ENGINEERING SECURITY METHODOLOGIES FOR DISTRIBUTED SYSTEMS

by

Anton Victor Uzunov

May 2014

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN THE SCHOOL OF COMPUTER SCIENCE
UNIVERSITY OF ADELAIDE
This page is intentionally left blank.
# Table of Contents

Abstract .................................................................................................................................................. ix  
Declaration ........................................................................................................................................... x  
Acknowledgements ............................................................................................................................ xi

**Introduction** ..................................................................................................................................... 1  
Background ......................................................................................................................................... 1  
Problem Statement and Research Objectives .................................................................................... 2  
Solution and Brief Outline of Contributions ...................................................................................... 3  
Thesis Structure .................................................................................................................................. 4

**Chapter 1: Engineering security into distributed systems: A survey of methodologies** .......... 6  
Prologue .................................................................................................................................................. 6  
1 Introduction ......................................................................................................................................... 8  
1.1 Scope and Organization .................................................................................................................. 9  
2 Taxonomy of Security Methodologies ............................................................................................. 10  
2.1 Background: Key Ingredients of a Security Methodology .......................................................... 10  
2.2 Classification Dimensions .......................................................................................................... 11  
2.2.1 Methodology Paradigm ........................................................................................................... 11  
2.2.2 Specificity ................................................................................................................................ 12  
2.2.3 Modeling Language and Notation ......................................................................................... 12  
2.2.4 Range of Security Properties ................................................................................................. 13  
2.2.5 Use of Formal Methods and Verification ............................................................................... 13  
2.2.6 SDLC Stages Supported ......................................................................................................... 13  
2.2.7 Ease of Use ........................................................................................................................... 13  
2.2.8 Tool Support ......................................................................................................................... 13  
3 Survey of Model-Based Security Methodologies ............................................................................. 14  
3.1 (General) Model-Based Methodologies ....................................................................................... 14  
3.1.1 Jürjens (MBSE / UMLsec) ...................................................................................................... 14  
3.2 Model-Driven Methodologies ...................................................................................................... 15  
3.2.1 Model Driven Architecture / Security (MDA/MDS) Methodologies .................................... 16  
3.2.2 Aspect-Oriented Software Development (AOSD) Methodologies .................................... 22  
3.2.3 Other Model-Driven Methodologies .................................................................................... 26  
3.3 Architecture-Driven Methodologies .......................................................................................... 26  
3.3.1 Ren / Taylor ............................................................................................................................ 27  
3.3.2 Ali / El-Kassas / Mahmoud ................................................................................................... 28  
3.3.3 Other Architecture-Driven Approaches ............................................................................... 28  
3.4 Pattern-Driven Methodologies .................................................................................................... 29  
3.4.1 Methodologies Using Security Components and/or Architectures as Patterns .................. 30  
3.4.2 Methodologies Using Security Patterns ............................................................................... 35  
3.5 Agent-Driven Methodologies ..................................................................................................... 40  
3.5.1 Mouratidis / Giorgini (Secure Tropos) ................................................................................. 40  
4 Evaluation ......................................................................................................................................... 42  
4.1 Criteria .......................................................................................................................................... 43  
4.2 Analysis Results ............................................................................................................................ 44  
4.3 Discussion ....................................................................................................................................... 45  
5 Conclusion and Future Directions .................................................................................................. 46  
References ............................................................................................................................................ 49  
Epilogue ................................................................................................................................................ 60
Chapter 2: Securing distributed systems using patterns: A survey ................................. 61
Prologue ................................................................................................................................. 61
1 Introduction .......................................................................................................................... 63
2 Background, Related Work and Organization of Patterns .................................................. 64
  2.1 Brief History of Software and Security Patterns .......................................................... 65
  2.2 Related Work on Security Patterns .............................................................................. 65
  2.3 Distributed Systems and their Security Concerns ......................................................... 66
  2.4 Scope and Organization of Patterns in the Survey ......................................................... 67
3 Survey of Security Patterns for Distributed Systems ......................................................... 67
  3.1 Security Contexts ......................................................................................................... 68
  3.2 Identity Management ................................................................................................. 68
  3.3 Authentication ............................................................................................................ 68
  3.4 Authorization and Access Control ............................................................................. 69
  3.5 Secure Communications ............................................................................................ 70
  3.6 Filtering ....................................................................................................................... 71
  3.7 Authentication/Authorization Frameworks and Languages ......................................... 72
  3.8 Distribution Control .................................................................................................. 72
  3.9 Processing .................................................................................................................. 73
  3.10 Monitoring ............................................................................................................... 74
4 Evaluation of Surveyed Patterns ......................................................................................... 74
  4.1 Classification .............................................................................................................. 74
  4.2 Analysis and Discussion ............................................................................................ 77
    4.2.1 Quality .................................................................................................................. 77
    4.2.2 Concerns covered ................................................................................................. 77
    4.2.3 Levels of abstraction ........................................................................................... 77
    4.2.4 Summary ............................................................................................................. 77
5 Methodologies for Applying Security Patterns ................................................................. 78
  5.1 Introduction and Classification ................................................................................... 78
  5.2 Survey ........................................................................................................................ 78
    5.2.1 Mature and/or larger-scale Methodologies ............................................................ 79
    5.2.2 Young and/or smaller-scale Methodologies ........................................................... 80
  5.3 Evaluation of Methodologies and Discussion ............................................................... 81
6 Conclusion and Future Directions ..................................................................................... 83
References ............................................................................................................................... 84
Epilogue .................................................................................................................................. 88

Chapter 3: A comprehensive pattern-oriented approach to engineering security
methodologies ....................................................................................................................... 89
Prologue .................................................................................................................................. 89
1 Introduction ........................................................................................................................ 91
2 Background ....................................................................................................................... 92
  2.1 Security methodologies ............................................................................................. 92
  2.2 Flexible security methodologies ............................................................................... 93
  2.3 Towards engineering security methodologies .............................................................. 93
3 Engineering a security methodology: security process aspect ......................................... 94
  3.1 SPPF: a framework of security process patterns and pattern modifiers ..................... 94
  3.2 Structure and presentation of SPPF .......................................................................... 96
  3.3 First SPFF level: generic life-cycle modifiers .............................................................. 96
    3.3.1 Requirements Analysis (reqAn) Phase pattern ....................................................... 96
    3.3.2 Design (des) Phase pattern .................................................................................. 97
    3.3.3 Implementation (impl) Phase pattern .................................................................... 97
    3.3.4 Deployment (deploy) Phase pattern ..................................................................... 97
3.4 Second SPPF level: generic security macro-process patterns ........................................97
  3.4.1 Overview ..................................................................................................................97
  3.4.2 Security Requirements Determination (SecReq) Phase pattern ...................................98
  3.4.3 Countermeasure Introduction (CounterIntro) Phase pattern ....................................100
  3.4.4 Security Implementation (SecImpl) Phase pattern ....................................................103
  3.4.5 Security Administration (SecAdmin) Phase pattern ..................................................104
  3.4.6 Refine task modifier [refine] ..................................................................................105
3.5 Using SPPF to construct security process models ...........................................................105
3.6 Engineering strategies for constructing security process models ......................................105
  3.6.1 “From-scratch” strategy: process pattern assembly ...................................................105
  3.6.2 Base methodology tailoring ..................................................................................106
4 Engineering a security methodology: conceptual security framework aspect ....................106
  4.1 Conceptual security framework meta-model ..............................................................106
  4.2 Engineering strategies for constructing conceptual framework models .........................107
    4.2.1 Meta-model instantiation ..................................................................................107
5 Notation for representing methodology models ....................................................................107
6 S-SMEP: a meta-methodology for engineering flexible security methodologies ................110
  6.1 Constructing S-SMEP ................................................................................................110
  6.2 Methodology Initiation phase ..................................................................................111
    6.2.1 Requirements elicitation and analysis (stage) .......................................................111
  6.3 Methodology Design/Modeling phase ..........................................................................111
    6.3.1 Selecting and organizing SPPF patterns (stage) ....................................................111
    6.3.2 Instantiating the conceptual framework meta-model (stage) .................................112
    6.3.3 Selecting existing methodology elements for tailoring (stage) ...............................112
    6.3.4 Verifying methodology model (stage) ..................................................................112
    6.3.5 Generalizing existing methodology elements (stage) ............................................112
  6.4 Methodology Construction phase ................................................................................113
    6.4.1 Instantiating SPPF patterns (stage) ......................................................................113
    6.4.2 Realizing conceptual framework model elements (stage) ....................................113
    6.4.3 Documenting the methodology (stage) .................................................................113
    6.4.4 Verification (stage) ..............................................................................................113
  6.5 Methodology Deployment phase ..................................................................................114
7 Applying S-SMEP to engineer a security methodology ........................................................114
  7.1 Methodology Initiation phase ..................................................................................114
    7.1.1 Requirements Elicitation and Analysis .................................................................114
  7.2 Methodology Design/Modeling phase ..........................................................................115
    7.2.1 Selecting and organizing SPPF patterns (stage) ....................................................115
    7.2.2 Instantiating the conceptual framework meta-model (stage) .................................115
    7.2.3 Selecting existing methodology elements for tailoring (stage) ...............................117
    7.2.4 Verifying methodology model (stage) ..................................................................118
  7.3 Methodology Construction phase ................................................................................118
    7.3.1 Instantiating SPPF patterns (stage) ......................................................................118
    7.3.2 Realizing conceptual framework model elements (stage) ....................................118
    7.3.3 Discussion ............................................................................................................119
  7.4 Feature analysis/screening of the approach ....................................................................119
8 Related work ....................................................................................................................121
  8.1 Security process patterns and method engineering .......................................................121
  8.2 Engineering development methodologies using process patterns ...................................122
  8.3 Mining process patterns .............................................................................................122
  8.4 Pattern-based process modeling notations .....................................................................123
  8.5 Relation to traditional SME approaches .......................................................................123
    8.5.1 Assembly-based SME in the context of security ....................................................123
## Chapter 8: A framework and process for security methodology quality assessment and improvement

### Prologue

1 Introduction .......................................................................................................................... 229
2 Background .......................................................................................................................... 230
   2.1 Cryptographic primitives and protocols ................................................................. 200
      2.1.1 Cryptographic primitives .................................................................................. 200
      2.1.2 Security protocols .......................................................................................... 201
   2.2 Security solution frames ............................................................................................ 201
3 Secure communications solution frame .............................................................................. 204
   3.1 Secure two-party communication pattern family .................................................... 204
      3.1.1 Secure message channel (abstract security pattern) .......................................... 204
      3.1.2 Pattern modifiers ............................................................................................ 205
      3.1.3 Solution design level ....................................................................................... 206
      3.1.4 Practical implementation details level .............................................................. 208
   3.2 Security process pattern family ................................................................................ 208
      3.2.1 Protocol design level ....................................................................................... 208
      3.2.2 Network configuration level ............................................................................ 209
      3.2.3 Implementation workflow for the secure communications solution frame (phase micro-process pattern) ................................................................. 209
4 Security information management solution frame .......................................................... 209
   4.1 Key management pattern family ............................................................................ 210
      4.1.1 Conceptual analysis level ................................................................................ 210
      4.1.2 Abstract architecture level ............................................................................. 210
      4.1.3 Solution design level ....................................................................................... 212
      4.1.4 Practical implementation details level .............................................................. 215
5 Identity management solution frame .............................................................................. 216
   5.1 Authentication pattern family .................................................................................. 216
      5.1.1 Conceptual analysis level ................................................................................ 216
      5.1.2 Abstract architecture level ............................................................................. 217
      5.1.3 Solution design ............................................................................................... 217
      5.1.4 Practical implementation details ..................................................................... 219
6 Analysis examples ............................................................................................................ 220
   6.1 Brief analysis: IBM Lotus Domino/Notes collaborative system ................................. 220
      6.1.1 Identity Management (Authentication) .............................................................. 220
      6.1.2 Security Information Management (Key Management) .................................... 220
      6.1.3 Secure communications ............................................................................... 220
   6.2 Detailed analysis: Plan9 distributed operating system ............................................. 221
      6.2.1 Identity Management (Authentication) .............................................................. 221
      6.2.2 Security Information Management (Key Management) .................................... 221
      6.2.3 Secure communications ............................................................................... 222
7 Related work .................................................................................................................... 222
8 Conclusion and future work ............................................................................................ 223
References ............................................................................................................................ 223
Epilogue ............................................................................................................................... 226
Chapter 9: A comprehensive security methodology for distributed systems

Prologue

1 Introduction

2 Background

2.1 Security methodologies

2.2 Construction and overview of ASE

3 ASE: conceptual security framework aspect

4 ASE: security process aspect

4.1 Requirements analysis (ReqAn)

4.2 Design (Des)

4.2.1 Security Requirements Determination (SecReq) phase

4.2.2 Countermeasure Introduction (CounterIntro) phase

4.3 Implementation (Impl)

4.3.1 Security Implementation (SecImpl) phase

5 Applying ASE in practice
Abstract

Over the last decade, researchers and practitioners have increasingly come to acknowledge that the introduction of security into software systems – especially complex, distributed systems – should proceed by means of a structured, systematic approach, combining principles from both software and security engineering. Such systematic approaches, particularly those implying some sort of process aligned with the development life-cycle, are termed security methodologies. While there are numerous methodologies in the literature, each with its own peculiar advantages and disadvantages, making it more or less suitable for a given set of project situations, none can lay claim to being universal, i.e. able to take into account all system-specific attributes, all technologies, all skill levels, and – in general – to be applicable to all project situations. In other words, the literature does not currently present developers with an “ideal” methodology (in an absolute sense); and, indeed, such a requirement would be infeasible, since “ideal” must necessarily be interpreted with respect to a given situation – encompassing system types, technologies, skillsets and whatever other qualities are seen as desirable. The problem facing the area is thus not so much the construction of “bigger and better” methodologies with novel or interesting features – i.e. (unattainably) ideal methodologies in an absolute sense – but the construction of (attainably) ideal methodologies for particular project situations.

This thesis proposes a comprehensive solution to the latter problem by developing a conceptual “toolkit” for engineering security methodologies, with an emphasis on security methodologies for distributed systems. The “toolkit” consists of a number of inter-related parts: a framework of process patterns, a domain-specific meta-model and a unifying meta-methodology for constructing and tailoring security methodologies (for any system type); a set of generic conceptual security artefacts – usable across different methodologies – for addressing various networked and distributed systems security features and for supporting threat modeling in a networked/distributed systems context; and a framework for assessing and improving security methodology quality, which, when combined with the meta-methodology, helps to ensure that all construction/tailoring efforts are sensible – i.e. of measurably good quality. Besides being part of an overall approach, each of these inter-related parts makes its own set of unique, self-contained contributions to the area of secure software engineering. Some of the parts are complete in themselves, while others require elaboration or specialization for different situations. In all cases the frameworks, artefacts and other “tools in the toolkit” can be customized and extended in various ways, providing developers, architects, methodology engineers and other team members with a high degree of freedom and flexibility. The latter point in particular, as well as the whole approach, is demonstrated incrementally throughout the thesis via the engineering of a specific pattern-based security methodology for distributed systems – a case study which in itself is another, final contribution. Of course, the case study methodology is also bigger and better (with respect to its predecessor) and contains novel features, but is only ideal with respect to its project situation. Through the presentation of the parts of the “toolkit” and the illustration of its use, the thesis accomplishes its task of both proposing and demonstrating the value of a comprehensive, holistic approach to engineering security methodologies, thereby offering a solution to the initial problem.
Declaration

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint-award of this degree.

I give consent to this copy of my thesis when deposited in the University Library, being made available for loan and photocopying, subject to the provisions of the Copyright Act 1968.

The author acknowledges that copyright of published works contained within this thesis resides with the copyright holder(s) of those works.

I also give permission for the digital version of my thesis to be made available on the web, via the University’s digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

___________________________
Anton Victor Uzunov
May 2014
Acknowledgements

First of all I would like to thank my two supervisors: Dr. Katrina Falkner and Prof. Eduardo B. Fernandez. I felt very fortunate throughout my degree to have two pairs of experienced eyes glance at my work and provide feedback from two different, complementary angles. I am particularly grateful to Katrina for promptly reading and always providing valuable feedback on all my manuscripts, even under excruciatingly tight deadlines; for encouraging me and supporting my work; for all the friendly discussions; for inspiring me towards excellence; and for allowing me to collaborate freely and trusting me to develop my ideas in the way I saw fit. I am particularly grateful to Eduardo for all his advice and his deeply insightful comments and suggestions; for always being “on-line”, even for late-evening Skype meetings spanning different continents, and being so generous with his time; for all the discussions on topics in software security and beyond; and in the end for helping me to realize what I myself had done (!). When I first spoke to Eduardo about external co-supervision a number of years ago, I was struck by his friendly unpretentiousness; in good time, I realized I had the privilege of meeting not only an immensely experienced and knowledgeable supervisor, but also a man with broad interests and a genuine friend. Collaborating with both my supervisors was always marked with mutual respect and understanding; in fact, I often felt as though I was working with peers or colleagues – something that I appreciated very much.

I would like to thank the University of Adelaide’s School of Computer Science administrative staff – Sharyn Liersch and Julie Mayo – for taking care of my conference travel arrangements; as well as the postgraduate coordinators – Dr. Michael Sheng and Dr. Frank Neumann – for their readiness to help with filling out all the Graduate Centre forms. I would particularly like to thank Michael for helping me to settle in the School of CS at the beginning and also for his friendly encouragement.

Thanks goes to the local IT-admin guru (and fellow PhD student) Yuval Yarom, not only for setting up my University machine (which was used rarely, but to good purpose), user accounts etc., but also for the comradeship. Shalom Yuval!

I would also like to thank my colleagues, supervisors and line managers back at work – especially Richard Appleby, Bradley Hopkins, Chris North and Dr. Olivier De Vel – for supporting me in my decision to undertake PhD studies. My thesis topic is all my own, but it really is in security; after gaining many skills and much experience and expertise, I believe that (besides coming out alive – not a bad feat) I managed to become a bigger and better security expert and, above all, a bigger and better security researcher.

Finally, I would like to thank my family – close and distant – for all their love and support, which, although mentioned here last, was indispensible. To my grandad I would say: no, I didn’t worry too much at the end, and, yes, I mostly “hurried slowly” – Eile mit Weile (as the Germans say).

Anton V. Uzunov
October 2013
This page is intentionally left blank.
Introduction

The mantra of any good security engineer is: “Security is not a product, but a process.”
It's more than designing strong cryptography into a system; it's designing the entire
system such that all security measures, including cryptography, work together.

– Bruce Schneier

Background

There can be little doubt that security is one of the most critical attributes of any software system. The consequences of the lack of security are grave indeed, and range from data theft, fraud, corruption of information and disruption of standard system functions to financial losses, loss of business and customers and even, in severe cases, loss of life (e.g. physical infrastructure systems). Perhaps with the exception of the last, most severe one, there is no dearth of examples of these consequences over the last 20 years from a correspondingly large range of systems straddling all levels of the software stack, beginning with large-scale commercial operating systems and ending in common networked applications [Anderson, 2008, Dowd et al., 2007, Gollmann, 2011, Hoglund and McGraw, 2004]. Security is particularly important in the context of distributed systems [Belapurkar et al., 2009] – defined most broadly from an architectural perspective as systems whose functional components are spread across multiple machines or nodes connected by some type of network. Such systems are uniformly susceptible to all the security threats of software designed for standalone machines, as well as a number of new threats stemming from the nature of distribution [Gritzalis et al., 1999, Muftic, 1989] and its realization in different middleware/platform varieties [Fairthorne, 1996, Schreiner and Lang, 2008], as well as the specifics of the particular distributed system type [Gymnopoulous et al., 2003, Pourzandi et al., 2005, Yue et al., 2009].

Given the great importance of security, it can appear surprising that with respect to the software development life-cycle (SDLC), security attributes are often considered according to the proverbial saying “too little, too late” [Fernández-Medina and Rodríguez, 2009, Sachitano et al., 2004, Meland and Jensen, 2008]. The research community has made numerous efforts to reverse this trend in the past decade [McGraw, 2004, Traore and Woungang, 2013], propelled by the realization that security should be introduced concomitantly with the standard software engineering activities performed throughout the SDLC – i.e. the introduction of security features into software systems should proceed by means of a structured, systematic approach, combining principles from both software and security engineering [Hein and Seadian, 2009, Mouratidis and Giorgni, 2006, Mouratidis, 2007, Rosado et al., 2010, Schumacher, 2003]. Such systematic approaches, particularly those implying some sort of process aligned with the development life-cycle, are termed secure software engineering methodologies, or simply security methodologies for short [Uzunov et al., 2012]. Security methodologies can be generic, or specific to a particular class of systems.

The advantages of utilizing a security methodology – for any system type – over an ad-hoc introduction of security features are completely analogous to the utilization of a software process for development over unprincipled code-hacking. For distributed systems in particular, security methodologies take on especial significance due to the higher degree of architectural complexity; the increased number of attack vectors (networked and host-based); and the larger number of components involved. Without a methodology, specific security solutions taken in isolation or as part of a security framework can easily degenerate into a collection of “techniques and mechanisms to solve disjoint problems” [Fernandez et al., 2006a, Fernandez et al., 2006b].
Problem Statement and Research Objectives

The secure software engineering literature presents a wide range of security methodologies – for general software systems, for general distributed systems, and for specific types of distributed systems – each with its own peculiar advantages and disadvantages, making it more or less suitable for a given set of project situations. Nevertheless, there is no single methodology that can lay claim to being universal, i.e. able to take into account all system-specific attributes, all technologies, all skill levels, and – in general – to be applicable to all project situations. In other words, the literature does not currently present developers with an “ideal” methodology (in an absolute sense), even though every individual methodology is, in many ways, an attempt to achieve this goal. Of course, if one asks “can there ever be an ideal methodology, in an absolute sense?”, then the only possible answer is that this would be practically unattainable, since “ideal” must necessarily be interpreted with respect to a given situation – encompassing system types, technologies, skillsets and whatever other qualities are seen as desirable. A fundamental problem facing the area is thus not the fixed, single-time construction of “bigger and better” methodologies with novel and/or interesting features – i.e. the construction of (unattainably) ideal methodologies in an absolute sense – but the construction of (attainably) ideal methodologies for particular project situations, which can be tailored further to suit new situations as the need arises, all “on-the-fly”. This problem gives rise to three research objectives, which are also the research objectives of this thesis:

**O1:** The creation of artefacts and/or processes addressing the construction, tailoring and/or combination of security methodologies.

It should be possible to devise a systematic approach whereby methodologies can be constructed from the ground up or tailored to suit the needs of a given project situation, defined in a loose sense as any set of constraints or requirements placed on the security methodology’s design.

**O2:** The creation of a collection of complementary artefacts and/or processes for supporting the introduction and/or analysis of security features – with relevance to distributed systems – across a range of methodologies.

Each methodology has unique characteristics: it uses a particular process together with a collection of (conceptual) artefacts – e.g. patterns, models, frameworks. These artefacts, and the methodologies themselves, are often not designed to be flexible in any way, and hence it cannot be expected that they are amenable to manipulation via some approach resulting from **O1** above. To support the construction and/or tailoring of methodologies, it should be possible to devise a collection of artefacts and/or processes that either reside on higher level of abstraction or are in some way complementary to a range of existing artefacts and/or processes, and that can be “deployed” as replacements or supplements in a range of methodologies. Since, as indicated at the outset of this Introduction, distributed systems form a particularly important class of systems from a security standpoint, these artefacts and/or processes should have relevance to such system types.

**O3:** The creation of artefacts and/or processes for qualitatively determining or measuring the suitability or quality of security methodologies resulting from construction / tailoring / combination efforts.

A very important notion when developing software products for a particular purpose is that of “quality” – whether the given product satisfies its requirements well, meets customer expectations and in general whether it possesses a number of desirable characteristics. This notion should be transferable to security methodologies and, in particular, it should be possible to devise an approach – either self-contained or as part of the artefacts/processes arising from **O1** or **O2** – to determine whether a given security methodology is of “good quality” (in some sense) and whether it can be
improved in any way to better suit the particular project situation.

The research question underlying these objectives can be posed in the following manner: *how can suitable, high quality security methodologies with relevance to distributed systems be designed and constructed for particular project situations delineated by a collection of security methodology requirements?*

**Solution and Brief Outline of Contributions**

This thesis answers the posed research question and realizes the latter research objectives by developing a comprehensive conceptual “toolkit” for engineering (constructing, tailoring, combining etc.) security methodologies, with an emphasis on security methodologies for distributed systems. The “toolkit” consists of a number of inter-related parts, corresponding to the research objectives:

- **O1**: a framework of process patterns, which represent a mixture of (security) best-practices and activities distilled from a number of existing security methodologies; a domain-specific meta-model, which captures the main (conceptual) artefacts used in security methodologies in general; and a unifying meta-methodology, which provides a systematic process for constructing and tailoring security methodologies.

- **O2**: a set of generic conceptual security artefacts – usable across different methodologies – for addressing various networked and distributed systems security features (authorization, authentication, secure network communications, security information management) and for supporting threat modeling in a networked/distributed systems context. These generic artefacts can be “deployed” independently or as groups into various methodologies to replace and/or supplement existing artefacts for the same or similar purposes (as defined in the aforementioned domain-specific meta-model). While really being generic, the artefacts can also be re-interpreted as being specific artefacts constructed for a particular methodology, i.e. they can be seen as illustrative examples in their own right of how (conceptual security) artefacts can and should be specified from scratch – as opposed to simply finding them in the literature – when engineering a security methodology.

- **O3**: a framework and accompanying process for assessing and improving security methodology quality, which, when combined with the meta-methodology, helps to ensure that all construction/tailoring efforts are sensible – i.e. of measurably good quality. The base framework can be extended via “situational profiles” for methodology features desirable on different project situations, allowing for fine-grained quality control.

Besides being part of an overall approach, each of these inter-related parts makes its own set of unique, self-contained contributions to the area of secure software engineering. Some of the parts are complete in themselves, while others require elaboration or specialization for different situations. In all cases the frameworks, artefacts and other “tools in the toolkit” can be customized and extended in various ways, providing developers, architects, methodology engineers and other team members with a high degree of freedom and flexibility. To give just a single concrete example, the generic conceptual artefact dealing with threats is based on a new type of abstract software pattern called a *threat pattern*, which encapsulates its architectural context as well as mitigating generic security policies. Using an elementary process, these patterns can be specialized for more specific system/technology contexts, allowing for the construction of an application-specific pattern-based threat library/taxonomy for supporting threat modeling and security testing.

Each of the aforementioned parts of the proposed toolkit are illustrated and evaluated separately, in keeping with their self-contained nature. The use of the toolkit as a whole is illustrated and
evaluated via the engineering of a specific pattern-based security methodology for distributed systems – a case study which in itself is another contribution. Of course, the case study methodology is also “bigger and better” (with respect to the predecessor on which it is based) and contains novel features, but is only ideal with respect to its project situation, delineated by a collection of methodology design requirements (desirable quality features). The actual evaluation procedure is thus based on a realistic example and feature screening (cf. [Kitchenham, 1996, Shaw, 2003]); however, the evaluation itself is somewhat more subtle and comprehensive, inasmuch as the part of the toolkit dealing with actual quality control essentially equips the meta-methodology with a method for evaluating itself, i.e. for evaluating the quality of the products (methodologies) produced. Thus, even if the approach overall was abysmal, it could still produce methodologies of measurably high quality regardless, which, in effect, allows most negative features, if present, to be negated (excepting learning curves and difficulty in application).

Thesis Structure

This thesis takes full advantage of a somewhat non-standard option for structuring PhD theses: a hybrid publications-narrative format. In keeping with the latter format, the body of the thesis is constituted of a mixture of publications/manuscripts (see Table 1) – some of which have been published (Chapters 1, 2, 4 and 5); some of which are under revision or submitted (Chapters 3, 6 and 7); and some of which are in preparation – Chapters in “manuscript form” (Chapters 8 and 9). The thesis begins with an introduction (this section) and ends with a conclusion, followed by a bibliography for the introduction and conclusion. Every (other) chapter has its own self-contained bibliography.

To make the account of the thesis more coherent, and thereby to reduce the effort required of the reader in piecing together the puzzle, so to speak, from the parts constituting the chapters, we preface each chapter with a Prologue, outlining where the chapter sits within the overall approach/solution; and end each chapter with an Epilogue, summarizing much the same. In this way the reader has the opportunity to consider the different publications/manuscripts (constituting the vast majority of the chapters) with a certain idea in mind, from the perspective of the thesis as a unified whole.

The chapter Prologues/Epilogues can also be read on their own as a single unit prior to reading the entire thesis from beginning to end, to give an overall picture of the individual chapters and the overall progression of ideas at a glance. Doing this also makes it possible to read some of the later chapters (after Chapter 1) in a shuffled order without any loss of coherency with respect to the central thesis solution. The fact that the generic artefacts (Chapters 4 to 7) can be interpreted as illustrative examples of how (specific) artefacts should be constructed – and hence can be seen as extensions of engineering the case study methodology – further permits a flexible reading approach.

The content of each chapter can be described as follows. In Chapter 1, we survey the state-of-the-art in security methodologies, which forms a foundation for the rest of the thesis. Following on from this survey, in Chapter 2, we confine ourselves to a particular artefact used by a particular class of methodologies – security patterns – and consider both the range of this artefact and how appropriate the methodologies which use it are for distributed systems. The core solution of the thesis, namely, the approach to engineering security methodologies, is presented in Chapter 3. The latter chapter also introduces the case study methodology as an illustration of how the approach can be applied. The next four chapters are concerned with presenting the generic conceptual security artefacts for characterizing distributed software architectures (Chapter 4); for constructing application-specific threat libraries/taxonomies (Chapter 5); and for various distributed system-specific security features – authorization and rule/policy management (Chapter 6); and authentication, secure communications and cryptographic key management (Chapter 7). Chapter 8 presents the quality framework and accompanying process for assessing and improving security methodologies, which is combined with the meta-methodology from Chapter 3 to show how methodology quality can be assessed and improved. The case study methodology is used as the
main example for the latter assessment/improvement process. A more complete description of this methodology appears in the final chapter, Chapter 9, together with an illustration of how it can be applied in practice in the design of a secure collaborative, distributed system. The final section of this thesis – Conclusions and Future Directions – summarizes the contributions and discusses future work stemming from the previous nine chapters.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Publication/Manuscript</th>
</tr>
</thead>
</table>

*Table 1: Publications/manuscripts constituting the thesis chapters*
Chapter 1
(Background & Literature survey I)
Engineering security into distributed systems:
A survey of methodologies

... though we cannot know these objects as things in themselves, we must yet be in a position at least to think them as things in themselves; otherwise we should be landed in the absurd conclusion that there can be appearance without anything that appears.

– Immanuel Kant

Prologue

Generally speaking, a secure software engineering methodology, or more simply, a security methodology, can be thought of as a collection of security-related activities and/or techniques – for example, threat modeling or penetration testing – and conceptual artefacts – for example, threat models or attack descriptions – employed in a systematic fashion during, or throughout, one or several phases of the software development life-cycle. Security methodologies lie at the centre of this thesis; therefore, in this chapter we comprehensively review a large portion of the existing security methodologies (emphasizing methodologies for, or applicable to, distributed systems), with a view to both introducing the reader to the domain and providing a foundation for the solution proposed in the rest of this thesis, namely, the engineering of security methodologies.
## Statement of Authorship

<table>
<thead>
<tr>
<th>Title of Paper</th>
<th>Engineering Security into Distributed Systems: A Survey of Methodologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication Status</td>
<td><img src="p" alt="Published" /> Published, <img src="o" alt="Accepted" /> Accepted for Publication, <img src="o" alt="Submitted" /> Submitted for Publication, <img src="o" alt="Publication style" /> Publication style</td>
</tr>
</tbody>
</table>

### Author Contributions

By signing the Statement of Authorship, each author certifies that their stated contribution to the publication is accurate and that permission is granted for the publication to be included in the candidate's thesis.

| Name of Principal Author (Candidate) | Anton V. Ljustov |
| Contribution to the Paper | Conceptualization of work (planned survey), its realization (research, analysis) and documentation (wrote manuscript), Acted as corresponding author. |

**Signature**

| Date | 11/10/2013 |

| Name of Co-Author | Eduardo B. Fernandez |
| Contribution to the Paper | Provided ideas and references, commented on manuscript versions. |

**Signature**

| Date | 13/10/2013 |

| Name of Co-Author | Katrina Falkner |
| Contribution to the Paper | Supervised development of work, helped to evaluate and edit the manuscript. |

**Signature**

| Date | 11/10/13 |

| Name of Co-Author |
| Contribution to the Paper |

**Signature**

| Date |
Engineering Security into Distributed Systems: A Survey of Methodologies

Anton V. Uzunov  
(University of Adelaide, SA, Australia  
anton.uzunov@adelaide.edu.au)

Eduardo B. Fernandez  
(Florida Atlantic University, Boca Raton, FL, USA  
ed@cse.fau.edu)

Katrina Falkner  
(University of Adelaide, SA, Australia  
katrina.falkner@adelaide.edu.au)

Abstract: Rapid technological advances in recent years have precipitated a general shift towards software distribution as a central computing paradigm. This has been accompanied by a corresponding increase in the dangers of security breaches, often causing security attributes to become an inhibiting factor for use and adoption. Despite the acknowledged importance of security, especially in the context of open and collaborative environments, there is a growing gap in the survey literature relating to systematic approaches (methodologies) for engineering secure distributed systems. In this paper, we attempt to fill the aforementioned gap by surveying and critically analyzing the state-of-the-art in security methodologies based on some form of abstract modeling (i.e. model-based methodologies) for, or applicable to, distributed systems. Our detailed reviews can be seen as a step towards increasing awareness and appreciation of a range of methodologies, allowing researchers and industry stakeholders to gain a comprehensive view of the field and make informed decisions. Following the comprehensive survey we propose a number of criteria reflecting the characteristics security methodologies should possess to be adopted in real-life industry scenarios, and evaluate each methodology accordingly. Our results highlight a number of areas for improvement, help to qualify adoption risks, and indicate future research directions.


Categories: C.2.0, C.2.4, D.2.0, D.2.1, D.2.2, D.4.6, K.6.5, L.4

1 Introduction

Ever since the advent of the Internet and its popularization, the face of computing has increasingly been defined by a physically distributed, collaborative model in which software components interact in parallel over a network – potentially spanning vast geographical distances – to perform complex computational tasks. Advances in hardware technologies for commodity parallelism and software paradigms for large-scale distribution [Erl 2009, Foster and Kesselman 2003, Milojicic et al. 2002, Vaquero- Gonzalez et al. 2009], as well as a range of new system varieties tailored for file sharing, resource aggregation, e-commerce and others, have ensured that businesses as well as social groups and individuals are increasingly exposed to a distributed computing environment.

Along with the fast-paced progress in technologies and paradigms, the corresponding dangers of intentional and unintentional security breaches have increased exponentially, so much so that among the more recent trends in distributed computing, security has become an inhibiting factor for adoption (see, for example, [Rosado et al. 2010a, Zhang et al. 2010]). While the importance and significance of security has never really been questioned, especially for open and collaborative distributed environments, its introduction into software by the industry has not been proportionally enthusiastic. Of course, there is no shortage of research in the area – a prodigious body of work addressing a myriad of access control models, intrusion detection techniques, cryptographic exchange protocols and many other concerns not only exists since the 1970's, but is being enlarged all the time. It appears, however, that much of this has remained in the realm of the theoretical, while software continues to suffer from vulnerabilities whose constant exploitation gives rise to ever new and interesting headlines in the popular news.

In a recent study of the state-of-the-practice in the industry, [Whyte and Harrison 2011] reveal a number of important factors influencing this slow uptake:

1. The business case for employing security best practices is missing;
2. Developers lack security expertise, which is currently required to employ security best practices;
3. The risk of committing to a particular security approach is too high.

Besides these points, and perhaps of greater importance, is the purport and overall conclusions of the study, namely: “a large majority of experts involved agreed that a very significant, if not the most significant, positive impact on secure software development would be concentration on measures that improve the overall quality of the ‘state of the
practice rather than the 'state of the art'." In this respect, "research will only have significant impact if it is compatible with the commercial environment of developers and their existent skills" [Whyte and Harrison 2011].

This indicates that for development teams to take security seriously it must be integrated into their everyday activities, i.e. security must be concomitant with software engineering practices, in a manner that is compatible with their current skillsets and (non-security) knowledge-base. In itself, this implication is not new: it has already been argued for in the research literature for over a decade [Fernandez 1999, Devanbu and Stubblebine 2000, Tryfonas et al. 2001, Ghosh et al. 2002, McGraw 2004, McGraw 2006, Jürjens 2005a, Mouratidis 2007, Anderson 2008, Jaatun and Tøndel 2008, Haley et al. 2008, etc.], with some researchers, e.g. [Mouratidis and Giorgini 2006] even envisaging an altogether new field of computing – secure software engineering – as its realization.

Such a full integration of security and software engineering is especially important in the context of distributed systems, where heterogeneity, collaborative behaviour and emergent properties all lead to increased complexity demanding systematic development approaches. A “systematic way of doing things in a particular discipline”, following [Gonzalez-Perez and Henderson-Sellers 2008], is termed a methodology; therefore, the systematic approaches referred to above can justifiably be termed secure software engineering methodologies, or more simply security methodologies, applicable to distributed systems.

A security methodology provides tools, techniques and processes throughout the software development life-cycle (SDLC) to guide the introduction of security and is thus, by its nature, focused on improving the state-of-the-practice in developing secure software. Such methodologies do indeed exist – in fact there are almost two dozen applicable to distributed systems alone – and, from our definition and previous remarks, their application certainly seems like a prudent best-practice that should be popular in the industry. Nevertheless, in light of Whyte and Harrison's findings, it appears that security methodologies are not being applied in any significant measure, or, if they are, they are not successful in their aims. Are these methodologies not widely known? If so, why are they not known? And if they exist and are well-known, why are they not receiving more attention from the industry?

Currently, there is no single survey that gives a broad and comprehensive overview of available security methodologies, not only applicable to distributed systems, but to general software systems, that can support answers to these questions. The more recent overviews of [Villarroel et al. 2005, Jayaram and Mathur 2005, Khan and Zulkernine 2009, Jürjens 2009, Fernández-Medina et al. 2009, Talhi et al. 2009 and Kasal et al. 2011] have all made important contributions in this regard, but, even taken together, do not provide essential details and comprehensive analyses that would allow a fair assessment of the range of methodologies available for practical use. Furthermore, very few (if any) security methodologies have ever been (independently) evaluated with respect to their potential for real-life adoption. This leaves a gap in the literature that, based on current practices and future directions, would inevitably grow in significance.

In this paper, we aim to fill the aforementioned gap by surveying and critically analyzing a wide range of security methodologies for, or applicable to, distributed software systems, both generally and for specific types of such systems, and evaluating their potential for real-life, industry use. Our selection of methodologies is based on their comprehensiveness, applicability and uniqueness, and is aimed at giving a comprehensive and detailed picture of the state-of-the-art. We have attempted to cover all relevant mature methodologies, as well as most emerging methodologies. In the instances where known omissions have been made, we either provide references or short summaries. It is hoped that our detailed reviews and subsequent evaluation provide a first step in increasing awareness about a range of security methodologies, allowing industry stakeholders to better qualify risks and make informed decisions in adopting a systematic security approach.

1.1 Scope and Organization

According to [Baskerville 1993], security design methodologies can be classified into several generations, akin to programming languages: first generation, ad-hoc approaches (approx. 1970s onwards), in which developers introduce security by consulting checklists of all possible solutions; second generation, requirements-driven approaches (approx. 1980s onwards), in which a set of activities are followed alongside (but apart from) the software development process to introduce security; and third-generation approaches (approx. late 1980's/early 1990s onwards), which are based on abstract modeling approaches concomitant with the development process. The focus of this survey stems from our definitions of security methodologies as systematic approaches combining security and modern software engineering, i.e. we consider third-generation security approaches based on some form of abstract modeling. As a side-effect to surveying these approaches in detail, which 20 years ago were only in their germinal stages, we extend Baskerville's work and bring the security methodology survey literature up to date, albeit from a distributed systems viewpoint.

To accomplish the goals set out above, we first present a taxonomy of security methodologies in [Section 2], discussing their main ingredients and most pertinent characteristics. In [Section 3], we review our selection of security methodologies according to the taxonomy. Our reviews attempt to provide a reasonable amount of detail (more rather than less), so as to promote a fuller understanding and appreciation of each approach, and to allow a more complete assessment of the (above all practical) value of any given methodology. This section therefore aims to address the question we posed previously – why are security methodologies not better known? In [Section 4] we
present a set of criteria reflecting industry requirements for adoption based predominantly on the study of Whyte and Harrison, and evaluate the different methodologies accordingly. The merit of the analysis results presented in this section lies in revealing the current deficiencies in existing approaches and establishing trends. This section is an attempt to address the other question we posed previously – why are security methodologies not receiving more attention from the industry? Finally, in [Section 5] we conclude and discuss future directions.

2 Taxonomy of Security Methodologies

2.1 Background: Key Ingredients of a Security Methodology

In this section we present some background material on security methodologies as a prelude to the taxonomy and the rest of the survey.

Previously, we defined a methodology as a systematic way of doing something in a given discipline. Another, more specific point of view on what constitutes a methodology is provided by [Booch 1994], according to whom “a methodology is a collection of methods applied across the software development life-cycle and unified by some general, philosophical approach”. The collection of methods referred to by Booch imply a set of activities performed by a methodology, which can provide further insight into the nature of security methodologies. To describe these activities, we first consider a methodology's alignment with a generic software development life-cycle [Ramsin and Paige 2008, Rodriguez et al. 2009] as illustrated in [Figure 1] below.

![Figure 1: Alignment of a security methodology with a generic SDLC](image)

In general, a software system passes from a set of requirements, to a set of models, to a set of implementation artefacts, until it reaches deployment. Analogously, a security methodology's activities can be aligned into (corresponding) requirements analysis, design, implementation and deployment stages. This alignment of activities may not perfectly fit the (generic) development process above, e.g. some or all of the “design” activities of a methodology may actually be performed during the “analysis” development stage; but in any given case the overall progression of activities will be the same, regardless of the overall development paradigms and process models used. In light of this alignment, we can briefly describe a (generic) security methodology's activities as follows:

— **Security requirements determination** (*Requirements analysis*): during this stage, activities are performed to determine a set of security requirements (see, for example, [Haley et al. 2008]). Some of these requirements are prescribed by the organization in the form of high-level policies, or are directed by regulations (government, state etc.). A methodology may further undertake some form of security assessment on conceptual or design models to determine and possibly enumerate threats and associated risks. This is often collectively termed “threat modeling” (see, for example, [Swiderski and Snyder 2004]). The outcome of these assessment activities are a set of resultant security requirements, which either augment or refine the initial prescribed set.

— **Security modeling** (*Design*): during this stage, security properties are modeled, usually as a fulfillment of the security requirements from previous stages, alongside the standard (architectural) modeling activities. Security modeling activities aim to embody the security properties in some form of architectural models, just as functionality is embodied in models during system design.

— **(Security) Implementation** (*Implementation*): during this stage, the security properties (as models, for example) from the previous stages are implemented, either manually, or automatically. This can include the generation of security configurations or parameters for certain software elements or the target platform, as well as manual coding activities and the selection of COTS components. Other activities may include testing, code-level security verification etc.

— **Configuration and monitoring** (*Deployment*): during this stage, a methodology may prescribe certain configuration activities, and may provide, through integral use of software support, monitoring facilities during system run-time. While not strictly part of a methodology's activities, such configurations and monitoring provisions may be an integral part of the overall approach (e.g. if a methodology is tied to a particular software framework).
2.2 Classification Dimensions

Security methodologies, like the systems they secure, can vary greatly from one another, depending on a set of fundamental characteristics. In order to encompass their wide variety, in this section we introduce a number of classification dimensions as part of a taxonomy of security methodologies. Our classification dimensions do not represent a complete set of features of a methodology, but attempt to capture the main characteristics. The dimensions are presented in order of those that are inherent to a methodology ([Section 2.2.1] – [Section 2.2.3]) and unlikely to change over time; those that are inherent but mutable ([Section 2.2.4] – [Section 2.2.7]); to those reflecting current work ([Section 2.2.8]).

The taxonomy complements the descriptions (in the previous section) of the general activities performed by a security methodology.

2.2.1 Methodology Paradigm

In [Section 2.1] we presented Booch's extended characterization of a methodology as collection of methods applied across the SDLC, unified by a general, philosophical approach [Booch 1994]. The philosophical approach referred to by Booch can be termed the methodology's paradigm, and is one of the most distinguishing features. A methodology's paradigm may influence its organization, prescribing certain activities to be performed during a certain life-cycle stage, and often determines corresponding development paradigms, integral use of standards and a concentration on certain software engineering artefacts or approaches. All the above allow us to divide security methodologies into a hierarchical set of distinct (though not always disjoint) classes.

2.2.1.1 Code-Based methodologies

Code-based methodologies attempt to introduce security into a system by enforcing certain security-related activities throughout a software process without explicit regard for a system's design or architecture. The main distinguishing feature of code-based methodologies is their focus of security-related efforts primarily or exclusively on requirements and implementation-level artefacts (hence the term “code”-based). Activities performed throughout the whole SDLC may include security requirements gathering, threat modeling (at either the requirements or design stage), penetration testing, developer education and others. Following [Baskerville 1993], code-based methodologies can be classified as second-generation security approaches.

The most widely recognized and used code-based methodologies are Microsoft's SDL [Howard and Lipner 2006] and OWASP's CLASP [OWASP 2011] [De Win et al. 2009, Khan and Zulkernine 2009]; other less known approaches include the early work of Breu and colleagues [Breu et al. 2004], who consider introducing security within the object-oriented development life-cycle, and Secure UP ([Steel et al. 2005], Chapter 8), a security-enhanced version of the Rational Unified Process (UP). Such code-based approaches have a number of benefits. The activities performed provide managed guidelines for software projects to reduce security flaws most broadly, and determine well-defined checkpoints at which to enforce security. Moreover, other, more specific methodologies, can be incorporated into a code-based methodology to enforce one or more security practices (e.g. secure coding).

Taken by themselves, however, code-based methodologies do not give concrete guidance on how to introduce security properties into a software system, or which such properties are required – they only provide a framework in which to use separate security improvement techniques (cf. [Schumacher 2003], Chapter 5). Thus, they only offer high-level solutions, without any “practical mechanisms” [...] that would permit [them] to implement the approach in a short space of time and with minimal effort” [Rosado et al. 2010a]. Their lack of a modeling stage in particular implies that security must be achieved via rigorous code inspections, testing and manual techniques.

By nature, code-based methodologies are generic, and tailoring them to a specific type of system (e.g. a distributed system) may require a significant amount of work, making them inappropriate for all system types (cf. [Rosado et al. 2010a]). All code-based approaches require significant security expertise in some form, such as a security expert (Microsoft SDL) who oversees the security-related activities or a dedicated team member (CLASP) who performs specific development tasks [Belapurkar et al. 2009].

[Gregoire et al. 2007] present a detailed comparison of Microsoft's SDL and CLASP, which is extended in [De Win et al. 2009] to include McGraw's Touchpoints [McGraw 2006]. Both papers discuss at length the activities done at each stage for each individual approach and in relation to each other. [Jayaram and Mathur 2005, Khan and Zulkernine 2009] collectively survey a number of code-based approaches, including Microsoft's SDL, CLASP and AEGIS [Flechais et al. 2003, Flechais et al. 2007], with concentration on the requirements and design stages.

2.2.1.2 Model-Based methodologies

In contrast to code-based methodologies, Model-based methodologies (the focus of this survey) are based on some form of abstract modeling and hence take a system's design or architecture into account explicitly. In this sense, they are the logical complement of code-based approaches.
Model-based methodologies form a very broad class of approaches that can be further divided into four hierarchical classes. A methodology can have characteristics from more than one of these classes, in which case the class into which most of its characteristics fall becomes its primary class, and the others secondary or even tertiary classes. In most cases, one characteristic will prevail over the others, allowing the methodology to be categorized in a single class. In what follows we briefly describe the (model-based) methodology classes. They are further described in detail throughout the survey ([Section 3]), in sections preceding the reviews of the methodologies belonging to a particular paradigm class.

— **Model-driven methodologies** ([Section 3.2]) focus on system models as first class entities within the MDE (Model-Driven Engineering) paradigm [Schmidt 2006], and add security properties by enriching the system's functional models with security artefacts. These methodologies can be subdivided further into Model-Driven Architecture/Security (MDA/MDS) approaches ([Section 3.2.1]), which use transformations on security-enriched models to generate implementation-level (security) artefacts; and Aspect-Oriented Software Development (AOSD) approaches ([Section 3.2.2]) – emphasizing separation of concerns – which model security as aspects and weave the aspects into system functional models.

— **Architecture-driven methodologies** ([Section 3.3]) are distinguished from model-driven and aspect-driven methodologies by the (broader) emphasis on software architectures as opposed to first-class models or aspects, and a focus on creating a “secure software architecture”. Architecture-driven approaches either typically use specialized Architecture Description Languages (ADLs) – [Taylor et al. 2010] to incorporate security at the architectural level – often employing formal methods to reason and analyze the resulting architecture – or base themselves on certain design principles.

— **Pattern-driven methodologies** ([Section 3.4]) emphasize the use of patterns as the central technique to achieve the introduction of security properties into a system, and can be subdivided further into methodologies utilizing components and/or architectures as patterns ([Section 3.4.1]) and methodologies utilizing on security patterns ([Section 3.4.2]).

— **Agent-driven methodologies** ([Section 3.5]) are based on the Agent-Oriented Software Engineering (AOSE) paradigm [Jennings 2001], and are distinguished by a goal-oriented approach and an explicit focus on software agents.

### 2.2.2 Specificity

A methodology can be applied to a wide range of systems or it can be tailored to only a small subset or a particular system type. Generic security methodologies can be applied to all software systems by providing universal guidelines, models, tools and techniques for the incorporation of various security properties. As a trade-off to their wide applicability, these methodologies do not address the challenges raised by particular system types. In such cases, a generic methodology's guidelines and techniques must be adapted for the specifics of the system, relying on the security expertise of the architects and developers. Specific security methodologies, on the other hand, are tailored and applicable to one or several types of systems, but may not be appropriate for securing more general systems. If one or more parts of a system employ, for example, different technologies or design approaches, specific methodologies can become inadequate. Between these two extremities is a whole spectrum in which some or all aspects of a methodology can make provisions for or be suited to specific system types. The latter can broadly be referred to as hybrid specific, or hybrid generic methodologies, depending on their accent.

### 2.2.3 Modeling Language and Notation

The modeling language and corresponding notation used by a methodology is often an important defining factor. Different modeling languages include a variety of ADLs (such as Darwin, xADL and AADL), UML, SysML, Petri nets and others. They differ in their syntax and semantics, ability to model the various aspects of a system, level of formality etc. [Taylor et al. 2010] provide a detailed taxonomy of ADLs that is applicable to modeling languages more generally, and can be referred to for more detail.

### 2.2.4 Range of Security Properties

Securing a whole system requires the use and application of a broad range of security properties. These are often divided among the categories of confidentiality, integrity, availability and accountability [Gollmann 2011], and include properties such as entity identification, non-repudiation, data origin authenticity and others. A methodology can support the introduction of many of these security properties or only one or several such properties. Some properties (e.g. access control) are arguably more important than others, particularly in distributed settings where systems are inherently more exposed to external threats (e.g. via the Internet). In the face of leaving vulnerabilities left open to exploitation, however, all security properties can become equally important. Ensuring that an attacker cannot gain access to a company database (access control), for example, is essential, however, it may be equally important to prevent an attacker eavesdropping on a communication channel carrying database data (communication secrecy). Ultimately, the aim of applying a security methodology is (or should be) to produce a secure system, and in this sense the scope of supported security properties to a large degree determines the utility of a methodology.
Adding support for the introduction of new security properties to a methodology is also an important characteristic in this context. In some cases it may take significant effort to interpolate or even extrapolate a methodology’s support for introducing new properties, while in other cases this addition may be trivial.

2.2.5 Use of Formal Methods and Verification

Using formal methods during the application of a methodology as an integral or optional element can give assurances that the security solutions introduced are correct, consistent and compatible, and, in some cases, verified or even certified to be such. The use of formal methods, however, also has disadvantages: it involves significant human labour, leading to increased project costs [Devanbu and Stubblebine 2000]; and it also impacts adversely on the ease of use of a methodology, requiring developer training in the relevant formal techniques.

Formal methods are usually used in conjunction with some form of security verification. Verification approaches may be manual, such as the creation of test cases; automatic, using specialized tools; or semi-automatic, and may be performed at any stage of development. In any of the above cases, the application of formal methods is almost invariably localized to critical aspects of a system: no single formal technique is capable of formally verifying a complete complex system.

2.2.6 SDLC Stages Supported

A methodology can encompass the whole SDLC, or only certain stages, i.e. one or more of the early stages, which we define to be the stages spanning across (early) requirements to (late) design. These stages are dependent on the paradigm used for development (e.g. object-oriented, functional, architecture-centric), and may include an analysis stage, an architecture-development stage and others.

2.2.7 Ease of Use

The ease of use or usability of a methodology is a qualitative measure of how easy it is, in relative terms, for an average software development team to introduce security into a system by applying the methodology. This is an extremely important factor in deciding the uptake of a methodology, since high complexity and high levels of expertise imply steep learning curves and, therefore, prolonged project schedules – making for an unattractive choice. Features such as tool-support, detailed guidance, usage of popular notations, support for re-usability and others all contribute to a methodology that is not only more usable but also more practical. Such methodologies would be classified as easy to use. Conversely, methodologies requiring detailed knowledge and application of formal methods, such as higher-order predicate logics and Petri nets, or domain knowledge, such as cryptography and complex security models, would be classified as difficult to use.

In assessing the ease of use of a methodology there is inevitably a factor of subjectivity, both from those performing the assessment, and the basis on which it is performed – namely, factors such as whether the most popular notation or modeling language is the easiest to use, the trade-off between a steep learning curve at the beginning and simple application of a technique thereafter etc. Case studies are more significant in this respect for showing how easy it is to use a methodology in actual experience, although here, too, it is not always clear how an approach can scale for larger and/or more complex systems. Despite the presence of subjectivity, the assessment of the ease of use of a methodology provides an important indicator when considering its real life application.

2.2.8 Tool Support

Tool-support for a methodology can be realized in many ways, for example, as a set of scripts to help transform models, a GUI-based application for specifying and cataloging threats, or a whole suite of CASE tools for modeling security architectures. Tools help to make a methodology more acceptable to a broader range of developers and, therefore, increase its practical value. If tool-support is not available, the methodology must rely on its ease of use in addition to using techniques realizable without significant developer or team effort; conversely, with tool-support a methodology can allow more sophisticated tasks to be accomplished automatically or semi-automatically.

3 Survey of Model-Based Security Methodologies

In what follows we review our selected range of security methodologies according to the taxonomy presented in the previous section. In order to provide more detailed reviews for more detailed approaches and retain relative similarity between review lengths, we have striven to review comprehensive methodologies in more detail, and partial methodologies in less detail proportionally, based on their scope and number of activities performed.

The methodologies are grouped based on their primary paradigms. Each review includes a very brief outline, capturing the essence or distinguishing elements of the methodology, a detailed description and a discussion, the latter concentrating on limitations and features of the methodology. The detailed description is segregated into the SDLC stages for which a methodology is applicable, though not always in the strict order (i.e. requirements analysis, then design, then implementation) of application. We preface each methodology paradigm with a very brief overview of the corresponding development paradigm and/or relevant concepts, insofar as these have relevance to
the security methodologies under review. The exception to this is the (general) model-based paradigm class, which was previously discussed in [Section 2.2.1.2].

3.1 (General) Model-Based Methodologies

3.1.1 Jürjens (MBSE/ UMLsec)

OUTLINE:
Jürjens [see Jürjens 2002a, Jürjens 2005a, Jürjens 2006a] was amongst the first to propose a modern model-based approach to securing distributed software. Jürjens’ Model-Based Security Engineering (MBSE) follows the standard life-cycle and corresponding engineering practices [see Section 2.1], and proceeds by first eliciting security requirements along with the system’s functional requirements; embodying the resulting security requirements in models during design; and finally implementing the models into code either manually or automatically. UMLsec—an extension of UML for the specification of security requirements—is the modeling language and core of Jürjens’ MBSE. Below we describe UMLsec and its associated tool-suite for design and implementation and then briefly mention approaches for security requirements determination.

DETAILED DESCRIPTION:
Design: UMLsec is a light-weight extension of the Unified Modeling Language, initially as a UML 1.4/1.5 profile in [Jürjens 2002a, Jürjens 2005a] and recently updated to conform to UML 2.3 via the UMLsec4UML2 profile in [Schmidt and Jürjens 2011b], defining stereotypes, tags and constraints for expressing a number of security requirements, such as data secrecy and integrity, message secrecy and integrity, information flow restrictions, role-based access control and others. Each stereotype is applicable to different UML diagrams, including deployment (accounting for communications between nodes), class (accounting for static data and relationships between elements), activity (accounting for intra-object properties) and sequence diagrams (accounting for interactions and protocols). By annotating UML models with the UMLsec security extensions, developers embody a system’s security requirements at the design level.

While useful in itself, simply annotating models with security requirements does not guarantee that the models will conform to their corresponding requirements. MBSE/UMLsec’s real value lies in the use of advanced tool-support for precisely this kind of security verification. The UMLsec tool-suite [Jürjens 2005b, Jürjens 2006b, Jürjens et al. 2008] consists of a number of analysis tools and plugins for verifying static and dynamic models as well as checking access control permissions. To support formal verification, the subset of UML employed by UMLsec is given a formal (behavioural) semantics via UML machines (see [Jürjens 2005a] for details), allowing for the execution semantics of UMLsec models to be converted to first-order logic and checked via automated theorem provers (e-SETHEO in earlier work, and more recently SPASS) against the relevant security requirements. If conformance fails, an attack sequence violating the security requirements is automatically generated and provided to developers for analysis.

For the purposes of verification, UMLsec allows the capabilities of adversaries to be specified via an adversary type, which formally models the threats related to the deployment view of a system, namely, those related to communication (delete, read, insert) and individual nodes (access). Two standard adversary types are defined in UMLsec that can be annotated on UML package diagrams: a default attacker, representing “an outsider adversary with modest capability” [Jürjens 2005a], and an insider, representing an attacker with internal access to a system.

Since the verification process can be applied to dynamic models captured as sequence diagrams, it is ideal for checking cryptographic security protocols (cf. [Jürjens 2006b]). In comparison to purely formal security protocol verification approaches [see Gritzalis et al. 1999] not employing the more intuitive UML notation, specifying and verifying protocols in UMLsec can be seen as one of its strongest points, albeit requiring expertise accordingly.

MBSE/UMLsec is tailored for object-oriented and component-based systems, making its applicability broad. UMLsec can also be used to specify security aspects [Jürjens 2006a] (see the beginning of [Section 3.2.2] for definitions) as well as security components and/or partial architectures treated as patterns ([Schmidt and Jürjens 2011a], see the beginning of [Section 3.4] for definitions), but, contrary to [Jürjens et al. 2002, Jürjens 2005a], is not appropriate for security patterns such as those found in [Schumacher et al. 2006], since the latter cannot be represented by a fixed, formally specifyable architecture.

Implementation: One of the problems with using MBSE/UMLsec is that heavy reliance on security verification at the design stage “has no implication on the real system, as long as the application code is not generated directly from the model[s] and there is no assurance that the generation process itself is correct” [Best et al. 2007]. According to [Jürjens 2006a], such assured generation is currently not feasible, leaving as an (only) alternative the necessity to perform verification procedures on the code resulting from UMLsec models. This is again achieved via the UMLsec tool-suite, augmented for the implementation stage, which can (1) help to ensure that the (manually developed) code resulting from UMLsec design models adheres to the specified security requirements by inserting run-time assertions; (2) automatically generate a series of test-cases to help guarantee code compliance; and (3) analyze (cryptography-related) source code by generating call-graphs, translating them into first-order logic—
analogously to the design-level verification procedures – and passing them to an automated theorem prover for checking [Jürjens 2006b]. Legacy code can also be a target of such analysis.

MBSE/UMLsec is unique among other methodologies in its direct support for implementation-level security.

**Requirements analysis:** To complete the MBSE approach, a security requirements stage is necessary in which the requirements that are later specified using UMLsec and verified to hold using the tool-support can be determined. There are several approaches that cover this stage. [Jürjens 2002b], for example, utilizes use cases and security goal trees to guide the introduction of requirements into UMLsec models; [Popp et al. 2003] employ UMLsec within an object-oriented methodology determining security requirements via security-extended use cases; [Hatebur et al. 2011] employ problem frames as patterns to capture requirements (cf. [Section 3.4.1.2]) and use OCL to generate UMLsec models; and [Mouratidis and Jürjens 2010] use Secure Tropos for security requirements modeling and architecture development, mapping the resulting architectural artefacts to UMLsec models during detailed design via a set of heuristics.

Despite the existence of several competing approaches for determining security requirements, none of them are really perfect matches for UMLsec. Hatebur et al.’s approach in particular is quite advanced and intricate, however, generating design artefacts straight from requirements ignores the activity of architectural design, making it appear a more theoretical approach applicable in practice to simple systems. The approaches of Mouratidis and Jürjens (which we discuss in more detail later from the perspective of the agent-driven Secure Tropos methodology in [Section 3.5.1]) and Popp and colleagues are attempts to integrate UMLsec into other methodologies, not the converse. Finally, Jürjens’ early work with goal trees does not include a comprehensive threat enumeration strategy. This leaves the requirements aspect less well-defined than the rest of MBSE.

**DISCUSSION:**

Overall, MBSE/UMLsec is a rigorous secure software engineering approach particularly well-suited for use during detailed design, with a powerful tool-suite for security verification. UMLsec's deployment view and network-related stereotypes («LAN», «Internet» etc.) make it quite applicable to distributed settings, (cf. [Jürjens 2002a, Best et al. 2007]), for example, demonstrate the use of UMLsec in the industrial development of a distributed (client/server) information system. UMLsec has also been used on a number of other real-life industry projects [see Jürjens 2005a, Jürjens 2006b]. Nevertheless, MBSE/UMLsec promulgates a formal approach that is at times complex [Best et al. 2007] and requires a reasonable amount of security expertise, though less so than purely formal approaches (cf. [Gritzalis et al. 1999]) due to the use of UML and the intuitive, often self-explanatory nature of the stereotypes. Where expertise begins to exponentially increase is in the modeling of security protocols and the development of custom stereotypes with custom tool-support (using UML machines). The latter, in particular, also requires strong mathematical skills and sound knowledge of formal methods, which not all developers posses.

Since no guidance is provided for actually designing secure systems, choosing security solutions etc., the MBSE/UMLsec approach can be seen as a “power user” modeling and analysis suite for guaranteeing security properties in systems for which security is a critical factor.

As slight digression, we should point out that more recently UMLsec has been described as a model-driven security (MDS) approach (see for example, [Fernández-Medina et al. 2009, Kasal et al. 2011]) as per [Section 3.2.1]. While MBSE shares certain similarities with MDS approaches, no model transformations are used (although the possibility for this does exist – see [Jürjens 2005a]), making it model-based, but not model-driven with respect to our taxonomy.

### 3.2 Model-Driven Methodologies

A model is an abstract representation, capturing some but not all aspects, of an actual system or entity. Model-driven security methodologies are based on the Model-Driven Engineering (MDE) paradigm [France and Rumpe 2007, Schmidt 2006], in which models are treated as first-class entities in the overall development process. A fundamental characteristic of MDE is the use of model transformations, either automatic or manual, between models (model-to-model transformations) across abstraction layers, or as a means for generating implementation artefacts (model-to-code transformations). While model-to-code transformations, otherwise known simply as code generation, is one of the attractive features of MDE and perhaps one of the most advertised, in practice the paradigm is also used as a means of obtaining re-usable models and as a disciplined and systematic development approach [Hutchinson et al. 2011]. All these benefits can be transferred to security methodologies within the MDE paradigm.

Model-driven methodologies can be further divided into two sub-classes described individually in [Section 3.2.1] and [Section 3.2.2].

#### 3.2.1 Model Driven Architecture / Security (MDA/MDS) Methodologies

Model-Driven Architecture is OMG’s realization of MDE [Frankel 2003], in which models are created in three levels of abstraction measured with respect to a particular platform and transformed gradually from one level to another. The levels correspond approximately to conceptual models (CIMs – computationally independent models), design models (PIMs – platform-independent models) and detailed, implementation-dependent design models
(PSMs – platform specific models). In MDA models can be transformed between and across levels to ultimately generate implementation-level artefacts.

Model-Driven Security (MDS) – a term first used by [Lodderstedt et al. 2002, Lang and Schreiner 2003] and, more definitively, by [Basin et al. 2006] – is an attempt to specialize the MDA paradigm to security by using models and transformations to enforce security properties from high-level designs to implementation artefacts [Fernández-Medina et al. 2009]. In MDS methodologies, a system's platform-independent functional models are annotated or composed with a well-defined set of security constructs, which can be verified and collectively transformed together with the functional models either directly or through intermediate stages to security configurations in some underlying platform or to actual software components (i.e. to code). [Fernández-Medina et al. 2009] survey the area of model-driven security both concisely and comprehensively, and we refrain from repeating their work here.

Although UML is the standard modeling language used in most MDS approaches, the use of other languages is also possible [see Dai and Cooper 2007]. Dai and Cooper's treatment of SecureUML as a modeling approach in that work is supplemented by our treatment of SecureUML in the context of Lodderstedt, Basin and Doeser's full Model-Driven Security methodology.

3.2.1.1 Lodderstedt / Basin / Doeser (Model-Driven Security)

OUTLINE:

The work of [Lodderstedt et al. 2002] arguably marked the beginning of the model-driven security paradigm with SecureUML – a security modeling language realized as a heavy-weight UML extension for allowing RBAC access control policies to be modeled concurrently with functionality at the design stage – within a code-generative approach. Lodderstedt's later work in [Lodderstedt 2003], and following it [Basin et al. 2006], generalizes and completes the approach begun with SecureUML for arbitrary security modeling languages while still retaining the focus on RBAC access control. The resulting (partial) methodology, termed simply Model-Driven Security, emphasizes the specification of security properties in design models (somewhat akin to MBSE/UMLsec), their verification and their automatic transformation to security configurations. The latter transformation can target arbitrary software platforms, however, in practice they are most useful in generating policy specifications for distributed middleware.

DETAILED DESCRIPTION:

Design and Implementation: The early work on Model-Driven Security begins with Lodderstedt et al.’s proposal of the SecureUML modeling language [Lodderstedt et al. 2002] for expressing extended RBAC policies in functional models during design that are subsequently transformed via a model-to-code transformation into security configuration parameters for the J2EE distributed platform. The latter ideas of modeling security at the design stage and transforming it to configuration parameters (or code artefacts, as required) are extended by [Lodderstedt 2003], appearing later in [Basin et al. 2003, Basin et al. 2006], as part of the fully developed Model-Driven Security methodology.

Model-Driven Security is based on a linguistic approach to modeling security, which proposes the creation of custom language schemas combining security modeling languages with functional (system or domain) modeling languages, enabling developers to model the security and functional aspects of a system simultaneously. To bridge the gap between the system and security modeling languages in each schema, a third, intermediate language – a “dialect” – is required, which tailors the security language in reference to a particular (system) language with more concrete or refined semantics and helps to merge the corresponding language metamodels (abstract syntax) and vocabulary. A dialect for a security language allowing the specification of access control policies such as SecureUML, for example, can specify which model elements of a system language can be valid protected objects and what actions can be performed on them. This general composition of languages results in families of secure design modeling languages that provide the security properties specifiable by each security language as system modeling constructs in a coherent fashion, with the concrete syntax (notation) used for functional models.

In order to realize this abstract vision of different language schemas and thereby produce a practical security approach, Lodderstedt, Basin and Doeser propose two example UML-based functional modeling languages in [Lodderstedt 2003] and [Basin et al. 2003, Basin et al. 2006] – ControllerUML for state-based systems; and ComponentUML for distributed component systems – along with SecureUML as a security modeling language. The proposed functional modeling languages are merged with SecureUML to produce two usable schemas: ComponentUML+SecureUML and ControllerUML+SecureUML, demonstrating the viability of the abstract approach. As in [Lodderstedt et al. 2002], the resulting languages can be used to create security-enhanced design models that can subsequently be transformed to security configurations in the supported middleware platforms; for example, ComponentUML+SecureUML models can be transformed to security policies for EJB as well as Microsoft’s .NET platform.

The verification of security-enhanced design models, first proposed as a goal for future work in [Lodderstedt 2003], is addressed in the work of [Basin et al. 2009], both abstractly and with relation to ComponentUML+SecureUML. The proposed verification strategy is based, like security modeling, on a linguistic approach, and consists of formulating queries in OCL about whether certain security conditions can hold and evaluating the queries with respect to the metamodel corresponding to a target design model. The approach requires...
the formalization of the modeling languages used (something already accomplished for SecureUML and ComponentUML in the previously cited publications), as well as the mapping of design models and possible run-time scenarios (expressed in object models, for example) to their associated metamodels for evaluation. Although supported by a modeling and OCL query tool (SecureMOVA), verification clearly requires an expert.

**Requirements Analysis:** The determination of security requirements is considered only briefly in [Lodderstedt 2003], who proposes a rights assignment scheme very similar to that of [Fernandez and Hawkins 1997] (see [Section 3.4.2.1] for more details on the latter). Nevertheless, security requirements determination is not an essential aspect of Model-Driven Security.

**Discussion:**

Most model-driven MDA/MDS methodologies have been influenced by the general strategy taken in Model-Driven Security of annotating higher-level models with security properties and transforming these to implementation-level artefacts. However, Model-Driven Security's abstract basis of creating general language schemas has not been as influential.

First of all, creating new security modeling languages is a demonstrably difficult task requiring expertise, as evidenced by the formal descriptions presented by [Lodderstedt 2003] and [Basin et al. 2006] of the abstract syntax and semantics of SecureUML (which, it must be kept in mind, can only specify RBAC policies). Formal specifications are necessary, however, for Basin et al.'s security verification strategy. Perhaps more problematic than complexity and high expertise requirements is that, in practice, the set of properties specifiable by a security modeling language is restricted by the capabilities of the target software platforms. Hence, according to [Basin et al. 2006]: “the modeling constructs in the security modeling language and their semantics should be designed with an eye open to the class of architectures and security mechanisms that will later be part of the target platforms. This requires care during the language design phase”. Thus designing new languages is not only complex, but can also be precarious. Even if a platform were to support a very wide (fixed) range of security mechanisms, there remains an issue of scalability: as the security semantics become richer, so does the complexity of the security modeling language.

As a practical methodology – consisting of SecureUML merged with ControllerUML and ComponentUML together with tool-support – aiming to generate security configurations for distributed platforms, Model-Driven Security is a sound security approach, albeit one focusing exclusively on RBAC policies.

The methodology has been trialled in the industry on at least one real project. [Clavel et al. 2008] report a successful use of Model-Driven Security with ComponentUML+SecureUML in the context of a test-report configuration utility, adding, however, that (good) tool-support is required for the wide-spread adoption of the methodology (in fact, at the time the developers did not actually implement a transformation function to generate the security configuration – this was done by hand). It is worth pointing out that Clavel et al.’s test-report utility had an almost identical design to the distributed calendar/scheduler application used as an example in [Lodderstedt et al. 2002]. Nevertheless, even the small-scale example in [Lodderstedt et al. 2002] showed an impressive improvement over creating manual security configurations in terms of effort and precision. [Lodderstedt 2003] also reports on using a prototypical e-commerce application for J2EE as a pilot study with similarly positive outcomes in generating security configurations.

As long as a new security modeling language is not required (i.e. if the existing SecureUML language schemas are sufficient for the task at hand), the ease of use of Model-Driven Security is relatively high, especially given the popularity of UML. However, since no guidance is provided as to where and why security specifications are introduced into design models, the methodology requires a security expert. In this sense, Model-Driven Security is akin to MBSE/UMLsec in helping experts to perform their tasks better.

Model-Driven Security is supported by ArcStyler plugins, and more recently by a custom Eclipse-based development environment for modeling and analyzing security design models (see [Basin et al. 2011] for details).

### 3.2.1.2 Hafner / Breu / Memon (SECTET / SECTISSIMO)

**Outline:**

SECTET [Alam et al. 2007, Hafner et al. 2006, Memon et al. 2008, Hafner and Breu 2009] is a comprehensive model-driven (MDS) (primary class) and pattern-based (secondary class) methodology consisting of a framework, run-time infrastructure and analysis process for securing SOA-based systems, concentrating on peer-to-peer, inter-organizational workflow scenarios. Like other model-driven approaches aspiring to produce security configurations from high-level models, SECTET's main goal is to support the annotation of functional business-process models with security properties, and to transform the resulting security-enhanced models to implementation-level artefacts in supporting web-service platforms.

**Detailed Description:**

**Analysis, Design and Implementation:** The main analysis, design and implementation level aspects of the methodology are embodied in the SECTET framework. In early work on SECTET, e.g. [Hafner et al. 2006], the framework's architecture closely adhered to the principles of MDA, and in particular Basin et al.'s MDS
specialization [Basin et al. 2006], by transforming security-enhanced functional models through two layers of abstraction corresponding approximately to PIM (security-enhanced functional models) and PSM to security configurations. The SECTISSIMO project [Memon 2011, Memon et al. 2008], which, judging by later publications, e.g. [Hafner et al. 2009] and presentations [Katt et al. 2010] seems to have been subsumed into SECTET, extends the latter architecture by adding an intermediate transformation layer, introducing an additional platform-independent, model-to-model transformation.

Using the SECTET framework (in its extended, current form), models are first created at the Security-enhanced Functional Models Layer using SECTET-DSL, a set of domain specific languages for business process modeling realized as UML 2.0 profiles [Hafner and Breu 2009]. Since the aim of SECTET is to support inter-organization workflows, the SECTET-DSL languages provide appropriate constructs to model documents, interfaces as well as global and local views for business processes. Together with modeling functionality using SECTET-DSL, models can be annotated with security requirements using SECTET-PL, a high-level policy language [Alam et al. 2007] to produce security-enhanced models. During development, these security-enhanced models are transformed via a model-to-model transformation to abstract security patterns and abstract policy models in the Abstract Security Models Layer, where they are subsequently refined manually by two degrees: the first by generating several platform-independent patterns and policy models at a lower level of abstraction; and the second by determining their platform-specific details (technologies, standards etc.). Pattern and policy model refinement thus gradually transforms security artefacts from PIMs to PSMs. In the last transformation step, the refined, platform-specific patterns and policy models are mapped and transformed, respectively, to security services and configurations in the Runtime Platform Layer (i.e. a compatible target platform), and subsequently deployed.

Implementation and Deployment: For the Runtime Platform Layer, SECTET utilizes a custom security as a service (SeAAS) infrastructure [Hafner et al. 2009] – inspired by the [ISO/IEC 7498-2 1989] security architecture for open distributed systems [Memon 2011] – in which security functionality is provided by a collection of flexible components or services. SeAAS is used during the SECTET framework's last (model-to-code) transformation step described above, in which the platform specific patterns are mapped to security workflows, i.e. they are transformed to compositions of SeAAS security services in BPEL, and policy models are transformed to security configurations of those services.

SeAAS is split into two layers of services: basic services like encryption and digital signatures, and advanced services like authentication, authorization, non-repudiation and others, which make use of the functionality provided by the basic layer. Instances of SeAAS are deployed in each participating organizational/administrative domain and connected by an Enterprise Service Bus (ESB), which can intercept all messages – thus playing the role of a reference monitor – and pass them to the SeAAS infrastructure. The decision of which security services to use, alone or in a composed sequence, at any given time is taken by the SeAAS Security Engine, thereby decoupling security functionality from application code. This approach promulgates a declarative security model, where most security-related configuration is placed in the underlying infrastructure of the application (i.e. SeAAS connected to the ESB).

Requirements analysis: The SECTET framework is complemented by a security analysis process, ProSecO [Breu et al. 2008, Hafner and Breu 2009], which consists of a series of steps to determine security requirements from general business-driven security goals. The latter is accompanied by a set of metamodels, which provide a (UML-based) conceptual framework in which to consider stakeholders, business concerns and system components (as services) at a large granularity level, from both global and local views. ProSecO is applied to a system's (local) model, which is decomposed into three layers according to the (local) metamodel: business layer, application layer and physical layer. The process proceeds by setting certain security goals, which in turn give rise to security requirements; considering the threats and their associated risks with respect to these requirements; and choosing a suitable (set of) countermeasure(s). This is done by further partitioning the system model into sub-models, determining a “dependency graph” of model elements that are related (top-down, starting at the business layer) and performing the analysis iteratively on each sub-model. The process lacks support for choosing the actual countermeasures (security controls), and this may be future work. ProSecO is supported by a tailored development environment to help with the task of annotating model elements with security artefacts (threats, risks etc.).

Although setting security goals and effectively refining them into security requirements is a useful endeavour in securing a system, associating threats and corresponding risks to model elements only with respect to security requirements fails to consider any attacks on the system, meaning that a valid threat arising from a possible attack will never be considered unless it falls within the scope of a security goal. The drawback of this scheme is that a system can be considered secure without actually being protected. In this respect, the remark by the authors that OCTAVE [Alberts and Dorofee 2002] can be used as a complementary process (i.e. a process to enumerate security requirements starting from threats, not the other way around) should be taken in practical terms, too, if complete security is to be achieved. Besides its goal-oriented nature, the main merits of the ProSecO security analysis process are in the use of tool-support, formalization via UML metamodels and the relating of security artefacts (threats and risks) to system models, as opposed to considering them in the abstract. ProSecO complements the SECTET framework (models to security configurations) with a corresponding set of security activities to determine the necessary security solutions (albeit with the limitation noted above).
DISCUSSION:

SECTET offers a detailed and strongly model-driven security approach. In comparison to Lodderstedt, Basin and Doeser's Model-Driven Security (see [Section 3.2.1.1]), SECTET can handle a much wider variety of security mechanisms, including authentication and non-repudiation. This is not altogether surprising, since SECTET takes its start from the latter approach, which is extended further and applied comprehensively. With ProSecO, SECTET also motivates the need for the security mechanisms as responses to threats within the context of set security goals. Nevertheless, SECTET is only applicable to inter-organizational workflow scenarios, and not to more general distributed settings.

In [Memon 2011], it is envisaged that the various activities performed when using SECTET are delegated to development team members with certain roles and experience. In particular, functional models are to be specified by a domain expert, who also determines the security requirements; a so-called framework engineer creates a catalog of security patterns, as well as their refinements, together with policy models for use in the abstract models layer; a security expert selects the apposite patterns and follows the process until code generation; and an application administrator takes care of deployment, configuration and management of the implemented security services. This distribution of roles fits the SECTET approach well, however, it implies that security expertise is essential to the success of the methodology. Security expertise is required in many forms: e.g. the application administrator should be versed in security concepts to configure and manage whole security infrastructures; the domain experts must possess sound security knowledge, otherwise it would be impossible for them to specify correct security requirements; and, finally, the framework engineer is really a security expert in disguise. These observations imply that SECTET is tailored for a very security-aware development team, making the methodology more restricted in scope. Re-use of the pattern refinements, models etc. across projects certainly helps to mitigate this situation, but not for the first project, nor with respect to differing security scenarios.

Regarding the use of patterns, strictly speaking, the security patterns in SECTET are not treated in the manner of the security patterns appearing in, for example, [Schumacher et al. 2006] (cf. [Section 3.4.2]), but are more properly intermediate abstractions of security mechanisms represented as models suitable for further transformation. Therefore, the advantages patterns have in guiding developers are not exploited. The modeling and refinement of security policies to configure the mechanisms represented by the patterns is a unique and important feature of the SECTET approach, and complements the use of models and patterns (as models) very well. The methodology has been trialled in real-life projects from the e-government and e-health domains [see Hafner et al. 2006] and [Hafner and Breu 2009], attesting to the applicability of the approach.

SECTET provides an extensive tool-suite, both for the security requirements process (ProSecO) and the model-to-code transformations. The SECTET-DSL profiles are compatible with a number of development environments [Hafner and Breu 2009].

3.2.1.3 Lang / Schreiner (OpenPMF)

OUTLINE:

[Schreiner and Lang 2008] present a partial model-driven (MDS) methodology building on a security enforcement system (PMF/OpenPMF) for various distributed middleware platforms. The approach is strongly influenced by the work of [Lodderstedt et al. 2002], aiming to specify high-level security policies in a domain-specific language and transform them to lower-level policies that can be enforced by PMF/OpenPMF. Below we describe Lang and Schreiner's the early work leading up to the PMF/OpenPMF system, and review the current form of the methodology afterwards.

DETAILED DESCRIPTION:

Design: Lang and Schreiner's early work [Lang and Schreiner 2003] presents an approach to securing distributed applications based on decoupling security policies from the software, security technologies and underlying distributed platform (middleware). The approach loosely follows the MDA philosophy, aligning the use of security policies and the relevant transformations to the MDA modeling levels (views) by analogy. Security policies are specified in PDL, a custom policy language that can capture complicated authorization and authentication semantics, allowing for the specification of a number of security models. Conceptually, security policies are mapped via transformations to the equivalent security functionality present in the underlying platform. With respect to implementation, this mapping is realized by extracting platform or technology specific attributes (transformation aspect) and matching policies against them at run-time (policy enforcement aspect). Access is mediated on a per-object invocation basis by the platform or middleware and intercepted by a custom Adapter software component, which passes on the mediated invocation to a dedicated Policy Enforcement component. Policy enforcement occurs by decomposing the set of policies into a tree-like hierarchy (which, most importantly, can take into account authority delegation), using a set of Transformer components individual to each platform to extract situational attributes and matching policy rules accordingly. The realization of this approach is embodied in the Policy Management Framework (PMF) and later OpenPMF [Lang and Schreiner 2004].

This approach is extended to address the limitation of manually specifying security policies in [Schreiner and Lang 2008]. There the authors state candidly their experience in a larger, real-life project with using OpenPMF alone and specifying policies by hand: “[the human administrator] had to use a primitive trial and error approach:
Start the system, wait for a policy violation, correct the policy, and go to start again! After about one week, the application was running without new policy violations, with a policy of about 1000 rules. The policy was far from optimal, and provided little assurance.”

In its extended (and now current) form the methodology strictly follows MDS principles by concurrently specifying security policies in SPL – a domain-specific language – together with functional system models as per MDA standards, transforming the security models to PDL policies and relying on PMF/OpenPMF for enforcement. This accomplishes Schreiner and Lang’s aim of “correct enforcement of correct policies” [Schreiner and Lang 2008]. The policies themselves reflect the available solutions in the underlying platforms supported by OpenPMF. As [Schreiner and Lang 2008] point out, the correctness of transformations, platform security capabilities, additional security functionality etc. all have to be considered, making the approach useful for security experts but not for the average developer. The emphasis of the methodology is thus on the design-level specification and run-time enforcement of security.

**DISCUSSION:**

Since Lang and Schreiner's work has close commercial ties including “patent-pending methods and algorithms” [Schreiner and Lang 2008] (one assumes for the high-level SPL policy transformations) the relevant publications are sparse in detail, making it very difficult to fully assess the merits and deficiencies of the approach.

As with Lodderstedt/Basin's MDS ([Section 3.2.1.1]), Lang and Schreiner's approach has been trialled on real projects. [Schreiner and Lang 2008] reports on their use of SPL and OpenPMF in an industrial air traffic simulation system, and note the positive outcome of producing a large number of valid PDL policies from a small set of high-level SPL policies.

The methodology is supported by an Eclipse-based tool-suite for specifying SPL policies, editing resulting PDL policies and performing automatic transformations. OpenPMF, currently in version 2.0, supports a wide variety of middleware platforms.

3.2.1.4 Gunawan / Hermann / Kraemer (security-enhanced SPACE)

**OUTLINE:**


SPACE is an MDE approach in which system functionality is specified using collaborative building blocks (or simply collaborations), which are higher level service models capturing the distributed functionality of a number of collaborating components, notated using UML 2.0 collaboration (static) and activity (dynamic) diagrams. SPACE collaborations can be composed hierarchically and combined in a “LEGO block” fashion to produce more complex functionality. The resulting architectural models can be checked due to their formal, temporal logic based semantics and transformed automatically to sets of components carrying the collaboration/service functionality, which are further transformed to Java code artefacts.

**DETAILED DESCRIPTION:**

**Design:** Gunawan et al.’s methodology extends SPACE by adding an additional security analysis stage and proposing a library of security building blocks that can automatically be integrated into a system’s functional models using tool-support. In the early work of [Gunawan et al. 2009], the process for introducing security consisted in: (1) creating a functional system design; (2) analyzing the design for valuable assets, assessing their value; (3) identifying threats to those assets; (4) assessing associated risks using a simple but effective matrix relating asset scores to threat scores; and (5) designing and integrating security countermeasures in the form of security collaborations/building blocks. The process is iterated between domain and security experts for each introduced solution.

In recent work on the methodology [Gunawan et al. 2011], emphasis is placed on automation via tool-support, thereby removing the need for domain/security expert iterations, but introducing a set of requirements on the original architecture for security integration that makes for a significantly more rigid approach. The current process replaces step (5) above with the following four steps: (5') identifying collaborations requiring the introduction of security countermeasures; (6) automatically checking that a system’s architecture fulfills certain pre-defined collaboration constraints for a given countermeasure; (7) automatically composing countermeasure security building blocks with identified collaborations via graph-based, information-flow algorithms; and (8) semi-automatically integrating the resulting secured collaborations into the functional models.

The use of security building blocks requires run-time support, and the approach currently considers only communications security between two participants (client/server), making its scope rather limited.

**DISCUSSION:**

The generative nature of SPACE and the high degree of automation in Gunawan et al.’s methodology is one of its most attractive factors. Since SPACE collaborations/building blocks are essentially template-like patterns or partial architectures containing specific functionality, they can be re-used across different systems. Nevertheless, no guidance is provided in actually choosing particular security building blocks. In both the older and most recent
variants of the methodology security is integrated after, not during, functional modeling, which implies that security introduction is effectively, despite the aim of early security integration, an afterthought to the design stage. With respect to implementation, it is not valid to assume as [Gunawan et al. 2009] do that the SPACE model transformations and subsequent code-generation preserve all security properties. It remains to be seen in future work whether the methodology can encompass security solutions other than TLS-based secure communications in a useful and relatively constraint-free manner.

3.2.1.5 Sánchez and colleagues (ModelSec)

OUTLINE:

Sánchez et al.'s ModelSec [Sánchez et al. 2009] is a partial model-driven (MDS) methodology applicable to general distributed systems, whose focus is on capturing security information and applying fine-grained model transformations across the development life-cycle to produce implementation-level security artefacts.

DETAILED DESCRIPTION:

Requirements analysis: In ModelSec, security introduction begins in the early development stages. During analysis, a domain-specific language (SecML) defined according to a general requirements metamodel is used for capturing security requirements. SecML allows common requirements and concepts to be represented graphically, helping to analyze security requirements, threats and goals in a single model. The resulting requirement model elements are synchronized with a corresponding conceptual model (which similarly conforms to the general requirements metamodel) by mapping assets in the SecML model to concepts in the conceptual model.

Design: During design, SecML requirements models are transformed via a model-to-model (PIM-to-PSM) transformation to generate “skeleton” security design models that conform to another metamodel capturing security design decisions. More specifically, security design models constitute platform-dependent encapsulations of technology-related decisions for each mechanism corresponding to a security requirement in the requirements metamodel. The latter are elaborated and transformed once again via a PSM-to-PSM transformation to a set of platform-dependent security implementation models, which further specify the design models with respect to a given platform. A number of such implementation models can be created, according to different target implementation artefacts.

Implementation: For implementation, the security requirements, design decision and implementation models, together with a set of platform metamodels capturing details about particular platforms for each implementation model, undergo another model-to-model transformation to produce a set of generation models from which code artefacts, such as policies or configurations, can be generated directly.

Discussion: ModelSec is a fine-grained model-driven approach aiming to encompass the whole SDLC and support developers with security expertise to generate security configurations more easily. The models used in ModelSec, however, do not model security solutions, but security-related information – requirements, technology decisions, platform details – in the form of schemas, meaning that the alignment to MDS is rather loose. ModelSec security models are not strongly related to a system's functional models, and in particular, security design models are not related in any way to the corresponding functional design models. Capturing design decisions as models is an interesting approach, but embedding technical decisions in the related metamodel implies that it is liable to change as new technologies appear. Regarding the transformation process, we should point out that the SecML models and corresponding conceptual models are actually computation-independent models (CIMs), not, as treated in ModelSec, platform-independent models. The implications of this are that the approach does not encompass PIMs, but transforms CIMs directly to PSMs, which carry platform-independent characteristics. Since design models do not capture solutions, but decisions, and are not related to functional models, verification is not an option.

The methodology is accompanied by an Eclipse-based tool-suite for SecML modeling and model transformations.

3.2.2 Aspect-Oriented Software Development (AOSD) Methodologies

An aspect, used as a technical term, is a particular part or feature of a program or design that cuts across or is present in numerous separate modules or components of a system. Historically, the use of aspects as a central concept takes its roots in Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] (see also [García et al. 2012] for a security-oriented discussion), where parts of a program's functionality are localized in aspects and woven (composed or integrated) at particular join points specified either manually or automatically. The ideas from AOP were subsequently applied to higher levels of abstraction in the work of [Clarke and Baniassad 2005], [France et al. 2004a] and others to form the Aspect-Oriented Software Development (AOSD) paradigm, focusing “on the identification and representation of crosscutting concerns, and their modularization in separate units [aspects], as well as their automated composition into a complete system” [Jakob et al. 2009]. In such aspect-oriented approaches, each concern embodied by an aspect is woven into a base or primary model describing some part (or the
whole) of a system's software architecture. A join point model defines where in that architecture aspects can be woven, while pointcuts determine a selection of possible join points. Depending on the approach, aspects can consist of pointcuts and/or valid join points and advice – the actual action or content of the aspect embodying the particular concern.

With respect to security, AOSD approaches allow developers to consider security concerns in isolation and weave them either into base design models to obtain integrated security-enhanced models or into implementation-level artefacts. AOSD methodologies are model-driven inasmuch as aspects are first-class models and aspect weaving is essentially a model-to-model transformation.

[Dehlinger and Subramanian 2006] survey several recent aspect-oriented approaches for securing software, among them [Georg et al. 2002a] and [Yu et al. 2005]. The former is treated without any detail, however, thereby not allowing a fair appraisal of Dehlinger and Subramanian's pointed criticisms. This is redressed by our review of the same approach in [Section 3.2.2.1] below.

3.2.2.1 Georg / Ray / France (AORDD)

OUTLINE:

[Georg et al. 2002a, Georg et al. 2002b, Georg et al. 2006, Georg et al. 2009, Georg et al. 2010], [Ray et al. 2004] and [France et al. 2004b] present a model-driven (AOSD) (primary class) and pattern-driven (secondary class) security methodology based on Aspect-Oriented Modeling (AOM) for securing general software systems. The approach is also applicable to distributed systems inasmuch as Georg et al.'s aspects can, at least theoretically, be constructed and/or tailored to account for more specialized security scenarios.

In Georg et al.'s approach, security solutions are expressed as aspects, which are treated as patterns based on the previous work of [France et al. 2002, France et al. 2004a]. The latter work is an attempt to make rigorous specifications of design patterns in UML by the use of role models (approximately meta-model level templates) and accompanying OCL constraint templates, which restrict the scope of possible pattern realizations in terms of model elements. Such rigorous specifications are useful because they determine the degree to which the design patterns (i.e. their models) can be varied. We should note that Georg et al.'s aspects [Georg et al. 2002a, Georg et al. 2009] are not to be confused with security patterns (cf. [Schumacher et al. 2006]), to which they are nevertheless related (see [Section 3.4] for more details). France et al.'s specifications are unable to capture the complete semantics of corresponding solutions, and, unlike security patterns, do not aspire to provide guidance or models of tried-and-tested solutions. In this sense, the aspects used by [Georg et al. 2002a, Georg et al. 2009] resemble the patterns used in the SECTET approach, which are likewise treated as first-class models, or in other pattern-based methodologies using template-like patterns ([Section 3.4.1]).

DETAILLED DESCRIPTION:

In the research group's early work [Georg et al. 2002a, Georg et al. 2002b], the approach can be described by a limited sequence of security activities: identifying necessary security aspects via a simple table mapping possible assets, attacks and corresponding countermeasures, and incorporating the countermeasures (aspects) by weaving them into a system's primary model. The (representative) countermeasures table is a good first attempt to identify security solutions, but is not a complete strategy.

With Georg et al.'s work on AORDD (aspect-oriented risk-driven development) [Georg et al. 2006, Georg et al. 2010] and the group's work on formal verification, the methodology is complemented by an attack modeling strategy, Alloy-based verification and trade-off analysis. In this latest version of the methodology, presented in [Georg et al. 2009, Georg et al. 2010], security introduction becomes an iterative six-part process, consisting of six consecutive steps: (1) determination of threats and risks using CORAS [Lund et al. 2011] alongside the elaboration of the primary model; (2) modeling of high-risk attacks related to threats as aspects; (3) weaving of the attack aspects into the primary model and formally analyzing the resulting misuse model to evaluate the attack's damage potential and hence to determine (resultant) security requirements, with the latter being mapped immediately to countermeasures; (4) introduction of relevant countermeasures as aspects into the primary model, producing a security-treated model; (5) re-weaving of the attack aspects into the security-treated model, followed by automated formal analysis, to evaluate the success of the security solution in mitigating the attacks; (6) calculating a fitness score for the solution using a BBN (Bayesian Belief Network). The process is repeated for each attack, to decide and trade-off between different solutions, and to discover newly introduced discrepancies such as solution conflicts. The approach is only applicable to the design stage. Below we describe the steps of this process in more detail.

Design: The first step in Georg et al.'s methodology is to use a security assessment strategy to evaluate the risks to an existing primary (system) model. CORAS is proposed for this purpose in [Georg et al. 2009], whose use results in a set of resultant security requirements embodied in the form of "treatments". The subsequent creation of misuse models and their analysis (discussed further below) effectively refines the risk (threat) model from CORAS in view of the primary model, allowing a more detailed set of security requirements to be specified. The overall approach is valid even if CORAS is replaced by another threat modeling/risk analysis method, making the methodology more flexible in this regard.
Following the determination of resultant security requirements, high-risk attacks are modeled as aspects, woven (as per the earlier work on the methodology in [Georg et al. 2002a, Georg et al. 2002b, France et al. 2004b, Ray et al. 2004]) individually into the primary model (one attack per iteration of the methodology) and verified, as explained below for security aspects, to produce a misuse model. If the verification results show that the attack is feasible, a relevant countermeasure is selected based on expert advice.

Currently, the methodology does not provide guidance or a structured approach for the selection activity. The range of countermeasures are those expressible as generic security aspects, which, judging by their availability, must be developed by security experts impromptu. Aspects appearing in the related literature include those for TLS secure communication [Georg et al. 2006, Georg et al. 2009], simple authentication and auditing [Georg et al. 2002b] and access control [Ray et al. 2004]. We should point out in this respect that [Georg et al. 2010] indicate their intent to create a repository of aspects associated with particular security properties, as well as to formalize dependencies between attacks and security solutions, which suggests that future improvements of the methodology may include features to support a more structured solution identification process using pre-defined solutions.

Once chosen, the countermeasures are subsequently woven into the primary model (from the previous methodology iteration, if applicable) to produce (a further) security-enhanced primary model. Aspect weaving is a two-phase process (following [Ray et al. 2004]), in which, firstly, the generic aspect (a role model-based pattern) is realized by manually mapping or binding the abstract roles (structural and behavioural) to existing primary model elements, if possible, or to new model elements, in both cases following the semantics and syntax (naming, concepts etc.) of the primary model, to create a context-specific aspect; and secondly, semi-automatically merging the context-specific aspect into the primary model, resolving conflicts where possible. Merging (UML) model elements can be accomplished via algorithmic strategies based on names [Straw et al. 2004] and signatures [Reddy et al. 2006], which can be tailored via composition directives, e.g. specifying which elements override others, rules for removal of elements etc. — see [Straw et al. 2004]. From a high-level perspective, the weaving process corresponds well to Fleurey et al.’s matching and merging model composition stages for AOM [Fleurey et al. 2008]. The aspect weaving process can also be thought of as a very rigorous but somewhat rigid form of pattern instantiation.

Once the aspect has been woven and all conflicts resolved, the attack aspect developed earlier can be re-woven into the security-enhanced primary model to determine whether the most recently woven countermeasure is effective in stopping the corresponding attack. This determination is accomplished with the aid of formal verification, which was also used to analyze the misuse models in the earlier stages of the methodology. The security-treated model (in UML) is translated into the Alloy formal language by simplifying the static UML models, writing OCL specifications of their methods (dynamic behaviour), and subsequently translating the simplified models together with accompanying OCL constraints using the UML2Alloy tool, see [Georg et al. 2009] for references and details. An Alloy Analyzer is used on the resulting model to automatically check for instances in which a particular condition – specified in OCL – fails.

The whole analysis process requires detailed specifications of the models such as would usually be obtained at the detailed design stage (complete method signatures, attributes etc.), which is slightly at variance with the aim of the methodology as stated in [Georg et al. 2009]: “our method can expose bugs and security issues of a system early in the development process before the model is refined enough to be implemented”. If verification is performed prior to detailed design, however, further refinement of the models may invalidate the results.

One of the most salient features of Georg et al.’s AORDD methodology is the trade-off analysis performed between different security solutions, following the Alloy-based verification process. A BBN is used to calculate a fitness score for a particular security countermeasure, which is “a measure of the degree that a particular security solution meets the security, development, project and financial constraints of the project” [Georg et al. 2010]. Each of these constraints, as well as the result from the verification (whether a solution mitigates an attack or not), is specified in the form of a number of parameters used as input to the BBN. The fitness scores can be used to guide the selection of which solution for a given attack should be chosen for implementation. This has the important ramification that a security solution failing to mitigate an attack can nevertheless be chosen for implementation, e.g. the case study in [Georg et al. 2010] based on its lower cost of implementation, project schedule constraints etc.

Trade-off analysis is a best-effort, semi-automated approach, but one in which subjectivity cannot be removed entirely, since the assignment of values in the BBN as well as its setup is a subjective process based on experience and expertise. Therefore, even more than the Alloy verification results, which are similarly best-effort, the fitness scores are not absolute indicators. Nevertheless, the approach offers developers support in making sound decisions in the face of numerous other project factors, and, given a correctly set up BBN, can be used by non-experts. Considering the impact resulting from the introduction of security aspects to other non-functional properties is a goal for future work in [Georg et al. 2010], which may be a valuable extension. Preliminary work addressing performance is presented in [Houmb et al. 2011].

DISCUSSION:

Overall, the methodology presented in [Georg et al. 2002a, Georg et al. 2009, Georg et al. 2010] is a design-focused approach distinctive for its use of rigorous security modeling and semi-automated trade-off analysis. While the trade-off analysis in particular is a unique and valuable feature, the methodology as a whole has several major drawbacks. Besides those already noted throughout the review, Georg et al.’s approach does not appear to be scalable to larger systems, and as the authors themselves state, “repeated generic aspect instantiation, composition
[weaving] and analysis can be tedious and error-prone” [Georg et al. 2009]. In many ways the introduction of security properties is also experimental, or ad-hoc: “it is important to continue integrating security mechanisms and analyzing the resulting security-treated system […] since some mechanisms may interfere with each other. When such conflicts arise, the designer can integrate alternative solutions until a usable combination is identified through achieving acceptable analysis results” [Georg et al. 2009]. The efforts required in aspect weaving, verification and trade-off analysis make the approach unrealistic in cases where more than just a (very) small number security solutions must be introduced.

With respect to guidance, in all the case studies the authors know a priori how to best construct and weave a security aspect into a system's primary model, but it never becomes clear how this is known, nor how the strategy can be generalized. Finally, from a more fundamental perspective, the application of the AOM paradigm as done by [France et al. 2004a], in which “fixed-to-measure” models of security solutions are rigorously specified, results in a need to consider fixed, automated weaving strategies via algorithms or the like carefully matching each UML model classifier, association etc., making for a rigid approach overall, in which the emphasis is on automation instead of guidance. While automation is a helpful feature, it is not possible to fully automate the securing of a system. The current lack of guidance and high expertise and effort requirements make Georg et al.’s methodology seem somewhat difficult to use.

The methodology is supported by a suite of tools: ArgoUML and UML2Alloy for the verification process [Georg et al. 2009], Hugin for the decision trade-off analysis [Georg et al. 2010] and custom tools for aspect weaving [France et al. 2004a, Mekerke et al. 2002]. It may also be possible to support the aspect weaving strategies using the KerMeta-based tool developed as part of the work in [Reddy et al. 2006]. More than other methodologies, the numerous descriptions of algorithms and processes of the AORDD approach indicate that using a software tool is essential for its practical use.

3.2.2.2 Mouheb and colleagues

OUTLINE:

[Mouheb et al. 2009, Mouheb et al. 2010] present a partial model-driven security methodology, similar in many ways to Georg et al.’s work described above. Mouheb et al. closely follow the AOSD paradigm by encapsulating security solutions in aspects using UML-based domain-specific languages, automatically weaving the aspects in base or primary models and subsequently (as a planned stage) generating code artefacts. Like Georg et al.’s AORDD, the approach is applicable to distributed systems on account of its support for constructing relevant, specific aspects.

DETAILED DESCRIPTION:

Design: One of the aims of Mouheb et al.’s approach is to lower the expertise required of developers in implementing security by localizing that expertise to security aspects. For this purpose, the approach provides a domain-specific language (DSL) in the form of a UML metamodel, together with a pointcut language, allowing both the structural and behavioural properties of aspects to be specified in the form of changes to the primary model (adaptations) together with the places in that model where the aspect can be woven (pointcuts). The resulting aspect specifications define parameterized model templates similar to the template patterns of [France et al. 2004b], in which the embodied security solutions can be re-used in different contexts. The simultaneous specification of valid join points in the primary model within aspects is a significant advantage over France et al.’s aspects [France et al. 2004b], since this information helps to better define the scope and applicability of a security solution. It should be pointed out that Mouheb et al.’s aspect and pointcut languages are not specific to security, but define general languages for specifying aspects in UML.

During design, developers select appropriate security aspects and specialize them using tool-support by choosing which elements (parameters) of the aspects map to which primary model elements according to the aspect-specified pointcuts. This can be likened to mapping pattern roles to valid model elements, and is very much analogous to Georg et al.’s context-specific aspect instantiation (see [Section 3.2.2.1] above). The mapping to model elements chosen by the developers defines a set of join points in the primary model around which the aspects are woven. The weaving process is defined in terms of model transformations written in QVT and is fully automated. Woven aspect behaviours appear in sequence diagrams as composed fragments that hide the underlying complexity, something especially useful in long security protocols. Unlike the similar process in Georg et al.’s approach of weaving using strategies, Mouheb et al.’s aspect weaving does not appear to address issues of conflict resolution. This may be problematic in instances where an aspect adaptation requires the addition of a class already contained in the primary model.

Requirements analysis: The specification of security requirements is mentioned as being part of the overall approach in [Mouheb et al. 2009], where Jürjens 2005a] is cited for further details. While UMLsec stereotypes do indeed appear in some of Mouheb et al.’s examples [Mouheb et al. 2009], the precise role of UMLsec, or requirements specification in general, remains unclear, which suggests that a security requirements stage is not an essential part of the methodology.
DISCUSSION:

Mouheb et al.’s work shares many similarities to Georg et al.’s AORDD approach with respect to aspect representation and weaving, arguably making improvements in both areas. One important element omitted from Mouheb et al.’s methodology, or at least its presentation in the literature, is a structured selection stage for the security aspects. Currently this selection requires security expertise, thus weakening the initial aim of the methodology to be developer-friendly. Missing support for conflict resolution in aspect weaving and a lack of security verification also leave open questions regarding the validity of the final security-enhanced models.

The methodology is supported by an IBM Rational Software Architect plug-in, used for aspect specialization and weaving.

3.2.2.3 Jakob / Loriant / Consel (DiaAspect)

OUTLINE:


DETAILED DESCRIPTION:

Design and Implementation: In the DiaSpec/DiaGen approach, a system’s software architecture is first modeled using DiaSpec – a lightweight ADL tailored for distributed (pervasive) systems. The architecture description is subsequently used to generate a typed programming framework with accompanying run-time support in the form of a custom middleware layer (DiaEnv) abstracting away underlying platform details.

To introduce security concerns, [Jakob et al. 2009] closely follow the AOSD paradigm and superimpose an aspect language called DiaAspect on top of the DiaSpec ADL that, together with its attendant join point model, associates pointcuts (effectively join point filters) with advice (security solution code) to allow the specification of a range of security concerns. DiaAspect's join point model is defined relative to DiaSpec's connector constructs encompassing synchronous RPC, publish-subscribe and session-based semantics, according to inter-component communication behaviour (e.g. a component publishing events).

In its particulars, DiaAspect is constructed to closely resemble AspectJ. DiaAspect's pointcut syntax and semantics allow for the specification of which join points should be selected for aspect weaving, while aspect advice (i.e. security solutions) is specified using Java. Aspect advice can encompass an arbitrary range of crosscutting security concerns. Since DiaAspect is built on top of DiaSpec, the advice in the aspects can refer to the architectural constructs of the ADL; however, since the latter are compiled by DiaGen to an implementation framework, the advice can also contain code referring to the associated (to-be-generated) framework via a specific API. Thus DiaAspect allows advice to refer to both architectural and implementation artefacts, making it quite powerful, albeit complex to use.

The process of introducing security properties consists in the (ad-hoc) modeling of additional security components and connectors, and the specification of security aspects (advice together with appropriate pointcuts) in DiaAspect at the architecture-development stage. Instead of weaving the DiaAspect aspects into the primary DiaSpec architectural models, however, they are translated to AspectJ aspects and, transitioning to the implementation stage, woven into the actual code-level artefacts, namely, the generated programming framework and the DiaEnv middleware layer. Since aspect weaving is performed on a codebase with known structure, precision can be ensured.

DISCUSSION:

Overall, Jakob et al.'s methodology is unique for its use of an aspect language on top of an ADL within a generative framework. The close adherence of Jakob et al.’s approach to the AOSD paradigm, coupled with strong similarity in syntax between DiaAspect and AspectJ helps to place the methodology in a more familiar context and thus increase its usability. Nevertheless, the absence of any guidance implies that security and domain expertise are required both in developing the security aspects and in determining via pointcut specifications where they should be woven. A real problem in this respect is that while the security solutions contained in the aspects can be verified independently, an overall correct security implementation relies on the correct specification of pointcuts, which is a potentially error-prone task even for experts. Some form of (security) verification would thus be required to make the approach more robust. Another, related, problem stems from the intermixing of architectural and implementation decisions in aspects. In particular, this may lead – at least on occasions – to a bottom-up (implementation-driven) approach to security, in which developers effectively code security solutions at the design stage.

The DiaGen/DiaSpec development framework comes with an Eclipse plugin, but it is not clear whether this is used in Jakob et al.’s methodology, which otherwise does not appear to be supported by any tools.
3.3.3 Other Model-Driven Methodologies

There are several other model-driven methodologies for securing general and distributed software systems that deserve mention.

[Menzel et al. 2009, Menzel et al. 2010] and [Menzel and Meinel 2009, Menzel and Meinel 2010], for example, specialize Model-Driven Security (see Section 3.2.1.1) to SOA-based systems using SecureSOA, a security modeling language inspired by SecureUML. Security requirements can be captured at the business process modeling level in platform-independent models and transformed to web-services policy specifications via a set of security configuration patterns (WS-specific security patterns formalized using a custom DSL). We do not cover Menzel and Meiner's work, since we have already reviewed a comprehensive model-driven approach for SOA-based systems (SECTET), and, furthermore, SecureSOA is a direct application of the principles outlined in [Basin et al. 2006].

[Yu et al. 2005] propose a partial model-driven AOSD (primary class) and architecture-driven (secondary class) methodology using the SAM ADL [see Dai and Cooper 2007]. A high-level SAM architectural model is separated into a base architectural model and a security aspect model containing the various security solutions in aspects. The latter are not discussed in detail, however, and there seems to be no structured approach to creating or selecting aspects. Only brief indications are given regarding aspect weaving, which, together with the extensive use of formal methods and lack of any tool support, implies a technically sound but unrealistic approach.

[Reznik et al. 2007] take a very similar approach to [Schreiner and Lang 2008], using Qedo (a CCM implementation) and OpenPMF combined with MDS principles for specifying high-level security policies. The emphasis of Reznik et al.'s work is on the creation of appropriate tool-support.

A more architecture-driven approach can be found in the work of [Oladimeji et al. 2007], who present a policy-based, software architectural perspective on secure model-driven development via a UML 2.0 extension. The emphasis, however, is on general software systems.

3.3 Architecture-Driven Methodologies

In contrast to model-driven engineering approaches, architecture-centric development approaches [Taylor et al. 2010] concentrate on a system's software architecture as the main artefact of importance. An architecture consists of more than a collection of models; it embodies a set of principal design decisions and determines the overall structure and behaviour of a piece of software in relation to various views [see Bass et al. 2003]. Accordingly, the aim of architecture-driven security methodologies is to create a secure software architecture: an architecture with embedded security properties.

As with their model-driven counterparts, architecture-driven security approaches generally begin with an architectural model of a system and annotate it with security properties to produce a security-enriched model (or security design model in the terms of [Basin et al. 2006]). The modeling languages used to achieve this security enrichment, the emphasis on architectures as opposed to first-class models and the absence of model transformation distinguish architecture-driven from model-driven methodologies. While model-driven approaches make use of extensions of UML, architecture-driven approaches typically use a precise architectural description of a system produced by ADLs and/or formal modeling methods. In architecture-driven methodologies, the verification of security properties often assumes a role of greater importance. Due to their nature, such methodologies generally concentrate on the design stage of the SDLC, without much regard for the remaining development life-cycle.

[Ren 2006] has surveyed a number of architectural-level approaches to introducing security, placing emphasis on modeling languages and older methodologies using formal methods such as [Moriconi et al. 1997] (architecture refinement through mappings) and [Deng et al. 2003] (Petri nets and temporal logic) aiming to implement security at the architectural level. We omit these, and review several more recent approaches instead.

3.3.1 Ren / Taylor

OUTLINE:

A representative (partial) architecture-driven security methodology is described in the work of [Ren and Taylor 2005], [Ren et al. 2005] and [Ren 2006]. The essence of the approach consists in extending an existing ADL to support security constructs and allow reasoning about security – specifically access control – at the architectural level via architecture analysis techniques [Taylor et al. 2010].

DETAILED DESCRIPTION:

Design: The substrate language used in Ren and Taylor's approach is xADL [Dashofy et al. 2005], an extensible XML-based ADL, whose semantics are enriched via a set of general security concepts to provide direct support for modeling subjects (users on whose behalf software components execute), principals (essentially the identity/ies of a subject), resources (protected entities), privileges (approximately subject permissions), safeguards (approximately object permissions), security policies (in XACML) and contexts.

In many ways, the resulting modeling language – Secure xADL – is a software architectural version of Lodderstedt et al.'s SecureUML. Unlike [Lodderstedt et al. 2002] and later [Basin et al. 2006], Ren and Taylor do not seek to generate code from their architectural models, but, as noted above, to analyze and reason about security properties prior to the implementation phase.
Secure xADL can model access control concerns at the architectural level by specifying “contracts” between software components via their associated access permissions and privileges. The security semantics are applied and enforced exclusively through connectors (for which see [Shaw 1996]), which are treated as first class citizens in all xADL architectural models. In this context connectors act as both policy enforcement and policy decision points [Gollmann 2011], and, above all, as multiple (or single) reference monitors. Higher-order connectors can be composed from multiple other connectors; connectors can be replaced (potentially dynamically); and connectors can be created from component specifications, to perform a (potentially) wide variety of security functions. Since xADL supports component compositions, the approach can account for multiple sub-architectures and extend access control to their constituent components and connectors. Since the approach works at the architectural level, it can also determine “architectural security” concerns, such as component creation and destruction, execution etc.

Analyzing the resulting secured architecture is achieved by using an algorithm that parses the Secure xADL model (which is represented as XML text) and validating the semantics. In doing so, the context for each component and connector must be taken into account and permissions and privileges aggregated to form an overall perspective on the access control (on a per component/connector basis). The algorithm provides assurances that the modeled access control is correct and consistent.

**DISCUSSION:**

Ren and Taylor's approach offers a comprehensive methodology for incorporating access control semantics at the architectural level. The approach has relevance to distributed systems inasmuch as it places preeminent emphasis on software connectors. However, the approach has several limitations.

Like [Lodderstedt et al. 2002] and [Basin et al. 2006], only access control semantics are demonstrated, although Ren and Taylor proceed further in allowing the semantics of multiple access control models to be captured – including those based on the traditional Matrix model (ACLs and capabilities), roles (RBAC) and trust-management (TMBAC). Nevertheless, delegation is not considered. In practice, access control policies are specified as part of the connector in XACML, limiting the range of access control models to those supported by the standard. Attacks and threats are also not considered, placing reliance on the security expertise of the architects. The largest drawback of the approach, however, is in its level of abstraction. Using connectors as security safeguards does not allow for the specification of detail, e.g. whether the connector is a firewall, a filter or some form of security proxy. Moreover, stipulating that security functionality should be placed in connectors is equivalent to stating that security should be placed in “special components”, which provides little practical help. A case study using a simple distributed (military) coalition application demonstrates the connector-centric approach by replacing a group of components and connectors with a single secure connector, which, while useful from a conceptual point of view, does not appear to address the issue of secure design.

Overall, the approach can be conceivably extended to encompass authentication and secure communications, but it is not entirely clear how easy it would be to encompass other security properties, and how many or what extensions (if any) would be required to Secure xADL and/or necessary verification algorithms.

Regarding ease of use, Ren and Taylor's methodology is quite easy to use, but requires a security expert due to the lack of support in modeling security properties. With respect to access control, the only currently supported property, the associated verification algorithm improves usability inasmuch errors can be detected and corrected early in the development process.

The methodology is supported by the ArchStudio development environment [see Dashofy et al. 2005]).

### 3.3.2 Ali / El-Kassas / Mahmoud

**OUTLINE:**

[Ali et al. 2009] present a partial architecture-driven methodology based on threat modeling, formal architectural modeling and formal verification approaches. The methodology builds on the work of a number of other architecture-driven methodologies such as [Deng et al. 2003] (see also [Ren 2006]), [He et al. 2002] and the threat modeling approach of [Myagmar et al. 2005]. Thus, it is representative of a range of formal architecture-driven approaches.

In Ali et al.'s approach, the ultimate aim is to construct a software architecture in which a number of security properties are embedded and verified to be correct. This is achieved by constructing a high-level architectural model, performing threat modeling to produce countermeasures, translating the countermeasures to formal models which are fitted back into the architecture and verifying the architecture using model checking. The approach applies only to the early architecture development stage, and only to SOA-based systems.

**DETAILED DESCRIPTION:**

**Design:** The methodology proceeds by first constructing a high-level architectural model of a system using Service-Oriented SAM (Software Architecture Model – [Dai and Cooper 2007]) – an ADL utilizing Petri nets and temporal logic as the underlying formalisms, tailored for web-services [Fu et al. 2006]. Petri nets in particular can be seen as modeling the information flow through the system. Threat modeling is then undertaken based on the architectural model, where the threats to vulnerable assets and access points are categorized according to STRIDE [Swiderski and Snyder 2004] with the additional extension that “architectural” mitigations (countermeasures), logical constraints...
and property specifications can be captured as well (using LTL – linear temporal logic – and informal descriptions). Security requirements resulting from the threat model are thus immediately mapped to particular countermeasures, which are translated to equivalent Petri net behavioural models (cf. [Deng et al. 2003]), and added to the architecture model as actual components to elaborate it further. Constraints link the new components into the architecture. Unless the actual mitigations are COTS components (modeled as “black-boxes” with just interfaces/ports), the threat modeling is re-iterated to refine their internal structure. Once the architecture with its components, connectors and constraints is completed, the architectural model is translated into an SMV (Symbolic Model Verifier) ready form. The security properties captured by the threat model (in LTL) are similarly translated to ensure that the properties hold in the presence of the corresponding threats (translated as a set of parameters influencing the model-checker). The counterexamples generated by the SMV model-checker are used to refine the architecture and the threat model iteratively.

[Ali et al. 2009] do not clarify whether the applicability of the approach to SOA-based systems extends beyond the ADL used (Service-Oriented SAM).

DISCUSSION:

Although Ali et al.'s approach is valid, offers justification for the introduced security properties using threat modeling and offers assurance via model-checking, the trade-off in complexity is not to the methodology's advantage. Extensive and almost exclusive use of formal techniques renders the methodology inaccessible to all but the most well-trained development teams. Besides expertise in formal modeling methods, a great deal of security expertise is demanded to specify the necessary security properties in the first place, without any guidance. While LTL is general and expressive, it only allows for low-level, technical policies to be captured. It does not appear realistic to apply Ali et al's methodology, and indeed any methodology with such a heavy use of formal methods, to large-scale and complex distributed systems; however, in small, critical applications, such approaches can help to provide the necessary assurance that certain security properties hold, at least theoretically, prior to implementation.

In the context of formal verification, the applicability of the approach only to architecture development seems to be a limitation: the guarantees that certain security properties hold may be broken during detailed design, for example.

Besides the SMV model-checker, there does not appear to be any tool-support for the methodology.

Related approaches relevant to distributed systems that concentrate on building and verifying an architecture with embedded security properties include [Deng et al. 2003], [Yu et al. 2003] and [Yu et al. 2004], all of who similarly employ SAM as a basis, but without suggesting any form of adversary modeling.

3.3.3 Other Architecture-Driven Approaches

Other, smaller-scale approaches that can be classified as architectural include [Schneider 1999], who considers security architectures abstractly in the context of information flow domains. Schneider's approach is surveyed in [Sachitano et al. 2004], who points out the unrealistic assumptions entailed by information domains when constructing actual systems.

[Jensen 1998] suggests securing distributed application architectures by considering component and connector security. Although the approach seeks to “integrate security specifications into the software architecture”, the (potential) solutions are described in the most general terms. Contrary to the stipulation that the approach applies to distributed applications, there is very little (if anything) specific to that setting, excepting several brief suggestions regarding access control (e.g. using domains and pseudo-users).

[Whitmore 2001] adheres to the Common Criteria (CC) and proposes a methodology in which security services are expressed as distinct subsystems, one for each class of security issues (e.g. authentication, authorization etc.). In this respect Whitmore's proposal resembles the [ISO/IEC 7498-2 1989] security architecture, transposed to meet the CC guidelines, as well as other methodologies discussed earlier which employ similar (generic) security reference architectures. It is not, however, altogether clear how Whitmore's separate security subsystems should be integrated with a system's functionality.

[Shin and Gomaa 2007] present a methodology in which the system is modeled using components and connectors, and afterwards made secure in some sense by evolving non-secure connectors to secure components and connectors "when the application requires security services". Only application level security is treated: the network and underlying operating system must be secured in some other way. Also, only a small range of well-known, standard security mechanisms are added when describing the process of evolution from non-secure to secure architectural constructs.

Finally, [Sachitano et al. 2004] argue that a software system can be made secure simply by good design or finding a correct architectural style that will ensure greater system security. The authors adduce qmail as an example of a secure application (cf. [Hafiz et al. 2004]), proposing that qmail has an architecture which is secure, since there have been no vulnerabilities found in the program. The design decisions taken by the qmail developer, however, such as "don't parse, do as little as possible" etc., are far too generic to be of use in particular situations or to dictate the structure or behaviour of a software system; principles such as keeping everything simple are not sufficient in guiding the system architect to create a more secure system, and do not constitute a universal approach.
3.4 Pattern-Driven Methodologies

The concept of a pattern in design and architecture began with the work of the architect Christopher Alexander in “The Timeless Way of Building” [Alexander 1979] and related works. Alexander conceived patterns as solutions that can be applied again and again in different contexts in a generative fashion. The idea was taken into the world of computer science by [Beck and Cunningham 1987], but reached its mass popularization in the book by [Gamma et al. 1995], in the form of design patterns. Software patterns have proved to be extremely useful in software engineering because they help organize expert knowledge into a standardized format; they directly support re-use; and they provide a domain-specific vocabulary, helping to improve communication in project teams [Buschmann et al. 1996, Erl 2009, Schmidt 1995, Taylor et al. 2010].

Pattern-driven security methodologies are distinguished by their aim to use patterns in a significant way for the introduction of security properties throughout the various stages of the SDLC. Over time, diverse interpretations of the pattern concept have emerged, leading to several distinct classes of patterns and, accordingly, of security methodologies using those patterns. For our purposes, we broadly distinguish between only two classes of patterns (and hence methodologies): (1) template-like patterns, appearing as software components and/or architectures (treated as patterns); and (2) security patterns.

Security patterns [Yoder and Barcalow 1997, Schumacher et al. 2006, Fernandez 2009, Uzunov et al. 2012] are full inheritors of the software pattern concept. Like software patterns in general, security patterns capture successful secure designs in a generic form that can be applied or instantiated to produce solutions with well-defined properties. They build on the success of design patterns and software patterns more generally, encapsulating the experience and expertise of security professionals [Schumacher 2003, Steel et al. 2005] in a form accessible to the security non-expert [Fernandez and Larrondo-Petrie 2006], including guidance for application, trade-offs, implementation hints etc. Security pattern instantiation implies highly flexible customization of a solution strategy, often (though not always) accompanied by representative architectural models. We review methodologies based on security patterns in [Section 3.4.2].

In contrast to security patterns, template-like patterns, in the form of parameterized model templates or interface specifications for security components or whole security architectures, are re-usable software elements that can be instantiated by replacing parameters in a fixed fashion akin to languages type-instantiation. Template-like patterns can be thought of as specific instances of the representative architectural models appearing as part of some security patterns. We review methodologies based on components and/or architectures used as patterns in [Section 3.4.1].

3.4.1 Methodologies Using Security Components and/or Architectures as Patterns

3.4.1.1 Rosado / Fernández-Medina / Lopez (PSecGCM / SecMobGrid)

OUTLINE:

PSecGCM ([Rosado et al. 2008] – also referred to as SecMobGrid in [Rosado et al. 2011b]), is a comprehensive methodology for Mobile Grid systems attempting to support all the stages of the SDLC, including system inception and requirements, analysis [Rosado et al. 2009a, Rosado et al. 2009b, Rosado et al. 2010a, Rosado et al. 2010b], design [Rosado et al. 2011a, Rosado et al. 2011b] and implementation. Every stage defines certain activities tailored to grid systems, taking into account mobility requirements, to output tangible software engineering artifacts. The latter include all the standard documents such as project scope and technology specifications for the planning stage, requirements specifications, architecture descriptions etc. Two of the main features of the PSecGCM methodology are requirements traceability and software re-use, realized in the form of a repository of re-usable artefacts including use cases and models. The latter are introduced to reduce development time and effort as well as to ensure that tried-and-tested solutions (in the spirit of software patterns) are being utilized. Individual tasks or activities in each phase are iterated with continual improvement and verification before proceeding to the next phase of the SDLC. Overall, the methodology aims to construct an "engineering process that defines the steps to follow so that starting from the necessities […] we can construct a secure grid system”. The three cornerstones of the approach are a specialized UML profile allowing security fine-grained use-case modeling at the analysis stage, an abstract security reference architecture used in the design stage, and a large repository of re-usable artefacts. Below we outline how each of these are used.

DETAILED DESCRIPTION:

Requirements Analysis: The introduction of security in PSecGCM begins in the analysis stage [Rosado et al. 2010a], where the analysis model is created using a custom UML profile called GSecUC-profile discussed in detail in [Rosado et al. 2009a, Rosado et al. 2009b, Rosado et al. 2010b]. One of the most important features of the profile is that it allows the modeling of security by providing semantics for graphically representing misactors, attacks, security countermeasures. Modeling elements can be tagged for greater detail, for example, by specifying kinds of attacks, assets involved etc. Use cases and models from previous iterations of the activity or the whole process can be taken from and recycled into the re-usable artefacts inventory to promote re-use. In terms of the analysis activity, a number of tasks are specified which, with respect to security, consist of functional modeling with UML using a standard software process (RUP, OPEN etc.), identifying assets, considering threats and attacks to the assets and
performing risk analysis. A number of security use cases, divided into general and grid-related, are the result of the latter task. The final analysis model consisting of an extended collection of UML diagrams with use cases, misuse cases and security use cases is carried over into the design phase.

**Design:** In [Rosado et al. 2011a] the authors present a “reference security architecture” for mobile grid systems with the idea of creating a service-oriented security infrastructure akin to SeAAS ([Hafner et al. 2009] - discussed previously in [Section 3.2.1.2]) to be used in the design phase [Rosado et al. 2011b]. Unlike SeAAS, the security reference architecture referred to here has an auxiliary methodological function, aiding the designer to relate security requirements to particular services. In essence, the reference architecture is a large architectural (template-like) pattern; its actual realization is dependent on the system architects. Rosado et al.’s reference architecture is thus somewhat similar in conception to the [ISO/IEC 7498-2 1989] Security Architecture or Cole's ECMA/TG9 security model [Cole 1990]. In terms of structure, the security reference architecture is divided into five layers with (approximately) each layer using the services of the layer below. Basic security services form the lowest layer (confidentiality, authorization etc.) while delegation, anonymity, trust and identity management and auditing services reside in the higher layers. The relationships between each of the services are considered in detail by Rosado and colleagues, as well as the interfaces of each individual service, determining its abstract operations. A very similar idea is proposed (but not realized) by [Naqvi and Riguidel 2004], who suggest a virtualized, pluggable security architecture for grid systems while also discussing mobility aspects. Although Naqvi and Riguidel's security architecture is more concrete, it is also less comprehensive in scope.

Rosado et al.’s reference security architecture is used in a central way during the design stage of the PSecGCM methodology, described fully in [Rosado et al. 2011b]. The design stage is separated into two parts: the construction of a software architecture and the construction of a security architecture, both of which must be integrated in some way and then documented according to IEEE 1471-2000. To create the security architecture, Rosado and colleagues mine and aggregate general security requirements for mobile grid systems from the open literature to form a tableau of requirements against which to consider countermeasures. Particular security use cases from the analysis phase are mapped to the set of requirements, which are then mapped, according to a predefined rule-set, to the security services of the security reference architecture in [Rosado et al. 2011a]. Only a subset of the reference architecture needs to be used for a given (general) security requirement, determined by the interrelations of the security services. For example, an “identify user” security use case can be mapped to the general requirement Authentication, which is mapped to the Authentication Service in the security reference architecture; however, the Authentication Service is related to services for identity management, confidentiality etc., thus combining a whole set of services to form the security architecture. The latter mapping process greatly helps with requirements traceability. Furthermore, mapping use cases to requirements to security services provides clear justification for the necessity of security mechanisms, and, if the reference architecture is considered as an abstract schema, the approach presents a basis for refinement to more concrete security solutions. The final security architecture is the largest interrelated subset of the security reference architecture obtained from the mapping of all possible security use cases via (general) security requirements to security services. As in previous activities of PSecGCM, the activity can be iterated for quality improvement.

**DISCUSSION:**

Unlike some of the other approaches surveyed so far, the PSecGCM methodology advocates good software engineering practices as a means to achieving higher quality software with less design flaws, which in turn implies improved software security. This is very much in accordance with the principles of secure software engineering [Mouratidis and Giorgini 2006]. Moreover, PSecGCM sees security required throughout the whole SDLC, not just a particular stage (as in, for example, Basin et al.’s Model-Driven Security), with use cases, generic security requirements and risk analysis helping to determine why and when a particular security solution is needed. Like other approaches though, the PSecGCM methodology has certain drawbacks and limitations.

One drawback of using abstract security architectures like Rosado et al.’s reference architecture lies in the fact that such architectures capture only general properties applicable to a broad class of systems. Rosado et al.’s reference architecture details interfaces and provides high-level requirements of the functionality of each security service as well as service interrelations, but does not consider actual service realization or internal design. According to [Rosado 2011b], “once we have defined the security architecture from the reference security architecture [by instantiating the relevant sub-architecture] […] it is necessary to design the final security architecture with UML diagrams, relationships between elements, protocols [etc.]”, which is where PSecGCM stops short of offering support. Another, related drawback of using the security reference architecture pertains to integration: it is not altogether clear in what way “[the resulting security] architecture must be integrated with the software architecture to obtain a software architecture with all the security aspects required by the users and the system” [Rosado et al. 2011a] in a systematic fashion. Understandably, security separation can be useful insofar that domain experts can concentrate on domain functionality rather than security aspects; however, the converse of this is also true: security designs disassociated from functional models or, as in PSecGCM, whole security architectures requiring some form integration with system functionality may, if not handled carefully, lead to design clashes and/or the traditional post-hoc introduction of security. It should be noted that such integration problems are symptomatic of using abstract security architectures generally, not just in PSecGCM.
In spite of these drawbacks, the reference security architecture is certainly a valuable tool in helping to build a global security architecture more efficiently, and the authors affirm its usefulness in their design case study (part of the GREDIA EU project – see [Rosado et al. 2011b]).

In [Rosado et al. 2011b] the authors suggest using MDS principles akin to [Basin et al. 2006] (see [Section 3.2.1.1]) to automate some aspects of the process, in particular transformation between different models, which may progress into a step towards automated integration.

PSecGCM is supported by SMGridTool, a custom CASE tool for building use case diagrams and managing the re-usable artefact repository during the analysis stage. Future work hinted in [Rosado et al. 2011b] may evolve the tool to encompass aspects of the design stage also (such as the security requirement mappings). The use of tool-support (essential due to the high number of development artefacts), clear traceability, re-usability features and standard development process features (including the use of UML as a modeling language) makes the methodology easy to apply.

3.4.1.2 Schmidt / Hatebur / Heisel (SEPP)

OUTLINE:

Security Engineering Process with Patterns (SEPP) [Hatebur et al. 2007a, Hatebur et al. 2007b, Hatebur et al. 2008, Schmidt 2010a, Schmidt 2010b, Schmidt et al. 2011] is a comprehensive, pattern-driven methodology applicable to the early stages of the SDLC, with a strong security requirements engineering focus. The approach is based on a partitioning and detailed analysis of the problem domain and the environment of a software system using abstract and concrete security problem frames and the elaboration of a corresponding security architecture via the composition of pre-defined sets of components, where both the security problem frames and corresponding components are treated as patterns.

DETAILED DESCRIPTION:

Requirements analysis: In SEPP, the process of introducing security begins after a set of initial security requirements are defined [Hatebur et al. 2008] both prescriptively and arising from some form of adversary modeling. These requirements are matched with a set of Security Problem Frames (SPFs), which are based on Jackson's problem frames [see Côté et al. 2008]) and treated as patterns. Problem frames are a technique to capture and analyze a system or software unit's problem domain in detail. The software system or system unit itself (called the “machine”) is placed in its environment, consisting of various physical and logical entities (domains) including users (biddable domains), data objects (lexical domains) and real-life entities (causal domains). SPFs in particular help to analyze a system's malicious and benign environments and capture the associated security requirements apart from any solution details. Concretized Security Problem Frames (CSPFs) on the other hand, make concrete, as the name implies, a particular SPF by embodying a particular security solution strategy, but without implementation details. By instantiating SPFs and CSPFs, a system's problem domain is effectively partitioned into a number of sub-problems and associated solutions. SPF and CSPF instantiation implies placing the software system or unit in a concrete context, whereby the problem-frame domains are mapped to context-specific domains. With respect to the development life-cycle, SPFs and CSPFs can be seen, respectively, as requirement and analysis stage artefacts.

SPFs and corresponding CSPFs in SEPP are organized as a pattern system by [Hatebur et al. 2007a, Hatebur et al. 2007b], in which relations such as “conflicts”, “complements” and others are defined. The pattern system can be represented as a 2-dimensional matrix or table [Hatebur et al. 2008] to aid the selection of relevant SPFs and CSPFs. A detailed solution selection stage via instantiating CSPFs is one of the salient features of the approach and also one of its strong points. After the initial security requirements have been analyzed and the relevant SPFs have been instantiated, corresponding CSPFs are chosen according to the columns of Hatebur et al.'s matrix in [Hatebur et al. 2008]. Related, required or alternative CSPFs are also selected and instantiated according to the chosen SPFs (matrix columns) and corresponding CSPFs (matrix rows) to cover new and existing requirements in an iterative fashion. After each CSPF has been considered, a threat and risk assessment is undertaken [Schmidt 2010b] to ensure that the CSPF's security solution cannot be bypassed or broken easily, and does not introduce any new vulnerabilities. Risk levels are considered according to assumptions about a system's environment, and, in the event of a weak solution for which the risk level is not tolerable, additional SPFs and CSPFs are instantiated. CSPFs and SPFs are thus iteratively and incrementally instantiated and assessed until all security requirements, both initial and those pertaining to the introduction of new solutions, are covered, thereby ensuring a good level of system protection.

The range of security solutions embodied in CSPFs focuses on confidentiality and integrity concerns and is relatively broad, covering communication and data security, as well as key distribution, security management and others. Extending this range, while requiring expertise, does not appear to be a difficult task, and would consist in fitting new CSPFs into the problem-frame pattern system (cf. [Hatebur et al. 2007b, Hatebur et al. 2008]) and evaluating conflicts, related patterns etc. for each existing SPF and CSPF.

As a next step, security requirements (and most notably confidentiality requirements) can be specified formally using CSP (Communicating Sequential Processes) ([see Schmidt 2010a] and [Schmidt et al. 2011] for details). In particular, the CSP specifications can be used to show that the instantiated CSPFs are step-wise refinements of the
corresponding SPF's. Although formal specification is an important feature of SEPP, we also consider it to be an optional feature, and do not describe it here.

**Design:** Once all CSPFs have been selected, indicating the end of the analysis stage, a system's security architecture can be elaborated. Security architectures in SEPP are comprised of a number of Generic Security Components (GSCs) and associated helper components termed Generic Non-security Components (GNCs), both of which are (simple) abstract components treated as patterns. GSCs include components for encryption, password input, hashing, access control and random number generation; GNCs are general purpose and include user interface, storage and communication manager components. The GSCs and GNCs can be composed in various combinations – somewhat like "LEGO blocks" to form Generic Security Architectures (GSAs), which are treated as larger-grained architectural (template-like) patterns, to satisfy the solution requirements embodied in particular CSPFs, and, from the point of view of problem frames, structure the CSPFs' machine domains. Each GSA also contains a manager (application) component, which acts as a facade to the interfaces of all the constituent GSCs and GNCs. [Schmidt 2010a] has defined a set of standard GSAs, each one corresponding to exactly one CSPF [see Schmidt 2010a], Ch. 14 for details). GSCs, GNCs and GSAs are all modeled using UML 2.0 composite structure and sequence diagrams.

During architecture development, individual GSAs as well as additional GSCs and GNCs are instantiated and composed to form a global security architecture. The approach is analogous to Hatebur and Heisel's problem-driven approach to constructing software architectures [Hatebur and Heisel 2009], in which a problem domain is recursively partitioned into problem frames and the solution to the individual sub-problems are composed and mapped to particular (pre-defined) architectural structures. In SEPP, dependencies between GSAs are analyzed according to the corresponding CSPFs and identical components across different GSAs are merged where possible (e.g. manager components). Wrapper components are used to resolve interface conflicts. The resulting global security architecture represents a complete solution set to the solution requirements in all relevant CSPFs.

In many ways SEPP's global security architectures are flexible and customized versions of Rosado et al.'s security reference architecture ([Rosado et al. 2011a], see [Section 3.4.1.1] above), sharing the similar drawback in requiring some sort of integration with a system's functional architecture. In contrast, SEPP requires careful composition of individual GSAs, which may not always be easy to achieve, and may give rise to emergent properties - something likely to be addressed in future work (cf. [Schmidt and Jürjens 2011a, Schmidt and Jürjens 2011b]).

**Detailed design:** In recent work, [Schmidt and Jürjens 2011b] extend SEPP to employ UMLsec via the UMLsec4UML2 profile (see [Section 3.1.1]) in the specification of GSAs. Individual UML model elements of the GSAs are annotated using UMLsec's stereotypes, tags and constraints such that they embody the security requirements contained in the CSPFs. This allows UMLsec's advanced tool-support (static checks, automated theorem provers etc.) to be utilized for verification, and is a logical extension of SEPP to the detailed design stage.

The view advanced by [Schmidt 2010a, Schmidt 2010b] as well as [Hatebur et al. 2008] that standard security patterns [Schumacher et al. 2006] are mainly used during detailed design (and, as a consequence, may be used after the stages covered by SEPP) is erroneous, since security patterns can be applied during both the analysis and architecture-development stages of the SDLC (cf. [Section 3.4.2.1]). Rather, security patterns are alternatives to CSPFs and GSAs, or, from another point of view, CSPFs and GSAs can be thought of as direct analogs of abstract and concrete security patterns respectively (cf. [Fernandez et al. 2008]).

**DISCUSSION:**

Overall, SEPP is a thorough security methodology with a strong focus on the problem domain and a detailed approach to selecting appropriate security solutions. The use of problem-frames and components as patterns promotes a flexible approach to introducing security, and helps to consider security properties within a system's unique context.

Although there is a good deal of formality associated with SEPP's SPF's and CSPF's, they are not difficult to apply. If the formal specification stage of SEPP is skipped, the approach becomes quite easy to use due to the clear SPF/CSPF selection strategy and the straightforward "LEGO block" approach to constructing security architectures via GSAs. Nevertheless, some security expertise is required since GSCs, GSAs etc. are abstract components and architectures without packaged guidance or trade-offs. Furthermore, familiarity and experience with problem frames is necessary to fully understand and appreciate SPF and CSPF descriptions. If the CSP formal specification stage is included, then the level of required expertise, as well as the complexity of application, is increased considerably and the methodology becomes cumbersome to use.

With respect to the development life-cycle, SEPP is quite a problem-domain heavy approach, i.e. the problem domain determines everything in the solution domain in a "surjective" fashion. Analysis models (contained in SPF's and CSPF's) are not used to guide the development process to a solution, but to enforce a set of possible solutions. Thus, solution-space considerations, such as constructing an optimal software architecture, the interplay between the software and security architecture of an application, are not considered.

Regarding the nature of the approach, problem frames are not as popular as, for example, use cases, which has a bearing on the adoption of SEPP in real-life projects. [Hatebur and Heisel 2010] have created a UML profile...
allowing for SPFs/CSPFs to be modeled with UML, which offers an improvement over using problem-frame specific diagrams, and may be a beneficial future improvement to SEPP.

The methodology is supported by CSFPTool, an Eclipse plugin for Linux that allows the creation and verification of SPFs and CSPFs. An Eclipse plugin is also available for the UMLsec extension of SEPP.

3.4.1.3 SERENITY

OUTLINE:
The SERENITY project [Maña et al. 2006, Spanoudakis et al 2009] is an effort to address the security and dependability (henceforth S&D) issues peculiar to Ambient Intelligence (AmI) environments. AmI environments can be characterized by possessing a high degree of dynamism, heterogeneity and, above all, distribution. Somewhat like SECTET ([Hafner and Breu 2009], [Section 3.2.1.2]), SERENITY is a methodology consisting of a software development and run-time framework with an accompanying development process.

DETAILED DESCRIPTION:

One of the major goals of SERENITY is to provide adaptable security via application run-time support and variable componentized security solutions. To realize this goal, the approach employs a formalized extension of security patterns embodied in S&D solutions [Gallego-Nicasio et al. 2009, Maña et al. 2006, Maña and Pujol 2008, Serrano et al. 2008, Serrano et al 2009a, Serrano et al. 2009b], which are the vehicle for security mechanisms. S&D solutions lie at the center of the SERENITY approach. The SERENITY framework supports a hierarchy of S&D solutions at several layers of abstraction:

1. S&D Classes, at the highest level, which guarantee particular semantics and a common interface, and abstract the S&D properties provided by a collection of S&D patterns.
2. S&D Patterns are concrete, individual specifications of solutions with precise formal semantics having multiple possible implementations. These implementations are represented as:
3. S&D Implementations, which are specifications of run-time solutions, realized as:
4. Executable Components, in the SERENITY run-time environment (described below).

The solution hierarchy is thus composed top-down, i.e. from abstract solution classes to concrete solution specifications to implementation-dependent specifications and finally to actual components. SERENITY’s S&D patterns are fundamentally implementation-oriented. According to [Sánchez-Cid et al. 2009], “S&D Solutions are well-defined mechanisms […] that provide one or more S&D properties”, while “S&D patterns are semantic descriptions of S&D solutions” containing the necessary information for the “selection, instantiation and adaptation, and dynamic application” of S&D solutions during run-time.

S&D Classes and Patterns must be rigorously specified by a security expert using an XML template. Patterns and associated implementations in particular can be formally certified, and are envisaged as “off-the-shelf” re-usable building blocks. Pattern specification include at least: (1) a well defined interface; along with Adapters bridging the interface to that of any associated S&D classes; (2) pre-conditions for applying the solution; (3) S&D properties provided; (4) some form of certification; and (5) a set of monitoring rules. The resulting artefacts are stored in a library for use during development and also at run-time. Some examples of S&D Patterns and Classes can be seen in [Dolinar et al. 2008] and [Sánchez-Cid et al. 2009].

The SERENITY framework itself is split into two parts: a development framework (SDF) and a run-time framework (SRF). The development framework provisions selection of S&D solutions by developers and also provides tools to develop new S&D solutions. The run-time framework deals with the monitoring of security solutions, their instantiation as executable components and their adaptation as the need arises. Monitoring and execution support help to realize the goal of providing adaptable security at run-time, allowing different security solutions (as executable components) to be swapped dynamically.


At the beginning of the process S&D experts must determine any new solutions to be used and codify them as S&D Classes or Patterns; the latter are then coded by S&D developers as Executable Components, and fitted into the framework by writing corresponding S&D Implementation specifications.

Requirements: During the early development stages, application developers must identify the necessary S&D properties from the target application’s security goals. A custom language [Serrano et al. 2009a] providing simple constructs for expressing, amongst other features, (canonical) S&D properties and contexts is used to specify the S&D requirements, which are then used as semantic queries in the SDF to determine a set of conforming S&D solutions that can be selected.

Analysis and Design: A UML profile is used for notating required S&D properties on classes during analysis and representing relevant S&D solutions (as single classes) during design. The requirements-as-queries from the earlier stages are used in conjunction with the SDF to determine the set of all available S&D solutions for selection, thereby establishing a conceptual mapping from S&D requirements expressed as required S&D properties to S&D solutions providing those properties. This mapping is realized by the selection of an appropriate solution at some level of
abstraction (i.e. Class, Pattern or Implementation). The higher the level, the more choice is left for the SRF
algorithms to make the decision as to what security solution is best during run-time. If an appropriate solution is not
found in the S&D solution catalog, a new S&D Class, Pattern or Implementation must be made by the S&D experts
and developers.

**Implementation:** For the approach to work, all applications must be made SERENITY-aware, i.e. they must use the
underlying interfaces and functions defined by the S&D classes and patterns, making security largely programmatic.
A Java API exists for this purpose, which provides handlers to applications to make calls on the Executable
Components instantiated in the SRF instead of communicating with them directly.

**Deployment:** One of the benefits of using SERENITY is that the SRF provides automated monitoring capabilities
for the security solutions during run-time. When the conditions or environment of the system change, the monitoring
rules contained in the S&D solutions can potentially be triggered, leading to a replacement of the solutions with a
different one (e.g. a stronger cryptographic algorithm). While not a methodological feature as such, monitoring has
implications on how S&D patterns are specified and is connected with the overall aims of SERENITY, making it an
essential feature of the approach.

**Discussion:**

Like SECTET and, to a lesser extent, PMF/OpenPMF [Lang and Schreiner 2003], SERENITY is only
concerned with application-level software. Essentially, applications run on top of the SRF, which acts as a
middleware layer providing adaptable security. The idea of having executable components encapsulating security
services is similar to Rosado et al.'s reference security architecture [Rosado et al. 2011a] and SeAAS [Hafner et al.
2009] and hence in some degree to the ISO/IEC 7498-2 Security Architecture model of security services in open
distributed systems. In a sense, this idea is taken to its logical conclusion in SERENITY, where all security solutions
ultimately reduce to pre-coded security components. However, the scope and scenario coverage of solutions is
variable and hence the number of (equivalents of) security services in SERENITY are potentially unlimited.

Comparing S&D Patterns in SERENITY to security patterns in SECTET reveals a number of similarities; for
example, both are used as intermediate representations of implemented security mechanisms in an underlying
platform or framework and therefore cannot capture software design decisions. An S&D pattern does not “describe
the internal functioning of [a security] solution, but its semantics (i.e. properties provided, limitations etc.)”
[Sánchez-Cid et al. 2009] in a formal fashion. The same could be said of S&D Implementations, which likewise
capture additional platform-dependent information about the corresponding Executable Component.

Although S&D solutions can capture an extremely wide variety of security mechanisms, the SERENITY
approach suffers from the drawback of relying exclusively on its own security framework. Interoperability between
the SRF and other middleware is not fully addressed, reflecting the specialization of SERENITY to AmI systems.

If SEPP ([Section 3.4.1.2] above) was portrayed as being requirements-heavy, then it could said of SERENITY
that it is, in the same measure, implementation-heavy. SERENITY patterns are simply encapsulations of
descriptions of run-time security components. While there is support for modeling security architectures via the
provided UML profiles, the activity of creating secure designs is a secondary concern. All analysis and design
activities ultimately tend towards the selection of appropriate solutions, which developers are required to manage
purely programmatically during implementation. The S&D artefacts themselves (classes, patterns etc.) are re-usable
only within the scope of SERENITY, and in particular its run-time constituent (the SRF).

Nevertheless, the SERENITY approach advances much farther than most other major security methodologies
outlined previously in catering for the needs of security non-experts, predominantly by allowing the prescription of
security solutions at the design stage at varying levels of abstraction, according to expertise; the SRF is then left to
decide (at run-time) the best implementation for a particular solution. This makes the approach easy to use.

SERENITY is supported by a tool-suite for the development and selection of S&D solutions, including a pattern
management tool to aid security experts in creating new S&D solutions and an Eclipse-plugin for solution searching
during development.

### 3.4.2 Methodologies Using Security Patterns

#### 3.4.2.1 Eduardo B. Fernandez and colleagues

**Outline:**

al. 2011] present a comprehensive methodology using security patterns throughout the early stages of the SDLC
within the object-oriented paradigm. The introduction of security properties proceeds in two complementary
dimensions: along the system's development life-cycle, closely following the stages of object-oriented analysis and
design [Fernandez et al. 2006a, Fernandez and Mujica 2010]; and across a set of functional and non-functional
hierarchical layers of abstraction, representing an extended software stack, superimposed on a system's software
constraints are defined in accord with this approach both at the earliest stages of the SDLC and at the highest layers
of abstraction – a dedicated security layer in [Fernandez and France 1995] or a meta-layer in [Fernandez 2003] where the conceptual model resides, as well as the application layer [Fernandez 2004] – and propagated horizontally and vertically respectively. Security patterns can be fitted into the various (lower) layers and across stages in increasingly more specialized variants [Fernandez et al. 2008] to enforce the high-level constraints (requirements).

**Detailed Description:**

**Requirements:** In Fernandez et al.'s approach, consideration of security starts from the requirements stage with an analysis of use cases, individually or in sequence (workflows). The purpose of this analysis is twofold: firstly, to determine a set of possible threats and thereby derive a set of security requirements for later stages of the methodology [Braz et al. 2008, Fernandez et al. 2006b]; and secondly, to determine a set of minimal authorization rights for each subject (or role) [Fernandez and Hawkins 1997].

For the determination of subject (or role) rights, use cases are extended via stereotypes to allow the explicit specification of security authorizations for each action by a given actor. “The union of the set of authorization rights across all use cases for every actor defines the complete set of authorization rights for the system” [Fernandez and Hawkins 1997]. Since only valid actions are authorized based on all possible (external) system uses, the resulting set of rights is minimally sufficient in accordance with the need-to-know principle. This approach, therefore, determines rights based on what subjects are allowed to do, not on what they are not allowed to do. The authorization rights determine the first set of security constraints for an application or system.

For determining threats and their corresponding security requirements, [Fernandez et al. 2006b] and [Braz et al. 2008] provide a systematic approach to threat modeling. First of all, threats are enumerated by analyzing each action within a single use case or a sequence of such use cases (per existing actor) and determining how that activity can be subverted, as well as adding external actors such as intruders or hackers and analyzing their specific actions. Each threat is related to one of the affected standard security requirement classes, namely, confidentiality, integrity, availability or accountability, and their related sub-classes [see Braz et al. 2008] for more details), as well as whether the actor initiating the given action is an authorized insider, unauthorized insider or outsider. The resulting threat model is expressed in the form of extended, composite UML activity diagrams. The actual determination of threats is based on experience and domain knowledge, as is the usual case, although threat patterns, which re-use this experience, are also proposed in [Fernandez et al. 2006b]. Performing risk analysis is also part of the process, although specific analysis techniques are not considered. The use cases, threat model, security requirements and initial security constraints are the main results of the requirements stage of the methodology.

**Analysis:** During the analysis stage, analysis patterns [Blaimer et al. 2010] and in particular Semantic Analysis Patterns (SAP – [Fernandez and Yuan 2000, Fernandez and Yuan 2007]) are used in addition to standard object-oriented modeling practices to create a conceptual (domain or business concept) model efficiently. The use case analysis results from before are carried over to determine a set of abstract security patterns [Fernandez and Mujica, 2010, Fernandez et al. 2008] that collectively embody a subset of the security requirements from before and also constitute a set of application-level constraints. From the point of view of [Braz et al. 2008], the abstract patterns can be seen as the mitigating policies corresponding to some or all of the security requirements. These include at least an overall access control model, which is realized by repeated instantiations of the desired (abstract) access control pattern defined by [Fernandez and Pan 2001], thereby realizing the authorization rights per subject (or role) determined from before. The final conceptual model represents a security-enhanced model of the system's main requirements, and is used to guide the rest of the development.

**Design:** During the design stage, the conceptual model from analysis is used to construct a corresponding software design (i.e. software architecture). This initiates a re-newed consideration of security attacks, since the threat model from before must now be considered with respect to the elaborated architecture, and, moreover, there is scope for new attacks at each layer of the (extended) software stack – i.e. the overall or superimposed hierarchical architecture. This is performed with the aid of misuse patterns [Fernandez et al. 2009], which describe in detail how an attack is performed (including what software components are required) from the attacker's perspective, which patterns can mitigate the attack, and how the attack can be traced once it occurs. Misuse patterns aggregate a sequence of attack patterns [Hoglund and McGraw 2004] that represent the steps or certain aspects of the whole attack. Misuse patterns can be cataloged according to their context, and, once instantiated, represent a specific attack with respect to a particular system's architecture at some level of abstraction. Using misuse patterns has a number of advantages in contrast to ad-hoc approaches, including guidance for the non-expert in considering possible attacks and easier identification of corresponding design-level countermeasures. This approach is combined with the previous security requirements and, following [Braz et al. 2008], the total set of unsatisfied or partially satisfied security requirements are mapped to appropriate groups of security patterns to secure each software layer. This ensures traceability from requirements to design, and aids in restricting the selection of patterns according to attacks and the previous security constraints.

At this stage, the abstract patterns in the conceptual model are also specialized into more concrete security patterns and, along with all other security patterns introduced during design, are propagated down the different software layers, thus giving rise to a set of concrete countermeasures that are required to collectively participate in enforcing the initial security constraints [Fernandez 2003]. To achieve this, the concrete or specialized security
patterns – representing security mechanisms at varying layers of abstraction – must be mapped through [Fernandez et al. 2006a] or coordinated between [Fernandez 1999] their respective layers. In [Fernandez 1999] this coordination consists in creating object models of each mechanism at each layer and formally defining the relationships (i.e. mappings of model elements), with Z and OCL being proposed as suitable languages for the purpose. Given that in the latest approach patterns represent the solutions at each layer, it does not seem strictly necessary to create object models, but simply to relate the security patterns to each other, which in turn implies at least the mapping of different security policies or rules across different mechanisms in each layer. It is beyond doubt that this is a difficult task, especially in the context of solutions crossing administrative boundaries, and a fully structured approach appears to be an open problem (cf. [Fernandez et al. 2005]). Since the patterns are instantiated in the system's design and elaborated along with the rest of the system, integration problems are avoided.

As a central example of satisfying access control constraints, and one that can be considered as an essential part of the approach, [Fernandez and Hawkins 1997] and later in a more finalized form [Fernandez et al. 2006a] suggest the (possibly) multiple instantiation of the Model-View Controller pattern [Buschmann et al. 1996] across the architecture, allowing for the division of objects into Model and View classes and the subsequent restriction of all external interactions (such as user access) to View objects (i.e. user interfaces). This can also be seen as a simultaneous instantiation of the Single Access Point (SAP) pattern [Yoder and Barcalow 1997], which likewise restricts external system accesses to a single point. In this scheme, the authorizations specified in the conceptual model are enforced at the View objects playing the role of policy enforcement points [Gollmann 2011] and mapped down the software stack, being reflected conceptually in database table privileges, filesystem rights etc. The practical application of this appears to require manual administration.

Distribution is considered as a special layer of the hierarchical architecture and is thus secured, like other layers, using patterns. Distributed concerns are therefore addressed during the design stage, whenever the conceptual model is mapped onto a pattern-based distributed architecture. [Fernandez and Larrondo-Petrie 2007] discuss this approach in the context of component middleware and propose secure variants of a number of standard distribution patterns (such as the Broker pattern in [Buschmann et al. 1996] – see [Fernandez and Uzunov 2012] and [Uzunov et al. 2012]). In [Fernandez et al. 2007b] an initial attempt is made to extend and tailor the methodology to SOA-based systems.

Implementation: During implementation, the necessary countermeasures as patterns are realized as software units or COTS components. Security can be tested at every stage by checking for the absence or presence of definite security patterns, alignment with security requirements, whether certain attacks are mitigated, etc. The most significant security verification activity employs test cases generated (manually) during or from the requirements stage, to test, during the design and implementation stages, whether the designed and implemented countermeasures can withstand the subverted uses and misuses of the system.

Discussion:

In comparison to other approaches, Fernandez et al.'s methodology is the only one to consider security in a holistic manner – i.e. the need to secure a system or application and all its related functionality in the underlying layers of the software stack. This is, however, a difficult and expensive task, and would require automation of certain aspects, such as determining lower-level policy rules, to make the approach practical for large-scale systems, especially those using black-box legacy sub-systems. On the other hand, coordinating security solutions throughout the whole stack as proposed by Fernandez and colleagues is the only way to ensure global system protection.

Close alignment to the object-oriented development paradigm has advantages – in that object-oriented analysis and design and UML are widely used and readily comprehensible – and disadvantages – in that the methodology may not be applicable in conjunction with other paradigms. The advantages in particular greatly aid to increase the acceptance of the methodology, while using patterns and guidance throughout helps to make the methodology easier to use. Future work in [Fernandez and Mujica 2010] and [Fernandez et al. 2011] indicates that the methodology is moving away from strict adherence to the object-orientated paradigm and heading towards an MDE paradigm.

The methodology relies on coherent catalogs of security patterns, which can cover a range of security concerns. Such catalogs exist in the form of books [Fernandez 2013, Schumacher et al. 2006, Steel et al. 2005], surveys [Uzunov et al. 2012, Yskout et al. 2006] and technical reports [Blakley and Heath 2004, Dougherty et al. 2009], as well as in individual papers, covering access control models, remote authentication, filtering, XML encryption and others. However, as yet there is no single repository. The same remarks apply to analysis patterns and their secured versions (cf. [Fernandez and Yuan 2007]). The main drawback of the approach in this respect is that there is a lack of tool-support for pattern selection, implying that developers must choose patterns manually by navigating through the (text based) pattern catalogs. Although this is offset by the guidance provided in classification schemes, e.g. [Hafiz et al. 2007, VanHilst et al. 2009] and [Washizaki et al. 2009], simple tool-support would certainly be a beneficial feature, and is a planned addition to future developments of the methodology (cf. [Fernandez and Mujica 2010, Fernandez et al. 2011]).
In a similar vein to PSecGCM ([see Section 3.4.1.1]), [Gutiérrez et al. 2005a, Gutiérrez et al. 2005b, Gutiérrez et al. 2006, Gutiérrez et al. 2009] present PWSSec – a comprehensive pattern-driven methodology for web-services (henceforth WS). The similarity in the approaches is understandable, since both methodologies were developed in the same research group (Alarcos, Universidad de Castilla-La Mancha). Like PSecGCM, PWSSec defines a rigorous and detailed software engineering process based on an incremental, iterative life-cycle model; PWSSec also uses a repository of re-usable artefacts, which helps to make the methodology scalable across different projects. Despite the similarities between PSecGCM and PWSSec, there are also a number of important differences. Firstly, PWSSec is exclusively tailored for web-services, and is not applicable to any other type of system, distributed or otherwise [Gutiérrez et al. 2009]. This makes the methodology very specific indeed, even though the overall approach has a number of interesting features applicable in wider settings. Secondly, PWSSec uses a finer-grained approach, especially during requirements analysis, to determine security requirements, risks etc. Finally, PWSSec employs security patterns, making it more flexible and somewhat more approachable to non-experts.

**Detailed Description:**

In PWSSec there are three main stages: WSSecReq (WS security requirements – Gutiérrez et al. 2005a, Gutiérrez et al. 2009)), WSSecArch (WS security architecture – Gutiérrez 2005b, Gutiérrez et al. 2006) and WSSecTech (WS security technologies/standards – Rosado et al. 2006), which roughly correspond to requirements analysis, design (or architecture development) and detailed design respectively.

**Requirements Analysis:** The ultimate aim of the requirements analysis stage is to define the WS application's security requirements. Activities in this stage for the most part follow the standard course: determining assets, then threats on those assets to form a threat model, refinement of the threat model by considering attacks, analyzing risks for each attack and finally defining the security requirements. The threat model is based on attack trees [Schneier 1999], which are a very detailed approach to enumerating threats. Each tree leaf is elaborated by considering attacks using misuse cases [Alexander 2003], which are subsequently mapped to corresponding countermeasures in the form of security use cases [Firesmith 2003]. Risk analysis is performed via a fine-grained risk allocation approach using percentages and any trade-offs are made to define and specify the final set of security requirements, which are carried over into the design stage.

**Design:** One of the most interesting aspects of PWSSec, as in PSecGCM, is the use of a reference security architecture (presented most fully in [Gutiérrez et al. 2005b]) during design. In the case of PWSSec, however, this architecture is of a somewhat different nature. Whereas the security reference architecture in [Rosado et al. 2011a] required integration with the software architecture, the reference architecture in [Gutiérrez et al. 2005b, Gutiérrez et al. 2006] makes use of security patterns, giving rise to a more flexible approach. In PWSSec, security requirements are satisfied by first identifying appropriate security patterns, whose range can, in theory, be arbitrary. This renders the approach more general and simultaneously allows for more detail and comprehensive coverage of security issues. The instantiated security patterns give rise to collections of (abstract) WS security components, which can be elaborated further until they are mapped to actual software components. Since security components are like any other WS components, they must be re-evaluated during another iteration through the requirements and analysis stage [Gutiérrez et al. 2005b].

Security patterns are selected and refined according to two levels of abstraction [Rosado et al. 2006], the first consisting of architectural security patterns, which cover more abstract security concerns affecting larger parts of a system's architecture; and the second consisting of design security patterns, which refine the latter by including specific implementation-related aspects. Design patterns can be mapped directly onto web-services standards and technologies, which can be seen as a final refinement stage for a particular security solution. The architectural security patterns are organized in a repository according to the requirements they address, together with corresponding design patterns, allowing for easier selection during the WSSecArch stage (architectural and design patterns) and the WSSecTech stage (technology-specific mapping).

Security patterns form one aspect of the PWSSec security reference architecture; the other aspect is the management of the resulting security components. During design, an application's (standard) constituent WS components are separated into security domains (termed zones) and assigned security management components called security kernels, with one single “master” kernel per zone. All communications pass through the master kernel for each zone, turning it into a type of reference monitor. The purpose of the kernels are twofold: firstly, they manage the WS security components in a zone; and secondly, they acts as a trusted centre or locus for all security functionality per zone. The last point makes kernels (and in particular the master kernel) a “one-stop shop” for all standard WS components, which can use the services they offer as indications to the WS security components. Regarding implementation details, standard WS components must specify, essentially, “require” policies determining what security requirements they have (e.g. needing to use authentication), while WS security components must specify “provides” policies determining what security services they provide. At run-time, both
these should be registered with the kernels, which check whether an available WS software component exists that implements particular security functionality. PWSSec thus promulgates a programmatic security model.

In spite of the theoretical generality offered by the use of security patterns, [Gutiérrez et al. 2006] only concern themselves with communications security via the “QoP” (quality of protection) pattern system of [Rosado et al. 2006], and only instantiate the QoP pattern with one security kernel in one zone in their case study in [Gutiérrez et al. 2009].

**Detailed design:** During the detailed design stage, a set of WS-specific technologies and/or standards are identified to be implemented by the WS security components, which are subsequently mapped to concrete software units. In this respect, the security patterns from the earlier stages play a central role in the transition between architecture and detailed design, in conceptually mapping WS security components to specific WS technologies and/or standards. Traceability is thus kept from requirements to architecture to technologies.

**DISCUSSION:**

PWSSec is a comprehensive and highly detailed security methodology covering all the early stages of the SDLC. Precisely this level of detail, however, is PWSSec's main drawback. In the case study used to demonstrate the features of the methodology [Gutiérrez et al. 2009], the various activities generated over 200 detailed artefacts for a WS application with a single use case. Even though the application was comparatively large when measured in lines of code (more than 100 kloc), the utilization of the methodology for sizeable, complex systems with numerous use cases appears to be infeasible. Certainly, the scalability features offered by the methodology in terms of re-use of artefacts across different projects is a mitigating factor in this respect, however, they cannot reduce the high amounts of developer effort involved, even with the use of the accompanying CASE tool (UMLPWSSec). The requirement to re-evaluate (via further iterations) WS security components following the instantiation of security patterns is a further complication.

Overall, the methodology makes a good case for a highly specific approach to securing a particular type of distributed system, but the significant labour involved reduces its benefits, and may imply the necessity to automate certain activities or at least increase their granularity.

PWSSec is supported by UMLPWSSec, an extension to the Rational Rose 2000 CASE tool. The support offered by UMLPWSSec is essential in helping to automate the generation of templates for use cases, risks etc.

3.4.2.3 **Delessy / Fernandez**

**OUTLINE:**

[Delessy and Fernandez 2008] and [Delessy et al. 2008] present a pattern-driven (primary class) and model-driven (secondary class) methodology for securing SOA-based systems, adapting and building on the work of Fernandez and colleagues ([see Section 3.4.2.1]) for the design stage. The approach is based on two central ideas: the use of pattern maps to guide the selection of security patterns; and the application of MDA principles to SOA-specific models.

**DETAILED DESCRIPTION:**

Pattern maps are graphs of (security) patterns divided into different layers of abstraction. The patterns (nodes in the graph) on a given layer of abstraction are interrelated by collaboration or refinement relationships (edges) and are classified by a security goal, such as enforcing confidentiality, integrity etc. Patterns across layers are interrelated by a realization relationship, i.e. a pattern on a lower abstraction level realizes the related pattern on a higher level.

In Delessy and Fernandez's methodology, the pattern map consists of two layers, an abstract layer containing patterns with general SOA applicability, and a concrete layer, containing patterns for web-service security. Collectively, the set of patterns in the map at some given point in time represent the set of possible solutions available to developers, which currently address authentication, identity management, access control and web-service cryptographic and policy standards among others [see Delessy and Fernandez 2008] for details and references). While pattern maps have also been utilized in other contexts, e.g. [Fernandez et al. 2007], in the work of Delessy and Fernandez they take a central role in selecting security solutions.

**Requirements analysis:** Since the methodology of Delessy and Fernandez is based on the work of Fernandez and colleagues, the activities performed during the requirements analysis stage for threat enumeration and the determination of security requirements are identical (see [Section 3.4.2.1]).

**Design:** To support MDA transformations and the modeling of SOA-specific systems, [Delessy and Fernandez 2008] present a UML metamodel defining in detail all or most of the necessary SOA-specific semantic elements such as orchestration, services, roles and others. The metamodel is used during design to construct a conformant (primary), platform-independent model, guided by the analysis model from the previous development stages.

The process of introducing security into the resulting model proceeds as follows: (1) a fixed list of security requirements for SOA-based systems (in [Delessy and Fernandez 2008]) mapping requirements to policy classes is
consulted to guide the initial selection of appropriate abstract security patterns, by mapping security requirements to a resulting policy and goal and finally to a related pattern from the SOA security pattern map; (2) selecting relevant patterns according to the relationships of the map, resulting in a decision tree [Fernandez et al. 2007b] that represents the set of chosen countermeasures as abstract and concrete patterns; (3) weaving the set of abstract security patterns into the platform-independent model in a manual process equivalent to pattern instantiation; (4) transforming the resulting security solutions by hand to the platform-specific level – along with the PIM to PSM transformation of the primary model – by specializing the abstract security patterns, according to the pattern map, to their related concrete web-services versions; (5) weaving the concrete patterns into the platform-specific model as before. The PSM in particular conforms to a web-services metamodel which is not presented in any relevant publications.

The PIM-to-PSM transformations are written in QVT, but there does not appear to be any tool-support at this stage, which implies that the weaving and transformation processes are manual.

DISCUSSION:
Overall, the intuitive guidance offered in selecting appropriate security solutions using the pattern map and the use of security patterns makes the approach easy to use. As for other methodologies based on security patterns, the range of possible security solutions is very large. However, the methodology currently lacks tool-support, which is essential if the model transformations are to be fully practical. The published information relating to the methodology is also somewhat sparse in detail, and in particular the last few steps of the security process described above are not demonstrated in full. Therefore, in its current state, the methodology presents a promising approach, but one that needs to be developed further for real-life use.

3.4.2.4 Other Methodologies Using Security Patterns

Other pattern-driven methodologies deserving mention (see also [Uzunov et al. 2012] for an overview from a pattern application perspective) include SODA [Meland and Jensen 2008], a security methodology “for the developer-on-the-street” in which patterns as well as general security guidelines and principles are presented to software developers via a web-based tool to guide them through the process of securing a system design. SODA borrows a number of ideas from process-driven methodologies like Microsoft's SDL, e.g. in its security life-cycle model: threat modeling, security reviews etc., except that security patterns are used as guides, thus reducing the amount of security expertise required.

[Schnjakin et al. 2009] present an approach to securing SOA-based (and in particular web-services based) systems using a “security advisor” CASE tool. The advisor helps developer map security goals to semi-formalized security patterns inspired by Schumacher's theoretical model [Schumacher 2003], which are subsequently transformed to security configurations in the underlying web-services platform using MDA principles. The set of patterns and possible security configurations used by the advisor must be determined by experts in a pre-configuration phase.

ISDF [Alkussayer and Allen 2010] proposes the use of security patterns alongside best practices in an integrated fashion throughout the whole SDL, while basing the process phases and activities on Microsoft's SDL. Despite the use of patterns, ISDF is a very high-level approach, which implies that it suffers from all the draw-backs of a (general) code-based methodology with respect to specific systems.

Finally, [Scandariato et al. 2008] present a security pattern-driven methodology based on their previous work of cataloging and refining available security patterns in the open literature [Heyman et al. 2007, Yskout et al. 2006, Yskout et al. 2008]. The latter catalog is used as an inventory, which is further structured according to a decomposition of security objectives, relationships such as dependencies and conflicts, and a separation of design phases of a system (application architecture, application design and system-level). Security patterns from the inventory are used according with the latter phases to help introduce security properties into a system, based on the security objectives defined at the outset.

3.5 Agent-Driven Methodologies

Agent-driven methodologies are centered on the Agent-Oriented Software Engineering (AOSE) paradigm [Ciancarini and Wooldridge 2001, Jennings 2001]. AOSE is based on several fundamental concepts borrowed from AI and social theories, the most important of which is that of an agent – a system entity with strategic goals and intentionality, or, from another angle, an encapsulated piece of software “situated in some environment and capable of flexible, autonomous action in that environment in order to meet its design objectives” [Jennings 2001, Wooldridge 1997]. There is a close similarity in some respects between the agent-oriented and object-oriented paradigms, in that both aim to encapsulate some element of system functionality in a higher-level abstraction, and both aim at autonomy, even if in different ways. The main distinction lies in the fact that agents are pieces of (sometimes mobile) code that, notionally at least, have some reasoning logic enabling them to make decisions based on environmental stimuli, rather than passively reacting to method calls or messages. [Jennings 2001] argues that AOSE is well suited to developing distributed systems, due to the collaborative, autonomous nature of agents. In
realism, the application of AOSE to distributed systems results in multi-agent systems (MAS), which can be considered specific types of distributed systems.

[Mouratidis et al. 2005] argue for the benefits of securing software using the AOSE paradigm, advancing the view that agents, with their intrinsic autonomy and intentionality (and hence goal-orientation), provide a suitable means to model high-level security concerns that can be propagated throughout the system. Although the argument itself does not stand to closer scrutiny (the same could be stated with respect to any paradigm), it is nevertheless true that AOSE allows for strategic goals to be captured early on, thereby allowing a goal-oriented approach to security that may be more natural for some developers.

A number of methodologies have been created for AOSE, however, very few incorporate security as a central concern (cf. [Mouratidis and Giorgini 2007a]). In this subsection we review Secure Tropos, an extension of the Tropos AOSE methodology, which integrates security throughout the early stages of the development life-cycle.

3.5.1 Mouratidis / Giorgini (Secure Tropos)

OUTLINE:
Secure Tropos [Mouratidis and Giorgini 2004, Mouratidis et al. 2005, Mouratidis and Giorgini 2007a, Mouratidis 2011] is an extension of the Tropos AOSE development methodology [Giunchiglia et al. 2003, Bresciani et al. 2004, Giorgini et al. 2004]. Since the AOSE paradigm is less familiar than other development paradigms, we will present the Tropos methodology in some detail below, before proceeding to review the extensions introduced by Secure Tropos.

DETAILED DESCRIPTION:

The Tropos methodology considers five development stages: early requirements, late requirements, architecture development, detailed design and implementation. Throughout each stage, Tropos uses the $i^*$ modeling framework, a modeling language tailored for agent-oriented systems. The range of concepts that can be modeled by $i^*$ include: actors (entities with strategic goals), which can be agents, roles (abstract characterizations of behaviour) or positions (sets of roles); goals (an actor's strategic interests); soft-goals (goals without clear satisfiability criteria); tasks or plans (ways of doing something); resources (physical or informational entities without intentionality); capabilities (context-specific ability of an actor to select and execute a task for the fulfillment of a goal); and dependency relations – dependor, dependum, dependee (inter-actor dependencies to attaining goals, executing tasks, and/or delivering resources). These concepts define the semantics of the modeling language; the syntax is comprised of a non-standard, custom graphical notation, which can be seen in all the Tropos-related publications.

According to [Mouratidis and Giorgini 2007a], “Tropos is based on the idea of building a model of the system that is incrementally refined and extended from a conceptual level to executable artefacts, by means of a sequence of transformation steps”. Here transformation does not imply automated model-driven transformation, but the conceptual transformation from models at one development stage to models of the next stage. At the early requirements stage, stakeholders (i.e. participants or users of the system, entities in the system's environment etc.) together with their intentions are modeled as actors and goals respectively, to create a strategic dependency model, which is somewhat analogous to a conceptual model in standard software engineering. This model is augmented by a set of strategic rationale models, one for each actor, with the determination of how the actor's goals can be achieved (i.e. with the addition of tasks and sub-goals from the perspective of the specific actor and internal to that actor, using various analysis techniques, such as means-end and contribution analysis and AND/OR decomposition).

During late requirements analysis, the software system itself, represented as an actor, is added to the previous models and the analyses performed to produce a rationale model for the system and to revise rationale models for other actors as needed. At this stage the functional and non-functional requirements of a system can be specified from the inter-actor dependencies. At the architecture development stage, the system's software architecture is elaborated using large-grained, agent-specific styles, and refined during detailed-design using agent-specific patterns [see Giorgini et al. 2004]). Throughout the architecture stage (as in the requirements analysis stages), all entities in the system are modeled as actors and the relationships between the entities are modeled as dependencies. In terms of functionality, architectural components (actors), which encapsulate elements of the system's functionality, can be added and new/existing ones can be recursively decomposed into sub-actors to which goals (of the original actor) are delegated. This is analogous to software components being refined into sub-components, which are assigned a portion of the larger component's functionality. Rationale models for each sub-actor are created as a result of analyzing the tasks and goals required to achieve the larger goal, as well as the dependencies with other actors (or sub-actors). The last steps in the architectural design stage consist in identifying capabilities needed by each actor [see Bresciani et al. 2004]) and assigning these capabilities to agents. The detailed design stage consists in further specifying each agent's capabilities and interactions with the help of Agent UML (AULM – [Huget 2004]).

Requirements analysis: Secure Tropos extends the Tropos methodology by introducing security constraints, which embody high-level policies and – during later development stages – security requirements, placed on the system and its entities (actors); as well as by expanding Tropos' modeling language to allow for the modeling of concepts such as secure goals, secure tasks, secure resources, secure dependencies and secure capabilities. In addition to extending the modeling capabilities of Tropos, Secure Tropos introduces four custom security modeling and analysis activities.
The first of these activities, security reference modeling, has the purpose of identifying the security needs of a system via threat and vulnerability analysis, the result of which is the determination of a set of security features (modeled as soft-goals), protection objectives (modeled as goals) and security mechanisms (modeled as tasks). This activity is not done with respect to the assets of an initial system model, but to the desired security features of the system. The second modeling activity, security constraints modeling, aims to model, decompose and assign security constraints to actors, whereby a set of secure goals are introduced into the system for the fulfillment of the constraints. The third modeling activity, secure entities modeling, is concerned with analyzing the secure goals introduced during constraint modeling, via, for example, means-end analysis, to determine a set of tasks and resources that can achieve the goals. Finally, the fourth modeling activity, secure capability modeling, seeks to identify a set of secure capabilities that can be assigned to agents in the system to satisfy the security constraints.

The modeling activities of Secure Tropos are used within its overall aims of identifying a system's security requirements at the early development stages; developing a viable design that satisfies those requirements; and verifying the security of the resulting system. This is achieved via a corresponding four step process [Mouratidis 2011], aligned with the (early and late) requirements analysis, architecture development and detailed design stages of Tropos.

During the early and late requirements stages, security reference, security constraints and secure entities modeling are employed alongside the Tropos strategic dependency and rationale models, to impose security constraints on the actors and their dependencies and to introduce secure entities (goals, tasks etc.) for the satisfaction of those constraints. The result of the requirements analysis stage is a set of models as well as functional, security and (other) non-functional requirements for the system.

**Architecture development:** During the architecture development stage, a suitable architectural style is selected, new actors are added and refined (i.e. recursively decomposed as per the Tropos approach) to elaborate the architecture; and secure capabilities are identified and subsequently assigned to actual agents. Architectural style selection occurs at the discretion of the developers, with the aid of a technique using fine-grained probability assignment to perform trade-offs (see [Mouratidis et al. 2005]). When elaborating the architecture, new security constraints can be introduced and analyzed as in previous stages, and an analysis of security criticality and complexity [Mouratidis and Giorgini 2004] for each actor can also be applied to determine if new actors need to be introduced in order to delegate overloaded functionality. Finally, a detailed verification is performed for the developed models, based on validation rule-sets, and for the design as a whole, based on such techniques as security attack scenarios [see Mouratidis and Giorgini 2007b]). Although realized quite differently, the use of attack scenarios in SecureTropos shares a similar purpose to attack model composition in Georg et al.’s AORDD methodology (see [Section 3.2.2.1]).

The mapping between the conceptual models developed during the late requirements analysis stage and the architecture stage models is aided by the use of agent-specific security patterns [Mouratidis et al. 2003, Mouratidis et al. 2006a]. More specifically, the secure goals and secure tasks from the analysis models are seen as a set of security challenges guiding the selection of particular security patterns. The advantage of this is that the analysis models are conceptually mapped to known, predictable sets of actors (existing and new) as the result of the pattern instantiations, thereby reducing the risks associated with an ad-hoc introduction of design solutions. In spite of their usefulness, the existing set of agent-specific patterns addresses only a limited range of security concerns. The requirements for agent-specific patterns [Mouratidis 2006a] implies that standard collections of security patterns, e.g. [Schumacher et al. 2006] are not applicable and thus cannot be used. It is worth mentioning in this context that Mouratidis et al.’s proofs of completeness [Mouratidis et al. 2006a] – a better term would have been “closure” – based on the ideas of [Schumacher 2003], show that their existing agent-specific patterns cover a given set of security problems and do not introduce new problems, but not that new patterns are not needed.

**Detailed design:** During earlier versions of the methodology (even including its presentation in [Mouratidis 2009]), the detailed design stage followed Tropos and used AUML to model the agent interaction protocols. In [Mouratidis et al. 2006b] and [Mouratidis 2011], Secure Tropos is combined with UMLsec [Jürjens 2005], predominantly to consolidate the detailed design stage. In the combined version, the $i^{th}$ models of the system are translated into UMLsec models at the architecture stage (after the initial architecture has been selected, patterns applied etc.), so that the further elaboration of the architecture and in particular the detailed design of individual components is performed using UMLsec. The resulting combined methodology, which can be described as Secure Tropos up to early architecture and UMLsec afterwards, has advantages and disadvantages relative to the complete Secure Tropos version. Firstly the mapping of (Secure) Tropos models to UMLsec models is not strictly one-to-one, which makes for a somewhat ad-hoc approach to converting Tropos goal-oriented models to models in UMLsec; secondly, the combined approach does not obviate the need for developers to learn Tropos notations, nor to think in terms of actors and dependencies during design to elaborate an initial version of the architecture, but adds the further requirement of learning UMLsec notations; and thirdly, the coherency of the approach, which used the same concepts, development approach and notation, is lost – contrary to the initial aims of Secure Tropos [Mouratidis et al. 2005] – and with it the relevance to AOSE. Conversely, the loss of relevance to AOSE implies that the combined approach is applicable to distributed software systems generally, as opposed to agent-based systems. Since Secure Tropos is so tightly bound to AOSE, employing specialized notation and modeling concepts, and since AOSE in the
context of distributed systems is so tightly bound to multi-agent systems, the applicability of the methodology is otherwise severely limited to a specific domain. The combined Secure Tropos/UMLsec approach overcomes this limitation. Moreover, UML is a familiar notation, and UMLsec follows standard software engineering paradigms, meaning that the design stages would be more approachable to a wider range of developers. Finally, UMLsec allows for the formal verification of interaction protocols, albeit at the cost of requiring greater expertise. In our opinion, the choice of which version of the methodology is actually superior is dependent on the problem domain and the project goals, as well as a trade-off between the existing approaches in that context, e.g. whether AOSE is the best choice of paradigm, whether the benefits of UMLsec outweigh coherency etc.

DISCUSSION:

Overall, Secure Tropos provides a very comprehensive security methodology that aims for high ease of use and flexibility. It differs significantly from other methodologies in its early requirement stages, predominantly due to the goal-oriented nature of AOSE, supporting the early introduction of security via high-level (informal) security constraints. Security expertise is still required for certain tasks (such as optimal assignment of secure capabilities), though less so in comparison to most other methodologies.

Secure Tropos is evaluated on a real-life case study [Mouratidis 2009] – eSAP, an electronic health-care system, which demonstrates the use of the methodology well. An independent application of the methodology appears in [Rojas and Mahdy 2011], which augments the approach with STRIDE threat modeling.

Secure Tropos is supported by the SecTro 2.0 CASE tool [Mouratidis 2011].

4 Evaluation

Just as it is difficult, perhaps even impossible, to produce perfectly secure software, it is also difficult to create a perfect security methodology. All the methodologies surveyed in [Section 3] have their own individual benefits and limitations, and may be more or less appropriate for practical use depending on the application, project demands, team skillsets and other factors. The real-life use of a methodology will very much depend, however, on the presence or absence of certain desirable characteristics.

4.1 Criteria

Based on Whyte and Harrison's study, taken as one of the most indicative reflections of the security needs in the software industry, we have formulated a set of criteria for the adoption of a security methodology in real-life scenarios. These criteria were subsequently compared and augmented where applicable using the more abstract requirement lists for a good security methodology presented by [Mouratidis 2009] and [Fernandez et al. 2011] to produce a definitive set, which we summarize below in Table 1.

Criteria numbers 1 to 5 (inclusive), represent essential criteria for industry projects, i.e. criteria that would indicate a low probability of adoption if they were not well satisfied by a given methodology. In some instances, poor satisfaction of these criteria may indicate a specialized approach, for example, an approach focusing solely on the generation of security artefacts from a particular SDLC stage. Criteria numbers 6 to 10 (inclusive) represent important characteristics that can help to qualify risks associated with adopting a methodology. Criteria 11 and 12 represent desirable or useful features of a methodology, whose presence may or may not be valuable for a given project.

With respect to the taxonomy presented in [Section 2], criterion 1 is related to classification dimension SDLC stages supported [Section 2.2.6]; criterion 3 is related to classification dimensions Specificity [Section 2.2.2] and Range of security properties [Section 2.2.4]; criteria 4 and 5 are both related to classification dimension Ease of use [Section 2.2.7], where criterion 5 in particular is related to classification dimension Tool-support [Section 2.2.8]; criterion 6 is related to classification dimension Modeling language and notation [Section 2.2.3]; and criterion 8 is related to classification dimension Use of formal methods and verification [Section 2.2.5].
<table>
<thead>
<tr>
<th>Criterion No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A security methodology should be end-to-end, across the whole system life-cycle (requirements analysis, design, implementation, testing, deployment, maintenance and even deposition); as well as the full spectrum of activities: modeling, producing test plans, coding etc. A methodology should attempt to address security threats as early as possible.</td>
</tr>
<tr>
<td>2</td>
<td>A security methodology should provide guidance on what security measure should be applied and where it should be applied; in particular, some form of early security assessment, such as threat modeling should be an essential part of the approach.</td>
</tr>
<tr>
<td>3</td>
<td>A security methodology should be sensitive to the system or application: in particular, “security measures should be proportionate” and specific “to the nature of the application” with “a range of solutions, each equated to a different risk and cost” [Whyte and Harrison 2011].</td>
</tr>
<tr>
<td>4</td>
<td>A security methodology should not require high security expertise for its application, but should rely exclusively on the skillsets of the developers (predominantly based on software engineering). In particular, the current knowledge-base of developers, including University graduates with low exposure to security concepts, must be taken into account.</td>
</tr>
<tr>
<td>5</td>
<td>A security methodology should provide processes and tools to increase its ease of use and aid developer productivity. In particular, “(naïve) user-friendliness and automation are important” [Whyte and Harrison 2011] to reduce development time and potential errors.</td>
</tr>
<tr>
<td>6</td>
<td>A security methodology should support “security modelling on top of well-established model-languages such as UML” [Whyte and Harrison 2011], to reduce developer training and effort.</td>
</tr>
<tr>
<td>7</td>
<td>A security methodology should demonstrate validity through complete case studies to build confidence in its real-life value.</td>
</tr>
<tr>
<td>8</td>
<td>A security methodology should utilize some means of assessment or verification to ensure the introduced security solutions correctly counter the relevant threats and improve the security of the system as a whole. The possibility of using “light-weight” formal methods for verification, even for parts of the system, is an advantage; however, excessive use of formal methods should be avoided.</td>
</tr>
<tr>
<td>9</td>
<td>A security methodology should not introduce disproportionate development overhead within the chosen software process.</td>
</tr>
<tr>
<td>10</td>
<td>A security methodology should promote the use of common repositories or catalogs of security knowledge (relating to solutions, threats, vulnerabilities, standards etc.) to encourage the application of best practices, increase productivity and/or aid developer training.</td>
</tr>
<tr>
<td>11</td>
<td>A security methodology should take into account (i.e. be applicable to) third-party and legacy software.</td>
</tr>
<tr>
<td>12</td>
<td>A security methodology should provide guidelines to balance security with other quality attributes.</td>
</tr>
</tbody>
</table>

Table 1: Criteria for industry adoption of a security methodology

4.2 Analysis Results

Based on our detailed reviews of the surveyed methodologies, we have analyzed and evaluated the conformance of each individual methodology to the criteria above, summarizing the results in Table 2. To avoid excessive repetition and reduce space, we omit the detailed analysis for each methodology and provide only a final result in the table, in which: “Y” for “Yes” indicates that a methodology fully satisfies a criterion; “N” for “No” indicates that a methodology fails to satisfy a criterion; and “P” for “Part” indicates that a methodology satisfies a criterion in part. The integer in the top row refers to one of the numbered criteria above. It should be emphasized that while our assignment of values for each criterion per methodology are guided by our reviews, they nevertheless reflect subjective assessment. Moreover, the reviews themselves are often constrained by the amount and/or quality of published information. The merit of the results lies in revealing the current deficiencies in existing approaches and establishing trends – they should not be seen as an absolutely fair comparison between methodologies, something that no restricted evaluation system can achieve. In this respect, the analysis can be seen as being coarse-grained, and finer distinctions of which methodology satisfies a given criterion more than another requires examination of the relevant reviews.
<table>
<thead>
<tr>
<th>Methodology / Criterion No.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model-based</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jürjens (MBSE/UMLsec)</td>
<td>P</td>
<td>N</td>
<td>P</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Model-driven (MDA/MDS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lodderstedt/Basin/Dooser</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(Model-Driven Security)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hafner/Breu/Memon</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>P</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(SECTET/SECTISSIMO)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lang/Schreiner (OpenPMF)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>P</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Gunawan/Hermann/Kraemer</td>
<td>P</td>
<td>P</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>P</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(security-enhanced SPACE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sánchez and colleagues</td>
<td>P</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(ModelSec)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model-driven (AOSD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Georg/Ray/France (AORDD)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>P</td>
<td>N</td>
<td>P</td>
<td>P</td>
<td>Y</td>
</tr>
<tr>
<td>Mouheb and colleagues</td>
<td>N</td>
<td>P</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>P</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Jakob/Loriant/Consel</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>P</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>P</td>
<td>N</td>
</tr>
<tr>
<td>(DiaAspect)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Architecture-driven</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ren/Taylor</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Ali/El-Kassas/Mahmoud</td>
<td>N</td>
<td>P</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>P</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Pattern-driven (components/architectures)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rosado/Fernández-Medina/Lopez (PSecGCM/SecMobGrid)</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>P</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Schmidt/Hatebur/Heisel</td>
<td>P</td>
<td>Y</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(SEPP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SERENITY</td>
<td>P</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Pattern-driven (security patterns)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fernandez and colleagues</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>P</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Gutierrez/Fernández-Medina/Spinetti (PWSSec)</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Delessy/Fernandez</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>P</td>
<td>Y</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Agent-driven</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mouratidis/Giorgini</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>P</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>P</td>
</tr>
</tbody>
</table>

Table 2: Conformance of security methodologies to criteria for industry adoption

4.3 Discussion

The results clearly show that, currently, no single methodology fully satisfies all requirements. The formal methodology of Ali et al. ([see Section 3.3.2]) satisfied the least number of criteria in the analysis, which was due to its complete reliance on formal methods and the lack of guidance offered throughout. SERENITY and PWSSec, which are among those to satisfy most criteria, are only applicable to AmI and WS-based systems in a very restrictive fashion. This should not be seen as an indicator suggesting that methodologies targeting specific system types (i.e. those with a high degree of specificity) are better able to satisfy the criteria than those of a more generic nature; it may, however, imply that the researchers were better able to tailor their approaches.

Perhaps not surprisingly, certain trends can be seen in the results, especially with respect to (primary) methodology paradigms. Pattern-driven methodologies, for example, fared well against criterion No. 10 (use of a repository of security knowledge), since patterns encapsulate security knowledge and best practices in one way or another, or, in the case of PSecGCM, a repository of solutions that were found successful allows re-use leading to increased productivity and good practice in subsequent projects. In fact, the general trend of the results showed that pattern-driven methodologies, even those in which this is a secondary paradigm (SECTET/SECTISSIMO, Georg et al.’s AORDD methodology) satisfied more criteria than others.
Methodologies that did not satisfy the first five (essential) criteria well, such as MBSE/UMLsec, Model-Driven Security and others, indicate approaches that are tailored for helping security experts do a better job. Some of these methodologies may best realize their benefits when used as part of a larger security approach (e.g. together with another methodology, as is done with UMLsec and SecureTropos) capable of guiding developers in introducing security solutions. The latter include methodologies that mostly satisfied the essential criteria, and which aim at providing guidance to non-expert developers throughout larger portions of the SDLC. 

Below we briefly comment on the analysis results for each criterion individually.

1. No methodology fully satisfied criterion No. 1 (end-to-end life-cycle support), which is perhaps the single most important criterion for real-life adoption, decidedly showing an area for further research and improvement. PSecGCAM was the only approach to propose activities for the planning and maintenance stages; however, those stages, as well as implementation (especially for the reference architecture), did not receive the same attention as the core development stages.

2. Guidance in application of security solutions was predominantly satisfied by pattern-driven methodologies, due to the guidelines present in patterns. Mouheb et al.'s methodology satisfied the criterion insofar that aspects in that approach capture security knowledge, as well as where in the design the aspect can be applied; however, no threat modeling/assessment strategy is proposed. Similarly, SERENITY does not offer support to developers in determining the initial set of security requirements.

3. Some (most notably model-driven) methodologies supported the introduction of only one or two security properties (e.g. access control, secure communication), rendering them – despite the presence of promising and novel ideas – unrealistic for many practical situations. Methodologies relying on security patterns (fully) satisfied the criterion in a slightly relaxed form, since such patterns do not always specify the costs and associated risks with their use, even though, in theory, this is one of their essential aspects.

4. Perhaps the second-most important factor for real-life adoption of a methodology is criterion No. 4. Satisfaction was generally achieved by encapsulating security expertise in some form, and/or relying on tool-support, guidelines etc.

5. Some methodologies, such as that by Fernandez and colleagues, suffer from a lack of tool-support, even though they otherwise provide processes promoting simplicity and ease of use. In the case of Fernandez et al.'s methodology in particular, the high ease of use of the process can to a large degree compensate for the missing tools, but not entirely within the context of the proposed holistic approach (i.e. securing all layers of the software stack). Most model-driven methodologies offer good tool-support, as befits the MDE paradigm. MBSE/UMLsec, SECTET and the methodology of Lang and Schreiner make particularly strong use of tools and/or run-time software support.

6. Most methodologies make use of UML both for architectural modeling and for representing security solutions, and hence satisfied criterion No. 8 easily. The methodology of Ren and Taylor uses Secure xADL, which is sufficiently simple to use and understand, and requires little extra effort from developers than using, for example, UML 2.x collaboration diagrams. SEPP, which uses problem frame diagrams during requirements analysis, and Secure Tropos, which uses the i* modeling framework either throughout or, in the UMLsec variant, during the early development stages, both suffer from non-standard notation, which may be a usability hindrance.

7. Jürjens' MBSE/UMLsec and Lang and Schreiner's methodology stand out for having demonstrated real-life use in a number of industry scenarios. Other methodologies that completely satisfied the criterion were trialled on singular but full case studies developed from inception to deployment. Methodologies satisfying the criterion in part or not at all were presented in the literature using real-life examples and designs, but not full case studies, or were otherwise trialled on smaller aspects of a project.

8. Approaches to verification range from manual inspection and iteration of the process (PWSSec), generation of test cases (Fernandez et al.'s methodology) and utilizing attack scenarios (Secure Tropos) to semi-automated verification (Georg et al.'s AORDD approach). Most methodologies eschew the use of heavy formal methods, the notable exception being Ali et al.'s approach. Jürjens' MBSE/UMLsec, Model-Driven Security and the methodologies of Georg et al. all employ some form of formal techniques for verification. The verification in Model-Driven Security in particular, while based on OCL, is complex and unsuitable for non-experts. Secure Tropos can make use of formal methods from the Tropos methodology, as well as protocol verification from UMLsec (for the Secure Tropos/UMLsec variant). It is important to note that in all the cases above, the verification approaches generally target a restricted set of development aspects, e.g. a system's design, protocols, specific attack scenarios etc., and a restricted set of development stages.

9. Excessive overhead is generated in PWSSec by numerous iterations for verification and the creation of many (non-deliverable) development artefacts, as well as Georg et al.'s AORDD methodology by the (currently) iterative process of weaving attack aspects, mitigating aspects, deriving OCL from models for verification etc. SEPP satisfied this criterion on the provision that the formal specification stage is skipped, as did Model-Driven Security, on the provision that only a simple verification is performed, and that a new security modeling language is not required.

10. As already noted, use of a common repository of security knowledge was satisfied best by pattern-driven approaches and those utilizing some form of encapsulation of security solutions. Methodologies such as
SECTET and PSecGCM partially satisfied the criterion, since their repositories require population of successful solutions, models etc. during an initial project. Other methodologies partially satisfying the criterion promote the use of security solution catalogs, but not with the purpose of being used as a common knowledge repository. It is worth noting that the catalogs or knowledge repositories discussed above are predominantly of security solutions. Common repositories of vulnerabilities such as the MITRE Corporation's CVE/CWE, attack patterns and others are not employed in any of the discussed approaches, with the exceptions of SEPP, which makes use of a catalog of problem-domain patterns; PSecGCM and PWSSec, which store (security-related) requirements analysis artefacts in a repository for subsequent re-use; and the methodology of Fernandez et al., which proposes the use of a catalog of misuse patterns, currently only in its initial stages of development.

11. Only Jürjens' MBSE/UMLsec provides support for legacy software at both the design and implementation levels. Some model-driven methodologies, especially the AOSD ones, also partially satisfied this criterion in virtue of their ability to separate security code or models and introduce it into ready designs. However, understandably, the latter satisfaction is often reliant on the existence of a documented software architecture for a legacy system. Lang and Schreiner's OpenPMF, which is used for security enforcement, may support legacy systems at run-time; however, the use of OpenPMF may entail vendor lock-in issues.

12. Only Georg et al.'s AORDD methodology and Secure Tropos offered additional support for balancing security with other factors. In Secure Tropos, this occurs only during the architecture selection stage.

As a final remark, we should point out that the inability of any single methodology to satisfy all the criteria set out in the evaluation above, which reflect, essentially, industry requirements, should not suggest that the surveyed security methodologies are without merit, or that they cannot or should not be adopted. Indeed, employing any one of a number of the security methodologies reviewed to guide some part of the software process should produce better outcomes than ignoring security altogether. Considering once again the remark at the beginning of the evaluation, we should stress that there is no perfect methodology, and compromises will inevitably have to be made in both directions – for industry developers and security researchers.

5 Conclusion and Future Directions

Throughout this paper we have attempted to fill a growing gap in the literature by surveying and critically analyzing the state-of-the-art in (model-based) security methodologies for, or applicable to, both general and specific types of distributed systems. In doing so we have aimed to address the first of our questions posed in the Introduction ([Section 1]) – “why are security methodologies not better known?”. Our detailed reviews can be seen as a step towards increasing awareness and appreciation of a large range of security methodologies among researchers and industry stakeholders.

Following the detailed survey, we proposed a number of criteria reflecting the characteristics security methodologies should possess to be adopted in real-life industry scenarios, and evaluated each methodology accordingly. The evaluation attempted to address the second major question posed in the Introduction – “why are security methodologies not receiving more attention from the industry?” – from a technical viewpoint.

Our evaluation showed that no single methodology fully satisfied all the criteria, suggesting that, as yet, an ideal security methodology tailored to industry needs does not exist. Although the lack of satisfaction in any given case by no means implies that a given approach cannot be adopted, it does indicate that a strong business case is missing. A number of factors contribute to this situation: high security expertise requirements, non-standard concepts and notations, partial coverage of the system life-cycle, excessive overheads, missing tool support and demonstrated credibility via case studies are just some of the main inhibitors towards more wide-spread practical use of otherwise valuable methodologies with a number of beneficial features. Of course, beneficial features alone do not necessarily add up to usable methodologies with industry-scale applicability. As a broad generalization, after all factors are weighed appropriately, the final outcome does indicate that more effort is required in advancing the current state-of-the-art in security methodologies as practical, usable approaches for securing real-life systems, distributed or otherwise.

The irony of the situation is that code-based approaches like Microsoft's SDL and OWASP's CLASP, while placing overwhelming emphasis on expertise and providing few (if any) concrete development mechanisms for introducing security at the early life-cycle stages, have – generally speaking – received greater acceptance in the industry. It may be seen as a supposition, but it would not be unfair to state that the main reasons for this are the weight of the organizations behind them, and the fact that such methodologies are, in a sense, complete and tailored for real-life use (within their paradigm, that is). Regarding the latter point, approaches that offer only a proof-of-concept, supporting one or two security solutions for example, will hardly receive the same acceptance. Regarding the former point, it should be noted that most of the methodologies surveyed in this paper originate from academia and not the industry. Indeed, many directly stem from, or are entirely within the confines of, PhD work (in the best sense), which would no doubt be a significant contributor to some of the negative factors enumerated in the paragraph above, and inevitably results in methodologies that can easily be classed as being “proof-of-concept”, rather than “production-ready”.
In most, if not all, of these cases, there is little that can be done. Nevertheless, one can draw strong analogies between young start-up companies and methodologies originating from academia: just as a startup can be taken over by a larger company and its range and quality of products improved, so a security methodology can be elaborated and made more practical if adopted on industry projects and expanded by industry professionals in the appropriate setting. Of course, this leads to a closed circle that can only be broken by industry stakeholders taking a risk, on the one hand, and independent researchers in secure software engineering improving existing security approaches, on the other.

One important question that should be asked, and indeed has already been asked for development methodologies (see [Henderson-Sellers and Giorgini 2005, Ramsin and Paige 2008]), given the current state-of-the-art, is whether it is at all possible to produce a single or ideal, fixed methodology that can gain any good degree of industry adoption in the first place. While we have already considered quality factors that could help in this direction, one important factor that has not been mentioned is that not all methodologies are suitable for all purposes and scenarios: e.g. a web-services specific methodology may not be usable for systems in which web-services play only a small, even if essential, role. In this sense, constructing flexible security methodologies, which are applicable to different types of systems (and in this context, different types of distributed systems) may be a promising approach. Currently, such methodologies do not exist in the published literature (cf. [Uzunov et al. 2012]). Taking this argument one step further, one could argue that a “one-size-fits-all” approach, even with a single flexible security methodology, would still be inadequate in a number of situations, and that what is required is an approach to engineer security methodologies, i.e. to construct (from scratch or from re-usable parts) and tailor methodologies on the fly for different project situations as required. This, of course, is not a new argument for development methodologies [Ramsin and Paige 2008]: a whole field called Situational Method Engineering (SME) exists to address such issues in that setting (see [Henderson-Sellers and Ralyté 2010]). First steps in this direction have already been taken in [Uzunov et al. sub1], which proposes a comprehensive approach to engineering security methodologies based on process patterns and meta-modeling. While this direction will almost certainly pose challenges with respect to industry adoption, especially since its source of inspiration – SME – has not received widespread industry attention as yet [Henderson-Sellers and Ralyté 2010, Low et al. 2010], the nature of these challenges may be different and perhaps lighter, inasmuch as security methodologies are something additional to a development process.

Besides constructing flexible methodologies or engineering security methodologies on the fly, our survey and analysis indicate a number of other important future directions, which are relevant for single, fixed methodologies, flexible methodologies and tailorable methodologies alike. We consider some of these directions below, and indicate how we have begun to address them in our own work where appropriate.

– Extending the range of security solutions: as noted in the evaluation ([Section 4]), one striking observation was the lack of a comprehensive catalog of security solutions in a number of methodologies, above all those within the model-driven paradigm. Simultaneously with this observation was another indicating that encapsulating security knowledge (using patterns, for example) is a beneficial feature. An important direction in this respect is not only to create or distill a wider range of solutions for developers to use, but also to encapsulate a wider range of security knowledge and guidance in each solution. In this sense, new security solutions types individually providing increased guidance and at the same time sufficient flexibility to allow freedom in application would be a valuable contribution towards improved security methodologies. Within the context of engineering methodologies in particular, it would also be beneficial if new solutions were methodology agnostic, and hence (re-)usable across different methodologies with different paradigms, allowing for even greater degrees of customizability to different project situations. The recently proposed security solution frames of [Uzunov et al. sub2], which organize security patterns vertically in hierarchies, and horizontally in related families, can be seen as an initial step in this direction. Other, related ideas include a consolidation of the relationships between classification schemes (e.g. [VanHilst et al. 2009] for patterns) and corresponding security solutions; extensions of existing solution types, such as the aspects of [Georg et al. 2009] and [Mouheb et al. 2009], with both fixed and flexible parts; as well as the construction of hybrid solutions combining both design and implementation elements.

– Encapsulation of other security knowledge: encapsulating other forms of security knowledge such as threats, attacks, requirements etc. is another important direction following on the latter one. The problem frames of SEPP [Hatebur et al. 2007a, Schmidt 2010a] and the semantic analysis patterns of [Fernandez and Yuan 2000] are good examples of how knowledge can be captured at development stages earlier than design, however, the former are strongly tied to a particular development paradigm, while the latter are more domain-oriented and place heavier demands on developers in drawing analogies for extrapolation. The recently proposed threat patterns of [Uzunov and Fernandez sub1] are another contribution in this direction, encapsulating threats for networked and distributed systems in a fashion that is complementary to security solutions in general and also methodology agnostic.

– Use of existing catalogs of security knowledge: for example, CAPEC attack patterns, CWE vulnerability lists and others. The threat patterns mentioned previously can already be realized using attack patterns, which can help construct penetration testing approaches [Uzunov and Fernandez sub1], however, little work has been done in this direction both by the current authors and by others.

– Expansion of security activities and interpolation to other SDLC stages: since a number of security methodologies suffered from a lack of (full) SDLC coverage as required by the evaluation criteria in [Section 4], interpolating methodologies to cover both earlier and later development stages is an important direction for future
work. The incorporation of activities such as penetration testing, already mentioned above, or systematic attempts to evaluate the improvements in security via, for example, security metrics as in the work of [Heyman et al. 2008], is a related direction. In the context of engineering security methodologies, the aforementioned approach of [Uzunov et al. sub1] promotes this by allowing security process patterns for different activities to be instantiated, e.g. for penetration testing, for various types of verification etc. Other engineering approaches for development methodologies could offer similar or different advantages (cf. [Hurtado Alegría et al. 2011, Wagner et al. 2011]) – in all cases providing guidance in the incorporation of activities into a methodology – and their adaptation would also be a worthwhile pursuit, together with the development of new security-specific approaches.

- **Incorporation of security activities into distributed system specific development methodologies and integration:** a number of methodologies specific for certain types of distributed systems, such as SOA-based systems, not falling within the scope of this survey, e.g. SOMA [Arsanjani et al. 2008] and the work of [Dias et al. 2009], mention security but do not seriously consider its introduction in an integral way. An interesting (additional) future direction would be to integrate security engineering activities, perhaps inspired by some of the approaches reviewed here, into these and other existing development methodologies (cf. [Low et al. 2010]), including architecture-centric approaches such as ADD and QAW [Bass et al. 2003]. Following this direction leads to a consideration of (complementary) issues pertaining to the integration of security methodologies into different development methodologies, and a consideration of their mutual interaction (cf. [Uzunov et al. sub1]).

On a final note, we should point out that what has been discussed so far originates from a purely technical perspective on security methodologies, aiming at their improvement. Adopting such methodologies in real-life scenarios also depends on various non-technical factors (see, for example, the study of [Whyte and Harrison 2010], [Mouratidis and Giorgini 2006], [Hein and Saiedian 2009]), which must be addressed. These include:

- **A code-centric view of security:** in many cases developers and managers ascribe to the view that security problems arise only during implementation and should be addressed predominantly at that level. While many vulnerabilities are indeed introduced during coding, a number of researchers and security practitioners have shown that design level problems are no less critical (see, for example, [Dowd et al. 2007, Hoglund and McGraw 2004, Jaquith 2002]) and that security features should be introduced early in the SDLC (cf. [Section 1]). The fact that many technical papers (white papers, development notes, practitioner books) are often written in a colloquial style and avoid more precise architectural diagrams (even UML) – with the idea that only words and code are understandable to developers – does nothing to improve the situation, which is, to a degree, further compounded by code-based methodologies such as Microsoft’s SDL. The latter point accentuates the need in some cases for a paradigm shift, and in all cases for apposite developer education at least highlighting the scope of potential security problems, corresponding mitigation strategies, and the related benefits of model based approaches to security.

- **A limited emphasis on software only:** in a related vein, while sometimes developing a secure software application in isolation can be sufficient, often one must consider the system as a whole, i.e. the application together with its whole environment. This implies the consideration not only of all software levels (cf. [Fernandez 2003]), but also any supporting infrastructure, and entails the necessity for security to be considered in an integral way by all project roles.

- **Stubborn vision of security as an expense:** despite much research and many sound arguments for the converse, some companies continue to see the incorporation of security features during development as an extra expense, and do not appreciate the value of protecting their information, much less the value of the information about their customers (the recent security breaches of Sony’s PlayStation Network and Zappos’ customer database, both of which had large-scale effects, are just two pertinent examples).

In conjunction with the technical points discussed earlier, the latter points essentially conclude our attempt to answer the question of why security methodologies are not receiving more attention from the industry, and how the situation can be reversed.

Looking ahead into the future, the example of successful software development methodologies can certainly be emulated in the realm of secure software engineering, and this should be one of the major goals for new and existing security methodologies. The results and accompanying discussions in this survey, as well as the reviews and analyses of the methodologies themselves, can avail the latter task with clear indications of areas requiring improvement and important research directions, a number of which have been considered in this final section. Of course, there can be little doubt that our vision of the future is to some degree subjective, but we nevertheless believe that our concluding remarks can serve as a catalyst to further research in the area, and, ultimately, to foster a greater appreciation of the nature of some of the challenges which still need to be addressed.

**Acknowledgements**

We would like to acknowledge the careful reading and insightful comments by several of the J.UCS anonymous reviewers, which were indispensable in instigating us to make valuable improvements to our paper. We are especially grateful for the encouragement to expand the concluding part of the survey with “what we really thought” of the state-of-the-art, to be a little more provocative, and to indicate promising future directions as we see them.
References


Epilogue

In this chapter we surveyed the state-of-the-art in security methodologies, with an emphasis on methodologies for, or applicable to, distributed systems. The survey forms a foundation for the rest of the thesis; its conclusion also provides a miniature roadmap for the ensuing chapters. From the point of view of the overall engineering “toolkit”, the survey provides a rich source of security-related activities and techniques that can be distilled in a developer-friendly form, i.e. as various types of patterns. In Chapter 3, we present a number of such patterns as part of our approach to engineering security methodologies.
Chapter 2
*(Background & Literature survey II)*

**Securing distributed systems using patterns: A survey**

*From a sequence of these individual patterns, whole buildings with the character of nature will form themselves within your thoughts, as easily as sentences.*

– Christopher Alexander

**Prologue**

As the last chapter established, methodologies using the principles of encapsulation with respect to the security solutions used generally satisfy the most criteria for practical applicability. Of all the encapsulated solution types, security patterns appear to be the most promising – and they are certainly the most widely recognized and acknowledged in both academia and industry. Most importantly, security patterns are conceptual entities, making them largely independent of the methodologies in which they are used; their realizations need not be other, more refined security patterns, but different solution types altogether. It is thus natural to consider security patterns as a basis (though not necessarily final artefact) for the systematic development of secure distributed systems.

In this chapter we seek to fulfil this aim by reviewing both the range of available patterns for distributed systems security, as artefacts in themselves; as well as the suitability of existing techniques for applying the patterns – i.e. partial or complete pattern-driven methodologies – for a distributed systems context. In relation to the previous chapter, a number of the reviewed methodologies overlap, and receive cursory treatments in this chapter. With respect to the whole thesis, the value of the ensuing discussions on pattern-driven security methodologies lies in their assessment with respect to a distributed systems context; the review of patterns, on the other hand, is a basis for further developments in creating more “advanced” security solution artefacts in later chapters (Chapter 6 and 7 in particular), and as a collection of patterns that can be utilized by the specific methodology that will be constructed beginning with Chapter 3.
# Statement of Authorship

<table>
<thead>
<tr>
<th>Title of Paper</th>
<th>Securing Distributed Systems using Patterns: A Survey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication Status</td>
<td>Published, Accepted for Publication, Submitted for Publication, Publication style</td>
</tr>
</tbody>
</table>

## Author Contributions

By signing the Statement of Authorship, each author certifies that their stated contribution to the publication is accurate and that permission is granted for the publication to be included in the candidate's thesis.

### Name of Principal Author (Candidate)

<table>
<thead>
<tr>
<th>Anton V. Uzunov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
</tr>
<tr>
<td>Signature</td>
</tr>
<tr>
<td>Date</td>
</tr>
</tbody>
</table>

### Name of Co-Author

<table>
<thead>
<tr>
<th>Eduardo B. Fernandez</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
</tr>
<tr>
<td>Signature</td>
</tr>
<tr>
<td>Date</td>
</tr>
</tbody>
</table>

### Name of Co-Author

<table>
<thead>
<tr>
<th>Katrina Falkner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
</tr>
<tr>
<td>Signature</td>
</tr>
<tr>
<td>Date</td>
</tr>
</tbody>
</table>

### Name of Co-Author

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
</tr>
<tr>
<td>Signature</td>
</tr>
<tr>
<td>Date</td>
</tr>
</tbody>
</table>
Securing Distributed Systems using Patterns: A Survey

Anton V. Uzunov\textsuperscript{a,*}, Eduardo B. Fernandez\textsuperscript{b}, Katrina Falkner\textsuperscript{a}

\textsuperscript{a} School of Computer Science, The University of Adelaide, Adelaide, South Australia, Australia, 5005
\textsuperscript{b} Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, 777 Glades Rd., Boca Raton, FL 33431

* Corresponding author, Ph: (+618) 8313 7561, Fax: (+618) 8313 4366

E-mail addresses: anton.uzunov@adelaide.edu.au (Anton V. Uzunov), ed@cse.fau.edu (Eduardo B. Fernandez), katrina.falkner@adelaide.edu.au (Katrina Falkner)

Abstract
Driven by expanding scientific computing and business enterprise needs, the last decade has seen a shift towards software paradigms in which distribution plays a central role. The increasing size, complexity and heterogeneity of the corresponding systems is accompanied by an increase of security vulnerabilities that require mitigation via combined security and software engineering strategies. In this respect security patterns, which build on the success of design patterns and software patterns more generally, are a tool of great value. In this paper we comprehensively survey the state-of-the-art in securing distributed systems using (security) patterns, considering both relevant patterns and methodologies for applying them. In the first part of the survey, we provide detailed reviews of our selected security patterns, classify the patterns using a multi-dimensional scheme and evaluate them according to a set of quality categories. This highlights deficiencies in the reviewed patterns and provides a basis for identifying new or “missing” patterns and pattern classes. The newly identified and surveyed patterns are a step forward in defining a pattern language for distributed computing. In the second part of the survey, we briefly review a number of pattern-based security methodologies and evaluate their maturity and appropriateness for securing distributed systems.

Keywords: computer security, patterns, distributed systems, software engineering, methodologies

1 Introduction
Over the last decade the already large variety of distributed systems has grown significantly, spurred by applications such as peer-to-peer file sharing (Milojicic et al., 2002) and scientific computing (Celesti et al., 2010; Foster, 2005), as well as by enterprises moving towards software technologies in which distribution plays a central role. This trend has been accompanied by a general increase of security vulnerabilities (McGraw 2004), both potential and real, as corresponding systems increase in size, complexity and heterogeneity, requiring a broad range of strategies for the incorporation and enforcement of security attributes.

Researchers generally agree that for any security strategy to be successful, a software system of any type must be designed for security from the earliest stages (Anderson 2008; Devanbu and Stubblebine 2000; Fernandez, 1999; McGraw, 2006), with a consistent maintenance of the “security-push” throughout the whole development life-cycle afterwards. The tools of software engineering thus play an integral part in supporting security development activities (Mouratidis and Giorgini, 2006), becoming powerful aids in reducing the number and severity of potential vulnerabilities.

In this respect, security patterns (Fernandez, 2009; Schumacher et al., 2006), which build on the success of design patterns (Gamma et al., 1995) and software patterns more generally (Buschmann et al., 1996, 2007b) can be seen as a software engineering tool of great value in the quest for building secure distributed systems. As software patterns, security patterns capture the experience and expertise of many professionals in a form accessible to the security non-expert (Fernandez and Larrondo-Petrie, 2006; Schumacher, 2003). Security techniques and countermeasures captured as patterns can be customized and applied in a number of contexts while guiding developers from a problem in a given context to a proven solution with predictable consequences. A large number of security patterns have appeared in pattern catalogs (e.g. Blakeley and Heath, 2004; Dougherty et al., 2009) and books (e.g. Fernandez, to appear; Microsoft, 2006; Schumacher, 2003; Schumacher et al., 2006; Steel et al., 2005) to cover a wide variety of concerns and contexts – ranging from operating systems and VOIP to cryptography and network security. Despite their usefulness and, one could argue, “proven track record” in the field, there are few comprehensive surveys of security patterns, and currently no surveys concentrating on security patterns – or the means for applying them – applicable to distributed systems in general.

Security patterns can be applied to two ways: individually or in groups to help (partially) secure a system; or as a part of an overall, coherent approach, i.e. a pattern-based security methodology. In either case, any factors of software distribution should be taken into account both by the patterns and by any methodology applying those patterns.

In this paper we comprehensively survey the state-of-the-art in securing distributed systems using (security) patterns, addressing both individual patterns and methodologies for applying them. The survey is divided into two parts, reflecting its two-fold purpose.
In the first part, we comprehensively survey individual and groups of security patterns specific for or with strong applicability to general distributed settings with the aim of identifying patterns that require development. The surveyed patterns and our suggestions and concrete ideas for new patterns or pattern languages can be seen as a first step towards a more complete security pattern language for distributed computing. As features and subsidiary contributions leading to the latter aim, we:

1. provide detailed reviews of each pattern or group of patterns according to a hierarchical classification scheme, attempting to cover as many relevant security patterns in the published literature as possible;
2. group and organize the patterns by adapting the classification schemes of Washizaki et al. (2009) and VanHilst et al. (2009), which promotes easier navigation of the existing pattern catalog;
3. evaluate the quality of the patterns according to an adaptation of the scheme proposed by Laverdiere et al. (2006), attaching quality indicators to each pattern as appropriate, which allows software professionals to quickly identify patterns with less reliability and/or missing features.

The subsidiary contributions above are made in light of the fact that, beyond their value in the software engineering process, security patterns are also used by software professionals for security educational and training purposes. Indeed, according to a recent study by Elahi et al. (2011), security patterns account for over 20% of educational resources used by professionals working in international and Chinese national IT companies, being favoured over books and government standards due to their reliability and ease of use. The ultimate aim of this part of the survey (to identify patterns requiring development) can also be seen in this light as a contribution towards identifying additional domain-knowledge resources.

Limiting the scope of patterns to those relevant for distributed systems enables us to review and analyze a number of patterns not present in existing surveys (e.g. Hafiz et al., 2011; Yoshioka et al., 2008; Yskout et al., 2006 and others) or research dealing with pattern quality analysis (e.g. Heyman et al., 2007, Laverdiere et al., 2006 and others), providing a considerably more in-depth and comprehensive overview than has been available until now.

In the second part of the survey, we attempt to briefly review all existing pattern-based security methodologies with the aim of evaluating their appropriateness for securing distributed systems. Quality indicators are proposed and attached to methodologies as was done for patterns, providing a simple and effective evaluation of whether a given methodology can be used in practical situations. As a whole, the evaluations support decisions as to which methodology is best suited for a given project, and also help to highlight tendencies and possible research directions. Taken by itself, this part of our survey also helps to increase awareness of a range of pattern-based methodologies, which so far have not received due attention in the literature.

Using the individual patterns or their composites from the first part of the survey can aid in (partially) securing a distributed system, but their greatest merits become manifest when they are used as part of an overall methodology. Using a pattern-based methodology for securing a distributed system implies that the relevant patterns must necessarily originate from a collection applicable to such systems, and hence the quantity and quality of patterns effects the methodology's capacities. Therefore, the two parts of the survey are mutually interdependent, with the first part being enhanced by the second, and the second relying on the first.

The rest of this paper is structured as follows. As a prelude to our survey of patterns, Section 64 briefly covers background material and related work on security patterns: in Section 64 we discuss the history and necessary concepts in software and security patterns to give a better foundation for the ensuing discussions; in Section 65 we provide a brief overview of other surveys and catalogs of general security patterns, as well as pattern classifications and quality evaluations to place our own survey of patterns in context; and in Section 66 we briefly outline a set of distributed system security concerns; and in Section 67 we outline the organization of the patterns in the survey. The survey of patterns itself is contained in Section 67.

In Section 74 we organize the surveyed patterns and evaluate their quality, attaching indicators to each pattern as appropriate. Based on an analysis of the existing patterns and distribution concerns, we subsequently propose ideas and concrete suggestions for a number of new patterns or pattern languages for distributed settings.

In Section 78 we survey pattern-based security methodologies, dividing the approaches according to their maturity, evaluate their appropriateness for securing distributed systems based on a number of proposed categories, as well as whether a given methodology can be used in practical situations. In Section 83 we conclude by summarizing our contributions and discussing future work.

2 Background, Related Work and Organization of Patterns

In what follows we provide some relevant background on security patterns (Section 64) and distributed systems security (66), discuss related work (Section 65) and outline the scope and classification scheme used in presenting the survey of patterns (Section 67).

2.1 Brief History of Software and Security Patterns

The concept of a pattern in design and architecture first began with the work of the architect Christopher Alexander, appearing in books such as “The Timeless Way of Building” (Alexander, 1979) and others. Alexander conceived
patterns as solutions that can be applied again and again in different contexts in a generative fashion. The idea was taken into the world of computer science by Beck and Cunningham (1987), and reached widespread acceptance in the software community via the book by Gamma et al. (1995), in the form of design patterns.

Software patterns possess a number of valuable characteristics, which have made their application successful in a variety of software engineering contexts. These include helping to organize expert knowledge into a standardized format, supporting design re-use and providing domain-specific vocabularies to improve communication in project teams (Buschmann et al., 1996, 2007b; Riehle, 2011; Schmidt, 1995). Software patterns are also useful for teaching and education, and can help in evaluating existing and legacy systems.

From an architectural point of view, patterns encapsulate design strategies, decisions (Taylor et al., 2010) or constraints (Bass et al., 2003), which determine a family of satisfying architectures, and result in actual (partial) architectures upon instantiation. More concretely, according to Schmidt (1995), “patterns capture the static and dynamic structures and collaborations of components in successful solutions to problems that arise when building software in [various] domains”. The design structures or strategies can be expressed as a set of architectural models (a representative partial architecture) accompanied by textural descriptions of trade-offs, consequences of application etc., usually in a standard template.

Security patterns take the concept of software patterns and apply it to the domain of computer security. Security patterns have gained significant momentum since their inception in Yoder and Barcalow’s (1997) seminal work, although similar approaches – object models of security in the context of object-oriented databases (Fernandez et al., 1993) – appeared before that time as well. Like software patterns in general, security patterns capture successful security designs and/or strategies in a generic form, which can be applied or instantiated to produce solutions with well-defined properties. Following Buschmann et al.’s (1996) classification of software patterns, security patterns can be classified as architectural or strategic if they encapsulate larger-grained solutions effecting a system’s architecture as a whole; or as design or tactical if they encapsulate smaller-grained solutions with localized effect on specific modules or sub-systems.

2.2 Related Work on Security Patterns

As mentioned in the Introduction (Section 63), a number of catalogs and books have appeared over the last decade in order to collect security patterns for use by professionals and researchers. Yoshioka et al. (2008), for example, present a broad and relatively comprehensive survey of general security patterns according to the phases of the software development life-cycle, briefly covering security-related patterns such as attack patterns (Hoglund and McGraw, 2004) and classifications, as well as discussing quality features. Yskout et al. (2006) and Scandariato et al. (2008) survey 218 patterns from a variety of sources, and refine and sort them according to their particular selection criteria for pattern quality to produce a system of 110 base security patterns. Wiesauer and Sametinger (2009) attempt a comprehensive survey and taxonomy of security patterns based on attack patterns; however, they do not cover any new ground with respect to their survey, and, other than an example, do not present an actual taxonomy in full. None of these surveys are concerned with or purposefully cover security patterns for distributed systems, or provide in-depth reviews and/or analyses of individual patterns or pattern groups.

With respect to catalogs, two books containing large collections of security patterns are (Schumacher et al., 2006), which captures the collective effort of a number of researchers who have published in the area; and (Steel et al., 2005), which presents a collection of security patterns applicable to Web-services and the J2EE platform. Schumacher et al. (2006) in particular collects a number of patterns published previously, most notably updated versions of Yoder and Barcalow’s (1997) patterns, Fernandez and Sinibaldi’s (2003) patterns for operating systems security and others. A compilation of approximately 60 patterns by Eduardo B. Fernandez and colleagues also appears in (Fernandez, 2011; Fernandez, to appear). Other catalogs include Dougherty et al., (2009), who present a somewhat reduced set of patterns similar in scope and functionality to those found in Schumacher et al., (2006); and Kienzle et al. (2002), who catalog a large collection of patterns dealing with a range of security issues in web application development. Kienzle et al.’s (2002) classification of patterns as structural and procedural, however, results in a number of procedural non-patterns, which reduces the scope of their catalog. The Open Group (Blakeley and Heath, 2004) and Romanosky (2001), both present mostly isomorphic collections of security patterns in comparison to previous catalogs. Most recently, Hafiz et al. (2011) catalog 96 security patterns (after eliminating duplicates from 174 candidates) as part of their work in developing a general security pattern language. Despite the comprehensiveness and the claim that “all” security patterns in the literature are covered, a number of patterns are missing in Hafiz et al.’s selection – as evidenced by the current paper – and many more do not meet the criteria for being full security patterns (e.g. they represent processes, not artefacts). None of these catalogs specifically collect security patterns for distributed systems, even if some of the patterns are applicable in that setting. In comparison to the work of Hafiz et al. (2011) in particular, which also addresses the development of a pattern language, we employ a different classification scheme and do not endeavour to present a general, accomplished language complete with relationships; rather, we indicate new pattern classes (Section 74) that can be combined with the surveyed security patterns in Section 67 to form the foundations for developing such a language specific to distributed computing.

With respect to classifications of patterns, a number of classification schemes have been devised to organize security patterns and aid the target audience in selecting appropriate patterns for their design situations. Classification schemes form a useful navigational tool through the space of available security patterns and thus their
solutions. Some examples include Hafiz et al. (2007), who present a survey of classification approaches such as classifying security patterns according to the Zachmann framework, Microsoft STRIDE and the classic CIA (confidentiality, integrity, availability) triad; Fernandez et al. (2008c), who consider hierarchical relationships between patterns, a concept extended later by Washizaki et al. (2009) to encompass pattern relations (through pattern graphs) and different classification concerns (through dimension graphs); Wiesauer and Sametinger (2009), who attempt to classify security patterns according to attack patterns; VanHilst et al. (2009), who classify security patterns according to a multi(six)-dimensional conceptual scheme with each dimension representing a particular concern; Alvi and Zulkernine (2011), who classify patterns according to development phases and vulnerabilities addressed; and Heyman et al. (2007) with Scandariato et al. (2008), who overlay an organizational structure on top of their collection of security patterns.

Besides helping to navigate pattern catalogs, classification schemes are a useful tool to determine “missing” patterns for different domains. We employ an adaptation of Washizaki et al. (2009) and VanHilst et al.’s (2009) approaches to classification in Section 74, both to help in categorizing patterns more clearly and to determine the patterns requiring development. The work of (Washizaki et al., 2009) and (VanHilst et al., 2009) was chosen as a basis because the multi-dimensional nature of their schemes currently represents the most flexible and comprehensive approach. Our approach to organizing and presenting the patterns in this survey is discussed below in Section 67.

With respect to evaluating the quality of patterns, Heyman et al., (2007) reveal that approximately one third of the patterns in the literature do not qualify as software patterns per se, but as processes, general guidelines or principles. The approach used in their study was based on a qualitative evaluation considering factors such as use of diagrams or models, consistent templates and others. The quality of security patterns is considered more generally by Laverdiere et al. (2006), who classify pattern documentation deficiencies in five categories: over-specification, under-specification, lack of generality, lack of consensus in literature and misrepresentation. Their evaluation of patterns predominantly from Blakeley and Heath (2004) shows that a number of published patterns fail to meet the necessary quality criteria, leading to the proposal of a more comprehensive pattern template. New pattern templates as a solution to quality assurance are also proposed by Ortiz et al. (2010), who argue that current security pattern documentation does not consider (among other things) deployment and trade-off scenarios for real-life, complex software. However, the latter work does not stipulate realistic criteria for the evaluation of patterns, since patterns cannot be expected to specify, for example, deployment scenarios, impacts on existing software components etc. with any precision. Despite discrepancies in the latter works, they uniformly acknowledge that the quality of a pattern's description is extremely important for the achievement of its goals as well as for its application and deployment. In light of this fact, we evaluate the quality of our reviewed patterns in Section 74 based on an adaptation of the documentation-deficiency categories presented by Laverdiere et al. (2006), which we believe provide a clear and relatively objective measure of pattern quality. Our pattern selection allows us to extend the analysis in Laverdiere et al.’s work to a much wider range of security patterns.

### 2.3 Distributed Systems and their Security Concerns

Distributed systems form a broad class of software systems in which, from an architectural perspective, the functional components are spread or distributed across multiple machines or nodes connected by some type of network. A number of characteristics such as autonomy, the implementation of a single-system image (SSI), resource sharing across multiple, heterogeneous systems and co-operation between processes towards a common goal can be added to the latter definition, depending on the research angle taken and the aims of the particular system (cf. Foster, 2005; Milojicic et al., 2002; Taylor et al., 2010; also see Coulouris et al., 2005; Leopold, 2001; Pourzandi et al., 2005; Tanenbaum and Van Steen, 2007).

Recent trends have seen the rise of middleware frameworks as the prevalent solution for realizing distribution and hiding (to different degrees) the details of the underlying network from application/system developers (Schantz and Schmidt, 2007; Vöelter et al., 2005), however, other approaches resting on operating system concepts, communication toolkits as well as raw network programming can also be utilized (Buschmann et al., 2007a; Schmidt et al., 2000; Tanenbaum, 1995).

Distributed systems are uniformly susceptible to all the security threats of software designed for standalone machines, as well as a number of new threats stemming from the nature of distribution and its realization in different middleware/platform varieties, the protocols used, and the specifics of the particular system (e.g. peer-to-peer, high-availability and others). This implies the need for comprehensive protection from threats at both the network and system levels via software components addressing a correspondingly wide range of security concerns (see, e.g. Belapurkar et al., 2009 or Coulouris et al., 2005 for more background).

Such concerns include distributed **authentication**, to protect against identity spoofing, which in turn implies the management of identities throughout the system (i.e. identity management concerns). Identities as well as other security-relevant information should be encoded or managed in **security contexts** (Benantar, 2006), which can be defined in a broader light as the embodiment of any security information passed throughout the system in the form of security sessions, ID tokens etc.

To protect against eavesdropping and unauthorized information disclosure a range of cryptographic techniques can be used, ensuring message secrecy, integrity and authenticity (**secure communication** concerns). Unsolicited
network information should be checked and/or filtered in distributed settings, requiring the design of components concerned with data filtering. The system should be protected from privilege escalation and actions of external entities via components encapsulating authorization and access control concerns. Sometimes the latter concerns are combined into authentication/authorization frameworks, especially for authorization models based on identity information.

Software distribution necessarily implies some form of networked process collaboration (or distribution control), as well as the processing of data either sequentially or in parallel via network messaging. These functional concerns, often designed (knowingly or otherwise) using software patterns for distribution (see Buschmann et al., 2007a; Schmidt et al., 2000), should also contain security properties a priori. Finally, the run-time security monitoring of communications and/or particular behaviour takes on special significance for distributed systems in light of the possibility of mounting distributed attacks targeting particular nodes.

Practical solutions addressing the aforementioned concerns can be realized at many levels of abstraction, in underlying platforms or at the application level in various domains to form a complete security architecture for a given distributed system. In our survey we use several (typical) domains of application: middleware frameworks – either distributed object or component based; distributed object applications; and general networked applications.

2.4 Scope and Organization of Patterns in the Survey

The patterns in this survey are grouped according to three hierarchical categories: (1) the security concerns they address, ranging from security contexts and identity management to providing a framework or collection of patterns for secure distributed middleware; (2) the domain to which they are most applicable; and (3) the paper(s) in which the patterns appear. This classification scheme is used in part for the purposes of presentation – papers presenting several patterns with one or more patterns addressing a different concern will appear under the concern (and domain) to which most of the patterns apply. In Section 74 we summarize the results of the survey, considering patterns individually under a more rigorous classification scheme.

The research process used for finding patterns was to exhaustively search the open literature, including books (e.g., Microsoft, 2006; Schumacher et al., 2006; Steel et al., 2005 and others), conference and journal papers (including the PLoP – Pattern Languages of Program Design – series of conferences) as well as industry and government technical reports. We utilized a wide variety of electronic resources such as Internet search engines, IEEE Xplore, ACM Digital Library, the Scopus database and others throughout the process, in addition to our own knowledge of the security patterns literature. The collections of patterns found were subsequently analyzed for relevance to distributed settings and classified according to the concerns delineated in Section 66.

Our selection of patterns is based on applicability to distributed systems in general; we do not attempt to survey patterns specific to particular types of systems that cannot immediately be interpolated to a more generic distributed setting. Thus, for example, a number of security patterns specific to Web-services and web-based scenarios, which can be found in (Microsoft, 2006), (Steel et al., 2005), (Kienzle et al., 2002), (Fernandez, 2004b), (Fernandez et al., 2006b), (Delessy et al., 2005), (Schumacher et al., 2006) and others, are omitted. However, the patterns among the latter references that are applicable to any type of networked system, and hence distributed systems in general, are included in the scope of this survey. Of the many generic security patterns (i.e. not specific to any type of system) in the literature we consider only those applicable to distributed systems without the need for extensions or specializations. Since the focus of the survey is explicitly on security patterns (which are full inheritors of software patterns), we do not consider other security development artefacts, including parameterized model templates and others that are sometimes termed patterns as well, e.g. in the work of Schmidt et al. (2011) or Rossebo and Bræk (2006).

3 Survey of Security Patterns for Distributed Systems

Our survey begins with considering patterns for security contexts (as defined in Section 66 above). Subsequent patterns, grouped in the order of security concern, address issues in identity management, authentication, authorization and access control, secure communications and filtering. This is followed by security patterns which are part of pattern frameworks or languages dealing with authentication/authorization concerns, security patterns dealing with issues in distribution control and patterns dealing with issues in processing. The organization of patterns within each of these areas of concern follows the schema outlined above in Section 67.

3.1 Security Contexts

Security contexts (Benantar, 2006), can be defined as the embodiment of any security information passed throughout the distributed system in the form of security sessions, ID tokens etc.

3.1.1 All domains

Morrison and Fernandez (2006a) present a Credential pattern for use in distributed systems. A credential is an encapsulation of certain security information in the system such as authentication IDs, access rights etc., which can be given by a subject as evidence to an authentication and/or authorization service to prove identity and thereby gain
authority to perform an operation (Coulouris et al., 2005). In many instances, credentials act like distributed capabilities (Delessy et al., 2007b).

Weiss (2006) describes a proxy-certificate based Credential Delegation pattern for grid systems using Public Key Infrastructures (PKI). Delegation can be restricted by (1) short timespan, (2) privilege restriction (i.e. restrict possible operations) and (3) single-time delegation (non-delegatable proxy credential). A proxy credential is created by generating a public/private key pair and signing the certificate. A user can then grant the certificate to some other entity, which can present its own credentials together with the proxy certificate to a server to gain any necessary authorization. The proxy credential acts like a capability in this setting. Credential Delegation also allows single sign-on semantics. The pattern forms the basis of a pattern language for grid security (Lu and Weiss, 2008), which we discuss further below.

3.2 Identity Management

The identity information of system entities is a specific form of security information used in authentication and (potentially) authorization mechanisms. Identities in a distributed system may require careful management, especially in the presence of multiple administrative domains and/or when utilizing multiple security contexts.

3.2.1 All domains

In (Delessy et al., 2007a), a pattern language for identity management in distributed systems (and more specifically in SOA-based systems) is presented, consisting of three new patterns – Circle of Trust, Identity Provider and Identity Federation – as well as Credential (Morrison and Fernandez, 2006a) and Authenticator (Schumacher et al., 2006). Different organizations may have different credential/identity management mechanisms and protocols; if subjects require use of services in security domains for which they are not authenticated, their identity may need to be federated (i.e. combined) from different identity provision services in disparate security domains that trust each other. This is the main idea behind the identity federation patterns. The Circle of Trust pattern proposes the establishment of business relationships between service providers to ensure that identities can be queried and combined; presumably this can also be embodied in technical relationships: caches of trusted domains, time-based trust depreciation algorithms etc. The Identity Provider pattern encapsulates the idea of a centralized identity/credential repository for a particular security domain. Finally, the Identity Federation pattern encapsulates (conceptually) the process of creating a federated identity by transparently gathering a subject's attributes (partial credentials) from its local identities within the security domains in the circle of trust between organizations. As pointed out earlier, this new federated identity can be used to authenticate a subject for services in another (trusted) security domain.

3.3 Authentication

Authentication in distributed systems is used to verify the identities of users or processes and to protect against identity/name spoofing. Distributed authentication uses specialized protocols and requires more careful designs than the standalone system case.

3.3.1 Distributed Object-based Applications

Brown et al. (1999) present an Authenticator pattern for application-level, distributed object-based systems by extending the Gang-of-Four Abstract Factory pattern (Gamma et al., 1995) with generic authentication/identity validation functionality. The Authenticator pattern is primarily a creational pattern, even though it can be modified to pass a reference to an existing object instead of a newly created one under given circumstances. The participants – disregarding an object factory for the creation of objects on successful authentication – consist of a single abstract “Authenticator” class with an “authenticate()” method and concrete specializations embodying different authentication algorithms. The choice of algorithms and any ensuing design bearing they may have is not discussed. In terms of dynamics, under the proposed design an authentication requester object must call the “authenticate()” method each time identity verification is required. This feature requires careful implementation, otherwise it may become vulnerable to abuse (malicious subjects repeatedly trying to authenticate) leading to denial-of-service attacks. Hays et al. (2000), in whose work the Authenticator pattern is used as part of a pattern-based security framework for distributed object-based applications, suggest that the Strategy pattern (Gamma et al., 1995) could be used (instead of pure polymorphism) for varying the authentication algorithms in Authenticator. The pattern does not attempt to treat global, system-wide authentication issues.

3.3.2 General Networked Applications

Fernandez and Warrier (2003) present a Remote Authenticator/Authorizer pattern, which is a condensation of the RADIUS architecture (with proxy forwarding) for network-based authentication and authorization. The pattern is based on forwarding a user's authentication request to a centralized proxy (authentication) server, which can then delegate the request to a spatially separated server elsewhere. The server thus acts as a proxy for the user's request.
Authorization can be enforced by checking the subject's rights through the indirection of a corresponding role (applying the Role pattern, Yoder and Barcalow, 1997). The pattern is more comprehensive than the Authenticator in Brown et al. (1999), offering a complete RADIUS-like (challenge-based) architectural model. In the proposed design, the server can act as both a proxy and an authentication server, and can also be chained to a number of proxies to increase flexibility. The server can be designed to handle multiple requests as well. Although not detailed further in the paper, this feature could potentially be implemented using concurrency patterns in (Schmidt et al., 2000).

3.3.3 All domains

In (Microsoft, 2006), two patterns are presented for distributed authentication: Direct Authentication and Brokered Authentication. Direct authentication deals with peer-to-peer scenarios where each subject authenticates to its corresponding object. The object takes the subject's credentials from a store, which (potentially) helps to place an extra level of security by attempting to prevent credential spoofing or replay. Brokered authentication interposes a trusted third party (authentication server) in between the authentication participants. In such an instance the authentication server could use push or pull semantics (Oppliger, 1996), although this is not mentioned as part of the pattern. Brokered Authentication is seen as an architectural pattern in (Microsoft, 2006), and refined towards implementation, as “child” design patterns, into Kerberos, X.509 (PKI-based) Certificate and Security Token variants. Kerberos-based authentication uses a centralized authentication server, X.509 certificates are a decentralized solution requiring a trusted Certificate Authority (CA), while Security Tokens are more specific to web-services and associated security standards. It should be remarked that the descriptions of the design patterns (variants) is web-services centered, and has to be interpolated for more generic settings. The process of refining a security solution embodied in patterns from requirements to architecture (high-level) to design (implementation-oriented) to specific technologies and protocols could be useful in guiding the designer through the application of the pattern. This process also appears in Rosado et al. (2006), similarly in the context of Internet-based (i.e. web-services based) applications. The Direct and Brokered Authentication patterns appear (unchanged) in Erl (2009) in the context of SOA.

3.4 Authorization and Access Control

Authorization in a distributed system implies controlling which operations or accesses are allowed and which are not allowed, to protect against misuse of assets via privilege escalation and illegal actions by both external and internal entities.

3.4.1 General Networked Applications

While all authorization models have applicability to distributed systems, some models have special features that make them more suitable. Besides patterns for RBAC (first appearing in Fernandez and Pan, 2001), which is widely used in middleware and distributed applications, Pribe et al. (2004) present a pattern for Meta-data or Attribute Based Access Control (MBAC/ABAC) – an authorization model which extends the Access Matrix model by adding the concept of meta-data or attributes for subjects and/or objects. By determining authorization decisions based on attributes during run-time, MBAC/ABAC obviates the need for relating subjects to pre-defined identities – a feature that makes it useful for systems in which subjects cannot be determined in advance (e.g. Internet-based applications). Attributes attached to subject/object descriptors can also define contexts – a fundamental concept for location-based authorization. Pribe et al. (2004) further define MBAC/ABAC with Session, an extension allowing subjects to include a subset of their permissions within a given security session. The MBAC/ABAC pattern can be used in conjunction with other models such as RBAC, since subject attributes can be assigned to roles as well as users.

Delessy-Gassant et al. (2004) and Fernandez (2004b) present Application and XML Firewall patterns, which are essentially composite patterns consisting of Firewall (Fernandez et al., 2003), Authorization (particularly RBAC – Fernandez and Pan, 2001; and more generally what is referred to as PBAC in Delessy et al., 2007b), Authentication (Fernandez and Sinibaldi, 2003) and Reference Monitor (Fernandez, 2002). The purpose of the Application Firewall is thus not so much to filter as to authorize (network) requests via a policy enforcement point. These enforcement points can be deployed in either a system-wide centralized location or on a per application basis. In the latter case the Application Firewall is an application of the Single Point of Access pattern (Yoder and Barcalow, 1997). In both cases policies as well as authentication information are stored in a centralized repository for easier management. Network requests or accesses are intercepted by the firewall's policy enforcement point(s), authenticated, and authorized by checking their permissions against the access control rule-set. The Application Firewall pattern can, therefore, be seen as a user-space/user-level replacement of the corresponding OS authorization pattern (Schumacher et al., 2006) functionality. The XML Firewall is an Application Firewall tailored mainly for web-services, adding XML-specific content inspection features.
3.4.2 All domains

In (Delessy et al., 2007b), three patterns for access control are presented for use in distributed contexts: policy-based access control (PBAC), ACL and Capability. The last two are put forward as concrete realizations of the Access Matrix pattern (Schumacher et al., 2006) which in turn is a realization of the PBAC pattern – an abstract security pattern (Fernandez et al., 2008b) embodying access control via an instantiation of Authorizer (Fernandez and Sinibaldi, 2003) with a centralized policy repository. The most interesting feature of the PBAC pattern is the requirement of policy resolution from heterogeneous policy sets to form a single uniform decision; the details of the resolution algorithm(s) needed are not specified. The ACL and Capability patterns apply to single-machine systems as well as distributed settings; their use in the presence of parallelism and local/global enforcement issues is not considered.

3.4.3 Distributed Object-based Applications

Neves and Garrido (1998) describe Bodyguard – a pattern that “simplifies the management of object sharing over a network”, and which is in many ways a simplified Broker (Buschmann et al., 1996). The essence of this pattern from the security perspective is the use of Bodyguard objects – second-level proxies attached to an object or resource, which determine the access semantics. Specialized “bodyguard” objects (i.e. expressing some particular authorization rule) are attached to every (shared) resource in the system together with a Proxy object. The access rights encompassed by bodyguard objects are not evolvable, and new bodyguard objects must be created for new sets of rights. The real advantages of this approach are (1) the decoupling of distributed functionality such as synchronization (contained in the proxy object) from access control (contained in the bodyguard object) and (2) hiding the authorization policy decision point (PDP, in the form of the bodyguard) for additional protection. The pattern is intended to be used in a peer-to-peer application environment where simple authorization is required without underlying platform support; authentication is not considered. There is no reason, however, why the idea of a 2-level proxy should not be applicable at the system or middleware levels as well.

3.4.4 Distributed Object Middleware Frameworks

A pattern with ubiquitous use in many object-based middleware frameworks is the Invocation Interceptor pattern presented in (Voelter et al., 2005). From a security perspective, the purpose of the Invocation Interceptor is to capture object method calls to enforce security constraints throughout an object-based system. Since in many middleware frameworks objects can only interact between each other via some form of method call, capturing calls tends to imply complete mediation of object actions in the system. This in turn implies that some form of mediating component must be interposed between objects, such as a security kernel, system controlled IPC etc. In Voelter et al.’s, pattern, invocation interceptors are hooked into an Invoker pattern (Voelter et al., 2005) and called prior to and after invocations. Several Invocation Interceptors can be chained (applying the Chain-of-Responsibility pattern, Gamma et al., 1995) or otherwise coordinated to add additional (security) functionality. In this respect the Invocation Interceptor acts like a dissociated Bodyguard (Neves and Garrido, 1998). In any case, out-of-band communications break the pattern since then the interceptors, and hence their security functions, would be circumvented. It may be possible to extend the semantics of the Invocation Interceptor to general software architectural connectors (Shaw, 1996) and thereby increase the pattern's scope and usefulness.

3.5 Secure Communications

Secure communication channels must be set up between any communicating entities in a distributed environment to protect against eavesdropping and unauthorized information disclosure. A range of cryptographic techniques can be used for this purpose, ensuring message secrecy, integrity and authenticity, or a combination of the latter, depending on the needs of the system.

3.5.1 All domains

Patterns for secure communications in networked environments and hence more specialized distributed settings are presented in (Yoder and Barcalow, 1997) (and later in Schumacher et al., 2006, Secure Channel), (Blakeley and Heath, 2004, Secure Communication), (Microsoft, 2006, Message Protection patterns – Data Confidentiality and Data Origin Authentication) and (Steel et al., 2005, Secure Pipe). The patterns in all cases embody the semantics of a secure channel (see Ferguson et al., 2010) and are implemented via security protocols at some layer of the network stack (e.g. SSL – transport layer, IPSec – network layer). For the exception of Blakeley and Heath's (Open Group) Secure Communication pattern, the aforementioned patterns are tailored for web-based environments, rendering their more general applicability somewhat limited.

A set of patterns for Virtual Private Networks (VPNs) are given in Kumar and Fernandez (2010), which include an abstract version together with two technology dependent variants based on IPsec and TLS/SSL for network and transport level security respectively. The VPN patterns can be seen as a technological realization of the secure channel patterns mentioned above.
A number of patterns also address cryptographic concerns in general contexts and for communications in particular. Lehtonen and Parssinen (2001), for example, present a pattern language for key management within the context of secure communications. The work arose from a pattern story (Buschmann et al., 2007b) during one of the PLoP conferences, which was converted into a series of patterns (nominally, at least) to correspond to the various activities in establishing a secure channel as per the above references. Nevertheless, on more careful scrutiny a number of Lehtonen and Parssinen’s patterns do not adhere to the pattern criteria, since they neither capture nor provide design solutions, and their application does not result in actual architectural decisions. An example is a pattern (“Alice and Bob”) used to define the subsequent terminology of the paper, as well as a pattern used to determine the context for subsequent patterns. Despite the shortcomings of the pattern language, Lehtonen and Parssinen identify the existence of three main key management stages during secure communication: (1) key generation, (2) key exchange and (3) key storage. The corresponding pattern descriptions in the language (“Sealed Envelope”, “The Real Thing” etc.) loosely correspond to this scheme.

More rigorous and well-defined is Braga et al.’s (1998) Tropyc: a pattern language for cryptographic software. Tropyc’s aim is to cover the security mechanisms of message communications by providing patterns for (1) secrecy (via encryption), (2) integrity (via message digests or a Modification Detection Code), (3) authenticity (via a Message Authentication Code) and (4) non-repudiation (via digital signatures). Strictly speaking, these concerns are overlapped and can be achieved using three of the four mechanisms. While a number of the patterns presented are, to a greater degree than in (Lehtonen and Parssinen, 2001), patterns per se, a number of the Tropyc patterns are also composites (i.e. not individual patterns), which introduces unnecessary redundancy in the language. The most interesting feature of Braga et al.’s (1998) patterns (and what ultimately redeems them from becoming pure descriptions of known cryptographic measures for communications security) is the use of a “meta-pattern” – really a parent – from which all the patterns inherit their structure and behaviour. This ensures that the patterns possess a higher degree of design-level applicability than would otherwise be the case. The meta-pattern can also be thought of as the template for all other Tropyc patterns, allowing one to consider them as a single unit for end-to-end communications security.

In a related vein to the latter two papers, De Souza and Matwin (1999) describe a Secure Client/Server Communication pattern in an attempt to capture the design of secure session establishment in the context of a client/server system, along with a set of cryptographic patterns supporting the necessary key management. In actual fact, the session establishment holds valid for any two communicating parties, with nothing being present in the pattern to specialize it to client/server architectures. The Secure Client/Server Communication pattern proposes the addition of a security layer (Yoder and Barcalow, 1997) at both ends of the communication, containing a single secure socket component at each end to handle the cryptographic functionality. The stages of establishing the session include (1) (symmetric) session key generation, (2) session key distribution/exchange using asymmetric encryption (with RSA) and (3) usage of the session key to encrypt subsequent communications. One salient feature of the pattern lies in the generation and use of two session keys (one for each communicating end) as opposed to the usual use of one symmetric key (e.g. Ferguson et al., 2010) in secure communications. Further break-down of the pattern into its key management constituents results in a similar set of pattern descriptions presented in (Lehtonen and Parssinen, 2001) and (Braga et al., 1998), distilling well-known encryption mechanisms into a textual template.

Hashizume and Fernandez (2009) and Hashizume et al. (2009) take a more detailed approach and present a set of patterns for symmetric and asymmetric encryption, XML encryption and digital signatures, largely within the context of web-services, but applicable to a more general range of systems as well. Symmetric Encryption (Hashizume and Fernandez, 2009), in particular, addresses the same problem as Braga et al. (1998) and both extends and specializes the latter’s (Information) Secrecy pattern, simultaneously placing greater emphasis on design strategy. The distinction is thus in the attempt to ensure that the encryption strategy becomes a coherent design solution with better-defined participants and more explicit architectural effect, documented via a UML model. The standard encryption techniques apply otherwise. Similar remarks hold true for the Digital Signature pattern in (Hashizume et al. 2009), which is a significant improvement over the closely related Message Integrity (MDC-based) pattern in (Braga et al., 1998) dealing with the same concerns, notwithstanding the merits possessed by the latter.

Together with the Secure Channel pattern (or one of its variants) mentioned previously, the cryptography patterns presented in the above references form a closed set of patterns for secure communication in a distributed setting.

### 3.6 Filtering

Unsolicited network information should be checked and/or filtered in distributed settings, requiring the design of components concerned with data filtering.

#### 3.6.1 Distributed Object-based Applications

Flanders and Fernandez (1999) describe the Data Filter Architecture pattern, whose intent, as the name implies, is to filter unwanted information from a data (most importantly network) stream according to a set of pre-defined policies. The pattern appears to have arisen from a concrete distributed system and bears this mark both in its level of detail (down to detailed design) and specificity. In essence, the pattern defines a whole fine-grained partial architecture for filtering, with numerous actors such as Filter Services and Filter Agents performing the different...
filtering tasks guided by filter policies stored in repositories. Given the rudimentary nature of data filters, which can be realized by two or three participants at most (a data filter object accepting or discarding information based on a ruleset and one or more decision algorithms), the complexity of the Data Filter Architecture pattern makes its use fully justified only for more specialized cases. Nevertheless, it offers useful insight into how a more elaborate architecture for filtering can be constructed in distributed middleware systems – especially CORBA, in which setting, judging by the examples used and the UML models, the pattern was originally implemented.

3.6.2 All domains

Extending the idea of data filters is the idea of a network firewall. Patterns for firewalls are presented by Fernandez et al. (2003) (and again later in Schumacher et al., 2006), Delessy-Gassant et al. (2004), Shumacher (2003b) and Fernandez (2004b) capturing the designs of basic, proxy-based, stateful, application and XML-inspecting firewalls. All of the above can be used in a distributed context to filter out and/or to provide protection from malicious network traffic or requests.

3.7 Authentication/Authorization Frameworks and Languages

Authentication and authorization (occasionally with other security concerns) can be combined into distributed authentication/authorization frameworks (or infrastructures – cf. Lopez et al., 2004), especially when the authorization models used require identity information. Such frameworks have the advantage of encapsulating a number of security solutions in a single design.

3.7.1 Distributed Object-based Applications

Hays et al. (2000) present perhaps the most complete set of security solutions for distributed object-based applications in the form of the Object Filter and Access Control Framework (henceforth OFAC framework). The OFAC framework employs a number of the patterns discussed previously, including Bodyguard (Neves and Garrido, 1998) for access control, Authenticator (Brown et al., 1999) for authentication and Data Filter Architecture (in simplified form – Flanders and Fernandez, 1999) for object filtering. The central participant is a Coordinator object – reminiscent of an Object Request Broker (Buschmann et al., 1996) – whose purpose is to mediate control between client objects and other objects. Requests to protected objects induce a chain of object creations, including filter policy applicator objects and bodyguard objects to check the validity of the request. Filtering of objects in this context most likely applies only to data objects. An interesting feature of the framework is the use of a Connection/ConcreteConnection participant, an adaptation of the RPC connection object in (Heuser and Fernandez, 1999), which encapsulates network protocol and connection details. In many ways the OFAC framework resembles a whole security architecture for distributed object middleware frameworks that do not support the provided security properties inherently.

3.7.2 Middleware Frameworks

Lu and Weiss (2008) outline a pattern language for grid security mined predominantly from the Globus Toolkit's (Foster, 2005) security infrastructure, GSI. The language applies equally well to general distributed settings. Taken together, the central security concerns addressed by the patterns are: mutual authentication using PKI (Mutual Authentication pattern), with single-sign on semantics (Single Sign-On); authorization (Gridmap Authorization pattern) and authority delegation (Credential Delegation pattern). Other patterns necessary to complement the language are Authorization, Secure Channels (Schumacher et al., 2006) and Credential (Morrison and Fernandez, 2006a). It should be remarked that not all the pattern descriptions in the language are valid software patterns. In particular, Short Lifespan and Credential Renewal do not result in any architectural design decisions by themselves, and are properly part of the Credential Delegation pattern (as it appears in Weiss, 2006). In fact, Short Lifespan is the result of refactoring the (original) Credential Delegation pattern. The Mutual Authentication pattern describes user-host authentication (Woo and Lam, 1992) in the context of a Public Key Infrastructure based system. Secure channels are used to transmit the relevant credentials required by each principal to verify the other's identity. The credentials (certificates) are used to store the relevant system rights and identification information, and can be delegated in the form of a proxy certificate using the Credential Delegation pattern (Weiss, 2006). Authorization is achieved via a distributed information solution in which the rights of a user are distributed to each participating node in the form of a "gridmap" file. The user is mapped onto a local system entity under the local security policies. Checks occur at resource provision service components residing on each node (see, for example, De Camargo et al., 2004) by determining whether a user is present in the gridmap file. Under this design, global changes to user rights must be propagated in some fashion (supposedly also initially), but the mechanism for this is not discussed in the pattern.

3.8 Distribution Control

Software distribution necessarily implies some form of networked process collaboration (or distribution control).
This functional concern is often designed (knowingly or otherwise) using software patterns for distribution (see Buschmann et al., 2007a; Schmidt et al., 2000), which can be secured \textit{a priori}.

### 3.8.1 Middleware Frameworks

Karp and Smathers (2003) informally describe three security patterns which were mined from the HP e-speak open-source collaboration system. Although they do not use a specific pattern template, their descriptions do indeed result in architectural decisions and/or partial architectures, and therefore are genuine software patterns. Nevertheless, the authors only hint at the precise security functionality and do not provide any models for the patterns (e.g. UML diagrams), making it difficult to determine their full scope and, to some extent, full value. The first of these patterns deals with authorization, however, its insistence on disregarding identity information encourages identity spoofing and thereby makes the pattern unsuitable for distributed, untrusted environments. The second pattern described is a minimized version of the Broker pattern (Buschmann et al., 1996) and is concerned with mediating control between client and server invocations, although the (security) details are left undetermined. In the latter case the pattern's functionality resembles the Mediator pattern (Gamma et al., 1995) or the Client-Dispatcher-Server pattern (Buschmann et al, 1996). The idea of a separate mediator between clients and servers could also be realized as a Proxy (Gamma et al., 1995, Buschmann et al., 1996), a variant of which forms the final pattern presented by Karp and Smathers. The latter proxy does not, however, add any further functionality, security or otherwise, to the Protection Proxy variant of the Proxy pattern in the latter references, and can therefore be considered identical with them. The Bodyguard pattern (Neves and Garrido, 1998) is closely related and could be substituted in this case also.

In contrast to mining patterns from actual software systems, Morrison and Fernandez (2006b) propose to secure existing patterns relevant in or tailored for a distributed context to produce a corresponding security-augmented version. In particular, Morrison and Fernandez (2006b) secure the Broker pattern (Buschmann et al., 1996) to create a Secure Broker. This is accomplished by considering attacks to the non-secure Broker, and, once determined, considering the security measures used in actual implementations (i.e. real systems) using the Broker pattern. The latter implementations are CORBA and Microsoft’s .NET framework. The security measures are then mapped back to the attack model and the relevant security functionality is added to the Broker pattern accordingly. This consists in augmenting Broker with a Reference Monitor (Schumacher et al., 2006) to check authorization rules and ensuring that every entity in the system (client, broker and server) possesses some form of identity. Secure Channels (Schumacher et al., 2006) is used to ensure message confidentiality and integrity.

### 3.8.2 Component Middleware Frameworks

The latter approach of Morrison and Fernandez (2006b) is further applied by Fernandez and Larrondo-Petrie (2007) in the wider context of distributed component middleware. The essence of the approach remains the same, namely, to secure functional patterns by augmenting them with security patterns and additional modifications required for enhanced security functionality. Security is considered in several layers, loosely in line with the methodology developed in (Fernandez, 2004a; Fernandez et al., 2006a), discussed in Section 79. A number of patterns are proposed in their secure versions – Secure Proxy, Secure Client-Dispatcher-Server, Secure Facade etc. – with attendant suggestions for what security functionality they may contain; however, no detailed explicit descriptions are provided. Some of the latter patterns have appeared elsewhere with full descriptions – Secure Broker (Morrison and Fernandez, 2006b) (discussed above) and Secure Adapter (Fernandez and Ortega-Arjona, 2009b) are two examples. Most notable in this work in comparison to (Morrison and Fernandez, 2006b) is that whole systems of patterns are secured by the augmentation of security patterns, not simply individual non-secure instances. This suggest a more comprehensive approach to constructing flexible security architectures for distributed middleware systems.

### 3.9 Processing

Besides networked process collaboration, software distribution implies the processing of data either sequentially or in parallel via network messaging. As for distribution control, when designing this functional aspect it can be useful to employ solutions which have security properties incorporated \textit{a priori}.

### 3.9.1 All domains

Continuing with the approach begun in Morrison and Fernandez (2006b) and extended in Fernandez and Larrondo-Petrie (2007), Fernandez et al. (2008a) describe a Secure Three-Tier Architecture pattern, with the purpose of enlarging the catalog of secure versions of existing architecture-level software patterns for distributed contexts. The secure Three-Tier Architecture is the result of applying a set of security patterns to the original three-tier distribution pattern to enforce global security properties, entailing a secure presentation tier (with authentication, authorization and digital signatures), secure business tier (with global authorization and enforcement mechanisms) and secure data tier (likewise with authorization), with the precise level of security introduced (i.e. precisely which security patterns are applied) left to the system designers.

Further patterns expanding the latter catalog are the Secure Pipes and Filters and Secure Blackboard patterns by Ortega-Arjona and Fernandez (2008) and Fernandez and Ortega-Arjona (2009a) respectively. These patterns are
likewise secure versions of the original patterns – Blackboard and Pipes and Filters (Buschmann et al., 1996) as well as Shared Resource (Ortega-Arjona, 2003) and Parallel Pipes and Filters (Ortega-Arjona, 2005) – however, the actual emphasis is on distributed realizations of these patterns in which communications also require security, whether used in a parallel context or not. The distribution of the security components suggested in the Secure Pipes and Filters pattern, for example, depend on the distribution architecture of the system, making for a more representative solution. The secure Pipes and Filters pattern instantiates security patterns from (Schumacher et al., 2006) to introduce authentication, authorization, logging and communication security at each stage of processing (i.e. at each filter) and communication (i.e. at each pipe) to ensure that both users and automated processes can perform only allowed, traceable operations at each step. The Secure Blackboard pattern instantiates the same security patterns to enforce the same security properties, except in this case at the knowledge sources and control components of the Blackboard pattern.

3.10 Monitoring

Run-time security monitoring of communications and/or particular behaviour is useful in all software systems, but takes on special significance for networked systems (and hence for distributed systems more specifically) in light of the possibility of mounting distributed attacks targeting particular nodes.

3.10.1 All domains

Fernandez and Kumar (2005) present a pattern for knowledge-based (and more specifically signature-based) intrusion detection. Since developing a custom IDS is an extremely complex task requiring a project of its own, the pattern's main value lies in indicating an appropriate solution in the design, e.g. where relevant components can be placed, and/or to denote the need to use COTS products in later development stages.

4 Evaluation of Surveyed Patterns

In this section we provide a rigorous classification of the patterns presented in Section 67, evaluate their quality, and identify classes of new patterns.

4.1 Classification

Classification schemes are a useful tool in helping to navigate pattern catalogs and in determining patterns for different domains that require development. Below we summarize the patterns in this survey in the form of a modified Dimension Table (Washizaki et al., 2009), which in turn is based on the 6-dimensional classification scheme of VanHilst et al. (2009). Our adaptation removes the life-cycle stage, code source and constraint dimensions from the latter schemes, and adds several new dimensions that determine the type of security pattern (strategic or tactical, see Section 64), whether the pattern accounts for global or system-wide concerns and the degree of centralization of the solution. To condense the visual presentation, we have denoted the values of the additional dimensions (where applicable) using letters in brackets next to the pattern name. These include:

- whether the pattern is strategic, denoted by 'S', or tactical, denoted by 'T';
- whether the pattern accounts for global (vs. local) effects, denoted by 'g' if it does; and
- whether the solution is centralized, denoted by 'c', or distributed, denoted by 'd' (no letter indicates either).

The concerns dimension of Washizaki et al. (2009) is used to group patterns vertically. The architectural level dimension present in the aforementioned schemes has been split into three categories corresponding to three main layers of the software stack at which a pattern can be applied:

- application (which includes business logic and data sub-layers);
- integration or system infrastructure (which includes distribution, communication/networking, and any middleware functionality), and
- system (which includes any local OS functionality).

In our quality evaluation, we adapt the quality categories of Laverdiere et al. (2006) as follows:

- Over-specification (denoted by 'O'): a pattern is over-specified when its description is overly fine-grained, containing “more details and properties than needed” (Laverdiere et al., 2006), or encompasses multiple, complex solutions.
- Under-specification (denoted by 'U'): the converse of Over-specification, a pattern is under-specified when its description is incomplete or lacking necessary detail for application, is very high-level, or lacks structure (e.g. the use of a standard textual template).
- Lack of generality (denoted by 'G'): a pattern lacks generality if it applies to a highly specific setting or is
the outcome of a one-time solution or a solution for a specific situation and/or system. Such patterns are difficult to use in multiple contexts.

- Lack of precision (denoted by 'P'): occurs when a pattern's description does not solve a specific, well-defined problem or provides a solution that is unclear or vague.
- Misrepresentation (denoted by 'M'): occurs when a pattern provides a solution different from its name or intent, or portends to apply in wider or narrower contexts to which in fact it does not.

We do not believe Laverdiere et al.'s “lack of consensus” category is a good measure of quality, since pattern authors cannot control the work of others and hence cannot be at fault if other, inconsistent patterns of lesser quality but solving the same problem are published.

Lowercase letters equivalent to those of a particular category (e.g. 'u' instead of 'U' for under-specification) imply that a pattern falls into the category to a lesser degree. The letters are attached to each pattern (in the last column of the table) as quality indicators. Patterns with no quality indicators in the table were found to reasonably satisfy the quality criteria set out above.

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Reference</th>
<th>Architectural Level</th>
<th>Response Type</th>
<th>Domain</th>
<th>Quality Indicators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Security Contexts</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Credential (S,g,d)</td>
<td>Morrison and Fernandez (2006a)</td>
<td>All</td>
<td>All</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>Credential Delegation (S,g,d)</td>
<td>Weiss (2006)</td>
<td>All</td>
<td>All</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>Circle of Trust (S,g,d)</td>
<td>Delessy et al. (2007a)</td>
<td>All</td>
<td>All</td>
<td>All domains</td>
<td>U</td>
</tr>
<tr>
<td><strong>Identity Management</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identity Provider (S,g,c)</td>
<td>Delessy et al. (2007a)</td>
<td>Integration</td>
<td>All</td>
<td>All domains</td>
<td>U</td>
</tr>
<tr>
<td>Identity Federation (S,g,d)</td>
<td>Delessy et al. (2007a)</td>
<td>Integration</td>
<td>All</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td><strong>Authentication</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authenticator (T, d)</td>
<td>Brown et al. (1999)</td>
<td>Application</td>
<td>Prevention</td>
<td>Distributed Object-based Applications</td>
<td>G</td>
</tr>
<tr>
<td>Remote Authenticator/Authorizer (T,c)</td>
<td>Fernandez and Warrier (2003)</td>
<td>Application, Integration</td>
<td>Prevention</td>
<td>General Networked Applications</td>
<td></td>
</tr>
<tr>
<td>Direct Authentication (T, d)</td>
<td>Microsoft (2006)</td>
<td>Application, Integration</td>
<td>Prevention</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>Brokered Authentication (T, c)</td>
<td>Microsoft (2006)</td>
<td>Integration</td>
<td>Prevention</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>Mutual Authentication (T, d)</td>
<td>Lu and Weiss (2008)</td>
<td>Application, Integration</td>
<td>Prevention</td>
<td>All domains</td>
<td>U</td>
</tr>
<tr>
<td>Single Sign-On (S,g)</td>
<td>Lu and Weiss (2008)</td>
<td>Integration</td>
<td>All</td>
<td>Middleware Frameworks</td>
<td>U</td>
</tr>
<tr>
<td><strong>Authorization</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MBAC/ABAC (S)</td>
<td>Priebe et al. (2004)</td>
<td>All</td>
<td>Prevention</td>
<td>General Networked Applications</td>
<td></td>
</tr>
<tr>
<td>Bodyguard (T, d)</td>
<td>Neves and Garrido (1998)</td>
<td>Application</td>
<td>Prevention</td>
<td>Distributed Object-based Applications</td>
<td>g</td>
</tr>
<tr>
<td>Feature</td>
<td>Authors</td>
<td>Categories</td>
<td>Type</td>
<td>Frameworks</td>
<td>Domain</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------------------</td>
<td>------------</td>
<td>-------</td>
<td>-------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>Invocation Interceptor (S)</td>
<td>Voelter et al. (2005)</td>
<td>Integration</td>
<td>Prevention</td>
<td>Distributed Object Middleware Frameworks</td>
<td></td>
</tr>
<tr>
<td>ACL (T)</td>
<td>Delessy et al. (2007b)</td>
<td>All</td>
<td>Prevention</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>Capability (T, d)</td>
<td>Delessy et al. (2007b)</td>
<td>All</td>
<td>Prevention</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>PBAC (S)</td>
<td>Delessy et al. (2007b)</td>
<td>All</td>
<td>Prevention</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>Application Firewall (T)</td>
<td>Delessy-Gassant et al. (2004)</td>
<td>Integration, System</td>
<td>Prevention</td>
<td>General Networked Applications</td>
<td></td>
</tr>
</tbody>
</table>

**Filtering**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Authors</th>
<th>Categories</th>
<th>Type</th>
<th>Frameworks</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firewall pattern language (T)</td>
<td>Delessy-Gassant et al. (2004); Fernandez et al. (2003); Fernandez (2004b)</td>
<td>Integration, System</td>
<td>Prevention, Detection</td>
<td>All domains</td>
<td></td>
</tr>
</tbody>
</table>

**Distribution Control**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Authors</th>
<th>Categories</th>
<th>Type</th>
<th>Frameworks</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Broker (S)</td>
<td>Morrison and Fernandez (2006b)</td>
<td>Integration</td>
<td>Prevention</td>
<td>Component Middleware Frameworks</td>
<td></td>
</tr>
</tbody>
</table>

**Processing**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Authors</th>
<th>Categories</th>
<th>Type</th>
<th>Frameworks</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Pipes and Filters (S)</td>
<td>Fernandez and Ortega-Arjona (2009a)</td>
<td>Application, Integration</td>
<td>Prevention</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>Secure Blackboard (S)</td>
<td>Ortega-Arjona and Fernandez (2008)</td>
<td>Application, Integration</td>
<td>Prevention</td>
<td>All domains</td>
<td></td>
</tr>
</tbody>
</table>

**Communication**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Authors</th>
<th>Categories</th>
<th>Type</th>
<th>Frameworks</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Channel (S)</td>
<td>Schumacher et al. (2006) and others</td>
<td>All</td>
<td>Prevention</td>
<td>All domains</td>
<td>O, G, P, M</td>
</tr>
<tr>
<td>Symmetric Encryption (T)</td>
<td>Hashizume and Fernandez (2009)</td>
<td>All</td>
<td>Prevention</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>Digital Signature with Hashing (T)</td>
<td>Hashizume et al. (2009)</td>
<td>All</td>
<td>Prevention</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>Secure Client/Server Communication (T)</td>
<td>De Souza and Matwin (1999)</td>
<td>All</td>
<td>Prevention</td>
<td>All domains</td>
<td></td>
</tr>
<tr>
<td>Tropyc pattern language (T)</td>
<td>Braga et al. (1998)</td>
<td>All</td>
<td>Prevention</td>
<td>All domains</td>
<td>O, U</td>
</tr>
<tr>
<td>Key Management pattern language (S)</td>
<td>Lehtonen and Parssinen (2001)</td>
<td>All</td>
<td>Prevention</td>
<td>All domains</td>
<td>U, P, M</td>
</tr>
</tbody>
</table>
4.2 Analysis and Discussion

A number of patterns requiring development can be identified based on Table 1 above, which we discuss below according to pattern quality factors, additional concerns requiring coverage and levels of abstraction. The classes of patterns identified can be realized either as individual, stand-alone patterns or as whole pattern languages.

4.2.1 Quality

Approximately half of the patterns evaluated were assigned negative quality indicators, indicating that a significant number of relevant patterns are, in fact, not entirely usable in practical scenarios. The results of this evaluation are in accord with previous studies mentioned in Section 65, which found that a significant portion of published patterns fail to meet requisite quality criteria. The implications are that a number of patterns relevant in a distributed setting require suitable replacements. Most notable among these are the patterns for secure network communications (collectively “Secure Communication” in Table 1), which were found to be variously over-specified (Blakely and Heath, 2004), highly specific to web-based contexts (Steel et al., 2005 and Schumacher et al., 2006), sometimes vague and, as a result, misleading in their intent. Lack of templates (Karp and Smathers, 2003), implementation details (Lu and Weiss, 2008, as well as Delessy et al., 2007a), and encapsulation of system-specific solutions (Brown et al., 1999 and Flanders and Fernandez, 1999) led to other patterns possessing less applicability than would otherwise be possible. The Tropyc pattern language was both over and under-specified, in that individual patterns do not contain sufficient detail for application, however, the language contains a large number of redundant patterns resulting from combinations of smaller (under-specified) patterns.

4.2.2 Concerns covered

It is possible to infer from the context domains present in Table 1 above that not all aspects of distributed computing are currently covered by existing patterns, including group communication, collaborations across multiple domains and distributed data management. Correspondingly with these aspects, patterns that can be developed for secure group communication (see, e.g. Zou et al., 2005), collaborative authorization models (see, e.g. Tolone et al., 2005) as well as secure (distributed) database access and transactions.

Besides those aspects of security specific for distributed systems already covered by existing patterns, there are several (security) aspects that have received little attention thus far. These include secure message routing; distributed credential management, key management and more generally security information management; authentication in mutually-suspicious settings; and more specialized authorization models. The latter aspects invariably represent promising candidates for security patterns or pattern languages.

4.2.3 Levels of abstraction

The patterns surveyed can often be applied to different levels of abstraction as befits the situation or context. Despite this apparent flexibility, there is often no clear indication of precisely how the relevant patterns can be applied, what protocols are required, or what technologies can be used. Such instantiation-dependent information is lacking above all in patterns that are underspecified, but also in patterns that are well-specified according to generally published standards. This point suggests the need for a hierarchy of patterns covering different levels of abstraction, something begun in the work on patterns of Rosado et al. (2006) and in (Microsoft, 2006). Hierarchies of patterns would extend both existing security patterns and those suggested as possibilities above.

4.2.4 Summary

The broad classes of patterns proposed for development are summarized below in Table 2, according to the concerns introduced earlier in the classification (see Table 1). The patterns surveyed in Section 67 without (negative) quality indicators, together with our proposed classes of new patterns or pattern languages below, can be seen as forming
the foundations of a security pattern language for distributed computing. Currently this language is in its infancy, requiring – as with other pattern languages for a given domain – the exact determination and elaboration of the most apposite patterns, making our proposals concrete, as well as the development of pattern relationships both between the patterns in the language and with security patterns in more general settings, e.g. from Schumacher et al. (2006) (see Section 83 for future work).

<table>
<thead>
<tr>
<th>Security concerns</th>
<th>Pattern classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization</td>
<td>Collaborative authorization models</td>
</tr>
<tr>
<td>Authentication</td>
<td>Mutual authentication with suspicion</td>
</tr>
<tr>
<td>Security Information Management</td>
<td>Credential management</td>
</tr>
<tr>
<td></td>
<td>Key management</td>
</tr>
<tr>
<td>Communication</td>
<td>Secure (network) communication</td>
</tr>
<tr>
<td></td>
<td>Secure group communication</td>
</tr>
<tr>
<td></td>
<td>Secure message routing</td>
</tr>
<tr>
<td>Data management</td>
<td>Secure distributed database control</td>
</tr>
<tr>
<td></td>
<td>Secure distributed transactions</td>
</tr>
<tr>
<td>Distribution control</td>
<td>Process/task serialization</td>
</tr>
</tbody>
</table>

Table 2: Security patterns requiring development

5 Methodologies for Applying Security Patterns

5.1 Introduction and Classification

Applying individual patterns or their composites to secure a distributed system is certainly a step forward in secure system design. The patterns presented in previous sections as part of our pattern survey (Sections 64 – 74) are, for the most part, abstractions of actual solutions used in real-life systems, with all the attendant benefits. It is perfectly possible for a system designer to glance at our table in Section 4, navigate to a particular pattern for a given context and domain and instantiate it in a system's design. Applying security patterns in this ad-hoc manner, however, is unlikely to be sufficient in tackling the complexity inherent in securing a whole system with all its facets (Fernandez et al., 2006a). It is when security patterns are used as part of an overall software engineering approach or methodology that their greatest merits can become manifest, since in that case the use and application of security patterns is guided by sound engineering principles.

In what follows, we briefly survey pattern-based methodologies with a focus on pattern application strategies, grouping them according to their maturity and/or scale. The measure of this classifier is in part subjective, and our criterion is that if a methodology attempts to cover all facets of security as well as pattern application, and/or has been developed over the span of several years, we can consider the methodology mature and/or larger-scale respectively. Young and/or smaller-scale methodologies are those that fail to satisfy the above criterion.

As for the patterns surveyed in Section 67, the research process for finding methodologies was to exhaustively search the open literature, considering relevant conference and journal papers, books, and various technical reports. The collection of all security methodologies found was subsequently reduced to those that are based on security patterns to form the final selection of approaches.

There are a number of methodologies using some form of pattern-like artefacts that are excluded from this brief survey, according to the definition of security patterns (Section 64). These include the work of Hafner and Breu (2009) and Memon (2011) – SECTET – as well as of Georg et al. (2002, 2009), which use parameterized model templates and aspects respectively; the work of Schmidt et al. (2011) – SEPP – which uses requirements-level patterns as well as pattern-like, generic security components and architectures; and the SERENITY project (Spanoudakis et al., 2009), which uses pre-defined specifications of various levels of abstraction as security and dependability (S&D) patterns. These approaches and others are surveyed in detail in (Uzunov et al., submitted).

5.2 Survey

In this section we survey our selection of pattern-based security methodologies, beginning first with mature and/or larger-scale methodologies (Section 79) and proceeding to young and/or smaller-scale methodologies afterwards (Section 80).
5.2.1 Mature and/or larger-scale Methodologies

5.2.1.1 Secure Unified Process

Steel et al. (2005) present a secure version of the (Rational) Unified Process (UP) using security patterns, applicable to systems using J2EE technologies. Additions to UP include activities for determining security requirements; for constructing a secure design; for black and white box testing and security analysis; and for post-deployment auditing. Pattern application occurs at the design stage, along with security analysis tasks such as threat modeling and policy development. Appropriate patterns are chosen from a catalog (some of which have been surveyed in Section 67) to satisfy the security requirements determined in earlier stages and mitigate architecturally significant risks. The catalog of security patterns is structured according to the four J2EE tiers: web (user interaction); business (application/business logic); web-services (infrastructure integration) and identity (identity management functionality). Secure UP, unlike the related SODA and ISDF, provides not only high-level advice on introducing security into a system's design, but a number of concrete activities and a structured approach to selecting patterns, albeit for a very specific software environment.

5.2.1.2 Secure Tropos

Secure Tropos (Mouratidis et al., 2005a; Mouratidis, 2011) is an extension of the Tropos agent-oriented development methodology (Bresciani et al., 2004), with the (recent) addition of UMLsec (Jürjens, 2005) for detailed design. Tropos considers five development stages: early requirements, late requirements, architecture development, detailed design and implementation. During early and late requirements, a set of models are developed that capture security requirements using modeling notions such as actors, security goals, tasks and actor dependencies. Secure Tropos employs security patterns during the architecture development stage with the aim of helping to map the conceptual models created earlier to known, predictable sets of design elements and thereby reduce the risks associated with ad-hoc realizations of the security requirements. The patterns used (not covered in this survey) are agent-specific (see Mouratidis et al., 2003) and, despite their usefulness in the context of the Tropos approach, address only a limited range of security concerns. The requirements for agent-specific patterns implies that standard collections of security patterns (e.g. Schumacher et al., 2006) are not applicable and thus cannot be used.

5.2.1.3 Fernandez and colleagues

Fernandez (2004) and later Fernandez et al. (2006a) present a comprehensive methodology using security patterns throughout the whole SDLC within the object-oriented paradigm. One of the main ideas in the latter methodology is that systems are structured into hierarchical layers of abstraction (Fernandez and France, 1995; Fernandez and Nair, 1998; Fernandez, 1999, 2003) each of which must be secured. Security patterns can be fitted into any of these layers, starting from the most abstract, and, considering them as constraints, mapping them through the other layers (Fernandez, 2003). Security pattern application starts from the very beginning of the SDLC: use cases are defined which already carry security information helping to determine an overall access control model – defined as a pattern in Fernandez and Pan (2001) – as well as attack scenarios, authentication and logging needs, and non-repudiation measures. These use cases are carried over into the analysis stage, where analysis patterns (Blaimer et al., 2010; Fernandez and Yuan, 2000) can be used to create a domain model efficiently. Authorizations are also determined at the analysis stage and incorporated into the conceptual model. The latter provides a means to consider security threats, which, when carried over into the design stage, can instigate countermeasures as security patterns through the different layers of the architecture as described previously. Distribution is considered to be (a part of) one of these layers (more specifically, the communication layer), and a number of the patterns surveyed in Section 67 are applicable in this context. Security can be tested at every stage by checking for the absence or presence of definite security patterns, alignment with security requirements and whether certain attacks are mitigated. Application of security patterns in Fernandez et al.'s (2006a) methodology thus covers both the vertical (architectural layers) and horizontal (development process) dimensions of a software system. The methodology of Fernandez and colleagues is the only one to consider security across the whole software stack, however, tool support is currently lacking, making the full realization of the methodology's vision more difficult.

5.2.1.4 PWSSec

Gutierrez et al.'s (2006, 2009) PWSSec is a methodology for securing Web-services in which security patterns play a central role. The development process is divided into three stages: the determination of security requirements; the creation of a system's architecture together with appropriate selection of security patterns to satisfy the security requirements; and mapping of the resulting security solutions to Web-services technologies. During design, security patterns are selected and refined according to two levels of abstraction (Rosado et al., 2006), the first consisting of architectural security patterns, which cover more abstract security concerns affecting larger parts of a system's architecture; and the second consisting of design security patterns, which refine the latter by including specific implementation-related aspects. Design patterns can be mapped directly onto Web-services standards and technologies, which can be seen as a final refinement stage for a particular security solution. Instantiated security
patterns give rise to collections of (abstract) Web-service security components, which can be elaborated further until they are mapped to actual software components. The security components are organized in PWSSec into separate domains (zones) with centralized management components (security kernels), in conjunction with policy specifications.

In spite of the generality offered by the use of security patterns, Gutierrez and colleagues only concern themselves with communications security via the “QoP” (quality of protection) pattern system of Rosado et al. (2006). PWSSec main drawbacks are its specificity and the high development effort required for application, demonstrated by Gutierrez et al. ’s (2009) case study, where the various activities generated over 200 detailed artefacts for a Web-service application with a single use case.

5.2.2 Young and/or smaller-scale Methodologies

5.2.2.1 SODA

Meland and Jensen's (2008) SODA is a security methodology “for the developer-on-the-street”, in which patterns combined with general security guidelines and principles are presented to a software developer via a user interface to guide them through the process of securing a system design. SODA follows a security life-cycle model akin to Microsoft's SDL (Howard and Lipner, 2006) with activities including the application of patterns and principles to create a secure design as well as performing design-level threat modeling and security reviews. SODA is an example of a CASE tool-support reliant methodology: a Web tool (SODAWeb) helps software developers select patterns and guidelines at any given phase. The usefulness of the methodologies is thus reliant on the usability and quality of the tool.

5.2.2.2 ISDF

Like SODA, Alkussayer and Allen's (2010) ISDF promotes the use of security patterns alongside best practices in an integrated fashion throughout the whole software development life-cycle. The structure of the software process itself is, like SODA, based on Microsoft's SDL. Although Alkussayer and Allen define pattern application at certain definite points in the life-cycle, descriptions remain at an extremely high-level with insufficient guidance for developers and architects.

5.2.2.3 Scandariato and colleagues

In comparison to SODA and ISDF, Scandariato et al. (2008) take a more fine-grained approach in defining a security pattern-based methodology based on their previous work of cataloging and refining available security patterns in the open literature (Heyman et al., 2007; Yskout et al., 2006, 2008). The latter catalog forms a pattern inventory, which is further structured according to a decomposition of security objectives and according to a separation of design phases of a system (application architecture, application design and system-level). A system's design is secured by choosing appropriate patterns from the structured inventory at each of the aforementioned development phases corresponding to different security objectives.

One of the most salient feature of the methodology is the consideration of pattern relationships, including dependencies and conflicts. These relationships are integrated into the security pattern inventory, allowing for a structured selection of patterns to cover the necessary security objectives defined at the outset. Scandariato and colleagues also consider trade-offs to other quality attributes in each pattern description, albeit in a relatively simple fashion with positive and negative qualifiers (labels). The methodology is formalized using set theory, which does not add significant value beyond the merits it already possesses.

5.2.2.4 Schumacher

A formal approach is used to good effect by Schumacher (2003), who proposes a theoretical model of security patterns and an accompanying methodology to pattern application, realized by formalizing pattern descriptions with F-logic and employing a corresponding CASE tool for search and retrieve capabilities. Schumacher also proposes security enforcement throughout the whole SDLC (without entering into detail), as well as pattern relationships.

5.2.2.5 Horvath and Dörges

Horvath and Dörges (2008) consider formalizing security patterns with Petri nets (specifically, with Reference nets) and applying the principles of Model-Driven Security (see, for example, Fernandez-Medina et al., 2009) in the context of multi-agent systems. Essentially, in Horvath and Dörges’ proposal security patterns are modeled using high-level Petri net formalisms and transformed to “runnable nets” using continuous refinement. Here runnable nets imply executable dynamic models, rather than implementation code, although the idea could be extrapolated via tool support. The refinement process consists in asking and answering particular questions, thereby detailing the actual Petri net model (or rather, what is not contained in that model). While Petri nets provide a mathematically rigorous and well-founded language for modeling concurrent and distributed systems, using the technique for modeling
security patterns poses limitations, e.g. a number of security patterns leading to design decisions, especially at different levels of abstraction, cannot be modeled using this approach. Nevertheless, the approach is unique for its applicability to systems in which the modeling language used is based on Petri nets.

5.2.2.6 Robinson

Robinson (2007) proposes a framework for reasoning about and introducing security using a formalized variant of security patterns. The formalisms used are non-standard, and the presented patterns take into account only a small number of SOA relevant scenarios (such as communication and service binding).

5.2.2.7 Schnjakin and colleagues

SOA security using patterns is also considered by Schnjakin et al. (2009) via the use of a CASE tool – a security advisor. The advisor helps system architects with the selection of appropriate security patterns, which are transformed to security configurations using Model-Driven Security principles. The central use of a tool helping developers select patterns at first sight bears resemblance to SODA, however, Schnjakin et al.’s tool does not aim to offer developer-friendly guidance in selecting patterns; rather, the aim of the advisor is to map security goals to semi-formalized security patterns (inspired by Schumacher's theoretical model), which can subsequently be transformed to security policy configurations of security modules in an underlying SOA (Web-services) platform. The approach requires developers to choose certain security goals, which are mapped to security patterns by the advisor and refined to more concrete “architectural patterns” (specifying solutions) and finally to “implementation patterns”, either automatically or with user input. The tool with its set of patterns and possible security configurations must be determined by experts prior to developers using the advisor, in a pre-configuration phase.

5.2.2.8 Blakely and Heath (The Open Group)

A simplified approach to pattern application is presented by Blakely and Heath (2004) as part of The Open Group's initiative for security patterns. The approach is entirely based on The Open Group's catalog of patterns (Blakely and Heath, 2004), and proceeds by determining security threats in an ad-hoc fashion or using some custom (unspecified) approach; and selecting patterns from the Available System and Protected System collections for system reliability and protection respectively. This ties the approach to a specific set of solutions, some of which possess negative quality indicators according to our survey in Section 67.

5.2.2.9 Shiroma and colleagues

Shiroma et al. (2010) present an approach to security pattern application within the context of Model-Driven Architecture (Frankel, 2003). Security patterns are first modeled using ATL and applied to a design via model transformations. Like approaches based on formal methods, the ATL models can only capture one instance of a pattern's representative partial architecture. The main emphasis of Shiroma et al.’s work is on reducing developer effort and hence saving time, however, this does not appear to be commensurate with the corresponding loss of flexibility introduced by the strict modeling requirements.

5.2.2.10 Delessy and Fernandez

Delessy and Fernandez (2008) present a methodology for securing SOA-based systems using security patterns by adapting and building on the work of Fernandez and colleagues. The approach is based on two central ideas: the use of pattern maps to guide the selection of security patterns; and the application of MDA principles to SOA-specific models. Pattern maps are graphs of (security) patterns classified by a security goal and divided into different layers of abstraction, with various relationships between the patterns. In Delessy and Fernandez's methodology, the pattern map consists of two layers, an abstract layer containing patterns with general SOA applicability, and a concrete layer, containing patterns for web-service security. The process of introducing security follows the approach of Fernandez and colleagues until the design stage, when an SOA-enabled metamodel (in the MDA terminology) is used to create appropriate design models into which security patterns are introduced using the pattern map. Model transformations are used to transform design models from the SOA level to the Web-service implementation level. The methodology currently lacks tool support, which is essential if model transformations are to be fully practical.

5.3 Evaluation of Methodologies and Discussion

In this sub-section we evaluate the appropriateness of the surveyed methodologies for use with distributed systems according to five categories, listed below.

- **Specificity**: The specificity of a methodology determines the scope of systems to which it can be applied, and ranges from highly specific to a particular distributed system type, to completely generic and applicable to all distributed systems. A generic methodology may be further tailorable to or provide support for more specific system types.
• **Provisions for distribution (general):** a methodology may make general provisions for distributed systems, such as as special abstractions or activities addressing (general) distribution-specific concerns.

• **Provisions for distribution (specific):** a methodology may or may not make specific provisions for different aspects of distributed systems, guiding solutions accordingly. These include collaboration between different software entities into account; multiple administrative or security domains; and parallelism. Addressing such concerns may be done via solutions (i.e. relying on particular patterns) and/or specific techniques (e.g. providing guidelines, activities etc.).

• **Integration of specific patterns:** Patterns for distributed settings (such as those surveyed in Section 67, and those proposed for development in Section 74) may or may not be integrated in the approach, being treated separately or in a different fashion from other patterns, or being an essential part of various activities.

• **Coherency of pattern selection:** The selection of patterns may or may not take into account distribution factors (e.g. selecting a pattern for a particular tier). In this respect, patterns may be selected in an ad-hoc fashion, based on security requirements or objectives alone; using a structured catalog or tool for guidance; following various guidelines or rules in an overall approach; or according to pre-defined schemas or conceptual frameworks.

While there is some overlap in the categories, they reflect aspects of a methodology that can make it more appropriate for securing distributed systems of different kinds.

In addition to applying the categories set out above, we have evaluated the methodologies according to five quality indicators listed below, analogous to the indicators used in Section 74 to evaluate patterns. The indicators address deficiencies in a methodology including its support for certain SDLC stages, generality, notations used and use of tools, and help to support decisions regarding whether a methodology is appropriate for practical use.

- **Underspecification** (denoted by 'U'): occurs when a methodology does not provide enough detail to be applied in practice (based on the relevant publications), or provides only high-level, general advice to reach certain goals, but not concrete techniques and/or solutions for achieving those goals. An underspecified methodology may be more difficult to be employed in large, real-life projects.

- **Lack of generality** (denoted by 'G'): occurs when a methodology is tightly restricted to a particular technology or system type, without the scope for being applied more widely (e.g. in situations where a specific technology or system type is used within a larger context).

- **Partial SDLC scope** (denoted by 'P'): occurs when a methodology does not support a sufficient part of the software development life-cycle, namely, at least the requirements analysis and design stages, but only a single stage (e.g. early design or architecture development).

- **Use of non-standard concepts or notations** (denoted by 'N'): occurs when a methodology employs concepts and/or notations that are not widespread or well understood by the majority of developers or IT professionals.

- **Lack of tool support** (denoted by 'T'): occurs when a methodology does not provide sufficient software tools for its application or the full realization of its objectives, when such tools are in fact required or strongly beneficial. Any tools must also be convenient to use.

Usability features, while not considered explicitly above, can be seen as an aggregate factor resulting from a mixture of tool support, standard notations, easy processes etc. Therefore, if a methodology is assessed as having the 'T' and 'N' indicators simultaneously, its usability is not likely to be high.

The results of evaluating the methodologies surveyed in Section 78 are listed in Table 3 below.
The evaluation results suggest that methodologies specific to a particular type of distributed system (e.g., SOA-based) make greatest provisions for distribution, which in all cases is traceable to the highly tailored nature of the approach. Conversely, generic methodologies make least provisions, suggesting that they are generally less capable of addressing a number of issues specific to distributed systems. The exceptions to this are Secure Tropos and the methodology of Fernandez and colleagues, both of which, however, do not consider specific aspects of distribution. Besides a generic methodology with provisions for specific distribution factors, perhaps more conspicuously missing from the table is a methodology that makes provisions for both specific distributed system types and general distributed systems, i.e. a tailorable approach.

Regarding the quality indicators, while all methodologies surveyed were found deficient in one way or another, the assessments should be considered in the context of a given approach. For example, Secure UP is perfectly applicable to projects employing J2EE technologies, but, due to its restricted scope (lack of generality indicator), is not as applicable to other projects. Most young and/or smaller-scale methodologies were found to be underspecified and hence potentially unsuitable for use on large, real-life projects. The quality indicators therefore highlight areas of potential improvement, and suggest that, in addition to introducing features to better support distribution, more significant use of tools, greater detail and, ultimately, maturation are required to increase the usefulness of most pattern-based methodologies in securing distributed systems.

6 Conclusion and Future Directions

In this paper we surveyed the state-of-the-art in securing distributed systems using patterns. In the first part of the survey (Sections 64 – 74), we comprehensively reviewed individual and groups of security patterns specific for or with strong applicability to distributed settings. The patterns were subsequently classified according to an adaptation of (Washizaki et al., 2009) and VanHilst et al.’s (2009) classification schemes and evaluated according to the quality categories proposed by Laverdiere et al. (2006), to help in categorizing patterns more clearly, highlight deficiencies in existing pattern descriptions and determine “missing” patterns. Following the classification, we proposed ideas for new patterns or whole pattern languages based on an analysis of current solutions and aspects of distributed computing requiring coverage.

The patterns considered in this survey, as well as those proposed as a result, can be seen as the foundations of a
security pattern language for distributed computing. Part of our future work in this area will be to consolidate and expand this language in line with our evaluation results. Furthermore the work begun in Fernandez and Larrondo-Petrie (2007) in securing existing patterns for distributed setting is another promising direction in contributing to the latter pattern language.

In the second part of the survey (Sections 78), we reviewed methodologies for applying security patterns, and subsequently evaluated their appropriateness for securing distributed systems. For the exception of highly specific methodologies tailored to, for example, developing Web-services (e.g. PWSsec), Secure Tropos and the work of Fernandez and colleagues, most methodologies reviewed were found lacking in explicit support for distributed systems and hence their particular concerns. Existing methodologies are also not currently tailorable for both generic and specific distributed system types. Our future work in this area will be to investigate the development of such tailorable methodologies for different types of distributed systems, as well as integrating security patterns more tightly within an overall approach.

### References


Frankel DS. Design Patterns: Elements of Reusable Object-Oriented Software. 1st ed. Addison-Wesley Professional; 1995.


Hashizume K, Fernandez EB, Huang S. Digital Signature with Hashing and XML Signature Patterns. In: Procs. of the 14th European Conf. on Pattern Languages of Programs (EuroPLoP 2009); 2009.

85


86


Epilogue

In this chapter, we reviewed the range of currently available security patterns for distributed systems, and also considered the suitability of existing pattern-driven methodologies for developing such system types. We found that security patterns for a number of the core security features of distributed systems are missing in the literature and/or are lacking quality in one or more ways. Regarding pattern-driven methodologies, very few were found to be mature and to make specific provisions for distributed systems. In fact, the latter two findings are strongly related, since the process of developing a general secure distributed system, and a general software system is essentially the same (although, of course, it can be made to vary); therefore, it is not so much the process but the artefacts used that can differentiate a distributed system specific vs. a generic methodology. The same argument can be extended to more specific distributed system types, however, in that case the process may also require specialization. These considerations in particular lead into the next chapter, which is concerned with developing the core of the thesis, namely, an approach to engineering security methodologies. The aforementioned considerations about artefacts will be addressed later, in Chapters 6, 7 and 9, with an altogether new conceptual security artefact type encapsulating and organizing security patterns.

The patterns that were summarized and categorized in this chapter also serve a dual purpose as a base catalogue for the case study methodology of this thesis, presented for the first time in the next chapter.
Chapter 3
(Engineering meta-methodology I)
A comprehensive pattern-oriented approach to engineering security methodologies

Security is a strange field; it is often easier to solve a problem in general than solving concrete instances.
– Dieter Gollmann

Prologue

So far we have reviewed the current state-of-the-art in security methodologies, with emphasis on methodologies for distributed systems; as well as what we considered to be the most promising security solution artefact: security patterns. In this chapter we develop the core of the thesis: a comprehensive approach to engineering security methodologies, in which patterns of a different kind – process patterns – also play an important role. The approach can also be seen as a logical development stemming from the survey of methodologies (Chapter 1 and second half of Chapter 2), and is a direct answer to the question posed in the Introduction: how can ideal methodologies be constructed for particular project situations?

As an illustration of the approach presented, the paper constituting this chapter introduces for the first time the case study methodology used in this thesis: a comprehensive pattern-based methodology for general distributed systems, which is also partially capable of taking into account (through its artefacts, which will be developed in Chapters 4 to 7) the specifics of peer-to-peer systems as well.
# Statement of Authorship

<table>
<thead>
<tr>
<th>Title of Paper</th>
<th>A Comprehensive Pattern-Oriented Approach to Engineering Security Methodologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication Status</td>
<td>O Published, O Accepted for Publication, O Submitted for Publication, O Publication style</td>
</tr>
</tbody>
</table>

## Author Contributions

By signing the Statement of Authorship, each author certifies that their stated contribution to the publication is accurate and that permission is granted for the publication to be included in the candidate's thesis.

<table>
<thead>
<tr>
<th>Name of Principal Author (Candidate)</th>
<th>Anton V. Uzunov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Conceptualization of work, its realization (research) and documentation (wrote manuscript). Acted as corresponding author.</td>
</tr>
<tr>
<td>Signature</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>11/10/2013</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th>Katrina Fallner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Supervised development of work, helped to evaluate and edit the manuscript.</td>
</tr>
<tr>
<td>Signature</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>11/10/15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th>Eduardo B. Fernandez</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Provided ideas and discussed manuscript.</td>
</tr>
<tr>
<td>Signature</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>13/10/2013</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td></td>
</tr>
<tr>
<td>Signature</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td></td>
</tr>
</tbody>
</table>
A Comprehensive Pattern-Oriented Approach to Engineering Security Methodologies

Anton V. Uzunov, Katrina Falkner, Eduardo B. Fernandez

A, School of Computer Science, The University of Adelaide, Adelaide, South Australia, Australia, 5005
b, Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, 777 Glades Rd., Boca Raton, FL 33431

* Corresponding author

E-mail addresses: anton.uzunov@adelaide.edu.au (Anton V. Uzunov), katrina.falkner@adelaide.edu.au (Katrina Falkner), ed@cse.fau.edu (Eduardo B. Fernandez)

Abstract:
Developing trustworthy, secure systems is an issue of ever-growing importance. Researchers have generally come to acknowledge that to develop such systems successfully, their security features must be incorporated in the context of a systematic approach, i.e. a security methodology. There are a number of such methodologies in the literature, but, just as for the development methodologies to which they are related, no single security methodology can be adequate for every situation. Situations in which an off-the-shelf methodology is inappropriate explicitly require an approach for engineering security methodologies, i.e. for constructing “fit-to-purpose” methodologies or for tailoring existing methodologies to the project specifics at hand. While a large body of research exists addressing the same requirement for development methodologies – constituting the field of Method Engineering – there is nothing comparable for security methodologies as such; in fact, the topic has never been studied before in such a context. In this paper we draw inspiration from a number of Method Engineering ideas and fill the latter gap by proposing a comprehensive approach to engineering security methodologies, embodied in three interconnected parts: a framework of interrelated security process patterns and modifiers, forming a basis for engineering the process aspect of a security methodology; a meta-model for engineering a methodology's conceptual security framework aspect; and a meta-methodology to guide engineers in using the latter artefacts in a step-wise fashion. The approach is illustrated and evaluated by tailoring an existing, real-life security methodology to a distributed system specific project situation.

Keywords: secure software engineering; security methodologies; method engineering; process patterns; software security; modeling

1 Introduction

Security in modern software environments is an issue of ever-growing importance, reflected not only in the expanding research literature on the subject, but also in the increase of reported vulnerabilities and their successful exploitation [1, 2, 3]. These advances have meant that, on the technical front, it is no longer feasible to apply isolated security strategies such as cryptography for secure communications, COTS monitoring packages and others, to secure only parts or specific aspects of a system [4, 5]. Researchers have generally come to acknowledge that for any security strategy to be successful it must be employed within a comprehensive, holistic approach firmly grounded in the principles of software engineering, where the incorporation of security attributes begins from the earliest stages and proceeds throughout the whole software development life-cycle (SDLC) afterwards [6, 7, 8, 9, 10, 11, 12].

Such security approaches are necessarilly systematic in nature, i.e. they constitute a “systematic way of doing things in a particular discipline”, which, according to Gonzalez-Perez and Henderson-Sellers [13] is what defines a methodology. Since the discipline combining security and software engineering can be designated secure software engineering [5], we can designate these systematic approaches secure software engineering methodologies, or simply security methodologies [14].

There are a number of such methodologies in the published literature – in fact, there are over 20 applicable to distributed software systems alone (see [14]). Selecting one of these security methodologies for adoption is not an easy task – a number of factors must be carefully analyzed and balanced, including the complexity of a methodology, the expertise required, the level of formality and others, which is to say, a number of project and, more generally, situation-specific factors (cf. [15]). The decision-making process is not made easier by the fact that security methodologies are rarely clear or well structured as befits a good software process from an industry viewpoint (cf. [16]). Even when a particular security methodology has been selected for adoption, certain aspects of it may be inappropriate or may require customization: a “one-size-fits-all” approach, which is increasingly seen as inadequate in the realm of software development methodologies (see, for example, [17, 18, 19]), is just as limiting for security methodologies with respect to project specifics such as target system types or the usage of different technologies (cf. [20]).
Addressing the above issues requires the utilization of a systematic approach to tailor existing security methodologies, to combine different features from different methodologies, or to create new “fit-to-purpose” methodologies to suit the specific needs of an organization and/or particular project – ideally using a common basis which is also suitable for methodology analysis and comparison. In other words, there is a need for a systematic approach to engineer security methodologies – something that does not currently exist, and indeed has never been studied before.

In this paper we draw on a number of ideas from the related field of Method Engineering (ME) – the “engineering discipline to design, construct and adapt” [21] development methodologies for particular projects (see, for example, [22]) or, more specifically, contexts or situations (Situational Method Engineering or SME – see [23, 24, 25]) – to propose a comprehensive approach to engineering security methodologies, which directly addresses the aforementioned need and its corresponding requirements. Our approach is embodied in three interconnected parts (we explain all unfamiliar terms in the body of the paper):

- a framework of interrelated security process patterns and pattern modifiers, forming a basis for engineering the process aspect of a security methodology;
- a meta-model for engineering a methodology’s conceptual security framework aspect;
- and a meta-methodology to guide engineers in using the latter artefacts in a systematic, step-wise fashion.

By applying the meta-methodology, patterns from the framework can be combined or assembled into various workflows and instantiated to construct security methodologies at varying degrees of abstraction and with varying degrees of specified detail. Methodology models provide clarity in methodology descriptions and aid analysis and comparison between methodologies. In the latter case the “model elements” (patterns) offer rich capabilities to (informally) reason about any given security methodology. Such models can also be used to help generalize certain aspects, activities etc. of existing methodologies, which can then be applied by analogy to realize the constituent patterns (simultaneously elements) of a new methodology during construction. Finally, the elements that make up a methodology model can be tailored and replaced by different instances even during methodology enactment (cf. [26]), making for an extremely flexible, modular approach (cf. [27]) to security methodology engineering and re-engineering (cf. [28]).

We evaluate our approach by applying the proposed meta-methodology to tailor an existing, real-life security-pattern-based methodology to a particular project situation, together with basic feature analysis/screening, closely following the evaluation approach used by Hug et al. [68]. The latter approach is based on standard validation techniques for software engineering research [103, 104], and was determined as being most appropriate, as our work effectively resides at a meta-level with respect to its target security methodologies. Our evaluation is further supplemented by a short comparative review of related work, within the confines of approaches for engineering development methodologies and relevant security research (since there are no directly comparable approaches for security methodologies at the time of this writing).

The rest of this paper is structured as follows. In Section 2 we introduce relevant background concepts for security methodologies and provide a high-level overview of our engineering approach. In Section 3 we describe our engineering approach for the process aspect of a security methodology, with the central focus being the framework of security process patterns and modifiers mentioned above. In Section 4 we present our corresponding engineering approach for the conceptual security framework aspect of a methodology. In Section 5 we present a notation for methodology models resulting from the application of the latter artefacts. In Section 6 we present our meta-methodology. In Section 7 we illustrate and evaluate our approach by applying the meta-methodology to tailor an existing, real-life security methodology to a distributed-systems specific project situation. In Section 8 we discuss related work in some detail. Since the engineering of security methodologies has never been studied before, we focus predominantly on similar features of general method engineering approaches based on process patterns. Finally, in Section 9 we conclude and describe future work and important research directions.

2 Background

In this section we provide some necessary background on security methodologies and provide a high-level overview of our approach to engineering such methodologies.

2.1 Security methodologies

A security methodology, as we indicated in the introduction constitutes a “systematic way of doing things in a particular discipline”, where in our case this discipline is secure software engineering. Security methodologies are strongly related to development methodologies1, however, in our work and for the rest of this paper, security

---

1 In this paper, and in our approach more generally, we have chosen to use the term “methodology” as opposed to the more commonly encountered “method”, for the reasons outlined by [42]. We thus refer to the approach presented in this paper as an approach for engineering security methodologies, however, when referring to the encompassing area in general, we employ the established nomenclature – “method engineering” (ME).
methodologies will be considered as separate entities in their own right, distinct from any development methodology to which, or in which, the latter might be related or integrated, respectively. The consequence of this view is that certain concepts that apply to development methodologies lose some or most of their significance for security methodologies, with the most relevant being the process-product dichotomy often cited in the existing method engineering literature (see, e.g., [29, 30]). The reason for this loss of significance is that a security methodology's primary aim is not to provide guidance for producing particular software artefacts, but rather, for improving the security attributes of a given system, i.e., a security methodology is process-centric. This is not to imply that security methodologies cannot prescribe the production of various software artefacts also (security-enhanced architectural models being one example—cf. [31]), but it does imply that certain method engineering approaches and their related notions, such as building a process around a product model, are not as applicable.

Despite their process-centric nature referred to above, security methodologies are not simply “bare” processes in the sense of simply being descriptions of activities—a consideration of existing methodologies (based on, for example, [14]) reveals another inherent dichotomy: that of process-framework, which is to say, a methodology is a process for improving the security attributes of a system via the use of a conceptual framework, consisting of various security improvement artefacts (e.g., security solutions and attendant conceptual artefacts—cf. [32]). A security methodology \( SM \) can thus be defined more formally as a couple:

\[
SM = (SP, CF)
\]

where \( SP \) is a security process—the activities and/or steps taken to secure a software system of some type; and \( CF \) is a conceptual security framework, consisting of the conceptual artefacts used by the methodology's process, the number and type of which, in theory at least, can be arbitrary, but as a minimum must include a set of security solutions, as well as a set of threats leading to those solutions, whether explicitly defined and cataloged or otherwise. Henceforth we will refer to the two constituents of a security methodology as a security process aspect and a conceptual framework aspect, respectively.

### 2.2 Flexible security methodologies

Another important characteristic of security methodologies is their target system applicability, which divides all methodologies into two broad classes (cf. [14]): generic methodologies, which apply to software systems of any type; and specific methodologies, which apply to a certain class of systems, e.g., distributed systems, mobile grid systems and others; or systems using certain technologies, e.g., web-services, distributed objects etc. In both cases developers using a particular methodology are faced with making trade-offs—on one hand, the wide applicability of generic methodologies implies that the challenges raised by particular system types or technologies are not addressed, making them useful “in the large”, but insufficient (in the sense of providing detailed guidance) for more specific development situations; on the other hand, the narrow applicability of a specific security methodology may render their use completely inappropriate in situations where a system is composed of sub-systems of different types or employing different technologies that are not accounted for by the methodology's security process, conceptual framework, or both. We term methodologies capable of being both generic and specific as required—and hence capable of fully addressing development situations pertaining to generic and specific methodologies—flexible security methodologies. An example of such a methodology is a methodology for general distributed systems, which is either tailored inherently or can be tailored dynamically to provide detailed guidance for peer-to-peer systems, grid systems, cloud systems etc., as the need arises.

### 2.3 Towards engineering security methodologies

In this paper our aim is to provide an approach capable of engineering—modeling, analyzing, tailoring and constructing—both “fixed” and flexible security methodologies. From the definition of a security methodology given above (in Section 2.1) it follows naturally that any such approach must take into account both a methodology's process and conceptual framework aspects. It cannot be expected, of course, that the approach taken for each of these aspects will be the same, and such is the case in the work presented in this paper: for the process aspect of a security methodology we employ a (methodology engineering) framework based on process patterns inspired by assembly-based [22, 33] and method configuration [34] techniques; and for the conceptual framework aspect we employ an approach based on meta-model instantiation [13, 30, 35]. The aforementioned method engineering techniques are well documented in the literature in the context of general development methodologies (see, for example, [24]), and we refrain from (essentially) duplicating that content here.

In what follows, we present the main constituents of our approach, first with respect to the security process aspect (Section 3) and then with respect to the conceptual framework aspect (Section 4), as a prelude to presenting our full engineering approach for security methodologies, embodied in a meta-methodology, in Section 6. We show how the latter constituents are related in Figure 1, which can be referred to throughout the next two sections. While we delay and spread the accompanying explanation across Sections 3.6, 4.2 and 6, we introduce the figure now to give the reader a schematic overview and roadmap for the rest of the paper.
Engineering a security methodology: security process aspect

3.1 SPPF: a framework of security process patterns and pattern modifiers

The core of our engineering approach with respect to the process aspect of a security methodology is based on security process patterns, which are used in the context of a multi-level framework for modeling security processes named SPPF (Security Process Pattern Framework). Security process patterns are essentially traditional (development-oriented) process patterns encapsulating re-usable collections of security-related techniques and/or activities in a generic fashion. They can be seen as problem-solution pairs, such that given a particular problem, a corresponding pattern can be instantiated to (notionally) produce a family of process fragments, each of which can be regarded as a solution specialization and variant (cf. [17]). Patterns are thus more abstract than detailed process descriptions, and their encapsulated solutions emphasize what should be done in a given context, rather than how (cf. [36]) – the latter details being specified in the pattern instances.

According to Störrle [37], describing processes in general (and hence security processes in particular) using patterns has a number of advantages, including:

- comprehensibility: easier to understand and apply than traditional process/methodology descriptions
- varying formality: individual patterns may have varying degrees of formality and/or constraints
- flexibility: changing one pattern instance with another, customized instance is a simple task
- scalability: patterns can be applied on various scales of granularity and abstraction

Security process patterns can be divided into two main categories (see Figure 2): macro-process patterns, which are used to describe (large-scale) parts of a security process (e.g. modeling adversaries or threats); and micro-process patterns, which are used to describe (small-scale) activities or tasks related to or stemming from a particular conceptual security artefact (e.g. setting up meta-authorization policies, or configuring a network infrastructure for security). Patterns (both the macro- and micro-varieties) can be categorized according to their granularity: task patterns encapsulate the detailed steps for a specific activity; stage patterns encapsulate a set of tasks; and phase patterns encapsulate a set of stages performed in some workflow (e.g. in sequence or iteratively). Conceptually, each successively larger-grained pattern “contains” the finer-grained patterns as sub-patterns of its own solution, which allows for nesting. Patterns can also be generic or specific, with the latter refining the former and being distinguished by context of applicability – e.g. system type, general approach used etc. – and level of abstraction.
Naturally, not every security process/methodology element or description qualifies as a process pattern – process patterns must possess at least four characteristics, which apply to patterns in general (cf. [38] or [39]):

- **abstraction**: a process pattern describes a generic solution to a well-defined problem. Generic solutions are not tied to a specific methodology, and are applicable in a variety of related contexts within the scope of the pattern.

- **recurrence**: a process pattern must embody recurring activities, techniques etc. in various methodologies, i.e. it should be possible to identify different instances of the same pattern in at least three or more methodologies.

- **flexibility**: a process pattern must be customizable upon instantiation, whereby instances of a particular pattern differ from each other and yet are still recognizable as being instances of that pattern (by analysis and comparison, for example).

- **best practice**: a process pattern should reflect, as much as possible, best practices. A new, “one-time” idea with a single application may or may not qualify as being a pattern.

The factors outlined above also distinguish process patterns from related method engineering process elements such as fragments and chunks (for which see [40, 41, 33, 28]), which can, however, be seen as particular realizations of a given pattern. Security process patterns should also not be confused with the meta-patterns for SME appearing in the work of Rolland and Plihon [43] or Denecke [44], which are used to guide aspects of methodology creation. Patterns are, however, closely related to what we term abstract process fragments, which can be seen as a type of specific process pattern that is abstracted from some part of methodology's process description. Such fragments can be stored in a repository, akin to method fragments, and used to realize particular patterns during methodology construction.

Besides (pure) process patterns, SPPF makes use of pattern modifiers [45], which are specific types of patterns defining variations, extensions, or refinements — nuances — of other patterns. Modifiers are used in SPPF explicitly (as stand-alone modifiers) and implicitly (as patterns that take the role of modifiers) in three ways: (1) as specific modifiers (for development phases/stages), to help define points of variation or choice and to add nuance; (2) as phase patterns, to determine contexts within which patterns of smaller granularity have a different purpose; and (3) as task and stage patterns, to extend, refine or detail patterns of larger granularity. In cases (2) and (3) above we term the implicit modifiers (informal) parameters to the original pattern. The specific modifiers referred to above are called generic life-cycle modifiers, and represent in an abstract, relatively minimalistic fashion, the phases and stages of a generic software development life-cycle (see, e.g. [46, 47, 19, 48]) — henceforth simply generic life-cycle. The
purpose of these modifiers is to provide a means for representing the integration of various SPPF process patterns with a corresponding software development process.

3.2 Structure and presentation of SPPF

At the beginning of the last Section (3.1) we referred to SPPF as a multi-level framework. In fact, SPPF is stratified into three levels, which can be summarized as follows:

- **First SPPF level**: consists of a pre-defined collection of generic life-cycle modifiers to help align second-level process patterns to a given development phase/stage, and to define variants of their solutions.
- **Second SPPF level**: consists of a pre-defined collection of generic security macro-process patterns resulting from deductive principles and the mining of existing security methodologies.
- **Third SPPF level**: consists of arbitrary collections of abstract process fragments, which have been generalized or abstracted from existing methodologies. Such fragments may exist in a methodbase (see [23]), or may be extracted “on-the-fly” during the construction process of a new methodology.

In the sections that follow, we present the first two levels of SPPF, with the generic life-cycle modifiers described briefly in Section 3.3, and the generic security macro-process patterns described in some detail in Section 3.4.

Regarding the third SPPF level, since abstract process fragments are obtained by generalization, any fragment collection will be merely representative and open-ended, not “closed” (relatively speaking) as is the case with the first and second level SPPF patterns. Therefore, in order to keep within reasonable space limits, we only (briefly) present two abstract process fragments from the third SPPF level in Appendix A, and illustrate the use of one of these in our example in Section 7.

3.3 First SPFF level: generic life-cycle modifiers

In this section we briefly present the patterns constituting the first level of SPPF: a collection of generic life-cycle modifiers to help align particular process patterns to a given development phase/stage, and to determine solution variants. The presentation follows the classical sequential order of development, i.e. from the gathering and analysis of requirements, through design, to implementation; while the patterns can be customized, their order cannot be shuffled. We use the simple micro-pattern template [49] for phase modifiers, augmented with two fields (cf. [26]): Role (describing the technical team roles partaking in the development process phase) and Artefacts (the work products being developed during a particular phase). Stage modifiers are given simple descriptions without a template. All pattern names are accompanied by a shorthand name in brackets, used in the descriptions of the second-level security process patterns (Section 3.4) as well as in a notation for security process models introduced later (Section 5).

3.3.1 Requirements Analysis (reqAn) Phase pattern

*Context*: The development of a software system is being initiated.

*Problem*: The problem-domain of the project must be defined, i.e. what problems is the software system solving?

*Solution*: Capture and analyze a system's needs and requirements of the project. Within the model-based paradigm a number of (conceptual and/or architectural) models can be created to embody the requirements, which allows discussion with different stakeholders. The Requirements Analysis phase represents the problem space of the project.

*Roles*: Requirements engineers, software architects

*Artefacts*: Requirements lists, Use cases, Conceptual models and/or Initial software architecture

3.3.1.1 Requirements (req) Stage pattern

*Context* and *Problem*: as for the Requirements Analysis phase pattern.

*Solution*: During the requirements stage the needs of the projects are elicited using customer interviews, brainstorming sessions and other standard techniques (see, for example, [47]).

3.3.1.2 Analysis (an) Stage pattern

*Context* and *Problem*: as for the Requirements Analysis phase pattern.

*Solution*: During the analysis stage a conceptual model of the system and/or an initial software architecture is created. The former may take the form of an object-oriented analysis model, a computation-independent model (CIM) for Model-Driven Development (MDE/MDA – [50]), actor models for Agent-Oriented Software Engineering [51] etc.; while the latter may take the form of a conceptual architecture.
3.3.2 Design (des) Phase pattern

**Context:** A software system is being specified.

**Problem:** A solution to the problems captured as needs and requirements must be developed.

**Solution:** Design the system by realizing the needs and requirements in a set of architectural models expressed in one or more Architecture Description Languages (ADLs). The models can be organized in various views (see, for example, [52]), including logical, execution and deployment views. Depending on the development paradigm, the resulting models can be object-oriented design models expressed in UML, platform-independent models (PIM) as used in MDE/MDA, aspect models etc. Choices for software packages and tools are made throughout. As the design proceeds to finer levels of detail, the higher-level models are refined to lower-level detailed designs of different components. The Design phase represents the solution space of the project.

**Roles:** Software architects, software developers

**Artefacts:** Architectural models of varying levels of detail

3.3.2.1 Architecture-development (arch) Stage pattern

**Context and Problem:** as for the Design phase pattern.

**Solution:** During the architecture development stage the complete software architecture is elaborated via architectural analysis, synthesis and evaluation [53], and documented appropriately [54].

3.3.2.2 Detailed design (detDes) Stage pattern

**Context and Problem:** as for the Design phase pattern.

**Solution:** During the detailed design stage the architectural models are refined to be ready for implementation. There can be several iterations of detailed design, each time coming closer to some specific implementation.

3.3.3 Implementation (impl) Phase pattern

**Context:** A system is being developed.

**Problem:** How to realize the system in software.

**Solution:** Implement the software architectural models from earlier phases into concrete software units: subsystems, modules etc. by coding in appropriate programming languages, using COTS components, frameworks, particular middleware platforms etc.

**Roles:** Software developers

**Artefacts:** Code-level artefacts (libraries, software modules)

3.3.3.1 Coding (cod) Stage pattern

**Context and Problem:** as for the Implementation phase pattern.

**Solution:** During coding stage the majority of the programming effort is expended, including connecting new and existing software units together into a working system.

3.3.3.2 Testing (test) Stage pattern

**Context and Problem:** as for the Implementation phase pattern.

**Solution:** During the testing stage the software is rigorously tested using standard software engineering practices, including white box, black box, boundary, stress and regression testing [47]. This pattern can also be applied at other stages, in conjunction with other patterns (e.g. as verification during Analysis (an) or Design (des)).

3.3.4 Deployment (deploy) Phase pattern

**Context:** A software system has been completed.

**Problem:** How to put the software system into production.

**Solution:** Deploy the software system by realizing the installation procedure, distributing among various nodes, and administering any configuration parameters [52].

**Roles:** Software developers, system administrators

**Artefacts:** Complete software system, Configuration parameters

3.4 Second SPPF level: generic security macro-process patterns

3.4.1 Overview

The second level of the SPPF framework contains a collection of security process patterns describing in an abstract
form the main elements of a security process aspect of any given (model-based) security methodology. The constituent patterns were collected using two synergistic strategies:

1. **Inductive pattern mining**: based on the comprehensive survey of methodologies in [14], we distilled essential methodological aspects by modularizing methodology descriptions as much as possible (cf. [55] and [56]) and subsequently analyzing and abstracting – guided by domain expertise – recurring activities and techniques.

2. **Deductive principle application**: starting from the principles, requirements and general characteristics of a security methodology (e.g. in [57] and [58]), we were able to pose certain activities and features as necessary and/or important, compare them with recommendations in the literature (e.g. [59, 60, 61, 62, 9, 10]), and subsequently identify them in existing methodologies.

The resulting patterns represent a mixture of distilled security best practices and generalized activities and/or techniques from existing security methodologies. Being generic patterns, they are abstract enough to allow for a wide degree of customization (according to different project situations) and for both more concrete specific patterns and process fragments to be used in their realization. In what follows we present these patterns in a uniform fashion using adaptations of two established textual templates.

For phase and stage patterns, we employ an adapted version of the (already modified) POSA template proposed by Ambler [36] (cf. [63]), where the Name and Type of pattern (phase, stage, task) appear in the subsection title, while the initial Context/applicability of the pattern, the Problem it attempts to solve, the Forces (constraints, conditions etc.) that need to be satisfied by the solution, the Solution itself and the Resulting Context (often indicating work products effected) are included as explicit fields.

Parameters, which refer to second-level (explicit) modifiers or patterns acting as (implicit) modifiers (see Section 3.1) to particular phase or stage patterns are given a separate Parameter field, with a general description of the effects. Parameters also appear in the Solution text using a shorthand notation in round brackets (stage patterns) or square brackets (task patterns). Generic life-cycle modifiers also appear within the Solution text when appropriate (to define variants) proceeded by the preposition At and followed by the phase/stage pattern name with shorthand in brackets for easier identification (e.g. At analysis (An)).

For selected patterns we also include an Implementation field to indicate concrete implementation strategies.

When presenting task patterns we employ the simple micro-pattern template [49] as for the generic life-cycle modifiers in Section 3.3, augmenting it with an Implementation field if or when required.

Individual patterns are presented in order of decreasing granularity, beginning with phase patterns, then proceeding to stage and task process patterns. The order of phase patterns is based on the general procession of activities in a security methodology (see [14]), beginning with a consideration of security requirements and continuing with the introduction of security solutions (countermeasures), verification and finally security-related deployment activities. Each phase pattern encapsulates one or more stage patterns, and, likewise, each stage pattern may contain several task patterns, where in each case the (representative or suggested) sequential workflow between the activities embodied in the patterns is reflected in the order of presentation. All pattern names are accompanied by a shorthand name in brackets, which is used later (Section 5) as part of a notation for security process models.

While we present the task and stage patterns under corresponding larger-grained phases, stage patterns may be applicable across phases, and task patterns across stages. In accord with their role as modifiers, some task and/or stage patterns do not always constitute a design solution in their own right; in these cases, their full value lies in their combination with stage or phase patterns.

### 3.4.2 Security Requirements Determination (SecReq) Phase pattern

**Context:** A software system is being developed in which some or all of the functional requirements have been specified, but not all of the non-functional ones. In particular, security requirements have either not been specified or need refinement.

**Problem:** We wish to know what security measures, if any, the system needs to implement.

**Forces:**
- Organizations can define high-level security policies, which need to be taken into account
- The final set of security requirements should describe sound protective measures for a system
- Determining requirements cannot be automated, and requires support from a structured approach

**Solution:** Specify security requirements in two consecutive stages, according to a division of requirements into two types: prescriptive and resultant.

Prescriptive security requirements are equivalent to high-level security policies. Such policies are first elicited from an organization, are specified directly for a given project, or stem from legal requirements (e.g. HIPAA), and dictate the overall security measures. The high-level security policies we choose to implement become the security requirements for the system under development.

Resultant security requirements are, as the name implies, the result of performing some form of assessment of the system. This is generally some form of adversary modeling (AvdMod), where attacks to the system are considered. The resulting requirements are then traded-off with other non-functional requirements.
The set of prescriptive and resultant requirements form the final set of security requirements of the system, which dictate what protective measures are necessary. They are problem-domain artefacts and are not yet security countermeasures – those are determined in another phase.

**Parameter (Stage):** Adversary Modeling (AdvMod)

If adversary modeling is performed, the requirements are taken as the set of protective measures which cover all threats.

**Resulting context:** The result of this phase should be a fully coherent (or partially coherent, if refinement is planned) set of security requirements that can be used to determine necessary countermeasures.

### 3.4.2.1 Adversary Modeling (AdvMod) Stage pattern

**Context:** Some parts of a system have already been determined. There is a need to determine how the system can be attacked, and what can be done to protect it.

**Problem:** How can we find out the attacks to a system and justify the introduction (or not) of security measures?

**Forces:**
- It should be possible to identify potential system vulnerabilities
- Attacks need to be considered individually, with respect to these vulnerabilities
- There should be a guided method to list or structure attacks
- Attacker capabilities (e.g. available access) and goals should be considered

**Solution:** Perform adversary modeling, which seeks to model attackers and their potential attacks (threats) to a system.

At requirements analysis (reqAn): this may be done with respect to a set of use cases or a conceptual model (preliminary software architecture).

At design (des): this may be done with respect to an initial or elaborated design, or after the design has been detailed and made ready for implementation to help determine effectiveness.

Adversary modeling generally proceeds as follows (with steps 3 and 4 being optional):

1. **Identify** [assets] (at analysis) or uses of a system (at requirements)
2. **[enumerate]** threats
3. **Detail and** [refine] threats
4. **Estimate** [risks]

**Parameters (Task):** [assets], [enumerate], [refine] (see Section 3.4.6), [risks] and [attacks] (see Section 3.4.3.3.4) extend and detail this pattern.

**Resulting context:** An adversary model (such as a threat model).

**Implementation:** There are a number of concrete approaches (besides those used in various methodologies, for which see [14]) that can be used to realize this pattern in part or in full, as well as the related task patterns – for example, Microsoft's STRIDE-based threat modeling [64], T-MAP (for COTS packages) [105], formal threat modeling [106], goal-based threat modeling [107] and many others. Risk analysis approaches such as CORAS [65], which can be interpreted as an adversary modeling process emphasizing elaborate [risk] task pattern realizations, can also be utilized, as well as elements of other risk-based techniques (see [108]).

### 3.4.2.1.1 Identify assets task pattern [assets]

**Context:** We have an initial representation of a system.

**Problem:** What information and/or system parts need to be protected in a broad, general sense, or with respect to the enforcement of certain security properties?

**Solution:** From the initial system representation, or from a secondary system representation (e.g. data-flow diagram, network model, derived architectural model – see the [model] modifier in Section 3.4.3.2.3), determine the important elements that require protection: data, interfaces, subsystems, processes and others. This can be done in consultation with domain experts, customers and (other) stakeholders. Protection can be considered either in general (e.g. a part of the system should be protected from all threats), or according to targeted classes of security properties or requirements, such as confidentiality (whether stored or in-transit data should be readable or accessible); integrity (whether data should be modifiable); authenticity (of data, users or processes); non-repudiation, auditability, system maintenance and others – see [78] for a complete taxonomy. In this case assets are taken to mean both software units and valuable information/parts of the systems. The precise semantics of protection and any targeted security properties/requirements for each asset type, and each asset instance, are determined contextually.

**Implementation:** See the Implementation section of the AdvMod pattern.

### 3.4.2.1.2 Enumerate threats task pattern [enumerate]

**Context:** We have identified elements in a system that require protection.

**Problem:** How can we determine the threats to these elements?

**Solution:** Consider all the attacks an attacker can make to compromise the vulnerable parts of the system using a structured approach. A list of common attacks (e.g. CWE) or attack patterns can help speed up the process and
reduce the required expertise.  

Implementation: There are many techniques that can be used in realizing this pattern, including threat trees [64], attack trees and misuse cases [109, 110]. See also the Implementation section of the AdvMod pattern.

3.4.2.1.3 Estimate risks task pattern [risks]

Context: We have a set of threats or possible failures to a system.  

Problem: How can we determine whether how much of a risk they pose?  

Solution: Analyse and assess the risks of each threat occurring, alone or in relation to others, assigning scores or giving qualitative measures. Risks can then be tabulated or attached to each threat or failure for further analysis or trade-off. The analysis of risks here is meant in a relatively narrow, targeted sense, not as a synonym for a whole AdvMod stage (sometimes ambiguously referred to as “risk analysis” in the literature).  

Implementation: One approach to assess risks in a more structured manner is via Microsoft's DREAD rating system [111]. A customized variant realization could also use HAZOP analysis [112] instead of direct risk measures. See also the Implementation section of the AdvMod pattern.

3.4.3 Countermeasure Introduction (CounterIntro) Phase pattern

Context: A software system is being developed and requires security features.  

Problem: We wish to introduce necessary security measures in a structured fashion.  

Forces:  
- Countermeasures should be introduced in a systematic fashion  
- The development paradigm is model-based, requiring a corresponding approach

Solution: Introduce security countermeasures by converting problem-space security requirements (cf. Security Requirements Determination process pattern in Section 3.4.2) into solution-space goals and policies, and incorporating model-representations of the corresponding mechanisms at the relevant development stage into conceptual (analysis) or more complete software architectural models (design).  

Resulting context: The result of this phase should be a coherent set of countermeasure representations (or partially coherent, if refinement is planned) that can be implemented in software.

3.4.3.1 Countermeasure Identification (CounterIdentif) Stage pattern

Context: A set of security requirements has been determined in some fashion.  

Problem: How can we find out what countermeasures to introduce in response to a system's security requirements?  

Forces:  
- It should be possible to identify appropriate countermeasures for each security threat  
- A set of countermeasures, whether as a list, in a repository or based on expert knowledge should be available

Solution: Map all problem-space security requirements to a set of available solution-space abstractions. This can be done by several levels of indirection, first mapping requirements to goals and policies (essentially converting problem-space to solution-space terminology), then to particular sets of solution representations.

This stage can be accomplished by [expert opinion], whereby a security expert considers all requirements and selects appropriate solutions; or by a [structured] approach, whereby developers use tools (e.g. tables) to realize the security requirement to solution/countermeasure mapping.  

At requirements analysis (reqAn): this may be done with respect to a set of use cases or a conceptual model (preliminary software architecture).  
At design (des): this may be done with respect to an initial or elaborated design, or after the design has been detailed and made ready for implementation to help determine effectiveness.  

Parameters (Task): The [expert opinion] and [structured] tasks extend and detail this pattern.  

Resulting context: A set of countermeasures corresponding to each relevant security requirement.

3.4.3.1.1 Expert countermeasure identification task modifier [expert opinion]

Context: When identifying countermeasures (CounterIdentif pattern).  

Problem: As for the Countermeasure Identification pattern.  

Solution: Take each security requirement into account and determine an appropriate security mitigation strategy based on knowledge and experience. Mitigation strategies could come from a predetermined set of solutions/countermeasures, which in turn correspond to COTS components, libraries etc., or they could be given in the form of recommendations.

3.4.3.1.2 Structured countermeasure identification task modifier [structured]

Context: When identifying countermeasures (CounterIdentif pattern).
Problem: As for the Countermeasure Identification pattern.

Solution: Use a structured approach that systematically maps requirements to well-known sets of security solutions/countermeasures. Analysis of requirements is accompanied by tables, rules etc. that help inexperienced developers to map requirements and determine the necessary countermeasures. Since the set of countermeasures is known in advance, the mapping procedure can enforce traceability.

3.4.3.2 Security Modeling (SecMod) Stage pattern

Context: The set of security countermeasures (for a particular iteration) requiring introduction into the system have been identified.

Problem: How can we incorporate security countermeasures in the context of the model-based paradigm?

Forces:
- All necessary countermeasures should be representable in some fashion in a system's architectural models at the relevant development stage
- Countermeasures can be represented in different ways, and each representation may require a different incorporation approach
- Although the security mechanisms should be developed together with the functional aspects, they should be separated so they can evolve and change on their own

Solution: Incorporate security countermeasures as model elements or in some representative fashion into a system's software architectural models or as a separate security architecture. There may not be a one-to-one mapping of identified countermeasures to available model elements or representable security solutions, hence the interrelationships between the countermeasure models should be considered for consistency (between each other and existing architectural elements, functional and non-functional), correctness (individually) and completeness (to cover all identified countermeasures). Consistency in this context implies whether the chosen models are consistent with the actual nature of identified countermeasures, as well as whether the models are compatible with each other. Consistency is closely related to another important relationship that should be considered: composability – i.e. whether the composition of two models has any side-effects [113], such as a weakening of security properties (as can occur, for example, in cryptography-based countermeasures – cf. [114]).

Security modeling can be accomplished with the help of security experts, who guide the process, or using tools and techniques approachable to more novice developers. The approach for incorporating the countermeasures may be automated or semi-automated using tool-support, or manual.

At analysis (reqAn): security countermeasures (i.e. their model representations) are incorporated into the system's conceptual model (preliminary software architecture).

At design (des): security countermeasures (i.e. their model representations) are incorporated into the system's elaborated design, or after the design has been detailed and made ready for implementation.

Parameters (Task): The [specify], [model] and [instantiate] tasks extend and detail this pattern.

Resulting context: A set of new models and/or enriched versions of a system's existing architectural models with incorporated countermeasure representations, indicating where and how security solutions are to be implemented.

3.4.3.2.1 Specify task modifier [specify]

Context: When modeling countermeasures (SecMod pattern) and there are no pre-defined countermeasures.

Problem: As for the Security Modeling pattern.

Solution: Specify security countermeasures descriptively or using a custom notation. All specifications must subsequently be realized in the architectural models by developers.

3.4.3.2.2 Instantiation task modifier [instantiate]

Context: When modeling countermeasures (SecMod pattern) and there are pre-defined countermeasures in the form of patterns, aspects, model templates etc.

Problem: As for the Security Modeling pattern.

Solution: Instantiate the countermeasures into the relevant architectural models. For patterns this implies customization (in keeping with the original pattern ideas) and mapping participants to components etc., while for model templates and aspects this implies type-instantiation, where parameters are replaced with new or existing model elements. Integration between the countermeasures and the architectural model elements may be required.

3.4.3.2.3 Model task modifier [model]

Context: When modeling countermeasures (SecMod pattern).

Problem: As for the Security Modeling pattern.

Solution: Model countermeasures directly as part of a system's (functional) architecture, according to a metamodel; using a pre-defined, fixed set of elements; or by capturing security information in models.
3.4.3.3 Security Verification (SecVerif) Stage modifier

**Context:** The set of security countermeasures (for a particular iteration) have been introduced into a system's software architecture (CounterIntro) or have already been implemented (SecImpl).

**Problem:** To determine whether a particular set of countermeasures provides adequate protection, or, more generally, to verify the correctness, consistency and completeness of a set of security-related artefacts.

**Forces:**
- It is not always clear whether particular countermeasures can offer the required level of protection, and must be checked in some fashion
- All necessary countermeasures should be representable in some fashion in a system's architectural models at the relevant development stage
- There must be a way to measure or qualify desirable characteristics of the artefacts under consideration
- It should be possible to check that countermeasures do not interfere with each other

**Solution:**
- For security countermeasures (CounterIntro pattern), verify that the security countermeasures introduced into the system's architectural models provide the necessary level of security and adequately protect the system against relevant threats via [automated] or [semi-automated] analysis or via manual inspection/[reviews] with respect to the security requirements. Tool-support is essential for verification of code-level artefacts (implemented security countermeasures) (SecImpl pattern). In all cases verification activities should be documented to provide assurance and/or aid with security certification.

At design (des): Verification approaches can include automated analysis of models using software tools (see, e.g., [8]), or using some form of security metrics. Security protocol verification can also be performed [66, 115, 116].

At coding (impl): Automated verification implies the use of static and/or dynamic analysis tools (such as Splint, CodeSonar, Klocwork, Fortify Static Code Analyzer, Coverity Prevent and others – see [117, 118, 119] for an overview and references) directly, or with reference to the design-level models (see, for example, [66]).

At testing (impl): Verification strategies during testing include penetration testing (against the countermeasures transferred into software), as well as the usual software engineering testing strategies to verify boundary conditions. Manual testing as well as automated scripting of standard (exploitable) use cases can also be performed.

**Parameters (Task):** The [reviews], [semi-automated], [automated], [attacks] and [penetration tests] tasks extend and detail this pattern.

**Resulting context:** A set of verified security models and/or enriched versions of a system's existing architectural models with verified countermeasure representations (analysis and design), or a verified set of code artefacts (implementation).

3.4.3.3.1 Manual reviews task pattern [reviews]

**Context:** When verifying introduced countermeasures or security-related artefacts (SecVerif pattern).

**Problem:** As for the Security Verification pattern.

**Solution:** Manually inspect the effects of all security countermeasures, and trace them back to the security requirements to ensure they are correct, consistent and complete. Reviews and inspections should be structured, going through requirements systematically, or using a checklist to guide the process. This can be done at all development stages and as part of verification activities in all phase (security) process patterns.

3.4.3.3.2 Semi-automatic verification task pattern [semi-automated]

**Context:** When verifying introduced countermeasures or security-related artefacts (SecVerif pattern).

**Problem:** As for the Security Verification pattern.

**Solution:** Use tool-support to help verify some of the models or code excerpts. The models, code or, more generally, security artefacts can be prepared manually for use with the tool, which should speed up the most mundane aspects of the verification, or provide extra capabilities that would be difficult to achieve in a manual analysis.

3.4.3.3.3 Automated verification task pattern [automated]

**Context:** When verifying introduced countermeasures or security-related artefacts (SecVerif pattern).

**Problem:** As for the Security Verification pattern.

**Solution:** Use tool-support to verify some of the models or code excerpts. The verification process should be automated, and developers should not need to expend any additional effort other than entering the security artefacts into the tool, running the tool and interpreting the results.

3.4.3.3.4 Attack scenarios task pattern [attacks]

**Context:** When verifying introduced countermeasures (SecVerif pattern) during Design (des), or when performing adversary modeling (AdvMod pattern) during Requirements Analysis (reqAn).

**Problem:** As for the Security Verification pattern.
Solution: Determine a set of attack scenarios that can be used to analyze any security-enhanced architectural models. The scenarios can be modeled as part of the system's architecture. Analysis minimally consists in checking qualitatively whether the existing countermeasures offer sufficient defence. This task pattern can be used in conjunction with semi-automated or automated analyses to provide additional, quantitative measures.

3.4.3.3.5 Penetration testing task pattern [penetration tests]

Context: When verifying introduced countermeasures (SecVerif pattern) during Testing (test).

Problem: Try to break and/or bypass the countermeasures introduced of the system, by systematically exploiting known weak points. Exploits should be run against both the introduced countermeasures and known weaknesses of the system, for example, using lists such as the MITRE's CWE, CVE and CAPEC (see www.mitre.org/work/cybersecurity.html) to test the system's defences. This task pattern can be used in conjunction with semi-automated or automated analyses.

Performing penetration testing requires trained developers or a team of security specialists.

3.4.4 Security Implementation (SecImpl) Phase pattern

Context: A software system is being developed in which some or all of the security requirements have been specified and modeled in the software architecture.

Problem: How can the security countermeasures required by the system's design be implemented?

Forces:
- Within the model-based approach, architectural models of sufficient detail are necessary to implement security
- Implementing security should proceed together with, not after, the implementation of the functional aspects of the system
- A single implementation approach may not be sufficient
- The approaches to implementing security should align with the development paradigm used and/or with any relevant prescribed standards or rules

Solution: Transfer the security-related (standalone or enriched functional) architectural models into concrete software units. This can be done via a combination of patterns, such as manually coding some aspects, using COTS components for others and generating code artefacts from certain design elements. The stages of Security Implementation can only be performed during the implementation development phase (the code stage in particular). Verification and validation can be performed during the testing development stage (see the Security Verification pattern, Section 3.4.3.3).

Parameter (Stage): Code

If security implementation is accomplished via manual coding activities, the implementation necessitates a more rigorous verification strategy during testing (see the Security Verification process pattern).

Parameter (Stage): Map

Mapping security-related models to existing code-level artefacts ensures integrity, however consistency checking is required to ensure compatibility between third-party/COTS and/or legacy software.

Parameter (Stage): Transform

Transformations of models to code artefacts may ensure correctness, however, any generated code may still require some form of verification.

Resulting context: The result of this phase should be a full (or partial, if iteration is planned) implementation of some or all security-related architectural models in software, resulting in a set of concrete software subsystems addressing the security concerns of the system.

3.4.4.1 Code stage pattern

Context: During the implementation development phase.

Problem: To implement security countermeasures determined during earlier development phases in software.

Forces:
- Models of sufficient detail are necessary to implement security
- Implementing security should proceed together with, not after, the implementation of the functional aspects of the system

Solution: Implement security countermeasures as concrete software subsystems in conjunction with the functionality of the whole system via manual coding. Throughout all, the standard software engineering and software project management practices should apply to ensure the quality of the resulting solution. This process pattern assumes that all security countermeasures under consideration have been traded-off with functional properties and that implementation is based on detailed descriptions of the target system's software architecture.

During manual coding, all security-related models are transferred by the development team into software units in conjunction with functional subsystems. The coding pattern implies that any necessary configurations to security
components are also performed by developers or prepared for administrators to modify during system deployment.

The Code pattern is applicable in all development paradigms.

**Resulting context:** A set of concrete software units that realize any security-relevant architectural models.

### 3.4.4.2 Map stage pattern

**Context:** During the implementation development phase.

**Problem:** To implement security countermeasures determined during earlier development phases in software while using minimal coding effort.

**Forces:**
- Models of sufficient detail are necessary to implement security
- Manual coding implies significant engineering efforts and hence costs – it would be good to make as much use of existing components as possible
- All existing software must conform to the security-related aspects of a system's software architecture

**Solution:** Map all security elements in a system's architectural models to existing components (COTS, legacy etc.), ensuring that the security requirements contained in the models are matched by the mechanisms offered in the components. Matching requires careful selection of software products, or the appropriate selection of suitable, pre-made components.

The Map pattern is applicable in all development paradigms, and is particularly well suited to instantiation within Component-based (CBSE) contexts.

**Resulting context:** A set of concrete software units that realize any security-relevant architectural models.

### 3.4.4.3 Transform stage pattern

**Context:** During the implementation development phase.

**Problem:** To implement security countermeasures determined during earlier development phases in software with minimal manual coding efforts.

**Forces:**
- Models of sufficient detail are necessary to implement security
- Manual coding implies significant engineering efforts and hence costs, which should be minimized where possible
- It is necessary to configure existing security countermeasures (e.g. in an underlying platform), and it is difficult to accomplish this manually

**Solution:** Use software transformations to generate code-level artefacts from higher level artefacts (usually architectural models). The generated code can be particular security configurations of existing software components, parts of functional components concerned with security, or whole security components (see the Map pattern in Section 3.4.4.2). Often the architectural models are required to be in some standard form (e.g. using a particular ADL) so that transformations can be applied (see, e.g. [67]).

Manual coding is minimized when applying the Transform pattern inasmuch as coding efforts are expended on transformation functions rather than directly realizing architectural models in software.

The Transform pattern is applicable in all development paradigms, and is particularly well suited to instantiation within Model-driven (MDE/MDA) contexts.

**Resulting context:** A set of concrete software units conforming to any security-relevant architectural models and/or a set of security configurations for existing components.

### 3.4.5 Security Administration (SecAdmin) Phase pattern

**Context:** A software system has been developed and put into production.

**Problem:** To configure the system at the deployment stage and to ensure smooth operation with respect to security features.

**Forces:**
- Security configurations are critical to the security subsystems' correct operation (e.g. rules for authorization)
- Problems may occur during system operation, and these should be handled and used as feedback
- New threats may appear in the deployment environment (overlooked earlier or due to code changes or lower level details)

**Solution:** Configure all component parts of the (implemented) security infrastructure, setting appropriate policies for the individual controls. All underlying software infrastructure (platform settings, networking, etc.) should also be configured in accord with any prescribed security requirements. Run-time observation tools can also be used in an integral way to gather information and oversee the security operations of the system during run-time, providing continuous feedback.

This pattern is only applicable once the system and its corresponding security infrastructure has been deployed in
its target environment.

**Resulting context:** A set of software infrastructure configurations and settings.

### 3.4.6 Refine task modifier [refine]

**Context:** Any activity that requires refinement.

**Problem:** How can the artefacts or results produced by a previous activity be refined?

**Solution:** The task of refining something implies performing the same activity at a finer level of granularity, introducing new or variant strategies to generally create a larger or more precise set of artefacts. In all cases the encompassing stage (or phase) pattern acts as an implicit modifier to provide a context for refinement, where the Resulting Context determines (for example) the set of artefacts under consideration.

To give an example, for **SecReq: AdvMod**, the AdvMod pattern acts as an implicit modifier providing a context where we have a list or some structured set of threats. The refine modifier implies that each threat should be analyzed in detail, system failures should be considered as well as threats, and new strategies for enumeration (see Section 3.4.2.1.3) should potentially be employed. Similarly, for **CounterIntro: SecMod**, the SecMod pattern acts as an implicit modifier providing a context where we have a set of architectural models with security countermeasures introduced. The refine modifier implies the application of previous approaches for detailing the models.

### 3.5 Using SPPF to construct security process models

Thus far we have summarized SPPF, which we presented as a basis for engineering the process aspect of a security methodology. The fashion in which it can be used for this is to create a **pattern-based process model** (henceforth simply **process model**), which reflects the activities, techniques etc. of the target methodology. There are two types of such models: **abstract** (which can be further sub-divided into task, stage and phase-level abstraction) and **concrete**, which we explain below.

Abstract process models are those in which a group of SPPF patterns have been arranged in a particular order (henceforth **workflow**), such as in sequence, iteratively etc., with parameters specified to each phase and stage pattern at some level of detail. Each pattern within this abstract model can be specified at three finer levels of abstraction: if no stage parameters are specified for a phase pattern, then that pattern is said to be at phase-level abstraction; similarly, if no task pattern parameters are specified for a stage pattern, that pattern is said to be at stage-level abstraction; and finally, if all parameters are fully specified, a pattern is said to be at task-level abstraction. An abstract process model can mix patterns at different levels of abstraction, and hence specification. Naturally, abstract process models do not constitute actual security methodologies, much as an architectural model of a system is not a system; nevertheless, in this case the model can actually be used as a (basic) security methodology providing high-level guidance, a feature that is worth noting, even in passing.

In contrast to abstract models, concrete process models are constituted entirely of SPPF pattern **instances**, which are the actual activities of a concrete methodology, i.e. fully tailored, elaborated phases, stages and tasks. In both cases the models can be represented in the same fashion, using the same notation (see later in Section 5), however, the corresponding process descriptions will, of course, differ.

### 3.6 Engineering strategies for constructing security process models

As mentioned in Section 2.3, our approach to engineering security methodologies takes inspiration from a number of the traditional method engineering strategies. Having already discussed SPPF and security process patterns in general, we outline below two different strategies for engineering the process model. Both strategies are encapsulated as part of S-SMEP (Situational Security Methodology Engineering Process), a meta-methodology for engineering security methodologies, which we present later in Section 6.

#### 3.6.1 “From-scratch” strategy: process pattern assembly

In the “from-scratch” assembly-based strategy, which is inspired by the traditional assembly-based method engineering technique (see [22, 33]), SPPF patterns are selected and “assembled” in some workflow to create an abstract process model (see the left-hand side of Figure 1, which we introduced at the end of Section 2.3). The patterns are then specified, refined and realized, potentially by instantiating abstract process fragments from a repository. A security process is thus constructed in a fashion analogous to applying a design pattern through a two-phase process: firstly, instantiating the pattern into a design, where roles are mapped to components, interfaces are defined etc. to determine the overall structure: this corresponds to creating an abstract process model; and secondly, coding the pattern with the rest of the system, where all the details are finalized: this corresponds to specializing the abstract model to a concrete one by customizing, elaborating and detailing the phases, stages and tasks. In relation to Figure 1, the “from-scratch” strategy implies top-down construction.
3.6.2 Base methodology tailoring

Base methodology tailoring, on the other hand, which is inspired by the configuration method engineering technique (see [34]), starts out by constructing a concrete process model by identifying SPPF pattern instances in a chosen base security methodology. The patterns constituting the model can then be replaced (which implies customization and re-instantiation), modified etc. as required. New patterns can also be added akin to the “from-scratch” strategy, which follows the same top-down procedure. A security process is thus constructed in a fashion analogous to software re-engineering (cf. also [28]): a design is taken from an existing implementation, re-worked, and re-coded. In relation to Figure 1, the methodology tailoring strategy implies a mix of bottom-up (creation of concrete model) and top-down construction.

4 Engineering a security methodology: conceptual security framework aspect

Thus far we have described, in greater or lesser detail, the constituents of our engineering approach for the process aspect of a security methodology. In this section we briefly describe the central artefact and the corresponding strategies for engineering a methodology’s conceptual framework aspect.

4.1 Conceptual security framework meta-model

Our approach to engineering the conceptual security framework aspect of a methodology is inspired by the paradigm-based method engineering strategy (see [30]), which relies on instantiating a meta-model (see [13, 68]) often to create a product model around which a process model can be built. In our adaptation of this approach, a meta-model is used to prescribe a set of constraints on the nature of, and relationships between, the various conceptual artefacts to be used in a security methodology. Models of conceptual security frameworks are constructed by instantiating elements of the meta-model semantically conforming to the prescribed constraints.

The meta-model referred to above is presented in Figure 3. As for the second-level SPPF security process patterns, the meta-model was constructed by abstracting various features from conceptual security frameworks used in existing methodologies [14], studying meta-models for security such as [69, 70] and our own security expertise.

The dashed boxes in the figure represent elements that cannot be instantiated and are included only for the sake of clarification. The colours (or shades of gray if not viewed in colour) in Figure 3 are used simply to improve readability and to emphasize the importance of the Conceptual security artefact model element.

![Figure 3: Meta-model for engineering a methodology's conceptual security framework](image)

The meta-model essentially dictates that all conceptual security frameworks should consist of a set of Conceptual
security artefact collections (see Fig. 3), which aggregate Conceptual security artefacts. Such artefacts can be specified or documented explicitly, thus forming an Explicit artefact collection, which may be structured (e.g. a catalog) or unstructured; or they may be based purely on implicit (expert) knowledge, in which case they form an Implicit artefact collection. In both cases conceptual artefacts may be related, or relatable, to a particular system characterization (e.g. vulnerabilities being related to a network model). A selection of the conceptual security artefacts from the meta-model are explained in Table 1, most of which have the usual semantics for the corresponding security concept; however, it must be kept in mind that they represent classes of specifiable artefacts, not singular concepts.

<table>
<thead>
<tr>
<th>Conceptual element</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability</td>
<td>Vulnerabilities represent some form of weakness in some aspect or feature of a system.</td>
</tr>
<tr>
<td>Attack</td>
<td>An attack represents a set of actions of an adversarial nature made to compromise the security attributes (secrecy, integrity, availability etc.) of a system. Attacks realize threats and exploit vulnerabilities.</td>
</tr>
<tr>
<td>Threat</td>
<td>A threat represents some form of potential harm (e.g. “unsafe code execution”) that can be realized by multiple attacks in different contexts (e.g. viruses, buffer overflows, and many others requiring the execution of a payload/code snippets).</td>
</tr>
<tr>
<td>Security requirement</td>
<td>A security requirement represents any project or system requirement that pertains to a security attribute (as listed above), determined, at least nominally, in the problem space. Security requirements dictate the security needs of a target system both at a high/organizational level and at a technical level.</td>
</tr>
<tr>
<td>Security policy</td>
<td>A security policy is the solution space form of a security requirement, which is realized by a security solution of some form.</td>
</tr>
<tr>
<td>Supportive artefact</td>
<td>A supportive artefact refers to any notable artefact which is not security-related, but which is essential to, or aids, the use of one or more security artefacts (e.g. a mapping table or a collection of models).</td>
</tr>
</tbody>
</table>

Table 1: Descriptions of selected conceptual security artefacts from the conceptual framework meta-model

4.2 Engineering strategies for constructing conceptual framework models

4.2.1 Meta-model instantiation

As explained above, our strategy for constructing a conceptual security framework is based on meta-model instantiation, which, in line with the corresponding division for security process models, produces abstract or concrete (conceptual security) framework models. The meaning of the terms is similarly related: abstract framework models simply instantiate the meta-model element classes as a model; whereas concrete models imply that the actual model elements exist, have been specified etc.

With respect to the “mechanics” of meta-model instantiation, when instantiating the meta-model, the semantics of the element instances (i.e. conforming elements) should correspond to those of their meta-model counterparts, i.e. element semantics should be retained. For example, a Security solution artefact class can be instantiated as a security aspect – in the sense of Aspect-Oriented Modeling – but not as an attack tree, which would be an instance of the Attack class (see Figure 3). Conceptual framework models should also retain the existence of the associations between element instances present in the meta-model, however, their precise semantics are allowed to vary (e.g. the “realizes” association between security solutions and security policies may change to an “encapsulates” association) and new associations can also be formed as required. Since the meta-model does not exhaust all the possible conceptual artefacts that can be used in a methodology, elements specializing Conceptual security artefact (Figure 3) can also be created and instantiated as required. Naturally, the resulting model elements should necessarily correspond to actual artefacts that are or will be used in an existing or target security methodology, respectively.

5 Notation for representing methodology models

So far we have presented the core artefacts of our engineering approach for both the process and conceptual framework models, in Sections 3 and 4 respectively, and, towards the end of each of these sections, outlined the engineering strategies for constructing corresponding process and conceptual framework models. Taken together, the two models form what we term a pattern-based methodology architecture model, or more simply methodology model. The term “architecture” in the latter definition refers to the fact that methodology models represent the organization of process elements and conceptual artefacts and their interrelationships, in way that is strongly analogous to how software architectural models represent the component parts of a system and their relationships. Since in a software architecture these relationships embody a set of (principal) design decisions [52], by analogy, methodology models could also be seen to represent the major design decisions taken to construct a methodology. If the analogy is to be kept complete, then it should also be possible to represent these models using some notation or
In this section we briefly present a UML-inspired notation – with both graphical and textual aspects – derived from the work of Van de Weerd and Brinkkemper [71], which allows us to represent both the process model created using SPPF patterns and the conceptual framework resulting from an instantiation of the meta-model in Section 4.

The purely graphical aspect of the notation (see Figure 4) consists of two interrelated parts: an extended UML-like activity diagram notation (left-hand side) for representing process models; and an extended UML-like class diagram notation for representing conceptual framework models. The diagrams are linked via dashed arrows proceeding from the process elements to the conceptual framework elements, which denote a “uses” or “pertains to” relationship.

Considering the left-hand part of Figure 4, we can see that process model elements can either be open (italics font), in which case their nested elements are shown explicitly either on the same or on different diagrams (see Note 5 in the diagram); or closed (bold font), in which case their nested elements are not shown (see Note 7 on the diagram). Open and closed process elements are said to be complex elements. Elements with nesting shown explicitly (Note 1) are open by default, while those without nesting, which are also not open, are said to be simple process elements. A gray background on any element implies that a given process element (pattern) is applicable to multiple contexts (Note 4). Black background implies that a process element (pattern) has been realized using an abstract process fragment, the name of which appears in brackets (Note 8).

The arrows between process elements imply that the activities encapsulated by the given patterns (or pattern realizations) are or should be performed sequentially, or follow from each other logically (Note 2). Conditions on the flow of activities can be placed, which also allows for iteration (Note 3). Besides being performed in sequence, activities can also be performed without order, in which case process elements appear without arrows, divided from other (sequential elements) by a horizontal rule (Note 6); and also in parallel (Note 9).

Generic life-cycle modifiers are represented by “sign-post” arrows in light gray background attached to each relevant pattern. The given modifier is distributive across the nested process elements, e.g., the first modifier in the diagram will apply to the phase pattern, all the nested stage patterns and all the nested task patterns.

The advantage of the process notation is that it can represent not only SPPF patterns and their organization, but also arbitrary specific patterns as well as abstract process fragments and even the activities of realized patterns. Process models can thus contain elements of essentially varying levels of abstraction and specification, making for a rather “universal” notation (hence the description above referring to “process elements” rather than patterns explicitly).

If we now consider the right-hand part of Figure 4, we will see similar innovations extending the usual UML class diagram notations: concepts can either be complex, namely, open (italics) or closed (bold); or simple. Concept elements with a gray background imply flexible artefacts that are applicable in multiple contexts. All other features of UML class diagrams have been retained, however, hence allowing for the construction of models conforming to the meta-model from Section 4 using standard (UML) semantics, with the only difference being the decision of which concepts to regard as complex and which as simple.
Respecting the textual aspect of the notation (not shown in the figure), we employ the shorthand names of the SPPF patterns, which makes it relatively easy to differentiate SPPF patterns from other, custom patterns or other process elements. The name of a pattern with slashes on either side (e.g. /AdvMod/) implies an optional activity. A tilde before a pattern name (e.g. ~SecReq) implies the process has only basic support for that activity, which is useful information for comparing methodologies at a glance by considering only their process models. In a similar vein, comments, brief explanations or useful information can be notated in pointed brackets next to a given pattern (e.g. enumerate <use cases> implies that the Threat enumeration task pattern employs use cases in some fashion). The latter construct can also be used in process framework elements, for example, to indicate their context of applicability.

An example of a full methodology model using the notation introduced thus far can be seen in Section 7, Figure 5, for the security methodology of Fernandez and colleagues [4].
6 S-SMEP: a meta-methodology for engineering flexible security methodologies

Having now presented the core artefacts of our engineering approach for security methodologies (see Figure 1 in Section 2.3) – SPFF (Section 3) and the conceptual framework meta-model (Section 4); and having discussed engineering strategies for how they can be used to construct a methodology model (Sections 3.5, 3.6 and 4.2), using a specific notation (Section 5), we have completed the task laid out in Section 2.3 to present the constituents of our approach to engineering a security methodology. There is one outstanding point, however, which is missing in the latter presentation, namely, a step-by-step guide on how the engineering strategies can be realized. We fill this gap in the present section, by describing a corresponding meta-methodology (cf. [21]) for engineering security methodologies called S-SMEP (Situational Security Methodology Engineering Process), based on the work of Asadi and Ramsin [72]. The value of using the meta-methodology is completely analogous to the value of using security or development methodologies to provide systematic guidance for the introduction of security features or the construction of software systems, respectively. In this case, the roles of developers are simply replaced with that of security methodology engineers. S-SMEP can also be seen as a refinement and encapsulation of the engineering strategies presented in Sections 3.6 and 4.2, which details how these strategies can be realized in practice. In what follows we first discuss how S-SMEP was constructed (Section 6.1), and subsequently present S-SMEP’s phases, stages and tasks in Sections 6.2 to 6.5.

6.1 Constructing S-SMEP

Our starting point for S-SMEP lies in the work of Asadi and Ramsin [72], who propose a generic process (meta-methodology) based on a collection of process patterns for SME. Asadi and Ramsin's approach in turn is based on the three prevalent strategies for method engineering as per [30] – chunk/fragment assembly [33, 28]; meta-model abstraction/instantiation [13]; and method configuration [34] – which define three “workflows” internal to SMEP's patterns. Like the traditional SME techniques which it encompasses, SMEP is a meta-methodology for engineering development methodologies, not security methodologies. Nevertheless, since our own engineering approach is inspired by the aforementioned SME techniques, SMEP contains a number of relevant features that can be adapted, and forms a good base upon which to create a more specific meta-methodology.

We have taken a two-phase approach to constructing our own meta-methodology based on SMEP:

1. Firstly, we tailor and instantiate the process patterns in SMEP to construct a methodology which we term SMEP-EI – SMEP extended instance – capable of working with process patterns in addition to the three standard SME approaches.
2. We use the instantiated SMEP-EI to engineer a new methodology by taking SMEP as a base methodology, instantiating it and tailoring it according to the stages of SMEP-EI, to produce our final meta-methodology: S-SMEP.

Asadi and Ramsin's SMEP consists of 3 phase patterns (Initiation, Construction, Deployment), a collection of 8 central stage patterns (Requirements Analysis, Define Infrastructure – for the Initiation phase; Select Construction Blocks, Configure Construction Blocks, Develop Method, Generalization – for the Construction phase; Test in the Large, Deployment – for the Deployment phase) divided into 3 workflows and over 40 task patterns collectively (3 of the additional stage patterns in SMEP are actually task patterns). In our instantiation of SMEP, we retain the structure of the phase patterns, however, we dispense with the workflows and apply the principle of analogy often found in the pattern literature (see, for example, [73]) to adapt the three workflows to a pattern-based context. The resulting SMEP-EI methodology can be briefly described as follows:

**Initiation phase:** In this phase the required infrastructure is set up and requirements are analyzed. In our case, the functional requirements pertain to creating a methodology for realizing the engineering strategies in Sections 3.6 and 4.2; there are no non-functional requirements. The infrastructure used is a base methodology (SMEP).

**Construction phase:** Since there is only one workflow (Pattern-based methodology construction), all tasks and stages pertain to that workflow. The original Assembly workflow's stages and tasks have been merged into an Pattern Assembly stage, concerned with ordering patterns in sequences. We have adapted the original Paradigm workflow's Instantiation/Abstraction task by analogy to become the stage Pattern Instantiation. The Configuration workflow has similarly been adapted to entail customization of the instantiated patterns.

**Deployment phase:** In this phase the methodology is tested and put into production.

In instantiating SMEP using the SMEP-EI methodology outlined above to produce our S-SMEP, we have modified the overall phase structure (initiation, construction, deployment) with the generic process model for assembly-based SME proposed by Saedita et al. [74], which is in turn based on ideas from [75]. In the latter model, methodology construction proceeds in three phases: methodology requirements engineering (including assessing the project...
situation, goals and requirements); methodology design (fragment/chunk selection); and methodology construction (fragment/chunk assembly). The final phase structure of S-SMEP thus consists of 4 phases: *Methodology Initiation* (concerned with methodology requirements), *Design/Modeling* (concerned with using SPPF and the conceptual framework meta-model to design the respective process and conceptual framework models), *Construction* (concerned with instantiating and detailing patterns and model elements to create the final methodology description) and *Deployment*.

Like SMEP, S-SMEP is iterative-incremental: i.e. phases are sequential, stages are iterative, and the overall approach is incremental inasmuch as methodologies are produced/modelled and refined by repeated application of the stages in the *Design/Modeling* and *Construction* phases according to the methodology's requirements.

Since S-SMEP can be used for both methodology analysis and construction, we have defined two (phase-level) workflows for each goal: the *Analysis workflow* applies only the *Design/Modeling* phase; the *Construction workflow*, on the other hand, applies the whole 4 phases of S-SMEP. Within the construction workflow, there are a further two sub-workflows: one for from-scratch construction (process pattern assembly); and another for base methodology tailoring.

We describe S-SMEP below in Sections 6.2 – 6.5 following the order of phases set out above.

### 6.2 Methodology Initiation phase

#### 6.2.1 Requirements elicitation and analysis (stage)

**Construction workflow:**
During the requirements analysis stage of S-SMEP the requirements for a given methodology are elicited from stakeholders and subsequently analyzed – what needs to be secured, what process-related factors (e.g. paradigm) require consideration. Requirements can fall into two categories as for software development: functional and non-functional, whereby the first class determines the security-related functions of the methodology (e.g. shall secure distributed systems) whereas the second class determines additional properties (e.g. shall require minimal effort); or general and situational (cf. [72]; see also [75]). In the context of a flexible methodology, the most important situational requirement pertains to the methodology's applicability to different system types.

This stage is repeated whenever a change is required to the methodology. Requirements analysis can determine both the selection of patterns and the scope of tailoring needed for individual patterns.

### 6.3 Methodology Design/Modeling phase

#### 6.3.1 Selecting and organizing SPPF patterns (stage)

##### 6.3.1.1 Identifying patterns in base methodology (task)

**Analysis workflow** and **Construction workflow – methodology tailoring:**
The description of the existing methodology is studied and analyzed for instances of SPPF patterns, beginning with the phase patterns and proceeding to finer levels of granularity as appropriate. For each instance identified the corresponding SPPF pattern is selected for inclusion in the (concrete) process model. A generic life-cycle modifier is attached to each pattern according to the development phases and/or stages in which it is used.

##### 6.3.1.2 Selecting new patterns (task)

**Construction workflow – “from-scratch” assembly-based:**
SPPF patterns are selected that fit the requirements of the methodology. The phase (security process) patterns dictate the general structure of most security methodologies, and hence define a completeness criterion. Methodologies that do not satisfy the criterion are termed *partial,* but may be useful to cover certain aspects of securing a system. A generic life-cycle modifier is attached to each pattern according to the relevant development phases and/or stages.

##### 6.3.1.3 Constructing an initial security process model (task)

**All workflows:**
The selected patterns are organized according to their encapsulated activities using the workflow constructs described below (as appropriate) to construct an initial process model for the methodology. The model is represented using the notation introduced in Section 5.

Valid workflows for organizing SPPF patterns include: (1) a regular sequence, with one set of activities leading to the next; (2) an unordered collection of activities, performed without a specific order; (3) a set of activities performed in parallel; (4) conditional selection of activities or iteration of activities.

**Analysis workflow** and **Construction workflow – methodology tailoring:**
For flexible methodologies, patterns that have applicability to multiple system-/technology-specific contexts are
marked as being capable of specialization (process elements with a gray background in the notation).

**Construction workflow – “from-scratch” assembly-based:**
When constructing the (abstract) process model, obvious contradictions can be easily avoided (e.g. the AdvMod pattern being placed after the SecMod pattern in a sequence) by loosely adhering to the logical order of SPPF patterns as presented in Section 3.4.

### 6.3.2 Instantiating the conceptual framework meta-model (stage)

**Analysis workflow and Construction workflow – methodology tailoring:**
The model elements from the meta-model in Section 4 are identified in the existing methodology's conceptual framework, and, as for the SPPF patterns, are modeled accordingly using the notation from Section 5. After identifying each meta-model element, or, alternatively, after the whole (concrete) conceptual framework model has been completed, the resulting framework model element(s) have to be linked with corresponding elements from the process model. This is done by considering which pattern instances in the methodology make use of the model's conceptual artefacts, artefact collections etc., and notating the relationship using the dashed arrows (stemming from the process elements to the conceptual model elements).

**Construction workflow – “from-scratch” assembly-based:**
Model elements from the meta-model in Section 4 are selected and instantiated to create an abstract conceptual framework model. Each element should either correspond to real conceptual artefacts that will be used in the methodology (e.g. security patterns, which are instances of the Security solution element; or attack models, which are instances of the Attack element), or to artefacts that will be created later. As for the Analysis workflow, the resulting framework model element(s) are linked with corresponding elements from the process model. We should note that this stage can also be performed prior to the selection of process patterns, in which case the process model will be built around the conceptual framework.

### 6.3.3 Selecting existing methodology elements for tailoring (stage)

**Construction workflow – methodology tailoring:**
Once a concrete process model has been constructed, certain patterns and/or conceptual artefacts from it can be chosen for tailoring according to the methodology construction requirements. There are four approaches to tailoring, which can be realized with the aid of four corresponding **generic operators** inspired by the operator typologies of Ralytė et al. [76]. We describe these operators below with respect to SPPF patterns; their application to conceptual framework elements is analogous.

1. **Replacement:** the chosen SPPF pattern(s), which correspond to actual instances in a base methodology, are replaced with their un-instantiated versions in the model.
2. **Deletion:** the SPPF pattern instance is deleted, and the pattern instances linked to it are tagged for modification accordingly.
3. **Augmentation:** new SPPF patterns are added.
4. **Specialization:** a pattern is tagged for specialization to a more specific context.

The actual realization of the tailoring is left to the Methodology Construction phase.

### 6.3.4 Verifying methodology model (stage)

**All workflows:**
At this stage the relationships between the process and conceptual framework model elements is finalized (if not already done) and checked for inconsistencies (i.e. patterns not correctly linked to meta-model element instances, or linked to incompatible/inappropriate elements). The process model itself is checked for any obviously incompatible pattern combinations or sequences (e.g. SecImpl pattern used during the ReqAn phase), and similarly, the conceptual model is checked for conformance to the meta-model. This may require re-iteration of some of the earlier stages and/or tasks to construct a more complete, refined methodology model.

**Analysis workflow and Construction workflow – methodology tailoring:**
The relationships above are also verified against the existing methodology's description.

### 6.3.5 Generalizing existing methodology elements (stage)

**Analysis workflow:**
During methodology construction, it is possible to incorporate aspects (activities, techniques) of existing methodologies that fully or partially realize a particular SPPF pattern. This requires the generalization of these aspects, consisting in their abstraction and distillation in a methodology-independent, generic form: i.e. in an abstract process fragment. This procedure is strongly related to traditional method chunk identification in assembly-based SME (see 55), however, the resultant abstract fragments are not used directly, but are customized according to
the corresponding SPPF process pattern and instantiated by analogy into the methodology's process model during the Methodology Construction S-SMEP phase. As explained in Section 3.2, the set of all abstract process fragments constitutes a third level of SPPF, alongside the generic life-cycle modifiers (first level) and generic security macro-process patterns (second level).

Abstract process fragments can be stored in a methodbase (as per assembly-based SME – see 23) alongside methodology descriptions to speed up methodology construction. In the latter case the methodbase should be structured according to the individual methodologies so that each pattern is considered in light of the whole approach, rather than as an isolated extraction.

6.4 Methodology Construction phase

6.4.1 Instantiating SPPF patterns (stage)

Construction workflow – “from-scratch” assembly-based:
Using the abstract process model from the Design/Modeling phase, it is now possible to instantiate the patterns to create concrete process elements, which are part of a methodology that can be enacted on a project. The instantiation of patterns can be realized by customizing, specializing, detailing and expanding the activities in the context of the particular methodology's requirements, refining them with each S-SMEP iteration; or by employing abstract process fragments.

Construction workflow – methodology tailoring:
The generic operators are realized as follows:
1. Replacement: the chosen SPPF patterns are customized and re-instantiated into the base methodology as outlined for the “from-scratch” sub-workflow. The effects to any pattern instances linked to the replaced patterns are checked, and the relevant patterns re-specified as needed.
2. Deletion: the SPPF pattern instance is deleted, and the pattern instances linked to it are modified accordingly by re-specifying them. If the changes are too numerous the affected patterns may need to be replaced individually using the Replacement operator.
3. Augmentation: any new patterns added during the Design/Modeling phase are instantiated above.
4. Specialization: a pattern's description is specialized to a more specific context.

This stage can be performed prior to the next one (Realizing conceptual framework model elements), or in parallel with it.

6.4.2 Realizing conceptual framework model elements (stage)

Construction workflow:
The realization of conceptual framework model elements, if they do not exist, implies their construction. Otherwise, it implies specification and organization into structured or unstructured collections that can be used by the corresponding process description elements. Flexibility is an important issue, insofar that the constructed elements must be applicable to different system-/technology-specific contexts. The same generic operators used for constructing the process model apply in this case, with analogous effects. As for the Instantiating SPPF patterns stage, the effects of the tailoring operators on associated model elements and directly linked SPPF patterns should be checked, and any re-specifications and/or further tailoring performed.

6.4.3 Documenting the methodology (stage)

Analysis workflow:
As part of analyzing a given methodology, the original methodology description is re-written with respect to the constructed methodology model from the Design/Modeling S-SMEP phase. Each element in the model is documented individually, beginning with the process elements in the sequential order of activities, and following with descriptions of the conceptual framework elements. Process elements should conform to a simple textual template, consisting of a Description field and a Conceptual artefacts field for documenting which artefacts are used by that element (as inferred from the dashed arrows in the model).

6.4.4 Verification (stage)

Construction workflow:
Verifying SPPF pattern instances:
In all customizations, methodology engineers should aim to preserve the conceptual integrity of the original patterns (i.e. their main ideas), in keeping with pattern instantiation in general. Unlike software (product) patterns, which have direct architectural effect via representative architectural models for example (see 63 or [20]), making it easier to distinguish when a particular customization of the pattern has supplanted the original pattern ideas embodied in the models, with process patterns it is not always so clear when a customization is in fact another pattern instance.
Hence, the preservation of conceptual integrity for process patterns must inevitably rely more on careful comparison and sound judgement.

Verifying the methodology as a whole:
After the methodology has been constructed (at a given iteration of S-SMEP), its soundness must be verified, first of all as a security methodology, i.e. is it sensible from a security viewpoint, can it really secure a system etc. The methodology must then be verified in the context of the generic life-cycle – do the activities contained in the SPPF pattern instances match the overall development paradigm well, is their integration with existing development activities sensible etc. It should be noted that fine-grained issues of integration are not considered in this paper (see Section 9 on Future Directions). Finally, the concrete relationships between the process and conceptual framework aspects must be checked to ensure that there are no incompatibilities and that all conceptual artefacts are appropriate for the given security process.

6.5 Methodology Deployment phase

Construction workflow:
During this phase a security methodology is prepared for deployment, i.e. for enactment on real project endeavours (following SMEP’s Deployment phase). This requires testing in the large, including integration testing with a corresponding development methodology. The final stages of deployment entail documenting the methodology and introducing it into the target particular project environment, training developers etc.

7 Applying S-SMEP to engineer a security methodology

In this section we illustrate and evaluate our engineering approach by applying the S-SMEP Construction workflow (tailoring sub-workflow) to extend and adapt the existing security methodology of Fernandez and colleagues [4] (see [14] for a comprehensive overview) to a project situation requiring applicability to both general distributed and peer-to-peer system-specific contexts. The latter methodology employs security patterns (full inheritors of the software pattern concept – see [77, 20]) within the object-oriented development paradigm. In presenting this case study we follow the order of S-SMEP phases, beginning with Initiation, then Design/Modeling and finally Construction. We do not consider any phases related to verification or deployment, which are outside the scope of this paper. In each phase, we outline the main results of applying the various S-SMEP stages and tasks, in greater or lesser detail as required.

7.1 Methodology Initiation phase

7.1.1 Requirements Elicitation and Analysis
The first step in tailoring the base methodology is to determine the set of (methodology) requirements. The foremost requirement relevant to our case study is the applicability situational requirement of using the methodology for general distributed systems and also for peer-to-peer systems, in each case accounting for the relevant system features. Secondary to this are a set of requirements arising from the (security methodology) taxonomy dimensions presented in [14] and the evaluations presented in that work. This also determines the order of presenting the requirements below.

Req1 (related to methodology paradigm): Focus on a system's software architecture. This implies the use of models in some fashion, and thus a model-based paradigm.
Analysis: This requirement implies that an instance of the Security Modeling stage pattern should be part of the final methodology, and that the corresponding activities should be related to the architectural models used for development. Furthermore, security should be introduced with relation to the models, or at least taking them into account in some fashion.

Req2 (related to methodology specificity): Explicit provision for distribution. The methodology of Fernandez and colleagues already has general provision for distribution, but not specific support [20]. Such support may take on many forms, such as the inclusion of security solutions specific to distributed settings, specific guidance or activities related to particular types of distributed systems, generating configurations for underlying distributed middleware etc. The methodology should also be able to account for the specific features of peer-to-peer systems.
Analysis: This requirement affects the Countermeasure Introduction phase pattern, since any explicit support for distribution implies a selection of applicable solutions (given Req1 above). More specifically, distribution concerns should be accounted for during the Design (des) development phase, although it is possible to make some initial preparations during Analysis (an) as well.

Req3 (related to methodology modeling language & notation): Use of a well-known and/or readily comprehensible modeling language and notation. Since familiar modeling concepts imply increased ease of use, and may even determine the applicability of a methodology, modeling notations such as UML or less formal ADLs should be used.
Analysis: This requirement implies that solutions used as part of the Countermeasure Introduction phase pattern instance (during design at least) would either be represented as or be compatible with the required modeling
notations. This is already satisfied by the methodology of Fernandez and colleagues, which employs UML for use cases and security patterns with (representative) UML models.

**Req4 (related to methodology use of formal meth. & verification):** Localized and/or optional explicit use of formal methods. There should be some way to verify that the introduced security countermeasures protect the system. Formal methods, including interactive model checking, theorem proving etc. should be confined to the security verification stages.

**Analysis:** This requirement implies that an instance of the Security Verification stage pattern should be part of the final methodology. 

**Req5 (related to methodology SDLC scope):** Guidance during every stage of the SDLC on where and how to introduce security solutions. The ultimate aim of a (security) methodology is to guide and support developers in introducing security into a software system, and this should be reflected in all activities.

**Analysis:** This requirement implies that guidance should be reflected in a well-defined Adversary Modeling stage, with consideration of threats (enumerate threats task – thereby justifying the introduction of certain security properties), and their mapping to corresponding security countermeasures via a comprehensive Countermeasure Introduction phase. More specifically, any instance of Countermeasure Introduction should also instantiate a Countermeasure Identification stage pattern that provides a guided or structured approach.

**Req6 (related to methodology supported security properties):** Support for a wide range of security properties, including properties for at least distributed authentication and identity management, distributed access control, communication security, data filtering and auditing. Ideally, the introduction of security solutions for every major class of security requirements as found, for example, in [78] should be supported as well, tailored to distributed settings where appropriate.

**Analysis:** This requirement, like Req2, affects the Countermeasure Introduction phase pattern and in particular the Security Modeling stage, whereby as part of the activities developers should be able to apply a range of solutions (as outlined in the requirement) and relate them to any relevant architectural models.

### 7.2 Methodology Design/Modeling phase

#### 7.2.1 Selecting and organizing SPPF patterns (stage)

**7.2.1.1 Identifying patterns in base methodology (task)**

**Analysis workflow and Construction workflow – methodology tailoring:**
The base methodology of Fernandez and colleagues was studied and analyzed for SPPF pattern instances using the Analysis workflow.

**7.2.1.2 Constructing an initial security process model (task)**
The selected patterns were organized according to their encapsulated activities to construct the initial (concrete) process model, shown in the left-hand side of Figure 5. Nearly all patterns were arranged in sequential order, as befits the nature of the methodology, with the only exceptions being the stage patterns in the Security Implementation phase, whose activities are performed without order. The methodology's security process does not contain any flexible elements (cf. Section 2.2).

#### 7.2.2 Instantiating the conceptual framework meta-model (stage)

Elements from the meta-model in Section 4 were identified in the existing methodology's conceptual framework, and, as for the SPPF patterns, modeled accordingly using the notation from Section 5 to construct the conceptual framework model shown in the right-hand side of Figure 5. The resulting framework model elements were linked with corresponding elements from the process model in accord with the methodology's description to create the finalized (initial) methodology model. In Tables 2 and 3, we briefly describe the corresponding SPPF pattern instances at a phase level of granularity, and a selection of relevant conceptual artefacts, respectively.
Figure 5: Initial model of the pattern-driven security methodology of Fernandez and colleagues
Security process pattern | Generic life-cycle modifier | Security process description
--- | --- | ---
SecReq | Req | Consideration of security starts from the requirements development phase with an analysis of use cases, individually or in sequence (workflows), to determine a set of possible threats and thereby derive a set of security requirements (AdvMod); and to determine a set of minimal rights for each subject (or role) for access control. The threats used to determine how the use case activities can be subverted rely on expertise.
CounterIntro | An | Secure Semantic Analysis Patterns (SAPs [79]) are used during the analysis phase, where appropriate, to create a secure conceptual model efficiently. The use case analysis results from before are carried over to determine a set of abstract security patterns that collectively embody a subset of the security requirements. These include at least an overall access control model, which is realized by repeated instantiations of a desired (abstract) access control pattern, realizing the authorization rights per subject (or role) determined from before.
SecReq | Des | Security attacks are considered to the software architecture during design. A (superimposed) layered architecture is used to separate the system into areas of concern, and misuse patterns or expertise (documented attacks) are employed for adversary modeling (AdvMod), which generates an additional set of security requirements.
CounterIntro | Des | The total set of security requirements from both instances of the AdvMod pattern are mapped to appropriate groups of security patterns manually to secure each software layer, and to create an overall security architecture. Security patterns are propagated down the different software layers, giving rise to a set of concrete countermeasures that are required to collectively participate in enforcing the initial requirements [7, 80]. To achieve this the security patterns must be mapped through their respective layers [4]. Distribution is considered as a special layer of the hierarchical architecture and is thus secured, like other layers, using patterns.
SecImpl | Impl | During security implementation, the necessary countermeasures (as patterns) are realized as software units or COTS components.

Table 2: Identified security (phase) process patterns in the base methodology of Fernandez and colleagues

<table>
<thead>
<tr>
<th>Conceptual security artefact</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misuse patterns</td>
<td>Misuse patterns [81] describe in detail how an attack is performed (including what software components are required) from the attacker's perspective, which patterns can mitigate the attack, and how the attack can be traced once it occurs.</td>
</tr>
<tr>
<td>Security patterns</td>
<td>Security patterns encapsulate countermeasures in a re-usable format suitable for use during architecture development.</td>
</tr>
<tr>
<td>Security architecture</td>
<td>The set of all security measures at the design level together constitute a security architecture, i.e. a slice of the software architecture dealing with security.</td>
</tr>
<tr>
<td>Semantic analysis pattern</td>
<td>A type of analysis pattern [73], which combines a ready-made domain/business design with (abstract) security patterns.</td>
</tr>
</tbody>
</table>

Table 3: Identified conceptual security artefacts in the base methodology of Fernandez and colleagues

7.2.3 Selecting existing methodology elements for tailoring (stage)

**Construction workflow – methodology tailoring:**

Now that we have constructed a concrete process model, we can identify patterns and/or conceptual artefacts for tailoring according to the requirements in Section 7.1.1. We only need to consider SPPF patterns and conceptual framework artefacts used at the Design (des) phase, since this is where the effects of distribution are (fully) manifest. Considering the methodology requirements once again, we note that **Req1** and **Req3** are already fully satisfied by the base methodology, while **Req4** and **Req5** are partially satisfied. To fully satisfy all requirements, therefore, we apply the generic operators as follows:

- **Replace** (operator) is applied to the **AdvMod** stage pattern at design (des). This implies that the encompassing **SecReq** pattern will need to be modified with the **Specialization** (operator) accordingly.
  - **Replacement** (operator) is applied to the **Misuse patterns** conceptual artefact, replacing them instead with the flexible artefact of **Threat taxonomies** (see [82]), which is applicable to general distributed systems and to peer-to-peer systems in equal measure. The threat patterns contained in threat taxonomies are related to misuse patterns, so the model element will be left.
  - **Deletion** (operator) is applied to the Attack list conceptual artefact.
  - **Specialization** (operator) is applied to the **Security pattern catalog** conceptual artefact, to give it additional structure (realized during the **Methodology Construction S-SMEP** phase).
  - **Replacement** (operator) is applied to the **Layered architecture** conceptual artefact, replacing it with a new
Distributed architectural decomposition framework artefact described in [83], which is specifically constructed for characterizing distributed software architectures.

**Req4:** Augmentation (operator) is applied to the SecImpl phase pattern, adding a new SecVerif stage pattern for testing the threats found during design.

**Req5:** Replacement (operator) is applied to the CounterIdentif stage pattern.

**Req6:** Augmentation and Specialization (operators) are applied to the Security pattern catalog conceptual artefact.

The actual realization of the tailoring (modification of the models, re-linking of elements etc.) is left to the Methodology Construction S-SMEP phase.

### 7.2.4 Verifying methodology model (stage)

At this stage we check the models against the existing methodology's description, and also verify (as much as possible during this phase) that the operators applied above will produce sensible results.

### 7.3 Methodology Construction phase

#### 7.3.1 Instantiating SPPF patterns (stage)

At this stage we can implement the operators from the Methodology Design/Modeling S-SMEP phase for the SPPF patterns as follows:

- **Replacement (operator) on the AdvMod stage pattern at design (des):** Adversary Modeling now proceeds by decomposing a distributed software architecture and considering threats for each area of functionality using the threat taxonomy, thus retaining, but customizing, the previous attacks task pattern. The use of threat taxonomies also allows threats to be systematically enumerated, giving rise to an instance of the enumerate task pattern. Since the threat taxonomy for distributed systems in [82] contains a meta-security level, we need to re-iterate the AdvMod stage (once only) to consider threats with respect to introduced countermeasures; therefore, we insert an iterative workflow construct between the CounterIntro and SecReq at Design pattern instances.

- **Replacement (operator) applied to the CounterIdentif stage pattern:** we decide to implement this change by realizing the CounterIdentif with an abstract process fragment from the third SPPF level. We assume that these have been stored in a repository, and have been created previously using the Generalization stage of the S-SMEP Analysis workflow. The particular fragment is abstracted from two methodologies: Rosado et al.'s PSecGCM / SecMobGrid [11]; and the SOA-based methodology of Delessy and Fernandez [84], and is briefly presented in Appendix A.2. We customize and instantiate this fragment, linking it simultaneously to the Distribution-aware classification scheme, which is used in [20] to structure security patterns, and in this case helps to realize the mapping pairs as (concern, security patterns).

- **Augmentation (operator) applied to the SecImpl phase pattern:** adds a new SecVerif stage pattern: the verification is based on using misuse patterns that realize the threat patterns to test whether all threats have been covered.

#### 7.3.2 Realizing conceptual framework model elements (stage)

The generic operators for tailoring are similarly realized for the conceptual framework artefacts by replacing the elements and re-linking the models. For example:

- **Augmentation and Specialization (operators) applied to the Security pattern catalog conceptual artefact gives rise to the Distributed security patterns catalog, and implies that the methodology will use the collection of patterns appearing in [20] for general distributed systems organized according to the Distribution-aware classification scheme, as well as patterns for peer-to-peer systems. The specifications of the corresponding Countermeasure Identification (CounterIdentif) and Security Modeling (SecMod) patterns remain unchanged after applying the operators.

We also realize the various element replacements (e.g. replacing the Misuse patterns conceptual artefact with the flexible Threat taxonomies artefact, which also implies the removal of the Attack list artefact; as well as the replacement of the Layered architecture conceptual artefact, with the Distributed architectural decomposition framework artefact for characterizing distributed software architectures), considering in all cases how the new elements will be used in the corresponding process pattern instances, and re-specifying the latter where needed.

The final methodology model is shown in Figure 6. The colours (or shades of gray) imply elements that have been replaced (green), specialized (yellow) or added (cyan). The Threat taxonomy conceptual framework artefact is a flexible element and hence appears in gray background, even though it, too, is new; similarly, the CounterIdentif stage pattern, which is realized using an abstract process fragment, appears in black background. Brief descriptions of the tailored Security Requirements Determination (SecReq) phase pattern, the new Security Verification (SecVerif) stage pattern, and several conceptual framework artefacts appear in Tables 4 and 5, respectively.

<table>
<thead>
<tr>
<th>Security process description</th>
<th>Generic life-cycle</th>
</tr>
</thead>
</table>

118
### Table 4: Selected conceptual security artefacts in the tailored security methodology

<table>
<thead>
<tr>
<th>Threat taxonomy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threat patterns</td>
<td>[82] capture security threats (potential attacks) and allow developers to consider those threats in well-defined architectural contexts.</td>
</tr>
<tr>
<td>Threat taxonomies</td>
<td>The (pattern-based) threat taxonomies of Uzunov and Fernandez [82] aggregate structured collections of threat patterns for general distributed systems as well as (derived) patterns for specific distributed system/technology contexts.</td>
</tr>
</tbody>
</table>

### 7.3.3 Discussion

It should be noted that in making all the modifications to the base methodology as per the tailoring workflow, we aimed to retain the existing features of the methodology that satisfied the requirements in Section 7.1.1, especially Req1 and Req3, e.g. the use of UML in all relevant models. As per the definition in Section 2.2, the resulting methodology is a flexible methodology, due to its applicability to both general distributed and peer-to-peer systems, a characteristic which stems from the use of flexible conceptual artefacts; the activities remain the same.

Further refinements of the methodology would imply iterations of S-SMEP, and could include improved threat modeling processes as well as new solution types (e.g. security solution frames [85]), or the incorporation of features from other methodologies. Indeed, the model in Figure 6 can be a good basis to determine the appropriateness of adopting the methodology for general distributed systems via comparison with other methodology models.

Since the methodology presented in this section does not merely serve the purpose of illustration (i.e. it can be used on real projects), we should point out one feature requiring development (outside the scope of this paper), namely, a collection of security patterns specific to peer-to-peer systems that would relate to the high-level mitigations of the corresponding threat patterns in [82], used during the Countermeasure Introduction phase. Prior to deployment, the methodology models also need to be fully specified and documented (see the Documentation stage of the Analysis S-SMEP workflow in Section 6.4.3).

### 7.4 Feature analysis/screening of the approach

As the case study in this section has illustrated, our approach to engineering security methodologies – embodied in S-SMEP – satisfies the initial requirements and central need for such an approach set out in the Introduction (Section 1). Based on these requirements, it is possible to collate a set of desirable features that support a feature analysis/screening of any approach to engineering security methodologies. We enumerate these features below, and following this, discuss explicitly how the features are implemented in the approach presented in this paper. As there are no other comparable approaches for engineering security methodologies in the literature (this being the first), the enumerated features are, in a way, provisional and more abstract, however, they can form a basis for more detailed feature sets as new approaches are proposed in the future, and be refined as the current approach is used more widely in practice (cf. Section 9 on future work).

- **broadness of applicability**: the approach should be applicable to a large range of security methodologies. To be of practical use, it should support at least the engineering of a methodology's security process aspect.
- **systematicity and guidance**: the approach should provide systematic guidance for developers to engineer methodologies, in the form of rules, guidelines, processes etc.
- **ease of use**: the approach should be easy to use, preferably with software tool support to reduce effort and increase productivity, and use well-known notations and/or techniques familiar to developers.
- **re-use of existing knowledge**: the approach should allow for secure software engineering knowledge, as embodied in existing security methodologies, to be applied and re-used during the engineering of (new)
methodologies. In particular, there should be support for re-using features of existing methodologies across other methodologies in a flexible manner, preferably using common knowledge repositories or catalogs of security activities and/or artefacts.
The approach to engineering security methodologies presented in this paper encompasses the latter features as follows:

- **broadness of applicability**: Our engineering approach – as embodied in S-SMEP – is applicable to all security methodologies, regardless of their specificity or flexibility, and allows for the engineering of both a methodology’s security process and conceptual security framework aspects.

- **systematicity and guidance**: S-SMEP provides a structured, systematic, step-wise process for using the two core artefacts – SPPF from Section 3 (for a methodology's security process aspect) and the meta-model from Section 4 (for a methodology's conceptual framework aspect) – both for methodology construction and analysis.

- **ease of use**: The guidance provided by S-SMEP, together with the use of the pattern paradigm (SPPF) and UML-based modeling (conceptual framework meta-model), helps to increase usability and reduce the learning curve required for using the approach. There is no software tool-support as yet (see Section 9 on future work), however, since the notation used for methodology models is based directly on UML (with which engineers are almost invariably familiar), any UML editor with support for activity nesting can be employed.

- **re-use of existing knowledge**: The patterns forming the second SPPF level represent a mixture of security best-practices and activities distilled from a number of existing security methodologies in an abstract, reusable format. Various techniques, activities etc. of existing methodologies can also be generalized as abstract process fragments using S-SMEP, which can then be applied by analogy to realize the constituent patterns (instances) of a new methodology during construction. This paper catalogs SPPF patterns, however, there is currently no common repository or database of abstract process fragments or specially-structured methodology descriptions (see Section 9 on future work).

- **support for re-engineering**: S-SMEP can be used in its tailoring Construction workflow to re-engineer methodologies, whereby the pattern and meta-model element instances that make up a methodology can be replaced, augmented, deleted etc. by different pattern instances.

- **flexibility**: Using S-SMEP, SPPF patterns and fragments can be combined or assembled into various workflows and instantiated to construct a methodology's process elements at varying degrees of abstraction and specified detail. Similarly, the meta-model elements can be extended prior to instantiation, and also freely associated (within what is deemed sensible), following instantiation.

- **support for structured descriptions**: The use of methodology models conforming to the SPPF framework and conceptual framework meta-model allows methodology structure to be expressed clearly, as befits a good software process from an industry viewpoint (cf. [16], also Section 1 – Introduction).

- **support for comparison and analysis**: The methodology models constructed using S-SMEP's Analysis workflow, together with the brief accompanying descriptions of major elements, provide a uniform basis for comparison between different methodologies. Since SPPF patterns have known semantics, it is possible to analyze the features of a given methodology simply from its model, or using its model as a base.

8 Related work

In this section we systematically compare our approach to related work the subject. Since the engineering of security methodologies has never been studied before, and, for that matter, the engineering of a methodology's conceptual security framework aspect, we concentrate solely on a methodology's process aspect with emphasis falling on the similar features of general method engineering approaches using process patterns. This also helps to highlight some of the distinctions.

8.1 Security process patterns and method engineering

The first serious, though limited, attempt to apply process patterns to security in general can be found in the work of Liu et al. [86]. Liu et al.’s patterns (which do not use a standard template) constitute a set of guidelines for security
requirements elicitation restricted to a very narrow domain – pervasive healthcare. They could be used to refine instances of our Security Requirements Determination (SecReq) phase pattern from Section 3.4.2, albeit only in the context of healthcare systems.

The work of Wagner et al. [87] represents the only known attempt to use security process patterns in some fashion for security in a method engineering context, albeit for development, not security, methodologies. In that work the authors seek to tailor RUP by incorporating high-level security patterns from [77], which are related to SSE-CMM process areas (PAs) (see [61]), into the development process as a single “Software Security” discipline (using RUP terminology). While most of the patterns used by Wagner and colleagues are actually specific security macro-process patterns, some of them really are proper security (product) patterns, which makes for a rather unusual collection of process elements and conceptual security artefacts. The patterns (or, one assumes, those of them that refer to activities) are described using concepts from a process meta-model (PRIMA-M). Wagner et al.’s work does not aim to be either general or comprehensive, and its alignment with SSE-CMM ensures an overwhelming managerial, as opposed to technical, bias.

As far as we are aware, and notwithstanding the collection of patterns of Wagner and colleagues, there is nothing really comparable to SPPF in the literature.

8.2 Engineering development methodologies using process patterns

There are a number of related approaches to methodology creation, analysis and tailoring with process patterns outside the field of security, most of which are concerned with formalization aspects and the representation of process patterns using UML-derived notations (see also further below). Most relevant among these is the work of Huang and Zhang [88], who propose the concept of Hierarchical Process Patterns (HPP). HPPs are classified into three types: life-cycle patterns, which describe the overall structure of a software process according to two classification dimensions (linear/evolutionary and sequential/concurrent); activity patterns, which describe activities in a software process; and workflow patterns, which describe the organization of the activity patterns in time. Huang and Zhang also propose a simple meta-process within their CAME (Computer Aided Method Engineering, see [22]) environment SPDM, consisting of three stages – pattern selection, pattern instantiation and instance configuration (allocating time-slices, people to tasks etc.). In contrast to HPPs, our process patterns are strictly based on the original idea in [36] (with the original segregation of patterns into phase, stage and task varieties) and allow descriptions of varying degrees of abstraction and granularity. The application of S-SMEE to assemble patterns in different sequences can be seen as an application by analogy of the Sequence, Cycle, Parallel Split, Join etc. workflow patterns of [89]. Life-cycle patterns have no analog in our work.

Another relevant approach can be found in [90], whose work is concerned with modeling process patterns using precise UML meta-models. Tran and colleagues propose three levels of process pattern abstraction – abstract, general and concrete, where abstract patterns correspond to Huang and Zhang's workflow patterns, describing process structures; general patterns describe specific aspects such as artefact structures (e.g. composition of documents); and concrete patterns correspond to process fragments. The main aim of the approach is to provide a semi-formal basis for defining process elements and for determining different process organizations (workflows). This aim leads to patterns providing formal parameters, which are bound upon instantiation with actual process elements (specified in UML activity diagrams) from particular (concrete) patterns to realize process descriptions. Different process organizations are realized by mapping process elements to a particular workflow structure embodied in an abstract pattern. In the latter respect the approach taken by Tran and colleagues is close to that of Huang and Zhang described above, except that the view of patterns is analogous to formalized software patterns, i.e. they are not really patterns, but parameterized model templates. In contrast to the work of Tran and colleagues, our work does not employ a formal approach, which would be inherently inflexible. Formality imposes rigour, but also the potential for conflicts (cf. [90]). Our use of parameters, therefore, while similar in aim to the formal parameters presented in [90] and earlier in [91], are not bound to certain phase or stage patterns during methodology creation or analysis in a template-like sense.

Other method engineering treatments of process patterns in the manner of fragments or chunks, e.g. in the work of Gnatz et al. [26, 92] as well as in [93], do not explicitly consider aspects of customization inherent to patterns, or different degrees of abstraction or granularity, i.e. patterns are seen more as static, pre-made components that can be fitted together in the manner of CBSE. Perhaps due to this perspective, a number of researchers have found the need for concepts such as specializations of process patterns, compound patterns, uses relationships etc. (see [93, 90]). In the context of SPPF, the latter relationships are unnecessary, since the second-level patterns represent a “closed set”; however, similar relationships could become valuable for specific macro-process patterns or abstract fragments realizing the generic SPPF patterns.

8.3 Mining process patterns

With respect to mining process patterns, a related approach can be found in [56]. The aim of that work is to provide a method for distilling domain-specific development methodology knowledge in the form of patterns, which (according to the authors) can be used in assembly-based SME contexts. In constructing SPPF we employed inductive approaches similar to those presented by Gholami and colleagues, but also applied a deductive approach
by utilizing sound secure software engineering principles (see Section 3.4.1), and not any sort of data-mining based algorithms. Clustering based on mere name similarity and other mechanical strategies were not employed – in this regard we are convinced that, due to the nature of patterns (which are not components, fragments or chunks), pattern mining strategies require both domain expertise and human guidance at every step to produce carefully-crafted, optimal patterns and to avoid potentially unwieldy, rule-generated patterns.

8.4 Pattern-based process modeling notations

With respect to notations for pattern-based process models, a number of authors have proposed graphical notations based on UML activity diagrams, e.g. [94] – PPDL, [93] – PROPEL, [88], [37], [90]. Only the notation of [88] is capable of addressing pattern nesting, and even then in a rather imperfect fashion (using several diagrams). Our notation, which adapts the notation of Van de Weerd and Brinkkemper [71] (also inspired by UML) is capable not only of representing pattern nesting, but also conceptual security framework models. The addition of textual constructs is a feature peculiar to our approach.

8.5 Relation to traditional SME approaches

8.5.1 Assembly-based SME in the context of security

The sole closest work to ours with respect to SME in the context of security is contained in [95], where, building on earlier research in [51], the authors break the Secure Tropos methodology [58] into OPF fragments (OPEN Process Framework – see [96]; see also [41]) to be used in the assembly-based construction of agent-oriented development methodologies. Despite the seemingly clear relation with our “from-scratch” strategy, Low and colleagues focus only on a single methodology and a single paradigm in a development methodology context, which, by itself, is insufficient with respect to constructing security methodologies. Our pattern-based approach should not be seen as exclusive, however. In particular, Secure Tropos OPF fragments could easily be used as a basis for abstract process fragments realizing some of our SPPF patterns.

8.5.2 Other SME approaches outside of security

8.5.2.1 Meta pattern based techniques

The so-called pattern-based SME strategy best explained in ([97]; see also [30]), which uses semi-formalized, high level guidelines for the construction of process or product models according to a meta-model is, at a general level, another related engineering approach, which should not, however, be confused with our use of process patterns. In light of patterns providing guidance in methodology construction, SPPF patterns can be seen as generators of concrete activities, tasks etc., however, this does not make them method construction patterns or meta-patterns as in the work of [43] or [44].

8.5.2.2 Meta-model instantiation techniques

In some respects our engineering approach for the process aspect of a security methodology can also be likened to meta-model instantiation, whereby SPPF patterns can be thought of as models of actual methodology components. This is quite distinct, however, from meta-modeling as used in SME (cf. [35, 68]), in which meta-model elements offer quite simple semantics (and in fact can be likened more to pattern template elements than to the SPPF patterns themselves).

8.6 Related development meta-methodologies

With respect to meta-methodologies for development methodologies, a different approach based on meta-modeling can be found in Engels and Sauer's MetaME [21], which has similar aims to Asadi and Ramsin's pattern framework [72] on which our S-SMEP is based. Unlike Asadi and Ramsin's work, the meta-methodology of Engels and Sauer proposes a whole engineering process based on the 4-layer OMG modeling stack, of complexity almost comparable to that of the created methodology itself. MetaME is based on a strict product/process model distinction (common to much of SME – see [24]), which is not as applicable to security methodologies as it is to software development methodologies (see Section 2.1).

8.7 Security process frameworks

Outside of method engineering, but within secure software engineering, Cabot and Zannone [98] present a process-based framework for Model-driven Security (see [99] or [14]) that attempts to unify various MDA/MDE-based security approaches. The result is a set of requirements and general guidelines/processes that can/should be
performed in model-driven development contexts. Cabot and Zannone's work is not, however, based on method engineering approaches and is strictly tied to the MDA/MDE paradigm. In a sense, Cabot and Zannone's framework can be seen as a particular abstract security process model tailored for model-driven development.

8.8 Security methodologies employing process modeling

In relation to specifying specific security methodologies using process modeling approaches we should mention PSecGCM (later MobSecGrid) [11], which is described using OMG's SPEM 2.0 (Software & System Process Engineering Metamodel – see [100]). The aim of Rosado and colleagues is not, however, to apply method engineering or analysis techniques, but rather, to formalize the PSecGCM security methodology and to provide a detailed, rigorous description.

9 Conclusion and future directions

In this paper we presented a comprehensive approach to engineering security methodologies that takes into account both a methodology's security process aspect, via a framework of interrelated process patterns and modifiers (SPPF – Section 3); and a methodology's conceptual security framework aspect, via a specific meta-model (Section 4). Our approach directly addresses the issues outlined in the Introduction (Section 1) by providing a means to tailor existing security methodologies and to create new “fit-to-purpose” methodologies – all using a common basis suitable for methodology modeling, analysis and comparison. Our proposed meta-methodology (S-SMEP, Section 6) provides simple, step-by-step guidance to achieving the latter aims, which was demonstrated in some detail in Section 7 by applying S-SMEP to tailor an existing, real-life security methodology. We believe our approach fills an important gap in the secure software engineering literature, and can become a catalyst for a more ubiquitous adoption of systematic security approaches, or at least features thereof, in a variety of development contexts.

9.1 Current and future evaluation

Regarding the evaluation method used, as stated in the Introduction, we followed the approach of Hug et al. [68] utilizing a real-life example as a case study (Section 7), together with basic feature analysis/screening (Section 7.4 – cf. [104]). This was supplemented with a short comparative review of related work (Section 8). While in a certain sense our evaluation approach can be seen as “synthetic” – i.e. S-SMEP is not applied directly in an industrial environment, but in an academic environment – it is no less valid, and is comparably efficacious in illustrating the value of the work in general and indicating its benefits in practical situations. Industry evaluation using case studies, questionnaires or other evaluation techniques (see [104]), can also be valuable in determining the scope and further confirming the value of the approach; above all, they could provide – beyond the current evaluation – a number of “lessons learned” that may be used to extend and/or refine any of the constituents of our engineering approach: the SPPF framework (new patterns, specification of patterns), the meta-model (new elements) or S-SMEP itself (new or different steps). It is important to note in this respect, however, that, given our engineering approach fundamentally resides at the meta-level with respect to the security methodologies produced, single-time, industry-based case-study approaches cannot form a complete (additional) evaluation solution. In particular, a single study will not demonstrate anything more than our current evaluation (which, in this context, is equivalent); a number of such studies for a variety of project situations would be required – something that is not an easy undertaking and not always a viable option. A further complication with industry-based case studies is that there are a range of factors introduced in industrial environments [120, 121], including a bias for specific security techniques vs. objective, project-suited techniques (cf. [122]), which will inevitably encumber the evaluation process as a whole.

Another avenue for further evaluation is the use of detailed or combined feature analysis techniques [123]. One significant hindrance in our context, however, is the fact that engineering security methodologies is a "first-time" proposal, and there are currently no industry guidelines from which to obtain specific criteria. This implies that feature analysis will need to be supplemented with secondary techniques, and in any case follow studies to obtain desirable features and metrics.

It is important to note that there are no well established approaches for evaluating SME-based meta-methodologies in the literature (cf. [24]) – a point which, when combined with the foregoing discussion, implies that evaluation beyond what is presented in this paper is a significant research problem in its own right. While it is clearly not possible to address this problem within the confines of the present paper, an exploration of possible solutions forms an important research direction, which would supplement, and possibly extend and deepen, our current evaluation approach.

9.2 Additional research directions

Regarding other important future directions, one related area of SME research that we have not pursued in this paper is that of methodology quality analysis (strongly related to the S-SMEP Verification stage). In fact, there is very little work done with respect to ensuring that engineered methodologies (of any type) possess qualities such as completeness and consistency (cf. [24]). Brinkkemper et al. [40] are among the few researchers to propose tangible
assessment criteria in the form of twelve rules governing assembly-based (development) methodology creation. The rules, however, are more specific to the SME approach of Brinkkemper, Harmsen and colleagues [22, 23], and are neither general, nor fully interpolatable to different (e.g. security) contexts.

Another important direction opened by our work in this paper is to explore avenues of integration between security and software development methodologies in detail. Our use of generic life-cycle modifiers can be seen as a first step, determining the alignment of security activities with different development phases at a large granularity and high abstraction level, which could facilitate the analogous consideration of finer-grained aspects of integration at lower levels of abstraction. Extending and refining the generic life-cycle with paradigm-specific process patterns (e.g. based on the work in [101, 102]) is another possibility.

9.3 Software support

For the case study described in this paper, S-SMEP was applied without the support of any specialized software tools. Indeed, applying the approach in practice only requires a UML-diagram editor for the documentation of methodology models, and some degree of secure software engineering knowledge for realizing SPPF patterns and meta-model elements. The latter requirement, in particular, could be made less onerous given a searchable database (methodbase) of abstract process fragments from the third SPPF level, indexed by different criteria, as well as actual methodology descriptions linked to respective patterns and/or meta-model elements. This would not only provide examples of how SPPF patterns can be realized using different approaches, but also give developers direct model elements they can use to construct and combine different methodology features, reducing the time/effort requirements where appropriate. Relating fragments and patterns with potential realizations in the research literature could also help to bring more research approaches into industry contexts.

In a similar vein, the development of a comprehensive CAME tool (see [22, 124]), in conjunction with the latter database, can be an important addition to improving the ease of use and efficacy of applying S-SMEP. The recent use of MDE as a basis for CAME tool support has already been shown to be of value for assembly-based SME approaches in a development methodology context [125, 126], and could be an interesting avenue for exploration.

Appendix A: Third SPPF level: abstract process fragments

In this appendix we briefly present two abstract process fragments from the third SPPF level in condensed format (see Section 3.2 for our rationale).

A.1 Decomposition-based threat modeling (Stage granularity, [83])

**Context:** When realizing Adversary Modeling (AdvMod) for general distributed systems.

**Problem:** How to determine a collection of security requirements based on threats.

**Solution:** A derived architectural model is constructed using the decomposition framework of [83] (realizing the model task pattern). This process uses the constructs provided by the framework's three levels (distributed systems conceptual model; functionality decomposition layers; technical realization abstractions). Threat analysis (realizing the enumeration task pattern) proceeds by considering security threats – based on expertise – at each functionality layer and for all the realization abstractions, potentially assigning risk values as necessary (risks task pattern).

A.2 Table-based policy mapping (Task granularity, [11, 84]):

**Context:** When working with security policies and/or requirements.

**Problem:** How to transition from one set of policies and/or requirements and another.

**Solution:** Security requirements (in the problem-space) or policies (in the solution-space) are mapped to countermeasures using a table of pre-defined pairs (requirement/policy class, countermeasure class). Finer-grained requirement/policy mappings are also possible, whereby a table maps or translates one set of requirements/policies or requirement/policy classes to another, thus creating a kind of “adapter”. All the mapping pairs in the table must be mined from the literature.

References


Epilogue

In this chapter we presented the core of our “toolkit” and comprehensive approach to engineering security methodologies, embodied (in this chapter) in three parts: a framework of (security) process patterns; a (security methodology) domain-specific meta-model; and a unifying meta-methodology. The approach was illustrated and evaluated by constructing the case study methodology for this thesis.

In constructing the case study methodology, several conceptual security artefacts were used that have not yet been presented. These artefacts are also essential parts of the “toolkit” for engineering security methodologies, and are the subject of the four subsequent chapters.
Chapter 4

(Generic conceptual security artefacts I)

Framework for decomposing distributed software architectures

All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can’t get them together again, there must be a reason. By all means, do not use a hammer.

— IBM Manual, 1925

Programming without an overall architecture or design in mind is like exploring a cave with only a flashlight: You don’t know where you’ve been, you don’t know where you’re going, and you don’t know quite where you are.

— Danny Thorpe

Prologue

In previous chapters we established that a security methodology is essentially composed of two inter-related parts or aspects: a security process and a conceptual security framework, aggregating collections of conceptual security artefacts. The latter are usually specific to a methodology; in the case of security patterns, they can also be used in a supplementary fashion in methodologies of different paradigms.

Continuing and extending this methodology-agnostic feature of security patterns, over the next four chapters we present a collection of generic conceptual security artefacts, which, as explained in the Introduction to the thesis, can be viewed in several ways. In the context of the overall approach or “toolkit” to engineering security methodologies, the generic artefacts can be seen as separate drop-in replacements or supplements to existing artefacts. Collectively, the generic artefacts also provide a skeleton for a basic conceptual security framework, together with a simple SecReq pattern instance (essentially, abstract process fragment) and micro-process patterns for a CounterIntro phase. Finally, since each generic artefact will ultimately be used in the case study methodology (with two of the artefacts already incorporated into it in Chapter 3), the generic artefacts can be seen as illustrative examples of how specific, fit-to-purpose artefacts would be constructed and specified “from scratch” when engineering a specific security methodology.

This chapter presents the first of the aforementioned generic artefacts: a framework for the decomposition – i.e. the formation of system (or more precisely, architectural) characterizations – of distributed software architectures. While the framework is applicable to non-functional requirements (NFRs) other than security (e.g. reliability, performance, safety), for our purposes and within the context of this thesis security is the central NFR of focus. The analysis process that accompanies the framework is a basic threat analysis process, which, when combined with the second generic artefact of Chapter 5, can be made into a full threat modeling process – i.e. a realization of the SecReq phase for a methodology. Besides serving as a basis for threat modeling, architectural decomposition – as supported by the proposed framework – also forms a basis for a structured introduction of security features.
# Statement of Authorship

<table>
<thead>
<tr>
<th>Title of Paper</th>
<th>Decomposing Distributed Software Architectures for the Determination and Incorporation of Security and Other Non-Functional Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication Status</td>
<td>Published, Accepted for Publication, Submitted for Publication, Publication Style</td>
</tr>
</tbody>
</table>

## Author Contributions

By signing the Statement of Authorship, each author certifies that their stated contribution to the publication is accurate and that permission is granted for the publication to be included in the candidate's thesis.

### Name of Principal Author (Candidate)
Anton V. Uzunov

**Contribution to the Paper:** Conceptualization of work, its realization (research) and documentation (wrote manuscript). Acted as corresponding author.

**Signature**

Date 11/10/2013

### Name of Co-Author
Katrina Falkner

**Contribution to the Paper:** Supervised development of work, helped to evaluate and edit the manuscript.

**Signature**

Date 11/10/13

### Name of Co-Author
Eduardo B. Fernandez

**Contribution to the Paper:** Provided ideas and commented on manuscript.

**Signature**

Date 13/10/2013

### Name of Co-Author

**Contribution to the Paper**

**Signature**

Date

*Presented at: 22nd Australasian Software Engineering Conference, Melbourne, Australia, pp. 30-39*

NOTE:  
This publication is included on pages 133-142 in the print copy of the thesis held in the University of Adelaide Library.  

It is also available online to authorised users at: 

http://doi.org/10.1109/ASWEC.2013.14
In this chapter we presented the first of our generic conceptual security artefacts: a framework for the decomposition of distributed software architectures. The framework itself forms the basis for both threat analysis – a process for which was provided with the framework – and also for the introduction of security attributes using any security solution types. We demonstrate the use of the framework and its accompanying threat analysis process in the case study methodology in Chapter 9.
Chapter 5

(Generic conceptual security artefacts II)

Pattern-based threat library and taxonomy
for networked and distributed systems

“There is no terror, Cassius, in your threats; for I am armed so strong in honesty
that they pass by me as the idle wind, which I respect not.”

– William Shakespeare (Julius Caesar: Act 4, Scene 3, words of Brutus)

Prologue

In this chapter we continue with the exposition of the generic conceptual artefacts, and present a
library and taxonomy (the meaning of this terminology is clarified in the paper constituting this
chapter) of threats for networked and distributed systems, encapsulated in a new type of abstract
pattern called a threat pattern. As a generic artefact, the library/taxonomy can be used by itself
independently, or in conjunction with the previous artefact for a (nominally) complete threat
modeling approach. A simple specialization process, described in the paper, can be utilized to
specialize the (base) library/taxonomy for a variety of system-/technology-specific contexts,
making this a flexible artefact in the terminology of Chapter 3. Threat patterns also provide a
basis for security testing and verification – i.e. checking whether a particular threat is addressed
during later development phases. With respect to the meta-model (for engineering security
methodologies) presented in Chapter 3, this generic artefact can effectively be seen as a
Structured security artefact collection aggregating Threat instances.
# Statement of Authorship

<table>
<thead>
<tr>
<th>Title of Paper</th>
<th>An Extensible Pattern-based Library and Taxonomy of Security Threats for Distributed Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication Status</td>
<td>◯ Published, ◯ Accepted for Publication, ◯ Submitted for Publication, ◯ Publication style</td>
</tr>
</tbody>
</table>

## Author Contributions

By signing the Statement of Authorship, each author certifies that their stated contribution to the publication is accurate and that permission is granted for the publication to be included in the candidate's thesis.

<table>
<thead>
<tr>
<th>Name of Principal Author (Candidate)</th>
<th>Anton V. Uzunov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Conceptualization of work, its realization (research) and documentation (wrote manuscript). Acted as corresponding author.</td>
</tr>
<tr>
<td>Signature</td>
<td>Date 17/10/2018</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th>Eduardo B. Fernandez</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Provided some references, read the paper and commented on several sections.</td>
</tr>
<tr>
<td>Signature</td>
<td>Date 19/10/2013</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td></td>
</tr>
<tr>
<td>Signature</td>
<td>Date</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td></td>
</tr>
<tr>
<td>Signature</td>
<td>Date</td>
</tr>
</tbody>
</table>
An Extensible Pattern-based Library and Taxonomy of Security Threats for Distributed Systems

Anton V. Uzunova,*, Eduardo B. Fernandezb

a, School of Computer Science, The University of Adelaide, Adelaide, South Australia, Australia, 5005
b, Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, 777 Glades Rd., Boca Raton, FL 33431

* Corresponding author

E-mail addresses: anton.uzunov@adelaide.edu.au (Anton V. Uzunov), ed@cse.fau.edu (Eduardo B. Fernandez)

Abstract:
Security is one of the most essential quality attributes of distributed systems, which often operate over untrusted networks such as the Internet to provide organizations and individuals with a range of data and computational services. To incorporate security features during the development of a distributed system requires a sound analysis of potential attacks or threats in various contexts, a process that is often termed “threat modeling”. To reduce the level of security expertise required, threat modeling can be supported by threat libraries (structured or unstructured lists of threats), which have been found particularly effective in industry scenarios; or attack taxonomies, which offer a classification scheme to help developers find relevant attacks more easily. In this paper we combine the values of threat libraries and taxonomies, and propose an extensible, two-level “pattern-based taxonomy” for (general) distributed systems. The taxonomy is based on the novel concept of a threat pattern, which can be customized and instantiated in different architectural contexts to define specific threats to a system. This allows developers to quickly consider a range of relevant threats in various architectural contexts as befits a threat library, increasing the efficacy of, and reducing the expertise required for, threat modeling. The taxonomy aims to classify a wide variety of more abstract, system- and technology-independent threats, which keeps the number of threats requiring consideration manageable, increases the taxonomy's applicability, and makes it both more practical and more useful for security novices and experts alike. After describing the taxonomy which applies to distributed systems generally, we propose a simple and effective method to construct pattern-based threat taxonomies for more specific system types and/or technology contexts by specializing one or more threat patterns. This allows for the creation of a single application-specific taxonomy, which in turn allows developers to consider the widest range of applicable threats in a given architectural context. We demonstrate our approach to specialization by constructing a threat taxonomy for peer-to-peer systems.

Keywords: distributed systems security attacks; threat patterns; threat modeling; pattern-based security threat taxonomy; pattern-based security threat library; peer-to-peer system-specific threats

1 Introduction
Over the last decade distribution has become one of the main characteristic features of software systems, prompted in large measure by the expanding needs of businesses, scientific organizations and individuals who wish collaborate across geographical distances, share data and resources or simply perform computations remotely. To support such features, however, the corresponding systems must often span untrusted networks – with the Internet being a prime example – making them susceptible to a wide range of attacks both at the individual host and network levels. Security attributes, therefore, are among of the most important quality attributes for distributed systems operating in untrusted environments, and have consequently received much attention over the years [1, 2, 3, 4]. To incorporate these attributes during the development of a distributed system, whether using a systematic approach (i.e. a methodology [5]) or in some ad-hoc fashion, requires the introduction of a number of security measures, which, in turn, are the result of analyzing the potential attacks or threats to a system in a given context. This analysis process is often termed threat modeling [6, 7], and is performed during the requirements analysis stage, the design stage, or both. In all cases the process generally requires developers to conjecture possible attacks to different assets or parts of a system, to assess their risk and likelihood, and to determine at a high level how they could potentially be mitigated.

Conducting threat modeling usually requires a sound knowledge of a system's technical domain and sufficient security expertise to consider both generic and specific attacks for various system- and/or technology-specific contexts. These security knowledge requirements can leave most “off-the-street” developers estranged (cf. [8]), with the net result that threat modeling is not performed or, when performed, is performed sub-optimally or with significant effort involved (cf. [9]). As Dhillon [9] points out, a threat library that collects common threats to a given system-/technology-specific context can greatly enhance the efficacy of the threat modeling process and hence put it back, so to speak, on the project map. A threat library such as the one used at EMC [9], or even by Microsoft for
web applications [10], can also go a long way in educating developers about common threats, rendering future threat modeling tasks easier.

Despite their value, threat libraries encompass only a set of specific, pre-defined threats, making the discovery of new threats or the same threats in different architectural contexts more difficult. In this respect the use of threat or attack taxonomies such as Microsoft's STRIDE [7] (an acronym for Spoofing, Tampering, Repudiation, Information disclosure, Denial of service and Escalation of privileges), can be more useful, since they allow an arbitrary number of threats to be considered that fall within one or more categories. However, most taxonomies are either at a very high level of abstraction and hence require significant security expertise to identify appropriate threats (cf. [9] for STRIDE), or, in general, are simply not appropriate for threat modeling or indeed any form of security assessment in the first place [11]. Those that are appropriate for security assessment and at the right level of abstraction are not necessarily useful during the earlier stages of the SDLC (e.g. they consider post-design vulnerabilities), do not provide appropriate categories for threat modeling, or are relevant only to specific contexts (see [11] for a broad overview and references). Finally, the taxonomies referred to above – excepting STRIDE – are for vulnerabilities and attacks, not threats, which is a subtle but important difference (e.g. unsafe code execution is a threat realized by multiple attacks in different contexts).

In this paper we combine the values of threat libraries and threat taxonomies and propose what we term (with some risk of using terminology loosely) a pattern-based threat taxonomy for (general) distributed systems. In our approach, each threat is encapsulated in a new type of pattern (see [12]) called a threat pattern, which can be customized and instantiated in particular architectural contexts to define specific threats to a system. This allows developers to quickly consider a range of relevant threats in various architectural contexts as befits a threat library, increasing the efficacy of, and reducing the expertise required for, threat modeling. Threat patterns can also be related to corresponding misuse patterns [13], which can detail the attacks realizing a particular threat and educate developers. The (base) taxonomy aims to classify a wide variety of more abstract, system- and technology-independent threats, which keeps the number of threats requiring consideration during a threat modeling process manageable, increases the taxonomy's applicability, and makes it both more practical and more useful for security novices and experts alike. Employing patterns also helps to establish a common domain vocabulary, promoting the use of consistent threat names and concepts by developers in their everyday security-related work.

Despite the breadth of our taxonomy, each distributed system type, and even the technologies employed to realize a system, can create a variety of specific threats, which may not be explicitly present among our proposed threat patterns, or, more precisely, may not be present at the base level of abstraction. To solve this problem, we propose a simple and effective method to extend threat taxonomies by specializing one or more threat patterns to new system-/technology-specific contexts. This allows for the construction of application-specific taxonomies by taking the union of the set of relevant system-/technology-specific taxonomies, which in turn allows developers to consider the widest range of applicable threats in any given architectural context. We demonstrate our approach to specialization by constructing a threat taxonomy for peer-to-peer systems.

The latter example also demonstrates the purely “taxonomic” feature of our proposal, where each pattern in the taxonomy for distributed systems acts as a “category” for more specialized patterns and pattern instances. This feature allows known threats to be classified in a way that has value during system development, i.e. specific attacks such as CodeRed worm, SQL slammer exploit and, indeed, thousands of others, can be seen more abstractly as collections of individual threats (scanning, probing, injection etc.), which require mitigation irrelevant of whether they are launched against a system automatically (malicious software) or manually (malicious hackers).

The rest of this paper is structured as follows. In Section 2 we introduce the concept of a threat pattern, relate it to the existing misuse patterns of Fernandez and colleagues [13], and discuss threat taxonomies (Section 2.1); we also define the architectural contexts of the threat patterns (Section 2.2). In Section 3 we present our (base) threat taxonomy for distributed systems and discuss pattern specialization and instantiation. In Section 4 we specialize a number of (base) threat patterns to construct a taxonomy for peer-to-peer systems. In Section 5 we consider related work; and in Section 6 we conclude and discuss future research directions.

2 Background and definitions

In this section we provide the necessary background for the rest of the paper by defining threat patterns and pattern-based threat taxonomies (Section 2.1) as well as architectural contexts (Section 2.2).

2.1 Threat patterns and pattern-based threat taxonomies

The concept of a pattern in software engineering has received much attention over the last fifteen years, both in academia and the industry, owed in large measure to the pioneering work of Gamma and colleagues in the field of object-oriented design [14]. Patterns have been found useful in diverse areas such as software architecture, fault-tolerance, parallel programming and security, with each area boasting a sizable catalog of patterns available for developers to use. Within security in particular, solutions in the form of security patterns have appeared steadily in the literature to cover most major security-related concerns (see [15]). The pattern concept has also been found useful for the reverse side of the security solution landscape, namely, for security attacks, in the form of attack patterns [16] and misuse patterns [13, 17, 18]. Attack patterns capture the steps required to perform a specific
security attack (exploit) in a generic fashion; misuse patterns, on the other hand, detail a complex attack on a system related to particular architectural components, capturing the structure and dynamics of the attack, forensic information and much else besides. Misuse patterns in particular, like security patterns, are full *software patterns*, which capture a set of principal design decisions [19] or define constraints determining a family of architectures that satisfy them [20]. In the case of security patterns this implies architectural impact [15], and in the case of misuse patterns, a strong architectural relation.

Not all software patterns, however, have *direct* architectural impact, or *concrete* architectural relation. From a more abstract point of view, a pattern can be seen simply as “the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts” [12]. One can thus define a type of software pattern that has a more generic architectural relation conforming to the latter characterization, which we term an *abstract software pattern*. The security solution analog of this general type of pattern was proposed by Fernandez et al. [21] in the form of abstract security patterns. In this section, we define the *threat* analog: threat patterns, which are the core constituents of our threat taxonomy.

A conceptual model relating the various patterns for security mentioned thus far, as well as many of the main concepts contained in the rest of this paper, can be seen in Figure 1, with the more important elements appearing in different colors or, if viewed without color, different shades of gray (purely to differentiate them from other elements and from each other). For associations that appear as being (solely) vertically aligned in the figure, an arrow to the left implies it should be read “upwards” (e.g. each security patterns addresses one or several specific security policies), and an arrow to the right implies it should be read “downwards” (e.g. each general threat pattern is for a given general architectural context). In what follows, words appearing in *italic* font refer to (class) elements in the model.

![Figure 1: Conceptual model of threat patterns and threat taxonomies](image)

Threat patterns, as the name implies, encapsulate security threats in a generic fashion. More precisely, threat patterns can be defined as triples of the form:

$$(A, T, SP)$$

where $A = \{a_x: i \in \mathbb{N}\}$ is a set of *architectural contexts* – determining the abstract relations of each threat to a part of a distributed software architecture – which can be influenced by one or more (distributed) *system-/technology-
specific contexts (see Figure 1); \( T \) is a generic threat description; and \( S^P = \{ p_i : i \in \mathbb{N} \} \) is a set of mitigating generic security policies, which can be refined by more specific policies and realized by appropriate security solutions (such as security patterns).

Being abstract software patterns, threat patterns can be seen as having a similar relation to misuse patterns as abstract security patterns to concrete security patterns [21], namely, misuse patterns realize threat patterns by providing detailed descriptions of an attack that realizes the threat encapsulated in a corresponding threat pattern in a given architectural context. The relationship between threat patterns and attack/misuse patterns thus mirrors the relationships between threats and attacks more generally, i.e. attacks realize threats by exploiting vulnerabilities on assets [22] (as shown in Fig. 1). In our model assets are taken to mean both important parts of a system or, equivalently, an application, that require protection (information or resources), as well as particular software units.

A concrete example of the relations between assets, vulnerabilities, threats and attacks is when there is an injection threat to a database management system (DBMS) (asset as software unit requiring protection) storing banking information (asset as application information requiring protection) with an input validation weakness (vulnerability) that can be exploited via a particular SQL injection attack (realizing the threat). As can be seen from this example, the threats we are considering are technical threats applicable to the architectural level, to which high-level threats can be mapped – e.g. “reading the balance of a bank account” maps to an “unauthorized access” threat pattern in a particular architectural context.

Threat patterns can be grouped together in threat classes, which together classify the patterns and form a (pattern-based) threat taxonomy. There can be one such taxonomy for each system-/technology-specific context, with a single taxonomy for the (general) distributed system context, which we term a base threat taxonomy. Derived taxonomies for different system-/technology-specific contexts (e.g. peer-to-peer systems, agent-based systems, web-service and Java technologies etc.) can also be constructed by specializing threat patterns from the base taxonomy. Since one cannot expect to encompass all threats in a single taxonomy, derived taxonomies may also augment the base taxonomy with new patterns or additional threat classes as required. Derived taxonomies can be combined to form a single application-specific taxonomy, which will contain a mixture of base (general, unspecialized) and specialized threat patterns.

Taxonomies can also be stratified into different levels according to the purpose of the threat patterns (this is not shown in Figure 1). Our base threat taxonomy, in particular, is stratified into two levels, the first of which aggregates security threat patterns with respect to the system itself (at the architectural level); and the second of which aggregates meta-security threat patterns, which are simply (threat) patterns encapsulating threats to the designed security infrastructure of a system (more precisely, threats to the security countermeasures corresponding to the policies introduced as mitigating factors for the first level threats). The details of the threat classes used in both levels are given in Section 3.

Having thus far defined threat patterns and informally discussed the main concepts and their relations as shown in the conceptual model (Figure 1), perhaps a few remarks are in order regarding the more formal aspects of the model itself. Regarding the coverage and completeness of the generalization relations, the generalizations stemming from the asset, security policy and taxonomy elements are complete; while all others are incomplete. The generalizations stemming from the asset and security policy elements are disjoint; while all the other generalizations are overlapping, although the degree of overlap is not always well-defined (e.g. attack patterns are patterns, but cannot be classified precisely as software patterns, since they lack explicit architectural relation). Regarding the associations, the mitigates associations (between threats and security policies, and threat patterns and generic security policies) should be interpreted more broadly as “stop, lessen or mitigate”, since a policy may altogether stop a given threat, depending, of course, on how it is implemented. Another noteworthy association is the influences association between system/environment contexts and architectural contexts, which should be interpreted as carrying over into the derivative (subclass) elements, so that the system/technology-specific contexts influence (and indeed define) the derived architectural context as appropriate for the given situation. All other associations in the model can be interpreted straightforwardly.

2.2 Architectural contexts for the threat patterns

As explained in Section 2.1 above, for threat patterns to be (abstract) software patterns in any real sense, they must have some form of architectural relation. In this section we summarize part of a conceptual framework for a form of system characterization called architectural decomposition, proposed in [23], which we use to define the architectural contexts of our threat patterns. The aforementioned framework allows distributed software architectures to be decomposed into different areas of functionality using five decomposition layers, and further characterized by means of a set of corresponding abstractions capturing the main aspects of the architecture's realization. When the framework is applied to decomposing an actual distributed software architecture (during its development or otherwise), certain parts of the architecture will fall into the various layers, and potentially implement one or more realization abstractions. The architectural context for a given pattern is defined precisely by the relevant decomposition layers and corresponding realization abstractions. This gives each pattern its necessary architectural relation, which is also important for pattern specialization (Section 3.3), and allows each threat to be related to the parts of the architecture that are themselves related to the different layers and abstractions, which is
necessary for pattern instantiation (Section 3.4).

In what follows, we briefly describe the conceptual framework's decomposition layers – in Table 1, Section 2.2.1 – and realization abstractions – in Section 2.2.2 (the reader is referred to [23] for additional details). The presentation order of layers and abstractions in the table and the bullet points follows their hierarchical structure: each functionality decomposition layer relies on the general distributed system aspects or concerns encompassed by the layers below it, and, similarly, the realization abstractions are ordered such that each abstraction in each layer relies on ones below it. For the decomposition layers (Table 1), the right-most column on a given row indicates the threat classes from the base taxonomy (with accompanying sub-section numbers in brackets), which have threat patterns with that layer (row) as a context. While the threat classes are not actually introduced until later, in Section 3, they included in the table for reference. The letters in brackets after the decomposition layer's name in the first table column is a shorthand that will be used later in presenting the base taxonomy (Sections 3.1 to 3.2).

### 2.2.1 Functionality decomposition layers

<table>
<thead>
<tr>
<th>Functionality decomposition layer</th>
<th>Brief description ([23])</th>
<th>Threat classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>User interaction (U)</td>
<td>The user interaction layer encompasses all aspects related to interfacing and/or interacting with users.</td>
<td>Identity attacks (3.1.1), Passing illegal data (3.1.4), Remote information inference (3.1.6), Loss of accountability (3.1.7), Uncontrolled operations (3.1.8)</td>
</tr>
<tr>
<td>Data / storage management (S)</td>
<td>The data management layer encompasses all aspects related to the storing and management of application or system-level information.</td>
<td>Passing illegal data (3.1.4), Stored data attacks (3.1.5), Remote information inference (3.1.6), Uncontrolled operations (3.1.8)</td>
</tr>
<tr>
<td>Resource management (R)</td>
<td>The resource management layer encompasses all aspects related to resource allocation, global scheduling, process and/or object migration, as well as the life-cycle or dynamic configuration of (active) software components.</td>
<td>Uncontrolled operations (3.1.8)</td>
</tr>
<tr>
<td>Distribution control (D)</td>
<td>The distribution control layer encompasses all aspects related to component collaboration or interaction, coordination of local/remote execution and synchronization/concurrency control.</td>
<td>Identity attacks (3.1.1), Passing illegal data (3.1.4), Remote information inference (3.1.6), Uncontrolled operations (3.1.8)</td>
</tr>
<tr>
<td>Communication (C)</td>
<td>The communication layer encompasses all aspects related to network communications between (remote) software components.</td>
<td>Network communication attacks (3.1.2), Network protocol attacks (3.1.3), Loss of accountability (3.1.7)</td>
</tr>
<tr>
<td>Addressing (A)</td>
<td>The addressing layer encompasses all aspects related to address, identifier and/or name allocation, distribution and discovery/lookup.</td>
<td>Network communication attacks (3.1.2), Network protocol attacks (3.1.3), Loss of accountability (3.1.7)</td>
</tr>
</tbody>
</table>

Table 1: Functionality decomposition layers of the conceptual framework in [23], with relations to the (first level) threat classes of the base taxonomy

### 2.2.2 Technical realization abstractions

**User interaction:**
- **Input ports:** software elements of any granularity by or through which an application can receive user input.
- **Output ports:** software elements by or through which a user can obtain information from an application.

**Data / storage management:**
- **Data structures:** from an application's point of view, most data is used in the form of particular data structures defined during detailed design and implementation, but also during earlier stages.
- **Storage abstractions:** data can be stored in objects or encapsulated in some other fashion, e.g. in file abstractions.
- **Database systems:** data can be stored in a database, which can be thought of as coarse grained architectural components.
- **File systems:** the data used by an application will ultimately be stored in files on persistent media (block devices), regardless of the storage abstractions placed on top. Files are usually part of a filesystem managed by the operating system.
Resource management:
- **Resources**: resources are any software components that are managed (created, destroyed, monitored etc.) for use by other components. Examples can include schedulable processes, active software components with a certain life-cycle, data stores, as well as virtual machines or hardware units.
- **Algorithms**: resources are allocated and shared via particular algorithms, which can address issues in resource contention, detecting deadlocks or be concerned with electing new locations for data replication. They can also relate to the management of dynamic (run-time) configurations of components in a system.

Distribution control:
- **Software component interfaces**: all active software components have interfaces to interact with each other, which may include programmatic functions, APIs or some implicit means of making a component perform some function.
- **Operations**: every active component can also perform certain operations, which can be divided into various levels of granularity, such as large-grained responsibilities (sharing load among worker threads), functions (an object's methods), syscalls etc.
- **Execution abstractions**: active software components usually map to certain concrete software units in some technology: e.g. objects, beans, applets and others.
- **Processes (OS)**: the execution abstractions in a system are ultimately executed as processes (or in processes) by an underlying operating system, whose semantics of the processes may differ from any abstractions placed on top.

Communication:
- **Messages**: messages are the unit of transport data in a networked process collaboration, and may include remote function calls and their parameters, XML documents and others.
- **Message channels**: messages are carried over logical message channels, which can be considered separately from the messages themselves, and be of different granularities and levels of abstraction (network links, publish/subscribe infrastructures etc.)
- **Protocols**: the actual passing of messages is accomplished via the use of protocols, with some examples being the TCP/IP protocol suite and security (cryptographic) protocols such as SSL.
- **Networking infrastructure (S/W)**: the messaging infrastructure is ultimately implemented on top of a physical network infrastructure consisting of routers, switches, hubs etc., which are controlled and configured by software and hence can form part of the system configuration itself.

Addressing:
- **Addresses / identifiers**: for components to be usable and/or reachable by others, they must be addressable, which can be achieved by assigning entities identifiers, addresses and/or names.
- **Protocols / algorithms**: the protocols or algorithms used in assigning addresses can be considered abstractly as a “lookup()” function that returns the address of a component.
- **Routing data structures (tables)**: information for routing messages to the correct address during communication is often stored in tables or other data structures, at the application level or in the network infrastructure.

3 Base threat taxonomy for distributed systems

In this section we present the threat patterns in the two levels of the base taxonomy for distributed systems (Sections 3.1 and 3.2 respectively).

The patterns constituting the taxonomy are the result of selectively synthesizing, generalizing and condensing in abstract form the relevant security knowledge contained in a number of existing attack and vulnerability taxonomies [24, 25, 26, 27, 28, 29, 30, 31], threat libraries [9, 10] and attack repositories (CAPEC – capec.mitre.org, [32]; OSVDB through the work of [33]), guided by the current authors' own security expertise. While we cannot claim that our collection of patterns is exhaustive, we believe they collectively span most threats against (general) distributed systems, both at the network and host levels.

As mentioned in Section 2.1, the base taxonomy is stratified into two levels, with the first aggregating security threat patterns encapsulating threats to the system itself; and the second aggregating meta-security threat patterns, which encapsulate threats to the designed security infrastructure of the system. The first level of the taxonomy organizes the threat patterns into eight threat classes, which can be described as follows (the threats described refer to varieties of encapsulated threats):

- **Identity attacks**: threats in which an attacker attempts to fabricate or misuse identities in a system;
- **Network communication attacks**: threats to the communications between distributed components;
- **Network protocol attacks**: threats specifically to the network protocols used for communication;
- **Passing illegal data**: threats in which input data is manipulated by an attacker for some malicious purpose;
- **Stored data attacks**: threats specific to on-storage data (excluding unauthorized access);
- **Remote information inference**: threats which are concerned with extracting information from a component or part of a distributed system remotely, i.e. over a network;
- **Loss of accountability**: threats impacting accountability attributes;
- **Uncontrolled operations**: containing threats which are concerned with exploiting existing system functionality in ways that would not normally be allowed, including race conditions, access to data etc.

The second level of the taxonomy organizes threat patterns into four meta-security threat classes, which can be similarly described as follows:

- **Cryptography attacks**: threats to countermeasures using cryptography;
- **Countermeasure design**: threats to the way certain countermeasures are (or may be) designed;
- **Configuration / administration**: threats related to the configuration and/or administration of the security infrastructure.
- **The Network protocol attacks** threat class from the first taxonomy level is also part of the second level (with the threat patterns being applicable to secure protocol design).

In relation to high-level taxonomies such as STRIDE, our threat classes are based on the nature of the threat (e.g. network protocol attack), not its impact (e.g. escalation of privilege).

In the two ensuing sub-sections we present the patterns according to the levels and order of threat classes defined above. Following this presentation, we outline a simple method to specialize the threat patterns for different system-/technology-specific contexts (Section 3.4), and briefly discuss pattern instantiation (Section 3.5).

### 3.1 First level (security) threat patterns

The first level threat patterns aim to encompass most common threats to general distributed systems. We present the patterns according to their threat classes (Section 3) in succession, beginning with patterns for Identity attacks in Section 3.1.1 (Table 2) and ending with patterns for Uncontrolled operations in Section 3.1.8 (Table 9). We use the following scheme for presentation: the name of the pattern appears in the left-most column in each table, followed by a brief description of the threat in the next column; then the architectural context (which is affected by the system-specific context for distributed systems by default), with the decomposition layer denoted by its shortened form (as introduced in Section 2.2) and the realization abstractions following in brackets; and finally a mitigating (generic) security policy in the right-most column, with a more specific policy in brackets if applicable.

Our table-row-based presentation replaces (without any loss of essential information) the more traditional pattern presentations employing a structured textual template, and can be related to the latter by mapping the description column (for each row) to a “solution”-equivalent section; the architectural context to a “context” section; and the mitigating policies column to a directly equivalent “mitigation” section. Since the threat patterns are abstract, we do not need to consider forces, architectural models etc., which is what would be required of misuse patterns that realize a given threat pattern in particular contexts.

As an aside, having introduced the first level threat classes at the outset of Section 3, the reader can refer to Table 1 (right-most column) as a complementary organizational aid to quickly find which classes contain threat patterns with a given functionality decomposition layer as a context.

#### 3.1.1 Identity attacks

<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity spoofing</td>
<td>The attacker fabricates a new identity or claims to possess an existing identity of some principal in the system.</td>
<td>U (input ports), D (software component interfaces, execution abstraction, processes), A (all)</td>
<td>Identity management (authentication)</td>
</tr>
<tr>
<td>Advantageous identity allocation</td>
<td>The attacker obtains a particular identity or an identity belonging to certain groups, roles etc. that can be leveraged to grant specific functionality, privileges or access to data.</td>
<td>U (input ports), D (operations), A (addresses / identifiers)</td>
<td>Identity management (identity assignment control)</td>
</tr>
</tbody>
</table>

*Table 2: Threat patterns for the Identity attack class*
### 3.1.2 Network communications attacks

<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message secrecy violation (<em>passive eavesdropping, reading plaintext message</em>)</td>
<td>Messages in transit are intercepted and their contents read by an attacker.</td>
<td>C (messages, message channels, networking infrastructure)</td>
<td>Secure communication (message encryption)</td>
</tr>
<tr>
<td>Message integrity violation (<em>active eavesdropping, modification</em>)</td>
<td>Messages in transit are intercepted and modified, replaced, corrupted or simply deleted by an attacker.</td>
<td>C (messages, message channels, networking infrastructure)</td>
<td>Secure communication (message hashing, error detection codes)</td>
</tr>
<tr>
<td>Message authenticity violation (<em>source spoofing</em>)</td>
<td>The source address of the message is fabricated or changed.</td>
<td>C (messages, message channels, networking infrastructure)</td>
<td>Secure communication (message authentication codes, digital signatures)</td>
</tr>
<tr>
<td>Traffic analysis, protocol sniffing</td>
<td>The message stream is analyzed for various characteristics such as size of messages and message frequency. Protocols fields can also be analyzed in the process.</td>
<td>C (message channels, networking infrastructure)</td>
<td>Secure communication (message padding)</td>
</tr>
<tr>
<td>Covert network channel</td>
<td>A hidden communication channel, either new or within an existing one, is created to smuggle or reveal additional data. The data is carried in messages (e.g. via steganographic means) or in the frequency of messages.</td>
<td>C (message channels, networking infrastructure)</td>
<td>Secure communication (protocol design), Logging and monitoring</td>
</tr>
<tr>
<td>Session hijacking</td>
<td>An attacker takes control of a communication session between two end-points.</td>
<td>C (message channels)</td>
<td>Secure communication (protocol design)</td>
</tr>
<tr>
<td>Session state poisoning</td>
<td>An attacker corrupts or modifies the state information used in a session.</td>
<td>S (data structures), C (message channels)</td>
<td>Authorization, Secure communication (encryption, integrity checks)</td>
</tr>
<tr>
<td>Route poisoning</td>
<td>The normal route information for messages is tampered with so that communication is redirected, dropped, or routed to incorrect destinations.</td>
<td>C (networking infrastructure), A (protocols / algorithms, routing data structures)</td>
<td>Secure communication (secure routing protocols)</td>
</tr>
<tr>
<td>Message flooding</td>
<td>A large stream of messages is injected into a new or existing communication stream to degrade the network performance or interrupt the normal service of a target node or component.</td>
<td>C (message channels, networking infrastructure)</td>
<td>Filtering (network level firewall), Secure communication, Logging and monitoring (IDS)</td>
</tr>
</tbody>
</table>

Table 3: Threat patterns for the Network communication attacks class

### 3.1.3 Network protocol attacks

<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message replay</td>
<td>An attacker re-sends an intercepted message that was previously sent during the same protocol run to obtain the same results.</td>
<td>C (protocols), A (protocols / algorithms)</td>
<td>Secure communication (protocol design)</td>
</tr>
<tr>
<td>Message re-use</td>
<td>An attacker re-sends an intercepted message that was previously sent during another protocol run to obtain the same results.</td>
<td>C (protocols), A (protocols / algorithms)</td>
<td>Secure communication (protocol design)</td>
</tr>
<tr>
<td>Protocol field</td>
<td>An attacker modifies one of the fields of the</td>
<td>C (protocols, networking</td>
<td>Secure</td>
</tr>
<tr>
<td>Security threat pattern</td>
<td>Threat description</td>
<td>Architectural context</td>
<td>Mitigating security policy</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------</td>
<td>-----------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Modification</td>
<td>protocol in order to produce unpredictable results (e.g. crash the protocol implementation).</td>
<td>infrastructure), A (protocols / algorithms)</td>
<td>communication, filtering (network level)</td>
</tr>
<tr>
<td>Use of abnormal packet sizes</td>
<td>An attacker sends messages of unexpected size.</td>
<td>C (protocols, networking infrastructure)</td>
<td>Secure communication, filtering (network level)</td>
</tr>
<tr>
<td>Use of abnormal packet sequencing (re-ordering)</td>
<td>An attacker re-orders the sequence of messages, or changes the protocol sequence numbers.</td>
<td>C (protocols, networking infrastructure)</td>
<td>Secure communication, filtering (network level)</td>
</tr>
<tr>
<td>Use of reserved protocol packets</td>
<td>An attacker sends packets with modified values in reserved fields, or special packets (e.g. for debugging) under standard conditions.</td>
<td>C (protocols, networking infrastructure)</td>
<td>Secure communication, filtering (network level)</td>
</tr>
<tr>
<td>Protocol initial/end state exploitation</td>
<td>The attacker exploits the protocol initiation procedure (e.g. initiating multiple connections) or the end conditions (e.g. not finalizing the protocol state).</td>
<td>C (protocols, networking infrastructure)</td>
<td>Secure communication (protocol design, client puzzles)</td>
</tr>
</tbody>
</table>

Table 4: Threat patterns for the Network protocol attacks class

3.1.4 Passing illegal data

<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Injection</td>
<td>The attacker manipulates the input and passes arbitrary data to a target component that will be processed as normal data. The injected data may include binary code (used in an overflow, see further below), SQL statements, XML, OS commands etc.</td>
<td>U (input ports), S (all), D (software component interfaces, execution abstractions, processes), C (messages)</td>
<td>Filtering (input filtering), Storage security</td>
</tr>
</tbody>
</table>

Table 5: Threat patterns for the Passing illegal data threat class

3.1.5 Stored data attacks

<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corruption</td>
<td>The attacker modifies stored data on persistent or transient storage. The corrupted data can be a cause for corruption of state in the system, or can be used for denial-of-service (e.g. if the data is sensitive configuration data that is rendered useless).</td>
<td>S (all)</td>
<td>Storage security (hashing, integrity checks, secure backup)</td>
</tr>
</tbody>
</table>

Table 6: Threat patterns for the Stored data attacks class

3.1.6 Remote information inference

<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanning (information gathering)</td>
<td>An attacker scans a network or system manually or automatically using software tools to gather information (e.g. OS fingerprinting, middleware type, software versions etc.)</td>
<td>C (message channels)</td>
<td>Filtering (network level firewall), Logging and monitoring (IDS)</td>
</tr>
<tr>
<td>Probing (vulnerability)</td>
<td>An attacker checks for the existence of certain known vulnerabilities in the components of the</td>
<td>U (input ports), D (software)</td>
<td>Filtering (network level firewall),</td>
</tr>
</tbody>
</table>

154
<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>checking</td>
<td>system either manually or automatically using software tools.</td>
<td>component interfaces, processes, C (message channels)</td>
<td>Logging and monitoring (IDS)</td>
</tr>
<tr>
<td>Output information disclosure</td>
<td>Information from the system is leaked to the attacker via some form of output, such as error messages or standard responses.</td>
<td>U (output ports), D (all)</td>
<td>Filtering (output limitation)</td>
</tr>
<tr>
<td>Data inference</td>
<td>The attacker purposefully performs a set of operations that allow internal information to be inferred, e.g. by sending a pre-defined sequence of queries to a database or waiting for message responses in a given time-frame.</td>
<td>U (output ports), S (all), C (protocols)</td>
<td>Authorization, Storage security</td>
</tr>
</tbody>
</table>

Table 7: Threat patterns for the Remote data inference threat class

3.1.7 Loss of accountability

<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track erasing</td>
<td>An attacker modifies or deletes auditing information to erase the tracks of an attack.</td>
<td>S (database systems, file systems)</td>
<td>Logging and monitoring (secure system logs, replicated logs)</td>
</tr>
<tr>
<td>Repudiation</td>
<td>A user denies having performed some operation, or denies having sent a collection of messages over the network.</td>
<td>U (input ports), C (messages, protocols), A (protocols / algorithms)</td>
<td>Logging and monitoring (logging, auditing), Secure communication (digital signatures)</td>
</tr>
</tbody>
</table>

Table 8: Threat patterns for the Loss of accountability threat class

3.1.8 Uncontrolled operations

<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unauthorized access</td>
<td>A user gains access to data or resources without proper authorization.</td>
<td>U (input ports), S (storage abstractions, database systems, file systems), R (resources)</td>
<td>Authorization</td>
</tr>
<tr>
<td>Invoking unauthorized operations</td>
<td>A software component invokes operations on another component without being authorized to do so.</td>
<td>U (input ports), D (all), R (all)</td>
<td>Authorization</td>
</tr>
<tr>
<td>Spoofing privileged processes (transitive actions)</td>
<td>An attacker uses a process with less rights to cause a process with more rights to perform some action on its behalf.</td>
<td>U (input ports), R (algorithms), D (software component interfaces, execution abstractions, processes)</td>
<td>Authorization, Execution control</td>
</tr>
<tr>
<td>Unsafe code execution</td>
<td>Malicious code is executed on a target system or environment without control. This can include mobile code or an executable payload.</td>
<td>D (processes)</td>
<td>Authorization, Execution control (controlled environment)</td>
</tr>
<tr>
<td>Exploitation of tight component coupling</td>
<td>An attacker leverages component coupling between software components to force one component to perform some action.</td>
<td>U (input ports), R (algorithms), D (all)</td>
<td>Authorization, Execution control</td>
</tr>
<tr>
<td>Process overflow attack (buffer / integer overflows)</td>
<td>An attacker injects arbitrary binary code into the process execution environment that will overflow certain pre-defined boundaries and allow</td>
<td>U (input ports), S (data structures), D (software component interfaces, execution</td>
<td>Execution control (buffer overflow prevention)</td>
</tr>
</tbody>
</table>
In this section we describe the second level (meta-security) threat patterns of the base taxonomy. In contrast to the first level threats, which can generally be realized with a single attack, second level threats are more high-level and can encompass multiple attacks to the corresponding security infrastructure. We use a variation of the presentation scheme established for the patterns in Section 3.1, where each meta-security threat pattern is described in the format name – description – corresponding policy realizations, in Sections 3.2.1 to 3.2.3, Tables 10 to 12, respectively. In all cases the architectural context is determined by the realization of the corresponding security policy.

### 3.2 Second level (meta-security) threat patterns

#### Table 9: Threat patterns for the Uncontrolled operations threat class

<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploiting concurrency flaws</td>
<td>An attacker exploits concurrent access to shared data to cause corruption or gain authorization rights. One example is a time of check to time of use (TOCTOU) attack, where under certain circumstances data can be checked after it is used.</td>
<td>R (algorithms), D (all)</td>
<td>Execution control (process serialization)</td>
</tr>
<tr>
<td>Resource exhaustion</td>
<td>A system is saturated with resource requests, or resources are used excessively (e.g. memory hogging applications).</td>
<td>S (storage abstractions, database systems, file systems), R (all), D (execution abstractions, processes)</td>
<td>Execution control (quotas and limits)</td>
</tr>
<tr>
<td>Targeted process crashing</td>
<td>A process is purposefully exploited to crash, losing availability for the rest of the system.</td>
<td>R (resources), D (execution abstractions, processes)</td>
<td>Execution control (process replication)</td>
</tr>
</tbody>
</table>

#### Table 10: Meta-security threat patterns for the Cryptographic attacks class

<table>
<thead>
<tr>
<th>Meta-security threat pattern</th>
<th>Threat description</th>
<th>Corresponding security policies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forging cryptographic credentials</td>
<td>An attacker generates equivalent security credentials by means of a known cryptographic algorithm.</td>
<td>Identity management (authentication)</td>
</tr>
<tr>
<td>Abuse of weak algorithm</td>
<td>A cryptographic attack (brute force, known plaintext etc.) is launched on a weak algorithm used for authentication, message encryption, hashing or some other security measure.</td>
<td>Identity management (authentication), Secure communication</td>
</tr>
<tr>
<td>Exploiting vulnerable security protocol</td>
<td>One or more of the threats in the Network protocol attacks class (Section 3.1.3) applies to a particular security protocol. Additional threats include exploiting type flaws (re-interpreting bit sequences in a protocol implementation), parallel session and oracle attacks (when information from one parallel protocol run are used in another), binding attacks (when using a public-key infrastructure), encapsulation attacks as well as cryptographic algorithm-specific attacks (see, e.g., [34]).</td>
<td>Identity management (authentication), Secure communication, Storage security, Security information management</td>
</tr>
<tr>
<td>Password attacks (guessing, brute force, rainbow tables etc.)</td>
<td>An attacker acquires passwords easily (e.g. passed as plaintext) or takes advantage of incorrect password storage (lack of salting and stretching).</td>
<td>Identity management (authentication)</td>
</tr>
</tbody>
</table>
### 3.2.2 Countermeasure design

<table>
<thead>
<tr>
<th>Meta-security threat pattern</th>
<th>Threat description</th>
<th>Corresponding security policies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of default credentials</td>
<td>An attacker uses default credentials for access or authentication.</td>
<td>Identity management</td>
</tr>
<tr>
<td>Bypassing controls</td>
<td>An attacker bypasses existing security controls via an operation that can be performed in more than one way, or by using an access point which is not secured.</td>
<td>Identity management, Authorization</td>
</tr>
<tr>
<td>Leveraging authorization model</td>
<td>The authorization model does not map to the application or system semantics, or is not enforced correctly, which reduces the efficacy of the resulting controls and fails to mitigate threats such as Unauthorized access and Invoking unauthorized operations (Section 3.1.8).</td>
<td>Authorization</td>
</tr>
</tbody>
</table>

*Table 11: Meta-security threat patterns for the Countermeasure design class*

### 3.2.3 Configuration / administration

<table>
<thead>
<tr>
<th>Meta-security threat pattern</th>
<th>Threat description</th>
<th>Corresponding security policies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploiting bad policies</td>
<td>An attacker leverages authorization policies that allow users or processes more control than required (breaking principle of least privilege), or takes advantage of underspecified or missing policies (for authorization or network filtering).</td>
<td>Authorization, Filtering</td>
</tr>
<tr>
<td>Unauthorized modification of rights (meta-authorization policies)</td>
<td>Users pose as administrators, or administrators modify authorization policies which they should not be able to.</td>
<td>Authorization</td>
</tr>
</tbody>
</table>

*Table 12: Meta-security threat patterns for the Configuration / administration threat class*

### 3.3 Specializing threat patterns

The patterns presented thus far have all pertained to a general context for distributed systems, where the individual architectural contexts have been defined abstractly by the set of related decomposition layers and corresponding realization abstractions, as they are presented in Section 2.2. In what follows we present a simple method for specializing the threat patterns to different system-/technology-specific contexts, in order to construct a system-/technology-specific (derived) taxonomy.

As a first step in our method, one has to determine how the target system-/technology-specific context affects a given threat pattern's architectural context. This can be done purely based on the general characteristics of the given system or technology: each decomposition layer is considered in turn and characterized, along with any relevant realization abstractions. Subsequently, one can take two complementary approaches to specialization:

The first (deductive) approach consists in further detailing the effect of the new system-/technology-specific context on the given threat pattern's architectural context. This can be done purely based on the general characteristics of the given system or technology – to determine as full an architectural context as possible without reference to the application's actual architecture, and considering how the threat in each selected pattern changes in the new context using domain and security expertise. For example, a Corruption threat can be considered with respect to storage abstractions in a distributed file sharing context in which files are hashed. Corruption then implies that not only the file, but also the hash values, are affected, possibly leading to a new attack. Instantiation of the pattern can be avoided if the architectural context is kept sufficiently general, e.g. for a system type; if one begins to consider the threat for a particular system, then, naturally, this leads to instantiation (Section 3.4), not specialization.

The second (inductive) approach aims to draw domain and security knowledge from the existing literature on the subject, and consists in identifying specializations of threat patterns based on well-known and documented attacks. These attacks are then related to one or more threat patterns in the base taxonomy, and a corresponding architectural context, together with a mitigating policy, is determined accordingly.

In both approaches to specialization, the pattern's architectural context and potentially mitigating policies should be refined and/or expanded by propagating the threat down the hierarchy of decomposition layers and realization abstractions and considering its applicability to each of them in turn.

The final list of specialized threat patterns will constitute part of a system-/technology-specific taxonomy in its own right; a full taxonomy can be obtained by taking the union of relevant threats from the base threat taxonomy.
(i.e. the more general context) and the specialized patterns, with the more general patterns selectively replaced by the specialized versions (whenever this does not reduce the scope of possible attacks). In this sense, the constructed taxonomy can be seen simply as an extension of the base taxonomy to a more specific context.

Section 4 contains several examples of pattern specialization that make the latter descriptions somewhat more concrete.

3.4 Instantiating threat patterns

Threat patterns are instantiated in their architectural context with relation to the actual software architecture of the system or application, i.e. with consideration of which components and what parts of the architecture are involved. This gives an abstract description of the threat in the given context (as befits an abstract pattern) that can be made more concrete by instantiating related misuse patterns encapsulating the corresponding attacks in detail. A proper illustration of this requires a concrete distributed software architecture and is outside the scope of this paper. As a simple example, however, we can consider a client-server architecture using a Broker pattern for implementation [35], in which users (clients) use the server for entering their bank details. A threat pattern instance in this case is a message secrecy violation threat in the message channel abstraction (architectural context) realized by the Broker, with any encryption (mitigating policy) to be applied to the Broker's messaging components.

4 Constructing a threat taxonomy for peer-to-peer systems

In previous sections we have referred to the fact that different taxonomies can be derived from the base taxonomy by specialization of individual threat patterns for more specific architectural contexts. In this section we construct such a (derived) taxonomy for peer-to-peer architectures. Before this, we present some background on peer-to-peer systems in Section 4.1, outlining some of their general system-specific characteristics. This will be used in Section 4.2 to set the architectural context of the patterns.

As we explained in Section 1, constructing threat taxonomies for different system-/technology-specific contexts allows a single application-specific taxonomy to be created, which can be used to improve the efficacy of threat modeling during development. We discuss this point again briefly in Section 4.4 in relation to contexts other than peer-to-peer systems. Finally, we conclude this section by briefly discussing the use of pattern-based taxonomies as classification schemes (Section 4.5).

4.1 Background on peer-to-peer systems

Peer-to-peer systems are distributed systems in which there is no central point of control and software components on participating nodes can play the role of both a client and a server [36]. While well known content-distribution and file sharing applications such as Napster and KaZaa are usually cited as paradigmatic, peer-to-peer architectures are used in a wide variety of other systems as well, including instant messaging (IM) applications, collaborative applications and scientific computing environments. One of the main distinguishing architectural features of a peer-to-peer system is the use of a network overlay that facilitates communication between components (henceforth simply peers) by providing application-level routing on top of an existing networking infrastructure (e.g. on top of the TCP/IP protocol suite and the Internet) [37, 38].

Peer-to-peer systems can be broadly classified according to two orthogonal dimensions [39]: the degree of centralization with respect to addressing (centralized, de-centralized, hybrid), and the structure of the overlay with respect to routing and communication (structured and unstructured). In centralized peer-to-peer systems, a central server is used for addressing (peer lookup functionality), after which peers can communicate between each other freely. In de-centralized peer-to-peer systems the lookup functionality is spread across the peers. Hybrid peer-to-peer systems use super-peers to perform the lookup functionality (and potentially other functions), acting as localized servers. Regarding the overlay structure, structured peer-to-peer systems enforce some kind of mapping (usually using Distributed Hash Tables or DHTs) between keys or identifiers and resources stored on peers to facilitate efficient lookup, with the overall routing structures being distributed throughout the overlay; unstructured peer-to-peer systems, on the other hand, use network flooding strategies for routing and lookup queries. This division of peer-to-peer systems allows the theoretical combinations of structured, centralized and structured, hybrid systems, but we should point out that in practice current structured (DHT-based) systems are uniformly de-centralized.

Peer-to-peer systems can also be classified according to their application domain [40]: communication and collaboration, distributed computation, Internet services support, database systems and content distribution.

Besides the custom operations performed by peers, there are several abstract operations that are common to most peer-to-peer systems [37, 41], including: join, leave, search/lookup (for a particular resource/item) and route (for a particular lookup query or communication stream).
4.2 Setting the architectural context

Following the method for pattern specialization in Section 3.3, when specializing threat patterns one first needs to determine the derived architectural context by considering in what way the system-/technology-specific context affects the patterns' architectural context. Given the description in Section 4.1 above, and the decomposition layers outlined briefly in Section 2.2, the manner in which the peer-to-peer specific context affects the decomposition layers, and the corresponding abstractions which pertain most to such systems, can be briefly delineated as follows:

- **User interaction**: users are peer nodes or represented by peer nodes, therefore all input and output occurs locally or between peers. Identities of peers are usually defined by the overlay, and can be used as identifiers for routing (see below).
- **Data / storage management**: for content-based peer-to-peer systems, all peers store parts or the whole of some data or resource. Peers may store meta-data about the content, and may share the content globally.
- **Resource management**: all resources are managed either in a de-centralized manner (by the peers) or using a centralized server.
- **Distribution control**: is de-centralized, with each peer providing the same interface or at most certain super-peers providing additional functions. Operations may include join, leave, search and, for content-based systems, store and share.
- **Communication**: uses an overlay network, structured or unstructured, on top of an existing networking infrastructure. Since messages are routed via the peers themselves, they are more vulnerable to communication attacks.
- **Addressing**: is performed by the overlay, mapping identifiers to peers, which often (though not always) assumes a one-to-one logical to physical mapping.

Once the architectural context is set, we can (1) make hypothetical choices for how the realization abstractions are implemented and determine how the base patterns are affected in the new context; and (2) consult the existing literature for documented threats in the given system-/technology-specific context and relate them to relevant base taxonomy threat patterns within the given architectural context. In the three examples below we take the second of these two complementary approaches, presenting first the context (decomposition layer and corresponding realization abstractions), the base threat pattern that is specialized, followed by an explanation of the (specialized) threat.

- **Addressing** decomposition layer – **Addresses / identifiers** (propagated down to **Protocols / algorithms** and **Routing data structures (tables)**) – Route poisoning threat pattern: a set of peers that are compromised may begin to manipulate the routing information of a single peer, thus filling its tables with incorrect information. This would effectively “eclipse” the peer from the network. If this is done to enough peers, it would also be possible to split the overlay network into (potentially controllable) segments. This is suitably called an **Eclipse attack** in the literature.

- **Communication** decomposition layer – **Message channel** abstraction – Message flooding threat pattern: besides flooding the network with fabricated messages from the outside, a peer that has been compromised may also send out false queries within the overlay. This would be particularly unsatisfactory in an unstructured peer-to-peer system, and would clearly cause the performance of the underlying network to degrade, possibly leading to denial of service. This is called **Query flooding**.

- **Resource management** – **Protocols** abstraction (which can be propagated down to be related to **Distribution control** – **Operations** abstraction) – Resource exhaustion threat pattern: A peer can continually join and leave the peer-to-peer system, hence calling the join and leave operations on one or more peers or a centralized server and causing routing tables to be updated, messages to be sent out etc. In the literature on peer-to-peer systems, this problem is called **Churn**.

In like manner the process of relating known threats to base patterns and attempting to (manually) specialize base patterns in the new architectural contexts can be continued with all other threats, until an exhaustive list of threats is obtained.

In general, the choice of exactly which base patterns to specialize for a given context is up to the developers or the team constructing the taxonomy. In the present case, since peer-to-peer systems are a sub-class of distributed systems, we have chosen to specialize only the threat patterns which refer to the peer-to-peer overlay and which would not be readily recognizable from the base patterns by developers lacking domain and/or security knowledge. All other base threat patterns still apply in the peer-to-peer context (without requiring explicit customization).

4.3 Threat pattern list

The final list of specialized threat patterns, shown in Table 13 organized according to the threat classes proposed in Section 2, adheres to the common attacks in peer-to-peer systems identified in the work of Yue et al. [42], Urdaneta et al. [43] and Wallach [44], as well as in different recommendations and RFCs by standards bodies [45, 46, 47, 41].
We did not consider so-called rational attacks [48] (free-riding, incentive avoidance and others), consisting of strategized attempts by users or their representative nodes to (effectively) swindle the peer-to-peer system for gain. Such attacks can indeed become causes for security breaches (in the form of denial-of-service), but the threats posed by them will correspond to one of the threats described below.

The complete threat taxonomy for peer-to-peer systems consists of the simple union of the list of specialized threat patterns with the patterns from the base taxonomy. Although with respect to a threat modeling process the specialized patterns can replace their base counterparts in the relevant contexts, the base patterns still apply in other contexts (e.g. the patterns from the Network communication attacks class are applicable to the underlying network infrastructure), hence they are also included. If they were not to be included, the scope of possible attacks would be reduced, which would in turn lessen the value of the final taxonomy.

As in Sections 3.1 and 3.2, Table 13 presents the patterns in the now established table-row-based format, adding a further Threat pattern specialized column (“section”, in the terminology of traditional textual templates) and updating the Architectural context to be the Derived architectural context.

<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Threat pattern specialized</th>
<th>Derived architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity attacks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sybil attack</td>
<td>In a peer-to-peer system, virtual identifiers (i.e. peer identities) are mapped to physical nodes, ideally in a one-to-one fashion. In a Sybil attack a peer can obtain multiple identities that are mapped to fewer physical nodes, thus controlling larger parts of the overlay network.</td>
<td>Identity spoofing, Advantageous identity allocation</td>
<td>Distribution control peer operations (join), Peer address / identifier assignment in the overlay</td>
<td>Identity management (centralized authentication / identity assignment control, identity expenses)</td>
</tr>
<tr>
<td>Identity mapping</td>
<td>In a structured peer-to-peer system, a peer obtains a particular identity related to a lookup key, which allows control of the associated resource.</td>
<td>Advantageous identity allocation</td>
<td>Peer address / identifier assignment components</td>
<td>Identity management (identity assignment control)</td>
</tr>
<tr>
<td>Identity theft</td>
<td>A peer claims to possess a particular identity related to a certain lookup key, when in fact it does not.</td>
<td>Identity spoofing</td>
<td>Addressing (all)</td>
<td>Identity management (authentication)</td>
</tr>
<tr>
<td>Network communication attacks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eclipse attack</td>
<td>The routing table of a peer is filled with entries of malicious neighbouring peers, which “eclipse” the node. This can lead to fragmenting the overlay network if sufficiently many nodes can be eclipsed along communication boundaries.</td>
<td>Route poisoning</td>
<td>Overlay routing protocols and routing tables</td>
<td>Secure communications (secure overlays, multi-peer routing)</td>
</tr>
<tr>
<td>Overlay route manipulation</td>
<td>The normal overlay route information for one or more peers is tampered with so that communication is redirected, dropped, or routed to incorrect destinations.</td>
<td>Route poisoning</td>
<td>Overlay addressing, particularly in the context of structured overlays</td>
<td>Secure communications (secure overlays, multi-peer routing)</td>
</tr>
<tr>
<td>Query flooding</td>
<td>A route or content/lookup query (often for a non-existent resource) is repeatedly sent from one or more peers, causing an overload of the network. This is an especially strong threat in unstructured peer-to-peer overlays.</td>
<td>Message flooding</td>
<td>Communication message channels</td>
<td>Filtering (query limits per peer)</td>
</tr>
<tr>
<td>Stored data security</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Content spoofing</td>
<td>In a content-based peer-to-peer system, a peer returns forged, corrupt or old/obsolete content for a given query. A variation on this is content pollution, where a peer introduces corrupt content</td>
<td>Corruption on each peer</td>
<td>Storage data abstractions Storage security (hashing and content authentication, content versioning)</td>
<td></td>
</tr>
</tbody>
</table>

160
<table>
<thead>
<tr>
<th>Security threat pattern</th>
<th>Threat description</th>
<th>Threat pattern specialized</th>
<th>Derived architectural context</th>
<th>Mitigating security policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loss of accountability</td>
<td>with the same meta-data into the system for other peers to store.</td>
<td>Repudiation messages (queries) and protocols, overlay routing algorithms</td>
<td>Storage security (hashing, replication), Logging and monitoring (auditing, replicated logs)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uncontrolled actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage exhaustion</td>
</tr>
</tbody>
</table>

| Unauthorized operation on data | A peer performs some unauthorized operation on its stored data or resource. | Invoking unauthorized operations | Resources, Distribution control (all) | Authorization |

Table 13: List of specialized threat patterns for peer-to-peer systems

We should point out that the security policies suggested in Table 13 should only be taken as indicative – a number of solutions have been devised to deal with the various peer-to-peer specific threats, for which the reader is referred to [47, 41, 46, 43, 42] for an overview. While presenting corresponding misuse patterns for the threat patterns outlined above is outside the scope of this paper, the reader can refer to [22] for a fine-grained model template that describes Overlay route manipulation in more concrete detail and that can be converted to a misuse pattern with relative ease.

4.4 Threat taxonomies for other contexts

In like manner to the peer-to-peer taxonomy one could construct taxonomies for various technologies and/or other types of systems. For example, if a web application (system-/technology-specific) context is set, then one can begin constructing a taxonomy by considering Network protocol attacks against the TCP/IP stack (see [49, 50]), e.g. SYN flooding [51] (specializing the Protocol initial/end state exploitation pattern), or Network communication attacks against the HTTP protocol, e.g. response splitting [52] (specializing the Message integrity violation and Session state poisoning patterns), in the Communication functionality decomposition layer. While specializing some (general) threats would require domain expertise, many (web-based) threats are simple extensions that would require little change to the base patterns, e.g. SQL injection, XPath injection and others, which simply set different contexts for the Injection pattern.

If an application employs a peer-to-peer architecture for communication and a web application front-end for initial user registration (to give a hypothetical use case), then the peer-to-peer taxonomy from Section 4.3 could be combined – as a simple union – with the web application taxonomy to create a single application-specific taxonomy, which would allow developers to consider a highly relevant set of threats for each part of the application as required. Doing so has clear benefits for (security-wise) inexperienced developers performing threat modeling or any security analysis process aiming to encompass the widest possible range of threats.

Of course, we should emphasize that web applications and peer-to-peer systems are just two relatively general examples – pattern-based taxonomies can also be constructed for other, more specialized system types, such as cyberphysical, industrial control systems (see, for example, [53, 54]) for water, energy or other industrial processes. In the latter case there may be a need to specialize patterns multiple times to construct derived taxonomies for the different contexts.

As a final note regarding the construction of derived taxonomies, we should point out that in real-life situations it may not always be realistic to expect development teams to use our proposed method to specialize threat patterns. Ideally, in these cases security experts with domain knowledge in the target systems and technologies would prepare the necessary patterns and hence construct the taxonomies beforehand. In all cases, however, once a taxonomy for a particular system-/technology-specific context has been constructed, it is re-usable across related systems/projects, reducing the work required in the longer term.
4.5 Pattern-based taxonomies as classification schemes

In the introduction and earlier sections we referred to the fact that our base threat taxonomy, which plays the simultaneous role of a threat library, can be used to classify known threats for distributed systems. This can be seen from the list of specialized threat patterns for peer-to-peer systems in Table 13, where each specialized threat pattern is categorized under the more general base pattern. The complete threat library for peer-to-peer systems, which includes this time the threat patterns as patterns, can be used as a classification scheme (in accord with the purpose of a “pure” taxonomy) for specific peer-to-peer threats, where pattern instances, for example, Eclipse attacks against DHT tables, overlay route manipulation for a given algorithm etc., can be categorized accordingly. The same is true of other threats: SQL injection, XPath injection, LDAP injection, parameter manipulation and many others can all be classified as Injection threats, since they are either specializations or instances of the latter.

5 Related work

The literature contains a large number of proposals for attack and vulnerability taxonomies, many of which have already been surveyed in [11]. Many of these taxonomies, including more recent ones such as AVOIDIT [31], strive towards the goals set out by Hansman and Hunt [26], namely, to provide a taxonomy useful for classifying existing attacks in the broadest sense of the term. This can be useful for penetration testing or security evaluation when a system has already been implemented, or at least designed in sufficient detail, and incorporates particular technologies that may contain vulnerabilities and/or be susceptible to known exploits; however, they are not useful for security analysis during earlier development stages, and hence not for threat modeling. Moreover, we do not believe that worms, viruses etc. can be labeled as specific attacks in their own right; rather, they are automatic means of launching collections of specific attacks, such as port scanning (on specific ports), OS fingerprinting and others. It is these specific attacks that should be protected against during development, by considering them, for example, in the context of a threat modeling process as threats that are instances of our threat patterns.

Besides security analysis, attack taxonomies can also be used for intrusion detection, where knowledge about attacks is formalized in a fashion suitable for automation. Such taxonomies are contained in the work of Herzog et al. [27] (in the context of security ontologies), which is especially comprehensive, and Álvarez and Petrović [55]. Besides proposing a different classification scheme and not aiming for applicability to automated intrusion-detection, our taxonomy also differs in its being based on patterns, allowing it to be used simultaneously as a threat library in which threats are related to architectural contexts.

Despite their different applicability, we have used the security knowledge contained in a number of existing attack and vulnerability taxonomies as a basis for our own work – in particular, for creating our threat patterns, as explained at the outset of Section 3.

With respect to approaches combining both a classification scheme and software patterns, a closely related effort can be found in the MITRE organization's Common Attack Pattern Enumeration and Classification (CAPEC – see http://capec.mitre.org) list, which classifies attack patterns hierarchically, extending the work of Hoglund and McGraw [16]. Attack patterns, as we noted in Section 2 are related to threat patterns by a “realizes” relationship, which implies that the CAPEC attack patterns are complementary to our threat patterns. Besides the distinction in patterns, the CAPEC attack classification into mechanisms – data leakage attacks, resource depletion, injection, spoofing, time and state attacks, abuse of functionality, probabilistic techniques, exploitation of authentication, exploitation of privilege/trust, data structure attacks, resource manipulation and network reconnaissance attacks – differs from our scheme, even if with some overlap. Several of these classes, such as “abuse of functionality”, are much too broad, while most next level classes grouping attack patterns are much too specific (as attack patterns generally are), e.g. “WSDL scanning”, which applies only to systems based on web-services. This approach leads to hundreds of specific patterns (in fact, CAPEC catalogs more than 400), classified in the most general ways, rendering the classification bewildering for developers with little security expertise, and cumbersome for those who are experts already. In contrast, our taxonomy of threat patterns provides enough abstraction to encompass most common threats to distributed systems, but avoids a high level of generality in the classes and a high level of specificity in the patterns, rendering the classification more manageable and usable. The level of abstraction is not only (arguably) generally beneficial, but is also essential for security analysis processes and for threat modeling in particular, where threats can be considered early in the SDLC independently of any implementation details. Using our taxonomy for some form of security analysis does not, however, prohibit the complementary use of CAPEC (during later development stages), since attack patterns can be seen as particular realizations of our threat patterns, e.g. the WSDL scanning attack pattern referred to previously can be seen as a realization of the Scanning threat pattern in our taxonomy. The aforementioned overlap in the classification scheme (by attack mechanism) is, in fact, a benefit in this respect, since it could be used to relate our threat patterns with the attack mechanisms that are used to realize them.

A brief discussion of other related approaches to encapsulating threats or attacks can be found in [13].
6 Conclusion and future work

In this paper we presented an extensible, two-level pattern-based taxonomy of security threats for distributed systems, encompassing both threats to a system (first taxonomy level, Section 3.1) and threats to corresponding countermeasure realizations (meta-security threats) (second taxonomy level, Section 3.2). Each threat in our taxonomy is encapsulated as an abstract software pattern that can be specialized for different system types and/or technologies, increasing the flexibility and applicability of the taxonomy. We demonstrated this in Section 4 by specializing a number of (first level) threat patterns in the base taxonomy to construct a taxonomy for peer-to-peer systems. Similar extensions could be constructed for different technology-specific contexts, such as web-services [56] and web applications [10]); and distributed system types, such as multi-agent systems (MAS) [57], industrial control systems [58, 59], grid systems [60] and cloud systems [61, 62, 63] – which constitute distinct taxonomies in their own right, consisting of the base and specialized threat patterns. By combining the taxonomies we can also construct a single, application-specific (tailored) threat taxonomy, which, in effect, is simply an extension of the base taxonomy.

The base threat taxonomy itself can be said to possess nearly all the properties set out by Iigure and Williams [11] required of a taxonomy suitable for security assessment, namely:

- **Application specificity:** as discussed above, our base taxonomy is extensible by specialization, which allows the construction of application-specific taxonomies.
- **Layering or hierarchy:** our base taxonomy is divided into threat classes and threat patterns, the latter of which can be specialized and instantiated in various contexts to create arbitrary sub-hierarchies. The taxonomy is also stratified into two levels, for system (security) threats and security infrastructure (meta-security) threats.
- **Use of system-specific attack types:** specialization of base taxonomy patterns allows threats for different system-/technology-specific contexts to be captured.
- **Relation to system components:** the architectural contexts of each threat pattern ensure architectural relation for a given decomposition of an application's software architecture.
- **Classes need not be mutually exclusive:** as seen in Section 4.3, some attacks can be classified as realizing patterns from more than one threat class in the base taxonomy, which can be useful in practice, e.g. when searching for all attacks realizing a given threat.

Naturally, it should be kept in mind that the purpose of our threat taxonomy is to allow developers to consider threats during application design, rather than to find vulnerabilities during later development stages.

With respect to related approaches (Section 5), our threat patterns are neither too generic, nor too specific, nor too detailed; can be easily related to an application's software architecture; and encapsulate related mitigating policies – making them useful in a variety of security analysis processes both for existing and new systems. In both cases the patterns can be used as constituents of a threat library, which, as evidenced by Dhillon [9], greatly enhances the efficacy of threat modeling and reduces the expertise required to perform it; and an abstract threat taxonomy in the spirit of Microsoft's STRIDE, which helps developers to find relevant threats in different contexts via high-level categories. Specialization of the patterns to construct tailored taxonomies can help to ensure that a maximum number of threats are available for consideration during a threat modeling process, further reducing its expertise and effort requirements.

A valuable future direction with respect to using our taxonomy in threat modeling processes would be to precisely correlate the threat patterns with appropriate CAPEC attack patterns, whether directly or via misuse patterns (which aggregate attack patterns as part of their solution), allowing developers to consider possible realizations of a threat in a given context during development. This would also be beneficial for testing and evaluation purposes, where corresponding misuse and/or attack patterns used at the design stage can form the basis of a plan for penetration testing (cf. [64]).

The integration of our taxonomy into a complete security methodology for distributed systems (see [5]) is another, related research direction. Utilizing application-specific taxonomies that encompass various system-/technology-specific contexts would be particularly valuable for methodologies capable of being tailored in some fashion (see [65]) for different types of specific distributed systems (collaborative, peer-to-peer, web-service based etc.) (cf. [15]), and is something that we intend to explore in the future.

References

Epilogue

In this chapter we presented the second of our generic conceptual security artefacts, addressing threats in networked and distributed systems. Although in many ways it builds on the first artefact for architectural decomposition, it is self-sufficient, and can be used independently. In the context of the whole “toolkit” for engineering security methodologies, the threat library/taxonomy provides an offensive artefact supporting threat modeling for general and peer-to-peer distributed systems (and, through specialization, arbitrary distributed system types) as well as security testing and verification.
Chapter 6

(Generic conceptual security artefacts III part I)

Security solution frames for distributed authorization and policy/rule management

I'm still a hacker. I get paid for it now. I never received any monetary gain from the hacking I did before. The main difference in what I do now compared to what I did then is that I now do it with authorization.

— Kevin Mitnick

Prologue

Having considered the offensive side of security in the previous chapter, in this chapter and the next we turn our attention to developing generic artefacts for protecting systems, namely, instances of the Security solution artefact from the engineering meta-model of Chapter 3. Given our previous focus on security patterns and their advantages, it should not be surprising that the artefacts we present — called security solution frames — are based on, and, in fact, encapsulate and organize, security patterns.

As with the generic conceptual artefacts themselves as a whole, security solution frames can be viewed from several angles. In a sense, they are a response to the gaps found in the security patterns literature, insofar that the patterns encapsulated by the relevant frames are “fillers” for the different missing pattern classes — collaborative authentication models, authentication, key management and secure network communication (recall Chapter 2). Solution frames go beyond the patterns they encapsulate, however, in providing an encompassing structure in which patterns can be used across the development life-cycle and down different levels of abstraction. The actual patterns can be realized, or even replaced in the organization scheme, by other solution types, making them strongly methodology-agnostic.

In this chapter in particular, we present security solution frames for authorization — with an emphasis on authorization for distributed collaborative systems — and the supporting policy management infrastructure. The reason why collaboration is emphasized is that collaborative systems — more than other (distributed) system types — require more complex and thus interesting authorization models, allowing for the development of a richer set of patterns. This is not to say that the patterns are collaboration-specific; they are applicable to all distributed system types, but are more useful in a collaborative setting simply due to the inherent complexity of the corresponding models.
## Statement of Authorship

<table>
<thead>
<tr>
<th>Title of Paper</th>
<th>A Software Engineering Approach to Authorization in Distributed, Collaborative Systems using Security Patterns and Security Solution Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication Status</td>
<td>O Published, O Accepted for Publication, O Submitted for Publication, O Publication style</td>
</tr>
</tbody>
</table>

## Author Contributions

By signing the Statement of Authorship, each author certifies that their stated contribution to the publication is accurate and that permission is granted for the publication to be included in the candidate's thesis.

### Name of Principal Author (Candidate)

Anton V. Uzunov

**Contribution to the Paper**

Conceptualization of work, its realization (research) and documentation (wrote manuscript). Acted as corresponding author.

**Signature**

Date 11/10/2013

### Name of Co-Author

Eduardo B. Fernandez

**Contribution to the Paper**

Provided references and commented on manuscript.

**Signature**

Date 13/10/2013

### Name of Co-Author

Katrina Falkner

**Contribution to the Paper**

Supervised development of work, helped to evaluate and edit the manuscript.

**Signature**

Date 11/10/15

### Name of Co-Author

**Contribution to the Paper**


**Signature**

Date
A Software Engineering Approach to Authorization in Distributed, Collaborative Systems using Security Patterns and Security Solution Frames

Anton V. Uzunova\textsuperscript{a,*}, Eduardo B. Fernandez\textsuperscript{b}, Katrina Falkner\textsuperscript{a}

\textsuperscript{a}, School of Computer Science, The University of Adelaide, Adelaide, South Australia, Australia, 5005
\textsuperscript{b}, Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, 777 Glades Rd., Boca Raton, FL 33431

* Corresponding author

E-mail addresses: anton.uzunov@adelaide.edu.au (Anton V. Uzunov), ed@cse.fau.edu (Eduardo B. Fernandez), katrina.falkner@adelaide.edu.au (Katrina Falkner)

Abstract:
The design and implementation of an authorization infrastructure is one of the most important aspects of engineering a secure software system. Unlike other system types, distributed, collaborative systems often require custom, fine-grained authorization models and enforcement approaches that are able to take into account a range of semantic subtleties. In this paper, we present a comprehensive, pattern-oriented software engineering approach to authorization for distributed, collaborative systems that allows developers to build custom, application-specific conceptual authorization models in a simple, efficient and extensible manner, and to make informed decisions regarding their enforcement in software. Our authorization approach is embodied in a novel security solution structure, which groups together related patterns in different sub-structures, horizontally and vertically, for a single high-level security policy (in this case authorization). Using the solution structure developers are led progressively from abstract concepts towards concrete designs via a number of levels of abstraction, which imply solution refinement and simultaneously correspond to stages of the software development life-cycle. Our approach can be used not only to create custom authorization solutions, but also to analyze existing ones. We use our approach in this capacity to briefly analyze two existing authorization infrastructures: one based on UCON for collaborative Grid systems and another based on ZBAC for SOA-based systems. Since software engineering approaches to authorization such as the one presented here have not received much attention before, we conclude with an extensive discussion of future research directions arising from our work.

Keywords: security engineering; software engineering; security patterns; authorization; access control; distributed collaborative systems; distributed systems security

1 Introduction

Security is becoming an increasingly important aspect of distributed systems \cite{1, 2, 3} and of collaborative systems in particular \cite{4, 5, 6}, where multiple groups of users and components interact not only across physical but often across organizational boundaries as well. One of the most important aspects of engineering a secure distributed, collaborative system is to determine which actions by system entities are and are not allowed – i.e. the design and implementation of an authorization infrastructure \cite{7, 8}.

Traditionally, models for authorization have been classified into discretionary (DAC), mandatory (MAC), role-based (RBAC) \cite{9}, and, more recently, attribute-based (ABAC) \cite{10, 11}. A number of arguments exist in the literature, however, which suggest that traditional authorization models are not always appropriate or sufficient for distributed settings \cite{12, 13, 14}, and certainly not for distributed, collaborative settings with fine-grained authorization requirements \cite{15, 4, 16, 17, 18, 19}.

In their survey of collaborative authorization models, Tolone and colleagues \cite{18} stipulate a number of characteristics which such models should possess, the most relevant of which can be summarized as follows:

- Authorization models should be generic and should permit rights “to be configured to meet the needs of a wide variety of cooperative tasks and enterprise models.” \cite{18}. It should be possible to specify rights based on various information, such as roles of users, contexts etc.
- Authorization models should relate to shared objects (environments, information, resources etc.) of any type and at different levels of granularity.
- Authorization models should permit, as required, for the specification and modification of authorization rules at run-time “depending on the environment or collaboration dynamics” \cite{18}.

Naturally, a number of models that satisfy some or even all of these requirements can be found not only in the references already cited, but also in a wide variety of publications containing models for general, distributed and collaborative settings, whose number is so great, in fact, that one could fill a whole compendium if one were to enumerate just a fraction of them. Such models are continually being created to satisfy the new authorization
requirements posed by the development of new systems [21].

While these models are valuable additions to the growing body of security knowledge, they also give rise to two important (even if speculative) questions. The first of these is whether the existing authorization models can take into account the semantics of the collaborative system to which they are applied. Since different systems have different needs, it is unlikely that a “one-size-fits-all” approach is viable. Indeed, the requirements stipulated above collectively imply that distributed, collaborative applications require custom authorization models (cf. [20]), that can take into account a particular system’s semantics. The second question pertains to whether the existing authorization models, many of which are theoretical, are actually being used in practice. More importantly, the question can be reformulated as whether software engineering teams, which often take a rather scornful attitude to incorporating security during development (see [22, 23, 24]), are sufficiently interested in taking up and realizing the often formal descriptions of the models in everyday collaborative software, and sufficiently experienced to do so. While the former point could easily be addressed, and indeed in many situations may not be relevant, the latter point remains unanswered. What is certain is that the large majority of publications on authorization originate from academia, not the industry, and hence emphasize formal, theoretical and purely scientific values. One would be pressed to find a comprehensive, software engineering approach to authorization that guides security-inexperienced software developers (the predominant workforce [24]) from the abstract concepts of authorization to concrete solutions that can be used when designing distributed, collaborative systems.

In this paper, we seek to address the latter points and provide a comprehensive software engineering approach to authorization, filling what we believe is an important gap in the literature. Our authorization approach is embodied in a novel security solution structure based on one of the most developer-friendly design constructs: software patterns [25, 26] and, more specifically, security patterns [27, 28]. The solution structure groups together related patterns in different sub-structures, horizontally and vertically, for a single high-level security policy – in this case authorization. Using the patterns from our authorization solution, one can build custom, application-specific conceptual models suitable for a range of distributed, collaborative systems satisfying the requirements set out by Tolone and colleagues, in a simple, efficient and extensible manner as befits pattern-based approaches. The enforcement of these models in software is similarly guided by the use of patterns. The benefits of the solution structure carry over directly to the development of the authorization infrastructure, as developers are led progressively from abstract concepts towards concrete designs via a number of levels of abstraction, which imply solution refinement and simultaneously correspond to stages of the software development life-cycle.

Naturally, in presenting the constituent patterns for authorization we do not seek to cover the whole spectrum of concepts used in the countless number of pre-built models, but only a broad selection that was deemed especially useful for distributed, collaborative systems. Indeed, since patterns imply customization, the range of models that can be created is significantly broader than what is achievable by utilizing our authorization model patterns and their combinations based purely on the descriptions provided. Taken together in their uncustomized form, the patterns fill another gap in the literature identified in [28], namely, the documentation of security patterns for collaborative authorization models.

The rest of this paper is structured as follows. In Section 2 we present our solution structure in some detail and discuss its benefits. In Section 3 we present our authorization solution, beginning with some brief background on authorization and proceeding to describe the various patterns for building an authorization model and enforcing it in software. Section 4 presents a complementary solution for handling the security policies required for the authorization infrastructure, which simultaneously serves to illustrate the value of the solution structure from Section 2.

Our approach can be used not only to create custom authorization solutions, but also to analyze existing ones. Therefore, to demonstrate the usefulness of our approach to authorization as well as the solution structure which embodies it, we analyze the UCON-based collaborative authorization framework of Zhang and colleagues [5] and the ZBAC model and enforcement approach of Li and Karp [29, 30, 31] for SOA-based systems.

Finally, in Section 6 we discuss related work both in security solutions and in authorization, and in Section 7 we conclude by summarizing the paper and providing an in-depth discussion of a number of important research directions stemming from our work.

2 Security tactics, patterns and solution frames

From a software engineering perspective, developing a secure system of any type requires a set of specific security solutions to be applied during the development life-cycle, especially during architectural design [32]. A number of such solutions have been presented in the literature, including security tactics [33], security patterns [27], generic security architectures and components [34], aspects [35, 36] and others, each with varied characteristics (see [37]). Of these the first two – security tactics and patterns – are by far the most mature and most widely used in the industry.

Security tactics attempt to capture high-level, security design decisions in a simple, usable format. Each tactic is related to a high-level security policy, however, by nature, a tactic does not provide any details on how that policy can be realized. In this sense, tactics simply encapsulate high-level policies for use at the architectural level. Security patterns on the other hand, are direct inheritors of the software pattern concept developed by the architect Christopher Alexander, and hence aim to provide guidance and representative architectural models for developers to
use in order to realize more specific security policies (see [28]). There are numerous security patterns in the literature dealing with a wide variety of concerns, however, they are not structured in any way, and require classification schemes external to the patterns, i.e. to the solutions themselves, in order to navigate the solution space. Previously, the relationships between security tactics and patterns has been seen as either antithetical [38] or complementary [39]; the bridge between tactics and patterns, however, has not received much attention.

Although tactics and patterns have been used in practice, their main advantage – that they encapsulate implementation-independent solutions – is also one of their disadvantages: both tactics and patterns, taken in isolation, or even used within the bounds of a systematic approach, only describe what must be done at a relatively high level of abstraction, leaving out a number of details that are required for the full realization of a given solution.

In this section we present a new solution structure called *security solution frames*, which form a bridge between tactics and patterns. Each tactic is realized by a unique solution frame, which encompasses a number of patterns vertically and horizontally that can be used by developers to progressively realize some or all aspects of the solution.

Figure 1 presents a conceptual model relating solution frames, tactics and patterns (with the individual solution types appearing in different colors or, if viewed without color, different shades of grey, purely to differentiate them from other elements and from each other).

Security solution frames draw inspiration from ideas present in Microsoft’s pattern frame structure [40] and the pattern frames of Delessy and Fernandez [41] – both of which seek to organize the space of patterns according to different levels of abstraction (vertically) and, for Microsoft’s frame, into development viewpoints (horizontally) – to encapsulate related patterns at different levels of abstraction stemming from a common, root problem or policy, encapsulated in a particular security tactic. In a similar fashion, solution frames partition the solution space horizontally into separate (but related) concerns via *pattern families*; and vertically, into levels of abstraction.

Pattern families are collections of related (simple) patterns, with their own independent division of abstraction,
for solving a root problem that is an aspect of the solution for the whole frame itself. Patterns between the levels of abstraction, whether with respect to the frame or the pattern families, refine the patterns from the previous level; at the same level, patterns can be related via several solution relations (see below). Families treat one part or aspect needed to fully realize a tactic. It is not necessary to utilize patterns from all families for a given tactic, since the realization of a tactic in a given context may only require a subset. Using a pattern from a given level of abstraction realizes the initial security policy (and hence tactic) in part, at that abstraction level. The whole sequence of patterns from appropriate pattern families realizes the tactic in full.

A given security pattern may be part of more than one family (e.g. a DEDICATED SERVER pattern applies to both authorization and authentication), depending both on the nature of the pattern itself, and can contain patterns that appear in other families. When patterns from one solution frame (source) are used in another frame (target), their effects on the existing patterns (in the same frame) can be documented as part of the relevant (target) pattern families, making frames self-contained.

Four canonical abstraction levels for the frame determine the overall vertical divisions into four areas of responsibility within the solution or, looked at from a process viewpoint, stages through which the solution passes, which simultaneously correspond to the stages of the software development life-cycle (SDLC):

- **Conceptual analysis**: contains one or more abstract security patterns [42] capturing the abstract concepts required in realizing the solution.
- **Abstract architecture**: contains patterns that determine an high-level, abstract architecture for the solution.
- **Solution design**: contains patterns refining the abstract architecture and providing representative architectural models that suggest more concrete software components and connectors.
- **Practical implementation details**: contains patterns that address the details and specialized aspects of the solution design, refining the solution so that it can be instantiated (with necessary customization, as befits a pattern-based solution) into the system's architecture.

The names of the abstraction levels are purposefully not aligned with any standard development stage names to emphasize the generality of the levels in mapping to various paradigms, e.g. object-oriented development, where one possible mapping is: Conceptual analysis and Abstract architecture to Requirements analysis, Abstract architecture and Solution design to Design, and Practical implementation details to Detailed design; or Model-Driven Engineering, where the mapping is Conceptual analysis to the Computation-Independent Models (CIM) layer, Abstract architecture and Solution design to the Platform-Independent Models (PIM) layer, and Practical implementation details to Platform-Specific Models (PSM) layer.

Pattern families can be divided vertically into any number of independent levels of abstraction, where they span multiple canonical levels or where one canonical level spans multiple family abstraction levels. Thus, canonical levels are a characteristic of the solution frame, and an independent set of abstraction levels is characteristic of each family. Of course, in a family structure, patterns do not have to appear at every level of abstraction in that family, or for the frame as a whole. At the top of every family, however, there is always an abstract security pattern capturing the high-level solution addressed by that family. Pattern families of security (software) patterns always end with a Technology realization level as part of their hierarchy of abstraction levels, which contains either low-level idioms (see [25]) or descriptions of various technologies as done in [43]. Therefore, the progression of patterns downwards in the pattern family, or, equivalently, across the development stages, guides developers from concepts, to architectural solutions, to concrete design enriched with enough detail, becoming closer to templates that can be “plugged” into an architecture (see [44]).

Besides patterns relating to each other, families and frames can also be related via a set of solution relationships, which adapt a number of standard pattern relationships appearing in the literature [45, 41, 46]:

- **requires**: where solution A requires solution B for its realization
- **supports**: where solution B supports solution A in its realization
- **depends on**: where the realization of solution A is dependent in some fashion on the way solution B is realized.
- **derives from**: where solution B specializes or derives certain characteristics from solution A, having a more specialized context of application.
- **alternative**: where solution B can be used in place of solution A
- **clashes**: where either solution B inhibits the effectiveness of solution A or solutions A and B are altogether incompatible and cannot be realized simultaneously.

Relationships can be either one-way or two-way, for example, when solution A and B rely on each other mutually. Security (software) patterns from different levels of abstraction also implicitly fulfill a “refine” relationship, analogous to the “realize” relationship of [41], where pattern B refines pattern A if A is at a higher level of abstraction than B. Figure 1 uses loose UML semantics for the Solution relationship association class, which should be interpreted in light of the explanation above, namely, as meaning that the same class of relationships can exist between the different solution types.
Besides encapsulating pattern families of security (software) patterns, each solution frame has a single family that encompasses a set of *security process patterns*. Security process patterns are applications of the process pattern concept of Ambler [47] introduced first in the context of object-oriented development methodologies. Like their general counterparts, security process patterns can be classified into three levels of decreasing granularity, ranging from phase patterns, stage patterns and task patterns. They can also be classified as macro-process patterns — pertaining to the phases, stages and tasks of security methodology; or micro-process patterns — pertaining to activities resulting from the application of a particular security solution (see Figure 1). The micro-process patterns in the single process pattern family (in a given frame) can also reside at different levels of abstraction, however, patterns at lower levels do not necessarily refine ones at higher levels.

Instantiating a particular security frame implies progressively selecting and instantiating patterns from the relevant families to create a sequence of patterns (see [48]), or even a whole tree-like structure, which is itself reusable. Such tree structures capture the design decisions taken to realize a solution in a particular situation, and can also promote greater traceability from high-level security policies, their progressive refinements into specific policies, and their respective (partial) realizations via the use of patterns (see [49]).

Since each solution frame, the encapsulated pattern families and even the abstraction layers within the families are autonomous, the various solutions can be used independently, while realizing one particular root policy (encapsulated in the tactic), their parts can be used in a wide variety of situations in a methodology-agnostic fashion, or within any pattern-based security methodology (see [28]).

In what follows we present our solution frame for authorization, which realizes the “Authorize users” security tactic of [33].

### 3 Authorization solution frame

The subject of authorization in general, as well as for distributed and collaborative systems in particular, has received much attention in the literature, with a variety of perspectives offered in the works of Bacon and Moody [50], De Capitani di Vimercati, Samarati and colleagues [51, 52, 53], Jie et al. [54], Tolone et al. [18], Woo and Lam [55] and many others. From an abstract point of view, authorization is concerned with placing restrictions on the operations (actions) performed by entities (subjects) in a system, dictating a set of allowable states. These restrictions can be expressed as authorization policies, which are enforced in the system by a concrete software infrastructure. The prevalent view in the works above, therefore, is that realizing an authorization solution in a distributed system consists in specifying high-level policies, creating a formal model to support the policies and enforcing the model in software.

In our view, high-level policies are equivalent to security requirements that, via a particular conceptual model, are embodied in concrete authorization rules, which implies that authorization is split into only two factors: *conceptual models* (dictating the structure of the rules and the notional capabilities of the authorization infrastructure, i.e. the *what* aspect) and *enforcement architectures* (containing the concrete implementation details, i.e. the *how* aspect).

Our authorization solution frame presented in this section reflects the latter two-fold structure accordingly, and consist of two related (via a two-way “requires” relationship) security pattern families for building a conceptual authorization model, and for realizing or enforcing the model in software, respectively. Since the conceptual model is generally created during the requirements analysis development stage, realized as an architecture during design, and refined further (including with concrete implementation-related details) during detailed design, we have partitioned the frame vertically as follows:

- **Conceptual analysis abstraction level (corresponding to the requirements analysis development stage):** contains patterns for building a conceptual authorization model beginning from the abstract authorization pattern in [56]. The patterns also extend into the next canonical abstraction level, since the model may need to be re-visited during architecture development.

- **Abstract architecture and Solution design abstraction levels (corresponding to high-level architecture development and design stages):** contain patterns for realizing the conceptual authorization model in software, beginning with an abstract base pattern adapting Delessy et al.’s PBAC [57], and for refining the architectural authorization model in two steps:
  - the type of enforcement point: interceptor or server; and
  - using push or pull semantics.

- **Practical implementation details abstraction level (corresponding to the detailed design development stage):** addresses issues in information handling, rule specification and contains patterns for mapping onto particular technologies (technology realization).

The process pattern family that is part of the solution frame consists of only one (micro-process) pattern considering issues in protecting the authorization infrastructure itself. The horizontal and vertical structure of the solution frame is shown schematically in Figure 2. In what follows we present the pattern families constituting the authorization frame. All patterns are referred to in the text using *SMALL CAPS* font, and the type of pattern is indicated in the headings in brackets.
3.1 Conceptual authorization model pattern family

A conceptual authorization model defines the structure of the rules used for authorizing actions and hence determines the (theoretical) capabilities of the authorization subsystem/modules in the final software system. The rules themselves can be described in some formal language, for example, which can be compiled [58] or encoded in a CREDENTIAL [59]. What can be expressed by the rules is determined by the model, e.g. group structures, attributes for objects etc., while the expressiveness of the rules as such is determined by the language used, e.g. whether multiple objects can be expressed using a wildcard (*) or must be specified one by one (see the POLICY LANGUAGE pattern further below).

In our approach, a conceptual authorization model is created by utilizing an abstract “base” authorization pattern and instantiating pattern modifiers [60] to build up the rest of the model to suit the custom needs of the application. In this context the modifiers act as extensions to the base pattern, enhancing it with various features as required; they likewise effect the architectural models which are used to enforce the conceptual authorization model, as well as the next level of patterns by adding and/or modifying the functionality of certain (actual or representational) components.

Using the patterns presented here allows application-specific models to be created in a simple and efficient manner. They also help to classify and evaluate existing models, both those captured as patterns and more generally, to determine whether they are appropriate for use (e.g. because a standard model is supported by formal proofs, or is accompanied by a canonical policy language) or extension.

As a final note, we should mention that the viewpoint our patterns take is action- (vs access-) centered. Often authorization is taken to refer to accesses, i.e. the part of access control dealing with authorizing access to particular resources. Every access, however, implies some action preceding it, i.e. authorization refers more precisely to the broader notion of action, rather than access, which entities in the system, representing a user or otherwise, can undertake. A simple example illustrating the distinction can be seen in a process reading a file and sending a packet of data over the network: the action of reading the file can be regarded as an access request (read operation), but the send operation is more properly an action rather than an access, even if the underlying system represents network interfaces as files internally, and could in fact consist of multiple accesses. In both cases, however, one speaks of authorization. From this perspective, rights should be assigned to operations or actions in a system to determine what can be done, or more precisely, what actions are allowed and not allowed, rather than simply what can be accessed or not to perform a given operation.

Regarding abstraction levels, the base authorization pattern in this family resides on the Conceptual analysis level, while the modifiers collectively span the Conceptual analysis and Abstract architecture canonical abstraction levels.
3.1.1 Abstract authorization (abstract security pattern)

The most abstract pattern for the conceptual authorization model is the ABSTRACT AUTHORIZATION pattern of [61] (called simply AUTHORIZATION in that reference). In any authorization model based on ABSTRACT AUTHORIZATION, rights or privileges (R) are defined on the actions or operations (A) which subjects (S) can perform on objects (O) (see the conceptual model in Figure 3). An object here may refer to the system as a whole, in which case the rights refer to “system-wide” actions, for example, syscall restrictions. The set of conditions or predicates (C) for each authorized action determine what must be satisfied for the authorization: the possession of certain tokens, satisfaction of certain time-based constraints (last access not less than 1 hour) etc. (see [62]). The latter conditions are made concrete depending on the conceptual model. Thus each authorization rule can be represented as a tuple (S, O, A, R, C), and the set of all authorization rules determine the restrictions of allowable actions in a system, i.e. the authorization policy.

The actions (A) defined by the authorization model refer to the operations abstraction in the distributed system conceptual model. The granularity of authorizations are thus determined by aligning rights with the requisite granularity of operations: function calls (for given interfaces), syscalls (system-wide operations) etc.

Since authorization may occur over multiple domains, a global namespace for subjects, objects etc. is required.

3.1.2 Pattern modifiers

Modifiers are specific types of patterns that can extend, refine or determine a variation of other software patterns (see [60]) when used in combination. They may not and need not present complete design solutions by themselves. Below we outline 17 pattern modifiers capturing a wide variety of authorization concepts that can be used to refine and extend the ABSTRACT AUTHORIZATION pattern presented above. Our modifiers can be applied to enrich existing patterns of pre-made authorization models as well, including RBAC, ABAC and others (see [63, 64, 27]), which are in effect encapsulated versions of ABSTRACT AUTHORIZATION with several of the modifiers included.

The purpose and effects of the modifiers (following [60]) are documented as part of the solution sections in the minimalistic template used below, which contains a Motivation section (the main idea or problem of the modifier), a Solution section (modifier core), optionally an Example implementation workflow section (giving hints on how the modifier can be realized) and optionally an Architectural effect section (indicating how the modifier influences the authorization architecture by introducing additional components or placing certain constraints). Where appropriate we refer the reader to existing patterns in the literature or examples that instantiate the relevant modifier as part of the solution section.

The modifiers can be loosely divided into two classes:

- **static** – affecting the structure of authorization rules only; and
- **dynamic** – affecting the run-time interaction of components as well;

which we present in sequence.

3.1.2.1 Group (static)

**Motivation**: A number of subjects, objects or rights are related or are too numerous; additional structure is required to form collections of subjects, objects or rights.

**Solution**: A group structure can be imposed on the subjects, objects, rights and even conditions for a given authorization (see, for example [20, 16]). Group structures are hierarchical in that each entity is itself a (minimal) group, and groups are therefore made of sub-groups, potentially recursively. In the case where grouping is ubiquitous, collections of rights relating to collections of objects can be assigned to collections of subjects, making for flexible and powerful authorization policies.

The group modifier allows arbitrary subject, object and rights specification by grouping the respective elements into a global “wildcard” category. In the latter context, one can specify sets of actions allowable to all subjects, or on all objects, however, this requires corresponding rule specification support (see POLICY LANGUAGE in Section 3.2.3.2.1).

A common group includes ownership, which determines which subjects “own” which objects. Groups structures can also impose a kind of type system on objects (as required), wherein objects belonging to certain groups can be considered as being of a particular type.
3.1.2.2 **Role (static)**

**Context:** Any multi-user system in which users are assigned different privileges according to their function.

**Motivation:** There is a need to support users with different security profiles and to reduce the number of authorization rules.

**Solution:** Following [65] and [61], a Role is interposed between subjects and objects for additional indirection. In contrast to the Group modifier, Roles are assigned based on functionality; a process or group of processes can possess multiple Roles.

**Example implementation workflow:**
- identify appropriate roles in the organization or relevant to the system
- assign rights to the roles
- assign roles to users

3.1.2.3 **History (static)**

**Motivation:** To base future authorization decisions on previous ones.

**Solution:** Keep a history of granted rights or actions performed by a subject in the system, and make future authorization decisions accordingly. This requires some form of logging of actions, or transactions that keep a trail of relevant operations and/or granted rights.

3.1.2.4 **Attributes (static)**

**Motivation:** To determine allowed actions based on dynamically changing properties of a subject or object instead of identities.

**Solution:** Define a set of attributes that together characterize a subject or object. Attributes are assigned to some form of subject and/or object descriptors (for example, CREDENTIAL [59] or a certificate), and checked at run-time. The use of attributes can substitute the need to identify a subject explicitly via authentication, verifying instead whether the subject's attributes satisfy certain criteria such as thresholds for length of time running, personal user characteristics etc. In such instances attributes must be protected from modification to avoid spoofing. See also the MBAC / ABAC model pattern in [64] and later [63].

3.1.2.5 **Inheritance (static)**

**Motivation:** Rights possessed by one set of subjects may be relevant to another set of subjects based on function or some other connection. Specifying them again is time consuming and inflexible.

**Solution:** Make rights inheritable by individual subjects (see, for example, [20]), roles (see Role modifier) or groups (see Group modifier) in a hierarchical manner. All rights belonging to a parent set of subjects will also belong to the children, reducing the number of rules requiring specification.

3.1.2.6 **Implication (static)**

**Motivation:** To reduce the number and complexity of rules when subjects and/or objects are related and certain rights are required naturally by other rights.

**Solution:** Implement an “implies” relationship between rights. For example, a right to write to a document may be related to a right to read a document by an implies relationship, in which case any subject possessing the write right will automatically possess the read right as well, without the need for explicit specification. See [66] as well as [20] for a practical application.

3.1.2.7 **Exceptions (static)**

**Motivation:** Certain special circumstances or states in the system may require special rules that are outside the scope of those specifiable in the authorization model.

**Solution:** Allow exception rules, which override the usual rules under certain pre-defined conditions, such as break-glass policies [67] (see also [29, 68]). Exception rules can be checked on each relevant access, or activated via particular events (see the ADAPTATION modifier).

**Architectural effect:** To detect whether a given authorization is an exception, either a monitoring component must be used to determine matching system states, or a set of additional checks must be introduced into the existing infrastructure.

3.1.2.8 **Fine Granularity (static)**

**Motivation:** The granularity of actions determines the amount of control the final authorization subsystem can exert in restricting certain actions. In collaborative applications one often wishes to protect an object by applying different
policies to different object parts.

Solution: Allow rights associated with objects and/or actions to refer to finer levels of granularity. Associate rights, for example, with individual function calls instead of whole transactions; or associate rights with sub-objects (e.g. lines of text in a file) as opposed to large groups of objects (files). This has impact on the implementation of authorization, in that finer-grained actions usually imply greater overhead since they can be performed more often.

3.1.2.9 Negative Rights (static)

Motivation: The number and complexity of rules must be reduced, and specifying rights only for what is allowed is inflexible.

Solution: Allow the specification of negative rights, i.e. to specify what cannot be allowed (see [20, 69]) as well as positive rights (what is allowed). Negative rights can simplify some rules, and can eliminate a number of repetitive rules.

3.1.2.10 Context (dynamic)

Motivation: To determine allowable actions in different run-time situations.

Solution: Decide whether a given action is allowed based on the run-time contexts of that action [70], i.e. by taking a snapshot of the system at a given point in time and determining allowable actions based on that snapshot, taking into consideration information about the system environment, subjects and/or objects. Contexts can be associated with subjects, objects or actions (see [71, 72]). The context of an authorization may also include specific run-time data or attributes (see ATTRIBUTE modifier).

3.1.2.11 Delegatable Rights (dynamic)

Motivation: When chaining together multiple actions using multiple software components, intermediary components may need the rights of the initiating component to perform an action.

Solution: Allow delegation of rights across subjects with time limits. To ensure the principle of least privilege, the delegatable rights must be minimal for the actions performed. Delegation can be single-time (restricted to one subject), or recursive, which allows transitive authorization. The rights should be revocable, using, for example, NEGATIVE RIGHTS or a blacklist. Not all rights should be delagatable, and there should be a mechanism (e.g. a Boolean copy flag) to distinguish delatagable from non-delegatable rights.

3.1.2.12 Security Session (dynamic)

Motivation: To allow continuous permission during a given context, and/or to delimit the authorization within certain temporal bounds.

Solution: Create a time-limited session during which certain groups rights (in conjunction with the GROUP modifier) or roles (in conjunction with the ROLE modifier) are activated – see the CONTROLLED ACCESS SESSION pattern in [73]. Once an action by a subject on an object has been authorized in a session, it can be performed subsequently without further re-authorization. This modifier can also be combined with the STATE modifier (below) to enforce state changes controlling the re-authorization policies for the session.

Architectural effect: Since some form of session tracking is needed to realize this pattern, the authorization infrastructure needs a separate session management component.

3.1.2.13 Trust (dynamic)

Context: A system in which components can join and leave dynamically, such as peer-to-peer systems or collaborative systems with dynamic group membership.

Motivation: Two subjects that are endpoints of an association may not have sufficient information about each other, and must rely on extraneous means of authorization.

Solution: Use a protocol in which the two subjects at the endpoints of an association incrementally reveal security-sensitive data [74], such as credentials (see the CREDENTIAL pattern – [59]) to build a trust relationship. This relationship can then be saved for subsequent sessions (see SECURITY SESSION above), scoped by a time restriction. Trust values can be estimated quantitatively using a trust function (metric) – see [75]. They may also depend on the CONTEXT of the authorization. Details on potential protocols and strategies for implementation can be found in [76, 75, 77, 78].

Architectural effect: Usually a trust management component is required, together with a repository of trust values that change over time.

3.1.2.14 Adaptation (dynamic)

Motivation: To change the authorization rules dynamically.
**Solution:** Adapt the access control rights according to changing contexts (in conjunction with the CONTEXT modifier), or as a result of events in the system. Well-defined points of adaptation should be set, so that certain changes in the system are reflected in the rules themselves. This allows limited variation, so that it is not possible to abuse the adaptation and create arbitrary rules via a sequence of valid system states that trigger changes. See also [79] for a survey of several different approaches to adaptation.

**Architectural effect:** Adapting the authorization requires trigger-points and/or event handlers and monitors, which can be realized using the OBSERVER and MEDIATOR patterns [26], respectively.

### 3.1.2.15 State (dynamic)

**Motivation:** There is a need for different rights to apply to different system states at run-time.

**Solution:** Assign rights to a subject that depend on the state of its processing, such as the processing phase, or on the state of the object being accessed. The former case is an essential part of the TBAC (Task-Based Access Control) model, where the authorization decisions can vary depending on the state in which a process exists at some time (see [18]). The STATE modifier can be seen as a variation of the Adaptation modifier, with the distinction that Adaptation implies run-time changes to rules according to certain heuristics or meta-rules, while State also implies static specification of rules for known states. State changes can also encompass certain run-time scenarios such as continued usage, function calls and others.

### 3.1.2.16 Location (dynamic)

**Motivation:** How to take into account physical location as part of an authorization decision.

**Solution:** Attach an identifier at run-time to each action requiring authorization, to determine the physical location of the source component. Such identifiers may indicate which node the component resides on (see [3]), as well as to what domain it belongs.

### 3.1.2.17 User-driven Authority (dynamic)

**Context:** Any applications in which the user must have control over assignment and/or management of rights, especially collaborative applications.

**Motivation:** Users who have control over a particular (shareable) resource may need to assign and/or manage rights dynamically, without the burden of pre-assignment or a special administrator.

**Solution:** Make authorization rights user-driven, i.e. assigned by users to objects during run-time. This may imply ownership rights (see GROUP modifier, Section 3.1.2.1) or control rights (see, for example, [17]) over the objects.

### 3.1.3 Relationships between modifiers

The modifiers are related between each other at a basic level as shown in Figure 4. Since each modifier can be customized upon instantiation, other relationships may arise during the creation of the authorization model, e.g. TRUST based on a metric function that adds points for belonging to a certain GROUP would imply a “require” relationship between those two modifiers, which is not shown in the figure.

### 3.2 Enforcement architecture pattern family

The enforcement architecture realizes the conceptual model in software. The pattern family is split into four levels, the first of which is concerned with creating an abstract architecture (mapping to the Abstract architecture canonical level); the second level addresses concrete design decisions in realizing the abstract architecture (mapping to the Solution design canonical level), the third levels addresses the authorization strategy and practical implementation details (mapping to the Detailed design level), and the last level addresses technological realizations. We only present the first three levels in detail.

#### 3.2.1 Abstract architecture level

##### 3.2.1.1 Abstract authorization architecture (abstract security pattern)

The base (abstract) security pattern for enforcing a conceptual authorization model is a variation of the PBAC pattern appearing in [57]. PBAC proposes an architecture consisting of four main components in accordance with the general principles of authorization systems [80, 81]:

- **PEP (Policy Enforcement Point):** enforces all authorization checks
- **PDP (Policy Decision Point):** makes all authorization-related decisions
- **PIP (Policy Information Point):** gathers all information necessary for authorization during run-time
- **PAP (Policy Administration Point):** handles administration of authorization rules
The last component addressing the administration of authorization rules is an essential aspect of the pattern, however, since it is related to the Security Information Management solution frame presented later (see Section 4), it will not be considered here explicitly. This leaves three main elements, which together make the core of our modified PBAC pattern, named ABSTRACT AUTHORIZATION ARCHITECTURE, whose structure is depicted in the representative model in Figure 5.
Looked at from an abstract point of view, the authorization process encapsulated in ABSTRACT AUTHORIZATION ARCHITECTURE can be summarized as follows:

1. the PEP is informed about the action initiated by a subject with respect to an object
2. the PIP collects a slice of the global system state pertaining to the subject, object, action and environment
3. the PDP evaluates the authorization conditions attached to the right for the given action, using the information collected by the PIP, determining whether the action should be allowed or not
4. the PEP enforces the decision

3.2.2 Solution design level

3.2.2.1 Architecture realization level

There are two patterns that refine the ABSTRACT AUTHORIZATION ARCHITECTURE at the design abstraction level: INTERCEPTOR and DEDICATED SERVER. The patterns are inspired by the different approaches taken in real-life middleware and application-level solutions as described in [82, 83, 84].

3.2.2.1.1 Interceptor (security pattern)

**Context:** When realizing an ABSTRACT AUTHORIZATION ARCHITECTURE in a distributed environment.

**Applicability:** Application level (PROXY, BODYGUARD); System Integration/Middleware level

**Description:** The INTERCEPTOR is a realization of the abstract REFERENCE MONITOR pattern in [85], and can be implemented in two variants: at the application level, as a PROXY [25] or a BODYGUARD [86]; or at the system integration/middleware level as an INVOCATION INTERCEPTOR [87]. In all cases, the task of the Interceptor is to implement the PEP of the ABSTRACT AUTHORIZATION ARCHITECTURE by capturing every applicable operation in the system and triggering authorization-related functionality in response, such as denying a function call/request, sending information to the PDP etc. The references above contain detailed information on each of the patterns, which are also surveyed in [28]. In this architectural realization the PEP is contained entirely within the infrastructure underlying an application, and hence frees the application from authorization-related logic.

**Relationships:** Alternative to DEDICATED SERVER.

3.2.2.1.2 Dedicated Server (security pattern)

**Context:** When realizing an ABSTRACT AUTHORIZATION ARCHITECTURE in a distributed environment.

**Applicability:** Application level

**Problem:** System designs that interleave significant authorization functionality in the business logic are problematic for both maintenance and development. There should be a way to encapsulate the majority of authorization-related functions in a stand-alone component that can be accessed by other components.

**Solution:** Create a dedicated authorization server to handle requests. Authorization servers (placed in each security domain) can be used at the application level by requiring software components to include some (perhaps minimal) authorization-related functionality to check requests/actions (see, for example, the SESAME approach in [88] and [89], or the purely policy-driven approach by [69]). The server can be replicated to increase availability. In this architectural realization the PEP spans both the server and the application.

**Relationships:** Alternative to INTERCEPTOR.

3.2.2.2 Authorization rule handling strategy level

As with the architectural realization level, there are two patterns that refine the patterns in the upper level and determine the authorization strategy for handling and passing rules: PUSH and PULL AUTHORIZATION STRATEGY. The patterns distill the two most common approaches found in real-life systems – see [83] and [84].

3.2.2.2.1 Push authorization strategy (security pattern)

**Motivation:** When a subject performs some action on an object, how can the rules containing the associated rights be encapsulated and passed in the system?

**Solution:** Encapsulate all the authorization-related information in a tamper-proof token, CREDENTIAL [59] or CAPABILITY [57], which are given to the subject and use the PUSH SECURITY INFORMATION modifier from the Security information management solution frame (see Section 4.1.3.1). In this approach the subject acquires the token notionally containing its rights and passes the encapsulated information to the PEP when performing the action. Depending on the architecture realization patterns chosen previously, this process can be described more fully as follows:

- **INTERCEPTOR:** the subject is granted the information either by a separate component, which must be
added to the architecture, or is generated locally. The information in either case can be stored locally or as part of the policy management infrastructure (see Section 3.2.3). Subsequent actions are intercepted by the PEP and the tokens, credentials etc. checked by the PDP to form a decision as per the semantics of ABSTRACT AUTHORIZATION ARCHITECTURE.

- **DEDICATED SERVER**: the subject can contact the authorization server to obtain necessary tokens, credentials, capabilities etc. In subsequent actions the subject passes the security information to the object or associated components, which contact the authorization server to verify them (PDP functionality) and enforce the decision (PEP functionality).

Clearly PUSH AUTHORIZATION STRATEGY is more compatible with DEDICATED SERVER, however, it is possible to implement the pattern using INTERCEPTOR also. In all cases the authorization rules are embodied in the special token and hence authorization is “client-driven”.

PUSH AUTHORIZATION STRATEGY can be implemented in a variety of ways in practice. An example using the CAPABILITY pattern can be seen in [90]; see also [91] for an object-capability architecture based on the MICROKERNEL pattern that combines CAPABILITY with ABSOLUTE OBJECT ID [87]. Another important implementation approach is using a variation of CAPABILITY called ACTIVE CAPABILITY (currently an undocumented pattern), in which capabilities carry sanitized executable logic (equivalent to encapsulated rules), which can be executed by the PDP or by a PROXY interposed between the subject and object pair (see [92] and [93]). Using the CREDENTIAL pattern in the form of certificates (sometimes called signed capabilities [94]) is also a popular approach in Grid systems (cf. [5]).

**Consequences:**
(+): Authorization is simplified since the subject possesses its rights
(+): It is easy to represent subject-based rules (see the CAPABILITY pattern in [57])
(-): Revocation of tokens, credentials etc. and associated rights can become a problem

**Relationships**: Alternative to PULL AUTHORIZATION STRATEGY.

### 3.2.2.2 Pull authorization strategy (security pattern)

**Motivation**: When a subject performs some action on an object, how can the rules containing the associated rights be encapsulated and passed in the system?

**Solution**: Encapsulate all authorization-related information in lists of rules written in some formal language (see POLICY LANGUAGE in Section 3.2.3.2.1 below) or ACCESS CONTROL LISTS (ACLs) – see [57] – attached to each object, handled entirely by a policy management infrastructure, and use the PULL SECURITY INFORMATION modifier from the Security information management solution frame (see Section 4.1.3.2). The typical dynamics of PULL AUTHORIZATION STRATEGY can be described precisely following the semantics of ABSTRACT AUTHORIZATION ARCHITECTURE pattern, with the PDP or PIP contacting the information management infrastructure to obtain the necessary rules. With respect to using the DEDICATED SERVER pattern, the object or associated software components must contact the authorization server to check the validity of the particular action and enforce it on their own. Since the authorization rules reside inside the policy management system, the authorization can be seen as being “infrastructure-driven”.

An example of encapsulating policies using the ACL pattern in distributed environments can be seen in the EACLs (extended ACL) of [95].

**Relationships**: Alternative to PUSH AUTHORIZATION STRATEGY.

### 3.2.3 Practical implementation details level

#### 3.2.3.1 Information handling level

Once the ABSTRACT AUTHORIZATION ARCHITECTURE pattern has been refined, it is possible to choose an appropriate strategy for handling the security (authorization) information. The patterns that can be used for this come from the Policy management pattern family in Section 4.1: ABSTRACT POLICY MANAGER and related refinements. The policies referred to become authorization rules, and the policy manager components are linked with the rest of the authorization infrastructure as realized in previous levels of this (current) pattern family. Using the latter patterns is especially important for a full realization of PULL AUTHORIZATION STRATEGY.

#### 3.2.3.2 Rule expression level

##### 3.2.3.2.1 Policy Language (security pattern)

**Motivation**: Sometimes a simple list or an unforgeable token is not sufficient to encapsulate the authorization rules in a system. There must be some way of expressing rules in a precise, unambiguous manner, in line with the
conceptual authorization model.

**Solution**: Create a language that allows all the features of the conceptual model to be expressed in a formal manner. It should be possible to evaluate the the rules written in the language and map them back to a set of abstract authorization model rule tuples (S, O, A, R, C), determining the set of allowable actions, i.e. restrictions on the actions of subjects in a system. The simplest such language would allow for the following expressions:

\[
\text{<subject S> on <object O> possesses <right R> for <action A> with <conditions C>,}
\]

where the conditions are evaluated by the PDP given the information collected by the PIP, as per the authorization patterns presented above. Constructs indicating "all" for a given category (e.g. all subjects, all objects) can significantly extend the power of the language.

Example languages include Ponder [96] and XACML [97]. XML in general is a good choice upon which to build the policy language due to its universality and easily available software support (parsing tools, schema editors etc.) (again, see [97]). A more detailed discussion of policy languages can be found in [98].

Authorization rules can quickly become complex and difficult to understand for administrators [99], and this should be kept in mind when designing the policy language. The use of modeling languages such as UML can ease the task of translating higher level policies to low level authorization rules in this respect – see, for example, Lodderstedt et al.’s SecureUML [7] and Slimani et al.’s UACML [101] for two different approaches (the first compatible with models instantiating the ROLE modifier), as well as [67], which uses SecureUML with an instance of the EXCEPTIONS modifier.

**3.2.3.2.2 Conflict Resolution Strategy (security pattern)**

**Context**: When implementing an authorization architecture.

**Problem**: Conflicts can arise when rules relating to the same subject and object are inconsistent.

**Solution**: Use heuristics, meta-rules or an algorithm to detect or resolving rule conflicts during run-time as follows [102, 58, 103]:

For static aspects of the conceptual model:

- Rules are prioritized (to avoid modality conflicts, i.e. when one rule for the same subject/object pairs allows and disallows some action)
- Specific rules override generic rules (to avoid redundant rules)
- Rules with negative rights (if expressible by the language and if present in the model – see the Negative Rights modifier) override corresponding rules with positive rights
- Rules are ordered in some fashion, and conflicts resolved on an order basis (first-rule, last-rule etc.)

For dynamic aspects of the conceptual model:

- Rules with fresh (newer) rights override rules with stale (old) rights

Several of the approaches above may need to be combined for a complete solution, e.g. prioritizing rules may result in redundancy, which requires specificity overrides [103]. In all cases a default deny policy (when a rule is not found or conflicts) should be employed as the base policy, since this is the most natural choice for system administrators (cf. [17]).

**Consequences**:

(+): The number of conflicts that occur are reduced *a priori*, without administrator intervention.

(-): The complexity of the decision algorithms may increase.

**3.2.3.3 Technological realization level**

The final stage of instantiating the enforcement architecture family is to select appropriate technological realizations for the patterns chosen previously. This may include mapping POLICY LANGUAGE to XACML to write authorization rules or to J2EE descriptors etc. Administration strategies must also be determined, e.g. manual administration vs. automatic generation from development artefacts.

**3.3 Security process pattern family**

There are only two micro-process patterns for the authorization solution frame: a meta-authorization patterns dealing with aspects concerning the administration of the final (software) authorization subsystem and its variant. Both patterns span the Solution design and Practical implementation details abstraction levels.
3.3.1 Meta-authorization control (stage micro-process pattern)

Context: Large enterprise (distributed) systems, in which there are a number of administrators.

Problem: In complex, multi-domain systems it is often the case that there are numerous authorization rules requiring administration by different people. It is not always prudent, however, to allow any administrator to make any assignment of rights; there may also be a hierarchy of administrators reflecting organizational structures.

Solution: Build up a meta-authorization conceptual model to complement the authorization model [104] and restrict the actions of administrators. The meta-authorization model is equivalent to a (simplified) version of the abstract Authorization pattern together with the User-driven Authority modifier, in which rights are assigned over the action(s) equivalent to “assignment and management of rights”. Meta-authorization necessarily implies ownership control, and hence also the GROUP modifier with an “owner” group for administrators. The enforcement of the meta-authorization model can be realized using the existing/chosen infrastructure from the Enforcement architecture family.

Consequences:
The application of this pattern results in a fine-grained authorization model and corresponding implementation for authorization rule management. The use of a separate authorization model to protect the rules provides the most flexibility in comparison to approaches proposing a fixed management structure (see, for example, Varadharajan's multi-domain approach [58]).

(+ ) Prevention of vulnerabilities associated with administrator spoofing
(+ ) Implementation of the principle of least privilege
(- ) Additional overhead with respect to implementing extra model in software

Example implementation workflow:
1. build up conceptual authorization model
2. design appropriate enforcement architecture
3. assign rights to administrators as per the POLICY LANGUAGE pattern, by distributing credentials/capabilities or setting ACLs

3.3.2 Variant: Meta-authorization Control with Multi-level Authority Checking (stage micro-process pattern)

Description: Policy rules are set by one administrator, and checked by one or more (distinct) administrators with equivalent meta-authorization rights. This ensures that critical policies receive additional review, and can help to stop singular insider attacks, whether with real or spoofed identities, attributes etc.

3.4 Example implementation workflow for the authorization solution frame

Implementing the authorization frame in practice can follow the following steps:

1. instantiate an authorization conceptual model using the ABSTRACT AUTHORIZATION pattern and adding relevant modifiers.
2. instantiate the ABSTRACT AUTHORIZATION ARCHITECTURE to realize the conceptual model. This will need to link with the Security information management solution frame.
3. instantiate a pattern for the authorization design strategy (DEDICATED SERVER, INTERCEPTOR etc.)
4. determine the effects of the authorization model modifiers, and add the relevant components
5. select an appropriate pattern for handling security information
6. select an appropriate authorization strategy: PUSH or PULL AUTHORIZATION STRATEGY, then CAPABILITY, ACL etc. as appropriate
7. determine the POLICY LANGUAGE, if any, that will be used to express the authorization rules
8. instantiate the CONFLICT RESOLUTION pattern: pick conflict resolution strategy/algorithm
9. finalize the storage and handling of rules (see Section 4 below).
10. select technological platform for realizing the various aspects of the enforcement architecture (COTS software etc.)
11. META-AUTHORIZATION CONTROL: instantiate another set of authorization patterns to protect the actual authorization subsystem itself (steps 1 – 8).

4 Security information management solution frame

In this section we present a Policy management pattern family, which forms part of a solution frame for managing the information used in a system's security infrastructure. The solution frame is related to the authorization frame via a “requires” relationship, since the enforcement architecture pattern family requires the management of authorization information – see Section 3.2.3.1 – and realizes a “Manage security data” tactic (this tactic is an addition to the usual list in [33], which, as indicated in [39], should not be taken as definitive and complete).
4.1 Policy management pattern family

The policy management pattern family is concerned with managing security policies in the broad sense of that term, including (as befits the purpose of this paper) authorization rules. It is divided strictly into the four canonical levels of abstraction, of which we present only the first three in some detail.

4.1.1 Conceptual analysis level

4.1.1.1 Abstract policy manager (abstract security pattern)

**Context:** Any distributed system requiring the management of security policies.

**Problem:** Security policies are used widely in distributed systems; they usually take on the form of rules expressed using a particular language. Because of their importance to the overall functioning of a system's security infrastructure, policies should not be defined and used in ad-hoc manner. This is especially important for security solutions such as authorization and filtering, where policy are used to define allowed actions and allowed information respectively.

**Solution:** A policy manager stores all policies and interacts with system components that wish to utilize the policies, as well as with administrators (users) who wish to manage the policies. A conceptual model for this appears in Figure 6.

Notionally, a policy manager consists of two components: a (passive) storage component containing the policies, and an (active) interface component that can be used to interact with the policy store. In practice, the interface component can be divided into a system interface, with an API for software components to use, and a user interface, for administrators to manage (create, delete, modify) policies in the policy store.

4.1.2 Abstract architecture level

4.1.2.1 Centralized policy manager (security pattern)

**Context:** When managing security policies in a distributed system.

**Problem:** There is a need to manage policies coherently so that administrators can access and change policies from different locations. Software components requiring the policies should also be able to retrieve them consistently.

**Solution:** Create a single cluster of software components that together store all policies for a system. REMOTE USER INTERFACE [105] is used to create a single user interface for administrators to access and manage policies from any physical location. The policies themselves are stored in a SHARED REPOSITORY [106]. The system accesses the policies using a PROXY [26], which can also sanitize requests. A representative architectural model (deployment view) of this solution can be seen in Figure 7.

**Consequences:**

(+): Administration is centralized, making it easier for policies to be specified.

(+): There are no data consistency requirements, since everything is in one location and access to policies by the administration controller can be serialized.

(-): The policy storage components and proxies form a single point of failure.
4.1.2.2 Replicated policy manager (security pattern)

Context and Problem: The same as for CENTRALIZED POLICY MANAGER.

Solution: The solution remains the same as in CENTRALIZED POLICY MANAGER, however, we introduce replication to increase the availability of the data and also the overall performance of the system by allowing components to access the policies at “nearby” locations. The replication of components also implies that the policies across the replicas must be kept consistent. To do this we utilize a PROPAGATOR [107] pattern, which requires the definition of a dependency network in order to propagate changes. In this case the dependency network consists of the different policy stores and their corresponding manager components. More complicated protocols to coordinate the replicas containing the policies may be implemented also, but are not part of this pattern – readers are referred to [108] for more details.

In a typical use case, an administrator can enter/modify/delete policies via the User interface, which effect the centralized repository. The Policy manager proxy then ensures that the policies are propagated to second-level proxies on different nodes that hold either a copy of all policies (complete replica) or only some (partial replica). The latter approach is taken, for example, in the cluster security infrastructure of [3], where policies are pushed by a central manager to local manager components.

When an active software component wishes to retrieve a policy, it contacts the local Policy manager proxy (if its address is known, or the central proxy in the first instance to determine the address), which can then interact with the component to provide the policies as appropriate.

The structure of this solution is shown below in Figure 8.

Figure 7: Partial deployment view of the centralized policy manager

Figure 8: Partial deployment view of the replicated (centralized) policy manager
Consequences:
(+ ) Administration is centralized, making it easier for policies to be specified.
(+ ) The availability of policy storage increases in proportion to the number of replicas.
(+ ) Performance of the system increases, since components requiring a policies need only interact with a “nearby” replica, reducing overhead due to network latency.
(- ) There are strong data consistency requirements between replicas of the storage components and/or interfacing management components that must be satisfied.

4.1.2.3 De-centralized policy manager (security pattern)

Context: When wishing to manage policies across multiple administrative or security domains, or individually for different components.

Problem: Sometimes each site in a domain or even each component must have separate, attached policy definitions.

Solution: Each site (node) or domain in a system determines its own policies locally, which are also stored locally (see Figure 9). A PROPAGATOR is used in which the dependency network is made up of all the policy proxies. Not all policy updates require changes to propagate through.

Consequences:
(+ ) There is no single point of failure, as each site manages its own policies.
(+ ) Each site can have its own administrator.
(+ ) Performance of the system increases, since components requiring a policies need only interact with local policy management components.
(- ) Policies defined at multiple sites can cause policy heterogeneity.
(- ) Policy management becomes more difficult as meta-policies may have to be defined determining what changes in which policies should be propagated to other sites.
(- ) There are strong data consistency requirements between the different policy management sites.

4.1.3 Practical implementation details level

4.1.3.1 Push security information (pattern modifier)

Context: When wishing to utilize security information.

Problem: How should a client component interact with any infrastructure holding the security information in order to use that information for some specific purpose (e.g. presenting credentials or policies).
**Solution:** The client component (Component A) first contacts the relevant security information infrastructure components and retrieves the information locally. The infrastructure components can perform any checks to ensure this is a valid, and that the client component can be trusted in some sense. The information can also be validated and/or vouched for as being authentic. Component A can then contact a target component (Component B) and pass the information for the required purpose. Component B can optionally check the validity of the information as needed.

For policy management, this implies that client components contact a policy manager proxy, which can retrieve any policies from the appropriate policy store(s).

**Consequences:**

(+): A client has already fulfilled everything necessary to obtain the appropriate security information, lessening the burden on the server of verifying it.

4.1.3.2 **Pull security information (pattern modifier)**

**Context and Problem:** As for PUSH SECURITY INFORMATION.

**Solution:** The client component (Component A) first initiates an operation on a target component that involves the retrieval of security information, leaving the onus on that component to interact with the relevant security information infrastructure components and “pull” the information. As for the PUSH SECURITY INFORMATION PATTERN, appropriate checks and validation can be performed.

**Consequences:**

(+): A client does not need to verify any security information, since the onus is on the target component to verify it.

(+): The target component should not need to check the validity of the information itself, since the information management infrastructure (e.g. policy management proxy) is able to do this.

4.1.3.3 **Technology realization level**

The patterns described above can be realized using a range of technologies, such as relational databases (Oracle DB, MySQL and others) or LDAP for the shared repository, graphical front-ends for the administrative interface and others. In a full documentation of the policy management pattern family, this section would contain descriptions of various technologies as done in [43].

5 **Examples**

In this section we illustrate an application of our authorization solution frame in analyzing two existing authorization models and their respective enforcement architectures: the UCON-based authorization framework of Zhang and colleagues appearing in [5], which is designed specifically for distributed, collaborative applications; and Li and Karp’s ZBAC [29, 30, 31], an authorization model tailored to SOA-based systems. In both cases our analysis proceeds by identifying which patterns were instantiated from our solution frame and how they were customized, i.e. what design decisions were taken in the construction of the latter authorization schemes. An analogous process would be followed when constructing a new authorization solution, with the exception that one would not analyze existing design decisions but make and realize those decisions by selecting, customizing and instantiating the patterns.
5.1 UCON for collaborative applications

5.1.1 Conceptual model

Zhang et al.'s authorization infrastructure is based on the UCON_{ABC} model of [14], which can be seen as a type of ABAC model [11] (see also [109] for a comprehensive overview), where the conditions of the ABSTRACT AUTHORIZATION pattern are defined in terms of the ATTRIBUTE modifier. Attributes in Zhang et al.'s UCON model can include ROLE and GROUP memberships (for subjects) as well as usage quota and inclusive/exclusive status (for objects). Environmental attributes include a straightforward application of the LOCATION modifier.

In UCON, actions (usages) by subjects are not only authorized statically, i.e. once prior to the action taking place, but also dynamically (ADAPTATION) within a SECURITY SESSION for usages, whereby the state of the system may cause a change in the authorization status (STATE) – the so-called decision continuity property of UCON. This also implies that the attributes defined by the ATTRIBUTE modifier for subjects, objects and the environment can change (the so-called attribute mutability property) as per the ADAPTATION modifier, causing run-time changes in usage rights. Another important aspect of the model is the use of obligations, which can be seen as customized instances of the CONTEXT modifier, with two pre-defined contexts: pre-authorization and post-authorization, which require the occurrence of a set of specified activities. CONTEXT is further applied to support SECURITY SESSION, in which changes in contextual information can further lead to changes in authorization status for a given subject. At a high level of abstraction, one can thus state that:

\[
\text{UCON} = \text{ABSTRACT AUTHORIZATION} + \text{ATTRIBUTES} + \text{SECURITY SESSION} + \text{ADAPTATION} + \text{STATE (continuous usage)} + \text{CONTEXT (pre-authorization, post-authorization)} + \text{GROUP} + \text{ROLE} + \text{LOCATION}
\]

with the last three modifiers being treated as ATTRIBUTES.

5.1.2 Enforcement architecture

Conceptual analysis and Abstract architecture: The conceptual authorization model is realized in software abstractly by instantiating ABSTRACT AUTHORIZATION ARCHITECTURE with the usual PEP and PDP components and adding, as required by the ADAPTATION and STATE modifiers, additional monitoring components to ensure that authorized object usages are monitored and any state changes implying attribute changes are appropriately propagated. The PIP component functionality spans the PDP and the monitoring components, with the subject (user), object (resource, service) and environment attributes being part of the information collected.

Solution design (architecture realization level): The ABSTRACT AUTHORIZATION ARCHITECTURE is realized in turn by using an INTERCEPTOR with components at the resource provider end within a given Virtual Organization (VO), which encompasses both the PEP and PDP as well as a usage monitoring (UM) component; and a proxy at the client end to intercept requests.

Authorization data is passed using both the PUSH AUTHORIZATION STRATEGY and PULL AUTHORIZATION STRATEGY patterns to make a hybrid solution where static attributes (which do not change over time, but only administratively) are pushed by subjects requesting a resource usage to the PDP component (on the resource provider end), while mutable subject and object attributes are pulled by the PDP from the mutable attribute (security information) management infrastructure and the UM, respectively. Attribute updates are the responsibility of the PDP. In line with instantiating ADAPTATION and STATE, which require event trigger points to affect changes, “any update of subject or object attributes and any change of system (environment) conditions triggers the re-evaluation of the policy by the PDP according to the ongoing usage session” (see the SECURITY SESSION modifier) “and may result in revocation of the ongoing usage or update of attributes if necessary.” [5]. This is affected by the UM, which, as mentioned above, is part of the INTERCEPTOR realization. The need to update LOCATION information of subjects gives rise to the inclusion of an additional “sensor” component on the user end that forwards the information to the mutable attribute management infrastructure.

Practical implementation details: The mutable attributes, which are used in the PUSH AUTHORIZATION STRATEGY instance referred to above, are managed by CENTRALIZED POLICY MANAGER instances within a VO, with (possibly) multiple instances for different types of subjects or attributes. The advantages of a centralized approach in this context is that multiple, concurrent usage sessions (SECURITY SESSION instances) can be handled more easily and efficiently. In the prototypical implementation of the UCON authorization infrastructure (see the original publication [5] for details), LDAP (OpenLDAP with OpenSSL) was chosen as the CENTRALIZED POLICY MANAGER's technology realization. Static subject attributes are stored at the user end.

For expressing UCON authorization rules, Zhang and colleagues chose a POLICY LANGUAGE that uses XACML as the technology realization. A default deny policy approach is used, where a particular action is not allowed if a matching rule cannot be found in the CENTRALIZED POLICY MANAGER's repository (cf. CONFLICT RESOLUTION STRATEGY).
5.2 ZBAC for SOA-based systems

5.2.1 Conceptual model

ZBAC is a model that strives to obviate the need for federated, cross-domain authentication when making authorization decisions in the context of SOA systems. While this may at first suggest a complex model like UCON, when analyzed using the patterns in our solution frame ZBAC appears surprisingly simple, consisting of the base ABSTRACT AUTHORIZATION pattern together with a customized instance of the DELEGATION modifier. Rights specified (abstractly) on services (objects) in a system can be delegated either in full or as a subset to subjects, which can then perform any necessary action. Transitivity is one significant feature of the model as a result of applying DELEGATION to pass rights in the “arguments” of service calls, which are themselves actions on other services (objects), creating delegation chains. At a high level of abstraction, one can thus state that:

\[
\text{ZBAC} = \text{ABSTRACT AUTHORIZATION} + \text{DELEGATION}
\]

5.2.2 Enforcement architecture

Conceptual analysis and Abstract architecture: To enforce the latter model, the ZBAC approach applies ABSTRACT AUTHORIZATION ARCHITECTURE where the PDP becomes a Policy Engine component within each organization domain and the PEP and PIP are left to be specified at lower levels of abstraction.

Solution design: The ABSTRACT AUTHORIZATION ARCHITECTURE pattern is realized in practice using a DEDICATED SERVER in each domain, in the form of a Domain Access Rights Controller (DARC) component. The PEP notionally spans both the DEDICATED SERVER and (potentially) part of the service itself (i.e. partial application-driven enforcement). All rules for authorization are encapsulated in certificates, which can be delegated (as per the DELEGATION modifier of the model). The ZBAC approach thus uses PUSH AUTHORIZATION STRATEGY with the certificates playing the role of CAPABILITY instances.

When a service is deployed in an organization, it is registered with the DARC component, which implies that the DARC will possess a capability with all possible rights to that service. Users or other services (subjects) that wish to use the initial deployed service (object) must contact the DARC to obtain the necessary capability, which is realized in practice by the DARC delegating a capability with a subset of its own rights. The latter can then be presented by the subject on requests (see the PUSH SECURITY INFORMATION modifier). DELEGATION also requires revocation, and to implement this the ZBAC scheme exposes a “revoke” operation for each service, which can be called by some subject (given the right to do so), which will cause that capability to be revoked in a per-service revocation list. Capabilities are also time-constrained. Delegation of capabilities also applies across organizations—a DARC in one organization can delegate its capability for a (local) service to the DARC in another organization (see [30] for more details), which can be delegated in turn to the services there, allowing (authorized) interaction. In this approach identification of services across organizational boundaries is not needed, since everything is contained in the delegated capability, achieving the original aim of the model.

Practical implementation details: The scheme above requires that each organization domain has its own policy manager (for capabilities, in this case), leading to a DE-CENTRALIZED POLICY MANAGER solution. In Li and Karp's implementation of the ZBAC scheme [30], the capabilities (certificates) possessed by subjects are represented using SAML (Security Assertion Modeling Language) – a Web-services standard – as their technology realization, together with a public-key infrastructure for digital signatures. The whole authorization infrastructure is implemented in the context of the .NET Web-services framework.

6 Related work

As far as we are aware, this paper represents a first attempt to provide a comprehensive pattern-based, software engineering approach to authorization for distributed, collaborative applications. It is also a first step towards the unification of security tactics and patterns via a unique solution structure.

Besides Microsoft's singular pattern frame mentioned in Section 2, our solution structure was inspired by the solutions used in several security methodologies for distributed systems (see [37] for a comprehensive overview), namely:

- SERENITY's hierarchy of Security and Dependability (S&D) solutions [110, 111], which is divided into [37]: S&D Classes (at the highest level), guaranteeing particular semantics and abstracting the security properties provided by a collection of S&D patterns, which in turn are concrete, individual specifications of security solutions with precise formal semantics and multiple possible implementations. These implementations are represented as S&D Implementations, realized as Executable Components in a SERENITY run-time environment. Unlike our (traditional) security patterns, which are architecture-oriented (see [27]), SERENITY's S&D patterns are fundamentally implementation-oriented.
• SECTET/SECTISSIMO's security pattern models and policy models [112, 113], which are refined through two levels of abstraction using model-to-model transformations to determine platform independent and platform specific details. Like SERENITY's S&D solutions, the purpose of refinement in the SECTET/SECTISSIMO approach is to map and transform the platform-specific patterns and policy models to security services and configurations (respectively) in a runtime platform.

Besides encompassing several levels of granularity, encapsulating process patterns and numerous other distinctions, our security solution frames aim to guide developers from concepts towards producing more complete, “pluggable” designs, not generated or component-based implementations. Furthermore, the solution frames presented in this paper are methodology-independent. Their constituent parts – patterns and families – can also be used independently of each other and of the frame itself. The aforementioned factors similarly differentiate our work from related approaches to organizing patterns in hierarchies, such as the use of pattern maps in the context of a Web-service specific three-layered abstraction hierarchy [41], the conceptual mapping between patterns from architectural, to design, to descriptions of technologies/WS-standards [114], or the use of child security patterns for solution refinement [115] – all of which have nevertheless influenced the construction of our security solution frames in one way or another.

With respect to generalized models for authorization, an early approach using security patterns to “grow” models based on the Access Matrix can be found in the work of Fernandez, Priebe and colleagues [63, 64]. In that approach models can be defined by analogy and as variants of the base Access Matrix pattern. Our approach contained in the Conceptual model pattern family is much more flexible and generic in contrast, allowing custom models to be built that result in a large (and potentially arbitrary, as more modifiers are added) variety of patterns that would be cumbersome to document individually. Our work further considers model enforcement, within an altogether different setting of a security solution frame.

In the attempt to provide a wide variety patterns for creating custom models our authorization solution frame bears relation to approaches such as [21] and [116], which aim to create generic meta-models or policy machines for expressing new and existing authorization models, either in the abstract or at an architectural level. The distinction (besides our use of patterns) is that the approach embodied in our solution frame does not attempt to provide such encompassing generality, nor does it aspire towards any type of formal rigour. However, the fact that building up a conceptual model using patterns lacks proof of conformance of the resulting models to certain theoretical and/or formal requirements, such as decidability of authorizations, in no way reduces the usefulness of the approach. Indeed, XACML similarly lacks formal foundations (cf. [21, 117]), but this has not precluded its widespread use and adoption for real-life scenarios. Similarly, the models resulting from the instantiation of the patterns below result – from a software engineering point of view – in fully implementable models that can be constructed to suit the particular needs of the target system or application, which is certainly an advantage over purely theoretical approaches (again, compare [21, 117]).

A number of other approaches have sought to provide more customized authorization models for distributed and/or collaborative settings, such as dRBAC [118], Zhou's RBAC- and TMAC- (see [119]) inspired TT-RBAC for collaborative environments [19], Pretschner et al.’s distributed usage control realization [121], Lu et al.'s process-centered TABAC [4], the fine-grained models of Shen and Dewan [20], Sikkel [16] and many others. There are also numerous highly specific cryptography-based authorization schemes (see [6]), which, while predominantly theoretical, may be useful for certain types of collaborative applications. In all cases these models are not customizable, and may not fit the specifics of the application.

For the exception of the earlier work of Fernandez, Priebe and colleagues on pattern-based authorization models, few of the references considered above take a concrete software engineering approach to authorization. The (relative) paucity of work in this area highlights the need for further research in the direction of custom authorization models from a software engineering viewpoint, as well as novel structures for security solutions that can support this.

Regarding our overall viewpoint and approach to authorization, the closest work is embodied in the OM-AM and later PEI (Policy/Enforcement/Implementation) framework proposed by Sandhu and colleagues [84, 122, 5], where authorization (and more generally any security property) is seen to pass from high-level security objectives to conceptual models, to be realized by concrete architectures, refined further by implementation-specific details, and finally implemented in software and hardware components. Our authorization solution frame realizes this abstract, high-level approach in a concrete, practical fashion, as an independent security solution.

7 Conclusion and Future work

In this paper we proposed a flexible, pattern-based approach to constructing fine-grained authorization models for use in distributed systems, with particular value for collaborative applications. The approach was embodied in a novel security solution structure – a security solution frame – discussed in Section 2, which encapsulates patterns in fine-grained vertical and horizontal divisions. Using the authorization frame (Section 3), authorization models can be constructed by modifying a base abstract security pattern, and enforced by realizing an abstract architecture by degrees, according to several layers of abstraction. The refinement provided by the abstraction layers provides guidance to developers and, as a benefit of the solution frame structure, a smoother progression from high-level
design patterns (concepts) to concrete designs (templates). Details that are often left out of consideration when discussing authorization, such as meta-authorization and the management of policies were also addressed, the latter as a separate solution frame (Section 4) that served both to reinforce the authorization frame and to further highlight the benefits of the solution frame concept itself.

The value of our approach was subsequently illustrated by examining the design decisions contained in two existing authorization models used in collaborative settings (Section 5): UCON and ZBAC. We showed that it is possible to capture and describe both models using our pattern modifiers, and that it is also possible to identify patterns from our enforcement architecture pattern family that were used in the model's software realization. We argued that our authorization approach, based on a software engineering perspective, is valuable above all in practical development situations where custom authorization models are required. Such situations are indeed very common, with the development of collaborative applications being a prime example.

When considering related work on the topic (Section 6), we emphasized that a software engineering perspective on authorization as espoused in this paper is largely missing from the literature, and that more work is required on this front. While we by no means identify “software engineering perspective” with the creation of “sloppy” or ad-hoc models for authorization (cf. [11]) – not having presented any rigorous formal analysis ourselves – we recognize the inevitable fact that the formal foundations of authorization models can, for better or worse, be easily ignored during system development, and that a solution based on the construction of “practical” models that closely fit the semantics of the application can be of real value. Aligning this work with a formal framework could be challenging due above all to the generality involved, but it is certainly a viable direction for future research. Particularly relevant in this context is a consideration of the relationships between the conceptual model and enforcement architecture pattern families, including the fidelity of the architecture in implementing the pattern-based model, which, as discussed by Kane and Browne [13], is especially difficult in distributed environments where local views of the global system state may be inconsistent or incorrect.

In the introduction, we noted that authorization is one of the most important aspects of security, especially in collaborative systems. However, authorization itself is also supported by other security measures, which we have not treated in this paper, including authentication, for ascertaining the identity of authorized principals or users, and secure communications, for ensuring that all security information (policies etc.) are passed in encrypted form throughout the system. When collaboration occurs over an open and adversarial environment this can be especially important. Our more imminent future work, therefore, will concentrate on defining (or re-defining, where appropriate) tactics and developing corresponding security solution frames for authentication and secure communications, as well as a range of other measures useful both for collaborative systems and distributed systems in general.

Another important point is that, by nature, security solution frames and their encapsulated pattern families include only simple patterns, not compound patterns (for which see [123]). A number of compound patterns do, however exist, including Secure Broker [120], Secure Pipes and Filters [124] and others (see [125] and [28] for more detail). One future direction in this regard would be to create compound patterns by securing existing patterns for collaboration, such as the Floor Control pattern [126, 127] to speed up development. The relation of such compound patterns and existing security solution frames is another topic that will be explored in future research.

Finally, the novel security solution structures outlined in this paper were demonstrated in a somewhat “synthetic” setting (i.e. using an already completed model and architecture for authorization), and in isolation from a systematic approach for securing software. Incorporating solution frames and pattern families into a comprehensive security methodology for distributed systems will allow a full scale demonstration of the benefits of the structures during development, especially for authorization as presented in this paper. Case studies with collaborative systems are also another important direction in this respect.

As pointed out in [28], a methodology capable of being tailored in some sense for different types of specific distributed systems (collaborative, Web-service based etc.) is an important future research direction for pattern-based methodologies. The flexibility and extensibility of solution frames and pattern families, together with the ability to modify patterns individually, may make such solution structures more amenable for use in the latter types of methodologies, and is something that will be explored in future research.

References


Epilogue

In this chapter, we introduced another one of our generic conceptual security artefacts, this time for a defensive purpose – security solution frames – and presented frames for addressing authorization and policy management. Besides the self-contained contributions, the artefacts can be used (or perhaps more precisely, “deployed” via S-SMEP from Chapter 3) across different security methodologies independently, or in combination with the other generic artefacts (the frames in this chapter realize a subset of the mitigating policies of a number of threat patterns from Chapter 5).
Chapter 7
(Generic conceptual security artefacts III part II)

Security solution frames for secure network communication, authentication and cryptographic key management

Prologue

In this chapter, we continue with our presentation of the security solution frame artefact, this time for three core distributed systems security features: secure network communications, authentication and key management. As explained in the Prologue to the previous chapter, the patterns encapsulated as part of the frames can be seen as filling the gaps identified in the security patterns literature in Chapter 2. However, the frames are artefacts in their own right, at a higher level of abstraction.
# Statement of Authorship

<table>
<thead>
<tr>
<th>Title of Paper</th>
<th>Cryptography-based Security Patterns and Security Solution Frames for Networked and Distributed Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication Status</td>
<td>○ Published, ○ Accepted for Publication, ○ Submitted for Publication, ○ Publication style</td>
</tr>
</tbody>
</table>

## Author Contributions

By signing the Statement of Authorship, each author certifies that their stated contribution to the publication is accurate and that permission is granted for the publication to be included in the candidate's thesis.

<table>
<thead>
<tr>
<th>Name of Principal Author (Candidate)</th>
<th>Anton V. Uzunov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Conceptualization of work, its realization (research) and documentation (wrote manuscript). Acting as corresponding author.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th>Eduardo B. Fernandez</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Provided ideas, references and comments on different versions.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
</table>
Cryptography-based Security Patterns and Security Solution Frames for Networked and Distributed Systems

Anton V. Uzunova,*, Eduardo B. Fernandezb

a, School of Computer Science, The University of Adelaide, Adelaide, South Australia, Australia, 5005
b, Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, 777 Glades Rd., Boca Raton, FL 33431

* Corresponding author

E-mail addresses: anton.uzunov@adelaide.edu.au (Anton V. Uzunov), ed@cse.fau.edu (Eduardo B. Fernandez)

Abstract: Over the last decade researchers and practitioners have increasingly come to acknowledge that introducing security features into large, complex and distributed systems requires the combined application of software and security engineering principles. In this respect security patterns, which build on the success of design patterns and software patterns more generally; and security solution frames, which encapsulate and organize security patterns into different levels of abstraction and different facets of a root security policy – are tools of great value. At present, the literature contains several frames and many patterns for a variety of concerns, but none of these address the core security features of networked and distributed systems that are based on cryptography in a comprehensive fashion. This leaves a conspicuous gap in the secure software engineering “arsenal” that must be filled by developers on-the-fly. In this paper we seek to fill the latter gap by presenting security solution frames and relevant encapsulated patterns for three core cryptography-based security features: two-party secure network communications; authentication; and cryptographic key management. Our solution frames promote a software architectural, implementation-independent perspective, in which the aim is to consider the design decisions employing cryptography first, and cryptographic implementation details only afterwards. We demonstrate how the proposed solution frames and their patterns can be used in practice by analyzing the security architectures of Bell Labs’ Plan 9 distributed operating system and IBM’s Lotus Domino/Notes distributed collaboration suite.

Keywords: secure software engineering; distributed systems security; security patterns; security solution frames; cryptography; authentication; secure communications; key management

1 Introduction

Over the last decade the already large variety of distributed systems has grown significantly, spurred by applications such as peer-to-peer file sharing and scientific computing, as well as by enterprises moving towards software technologies in which distribution plays a central role. This trend has been accompanied by a general increase of security vulnerabilities, both potential and real, as corresponding systems increase in size, complexity and heterogeneity, requiring a broad range of strategies for the incorporation and enforcement of security attributes.

Researchers generally agree that for any security strategy to be successful, software must be designed for security from the earliest stages [1, 2, 3, 4], with a consistent maintenance of the “security-push” throughout the whole development life-cycle afterwards. The tools of software engineering thus play an integral part in supporting security development activities [5, 6, 7], becoming powerful aids in reducing the number and severity of potential vulnerabilities.

In this respect, security patterns [8, 9, 10], which build on the success of design patterns and software patterns [11]; and security solution frames [12], which organize groups of security patterns into different levels of abstraction (vertically) and different aspects or facets of a root policy (horizontally) – can be seen as software engineering tools of great value in the quest for developing secure networked and distributed systems.

As software patterns, security patterns capture the experience and expertise of many professionals in a form accessible to the security non-expert [13, 9]. Security techniques and countermeasures captured as patterns can be customized and applied in a number of contexts while guiding developers from a problem in a given context to a proven solution with predictable consequences. Security solution frames take these beneficial features a step further by allowing developers to employ patterns in a holistic context for realizing a given security policy – from concept to design – by refinement, throughout the software development life-cycle and (simultaneously) across different abstraction levels.

Given the benefits of security patterns and security solution frames, it is a surprising fact that the current literature on the subject has surprisingly few specific patterns for a number of the core security features of networked and distributed systems [10]. This is especially true for features based on cryptography, for which there are not only few security patterns, but no dedicated security solution frames [12]. That this is a problem within the context of pattern-based secure software engineering approaches is easy to see when one considers how – to a far greater degree than isolated systems – networked and distributed systems rely on cryptography as the basis for implementing a number of essential features: the protection of data in transit (secure communications), the
establishment and verification of user identities, and the secure management of sensitive and/or security-related information, among others. Given the ubiquity of distributed systems, and the increasing industry/practitioner awareness of patterns and pattern-based security methodologies (cf. [8, 14, 15, 16]), this leaves a significant gap that must be filled by developers, architects and other team members – who are often inexperienced with respect to security [17] – “on-the-fly”.

In this paper we seek to fill the latter gap by presenting security solution frames and relevant encapsulated patterns for three core cryptography-based security features: secure network communications between two-parties (Secure communications frame); user/process authentication (Identity management frame); and cryptographic key management (Security information management frame).

Solution frames and their encapsulated patterns are not only valuable in the design of new secure networked and distributed systems, but also in analyzing the security architectures of existing systems [13, 12]. We demonstrate how the solution frames in this paper can be used in practice in the latter capacity by analyzing the security architectures of the Plan 9 distributed operating system (in detail) and IBM's Lotus Domino/Notes distributed collaboration suite (in brief). Analyzing the IBM Lotus and Plan 9 security architectures briefly and in detail, respectively, shows how the solution frames can be used to capture relevant (security) design decisions at different levels of granularity and abstraction – providing either an overview or more in-depth description – depending on available documentation and/or purpose of analysis. The examples also indicate the analogous process that would be followed when applying the solution frames and/or their patterns to design new security architectures from scratch.

In comparison to existing literature dealing with cryptography, which promotes a low-level, algorithmic perspective stressing implementation details, our solution frames promote a software architectural, implementation-independent perspective, in which the aim is to consider the design of (new or existing) features employing cryptography – i.e. crypto-architectures – first, and to consider implementation afterwards. Since on real-life projects developers often employ trusted and well-verified cryptographic libraries and/or frameworks to reduce effort requirements and provide stronger implementation stability, the proposed solution frames can also make the transition from (cryptography-related) high-level security requirements to implementation details more seamless and traceable.

The rest of this paper is structured as follows. In Section 2 we provide some (very) basic background on cryptography and also outline security solution frames. In Sections 3, 4 and 5 we present, respectively, the solution frames addressing two-party secure communications; cryptographic key management; and user/process authentication. In Section 6 we show how the solution frames are used in practice to analyze the IBM Lotus Domino/Notes and Bell Labs Plan 9 security architectures. In Section 7 we discuss related work. Finally, in Section 8 we conclude and briefly consider future directions.

2 Background

2.1 Cryptographic primitives and protocols

In this section we briefly provide some (relatively superficial) background in cryptography for the rest of the paper, based predominantly on [18] and [19]. For more information and details, the interested reader can refer to [20, 18, 19, 21, 22], as well as numerous other introductory and advanced textbooks on cryptography.

2.1.1 Cryptographic primitives

We define a cryptographic primitive as a building block – either an algorithm, a collection of mathematical functions or a type of parameter – which can be used to construct software with some form of cryptographic functionality. Algorithmic primitives can be classified according to the use of cryptographic keys as unkeyed, symmetric or shared key; or public key (see [19]).

Within this uniform terminology, we explicitly distinguish two hierarchical levels of cryptographic primitives based on abstraction, where primitives in one level use those in the level below: (1) algorithms (e.g. symmetric block ciphers, unkeyed hash functions); and (2) mathematical primitives and parameters (e.g. PRNGs, prime number generators, time-variant parameters). The algorithms level is further divided into three sub-levels – (1) signature schemes; (2) hash functions; and (3) encryption ciphers – again with each level containing primitives that use other primitives from lower (sub-)levels. An example of one primitive using another at the algorithm level is hash functions that are implemented (internally) using symmetric ciphers, in which case a key is generated and passed to the symmetric algorithm. Similarly, keyed functions can use both symmetric ciphers and unkeyed hashes in their construction (e.g. HMAC).

Below we briefly describe the levels of cryptographic primitives in reverse order of abstraction, as background for the solution frames introduced later on.

2.1.1.1 Mathematical functions and parameters

Mathematical functions include random number generators (RNGs) and pseudo RNGs (PRNGs), permutation functions and others. PRNGs generate predictable sequences of random numbers based on a seed, whereas RNGs
generate “true” random numbers by using an entropy source (usually hardware-based). Other functions such as simple permutations, which “mix” or swap symbols in an alphabet (in this context, sequences of bits) are also included in this level of primitives.

Parameters are specific values or sequences of bits used in cryptographic operations or to support cryptographic functionality. The most prominent examples are time-variant parameters (TVPs) – nonces or random values and sequence numbers – which promote freshness.

2.1.1.2 Cryptographic algorithms

Encryption ciphers

Encryption ciphers are algorithms that, given a cryptographic key (a defined series of bits), reversibly transform one piece of data (plaintext) to another, “unreadable” piece of data (ciphertext) in the same alphabet. Ciphers can be symmetric or asymmetric: symmetric ciphers (also sometimes called shared key ciphers) use the same (or a trivially derivable) key for both encryption and decryption, and can be thought of as a key-parameterized family of permutation functions (usually composed of a series of alphabetic substitutions and transpositions for their implementation), while asymmetric ciphers use a key pair – a “public” key known to all parties, and a “private” key known to one party only – and generally rely on “intractable” mathematical problems for their implementation, such as integer factorization and discrete logarithms.

Symmetric algorithms can be further broken down according to whether they operate on blocks of data or data streams. The latter case implies the generation of a key stream, while the former implies padding and, according to the different modes of the cipher, the potential use of an initialization vector (IV) (another defined series of bits) to avoid predictable (bit-)patterns in the encrypted output.

Cryptographic hash functions

Loosely speaking, cryptographic hash functions reduce input data of some (arbitrary) length to an output of fixed length in a non-invertible (one-way), collision-resistant fashion. The latter properties can be strictly required or relaxed depending on the application required. Hash functions can be categorized as keyed (requiring a cryptographic key for operation) or unkeyed, and within these categories can be further divided according to their purpose as Modification Detection Codes (MDCs) or Message Authentication Codes (MACs), respectively. For the sake of simplification, and within the scope of this paper, unkeyed hash functions will be considered identical with MDCs, which in turn will be considered to be both non-invertible and collision resistant; while keyed hash functions will be considered identical with MACs.

As indicated previously, MACs can make use of MDCs for their implementation (e.g. HMAC uses an MDC and adds to it the cryptographic key via several bit-wise operations). Both MACs and MDCs can be built from symmetric cipher algorithms, e.g. DES in CBC mode with a pre-generated key.

Digital signature schemes

A digital signature scheme is the asymmetric-key analogue of a MAC, with the additional feature that digital signatures provide non-repudiation. In a digital signature scheme a public-private key pair is generated, the private key (still a series of bits, but can be interpreted as a series of compound values) is used to sign the message via a signature function, and the public key is used to verify the signature using a verification function.

2.1.2 Security protocols

According to Chen and colleagues [23], “a security [or cryptographic] protocol is a sequence of operations that perform a security-related function by using cryptographic methods” or, in our terminology, algorithms, parameters and mathematical primitives from the two levels of primitives described above. A protocol is more than simply a collection of steps between two or more entities; in its full specification, a protocol should determine how the lower-level primitives should be used, and should include details on supporting data structures and representations [23] (see also [21]). Besides thinking of a protocol as a sequence, one can also consider it as a template, parameterized by a fixed number of agents (or parties) acting as participants and variables, in which case the protocol is instantiated by assigning concrete values to the parameters (cf. [24]). Such a viewpoint cannot encompass all protocols, but it is particularly useful for symbolic reduction techniques to protocol evaluation (see [24]), and to conceptualize protocols as sequences of groups of algebraic terms. One should naturally distinguish the protocol in the abstract from the protocol realization in a given environment (e.g. particular network model, such as the OSI layered model), and then in turn to its implementation in software.

2.2 Security solution frames

From a software engineering perspective, developing a secure system of any type requires a set of specific security solutions to be applied during the development life-cycle, especially during architectural design. A number of such
solutions have been presented in the literature, including security tactics [25], security patterns [9], generic security architectures and components [26], aspects [27] and others, each with varied characteristics (see [16]). Of these the first two – security tactics and patterns – are by far the most mature and most widely used in the industry.

Security solution frames are solution structures that form a bridge between tactics (interpreted as direct encapsulations of generic security policies) and patterns [12]. The security patterns encompassed by the solution frames can be replaced by, or related to, other conceptual security artefacts – e.g. security aspects [27] – while retaining the overall intrinsic structure of solutions (as specific security policy encapsulations), making solution frames methodology agnostic. Figure 1 presents a conceptual model relating solution frames, tactics and patterns (with the individual solution types appearing in different colors or, if viewed without color, different shades of grey, purely to differentiate them from other elements and from each other).

Drawing inspiration from ideas present in a number of existing solution structures (see [12]), security solution frames partition the solution space horizontally into separate (but related) concerns via pattern families; and vertically, into levels of abstraction.

Pattern families are collections of related (simple) patterns, with their own independent division of abstraction, for solving a root problem that is an aspect or facet of the solution for the whole frame itself. Patterns between the levels of abstraction, whether with respect to the frame or the pattern families, refine the patterns from the previous level; at the same level, patterns can be related via several solution relations (see below). Families treat one part or aspect needed to fully realize a tactic. It is not necessary to utilize patterns from all families for a given tactic, since the realization of a tactic in a given context may only require a subset. Using a pattern from a given level of abstraction realizes the initial generic security policy (and hence tactic) in part, at that abstraction level. The whole sequence of patterns from appropriate pattern families realizes the tactic in full.

A given security pattern may be part of more than one family, depending both on the nature of the pattern itself,
and can contain patterns that appear in other families. When patterns from one solution frame (source) are used in another frame (target), their effects on the existing patterns (in the same frame) can be documented as part of the relevant (target) pattern families, making frames self-contained.

Four *canonical abstraction levels* for the frame determine the overall vertical divisions into four areas of responsibility within the solution or, looked at from a process viewpoint, stages through which the solution passes, which simultaneously correspond to the stages of the software development life-cycle (SDLC):

- **Conceptual analysis**: contains one or more abstract security patterns [29] capturing the abstract concepts required in realizing the solution.
- **Abstract architecture**: contains patterns that determine an high-level, abstract architecture for the solution.
- **Solution design**: contains patterns refining the abstract architecture and providing representative architectural models that suggest more concrete software components and connectors.
- **Practical implementation details**: contains patterns that address the details and specialized aspects of the solution design, refining the solution so that it can be instantiated (with necessary customization, as befits a pattern-based solution) into the system's architecture.

The names of the abstraction levels are purposefully not aligned with any standard development stage names to emphasize the generality of the levels in mapping to various paradigms, e.g., object-oriented development, Model-Driven Engineering and others.

Pattern families can be divided vertically into any number of independent levels of abstraction, where they span multiple canonical levels or where one canonical level spans multiple family abstraction levels. Thus, canonical levels are a characteristic of the solution frame, and an independent set of abstraction levels is characteristic of each family. Of course, in a family structure, patterns do not have to appear at every level of abstraction in that family, or for the frame as a whole. At the top of every family, however, there is always an abstract security pattern capturing the high-level solution addressed by that family. Pattern families of security (software) patterns always end with a Technology realization level as part of their hierarchy of abstraction levels, which contains low-level idioms (see [11]), descriptions of various technologies as done in [30], and/or technology-/implementation-specific patterns. Therefore, the progression of patterns downwards in the pattern family, or, equivalently, across the development stages, guides developers from concepts, to architectural solutions, to concrete design enriched with enough detail, becoming closer to templates that can be “plugged” into an architecture (see [31]).

Besides patterns relating to each other, families and frames can also be related via a set of *solution relationships*, which adapt a number of standard pattern relationships appearing in the literature: requires, supports, depends on, derives from, alternative, clashes (see [12]). Relationships can be either one-way or two-way, for example, when solution A and B rely on each other mutually. Security (software) patterns from different levels of abstraction also implicitly fulfill a “refine” relationship, analogous to the “realize” relationship of [32], where pattern B refines pattern A if A is at a higher level of abstraction than B. Figure 1 uses loose UML semantics for the Solution relationship association class, which should be interpreted in light of the explanation above, namely, as meaning that the same class of relationships can exist between the different solution types.

Besides encapsulating pattern families of security (software) patterns, each solution frame has a single family that encompasses a set of *security process patterns* [33]. Security process patterns are applications of the process pattern concept of Ambler [34] introduced first in the context of object-oriented development methodologies. Like their general counterparts, security process patterns can be classified into three levels of decreasing granularity, ranging from phase patterns, stage patterns and task patterns. They can also be classified as macro-process – pertaining to the phases, stages and tasks of security methodology; or micro-process – pertaining to activities resulting from the application of a particular security solution (see Figure 1). The specific process patterns in the single process pattern frame can also reside at different levels of abstraction, however, patterns at lower levels do not necessarily refine ones at higher levels.

Instantiating a particular security frame implies progressively selecting and instantiating patterns from the relevant families to create a sequence of patterns (see [35]), or even a whole tree-like structure, which is itself reusable. Such tree structures capture the design decisions taken to realize a solution in a particular situation, and can also promote greater traceability from high-level security policies, their progressive refinements into specific policies, and their respective (partial) realizations via the use of patterns (see [36]). Thus, by virtue of their structure, solution frames embody an implicit process for realizing the security policy encapsulated in their corresponding tactics, which can also be captured explicitly in the form of a phase micro-process pattern (belonging to the process pattern family of the relevant frame).

Since each solution frame, the encapsulated pattern families and even the abstraction layers within the families are autonomous, the various solutions can be used independently, while realizing one particular root policy (encapsulated in the tactic), their parts can be used in a wide variety of situations in a methodology-agnostic fashion, or within any pattern-based security methodology (see [10]).

For the rest of this paper we present three security solution frames, each of which uses cryptography in some essential fashion: a *Secure communications* frame; an *Identity management* frame (concentrating on authentication in particular) and a *Security information management* frame (concentrating on key management).
3 Secure communications solution frame

Since components in a distributed system are required to interact across machine boundaries and hence over (potentially) untrusted networks, protecting all data in transit, in the broadest sense, is one of the central concerns of distributed systems security. The Secure communications solution frame is constructed with the aim of addressing the latter concern comprehensively. It consists of three pattern families: Secure two-party communication, which contains basic patterns for setting up a secure channel, secure communication sessions etc.; Secure group communication, which contains a set of variants of the basic patterns, extending them to a group communications setting; and Secure routing, which contains patterns concerned (predominantly) with application-level routing issues. In the following subsections we present the first (most important) of the aforementioned families, namely, the Secure two-party communication pattern family.

This security solution frame realizes a “Protect communications” tactic, which encompasses “Maintain data integrity” and “Maintain data confidentiality” from [25] in a networked communications context. It requires (solution relationship) the Security information management frame in Section 4.

3.1 Secure two-party communication pattern family

A secure two-party channel (or simply secure channel for the rest of this sub-section) defines a protected, logical communications medium between exactly two parties interacting in a point-to-point fashion over an open network. In our approach, a secure channel design is created by utilizing an abstract “base” pattern and instantiating pattern modifiers [37] (cf. [12] for authorization) to determine the precise protection semantics, i.e. which attributes will be enforced: secrecy, integrity, authenticity and/or availability. This abstract channel can then be realized, guided by other patterns in the family, using various cryptographic primitives – custom or pre-defined security protocols, or an ad-hoc mixture of cryptographic algorithms. If a channel is flooded with messages, then another solution frame should be used – Filtering (covering firewalls and other concerns) – which is outside the scope of this paper.

Regarding abstraction levels, the base pattern in this family resides on the Conceptual analysis level, while the modifiers collectively span the Conceptual Analysis and Abstract Architecture (canonical) abstraction levels. Patterns for refining the design of the secure channel with session-based semantics, or encapsulating design decisions as to where the channel should be incorporated, what protocols to use etc. appear in the Abstract Architecture, Solution Design and Practical Implementation Details abstraction levels. We have chosen to omit the technology descriptions that would otherwise make up the customary Technical realization abstraction level. While valuable, they are not essential to the use of the pattern family, and are more appropriate for a different type of publication, such as a publicly available, online database.

The secure channel presented in this section also forms the basis of secure group (n-party) communication (outside the scope of this paper), since all group communications can be reduced to instances of one-to-many interactions, and the latter into many one-to-one or two-party interactions.

While the pattern family encapsulates a number of new security patterns, a fair percentage of patterns are also summaries or adaptations – with minor improvements – of existing ones in the literature (see [10]). In these cases the latter are indicated in a Reference patterns section as part of the textual template (this remark holds true for the all pattern families presented in this paper). We make the assumption that the reader will venture to read some of these reference patterns, to gain a fuller understanding of the relevant solution(s). This assumption allows us to concentrate more on the general content of each pattern rather than the finer-grained details, and hence to present a more complete pattern family, as opposed to a few constituent patterns in detail. It also helps to counter the natural tendency of software patterns – to which our work is also not immune – to “re-invent the wheel”.

All patterns are referred to in the text using SMALL CAPS font, and the type of pattern is indicated in the headings in brackets. We employ the micro-pattern (textual) template as described in [38] for presentation, with the additional section for reference patterns as described above.

3.1.1 Secure message channel (abstract security pattern)

Also known as: ONE-TO-ONE SECURE MESSAGE CHANNEL
Reference patterns: SECURE COMMUNICATIONS CHANNEL [9].
Description: The Secure message channel adapts the Secure communications channel of Schumacher and colleagues [9] by extracting the essential elements underlying a protected two-party communication stream. In such a channel messages can be protected from passive (eavesdropping) attacks by enforcing secrecy, and from active (modification) attacks by enforcing integrity, authenticity and (simple) availability properties. Secure routing of messages can also be enforced for applicable systems.
Conceptually, a SECURE MESSAGE CHANNEL can be represented simply as a “secured” version of the MESSAGE CHANNEL pattern for distributed communication in [35], which, from a software architecture viewpoint, is nothing other than a coarse-grained, message-based connector between distributed components (see Figure 2 for a simple conceptual model). In this case we only consider the message channel to be between two components (cases where multiple components are connected by the same channel, e.g. as in a publish/subscribe infrastructure [39], are considered as part of the Secure group communication pattern family (which, as mentioned earlier, is outside the scope of this paper). The level of abstraction of the channel is arbitrary: it may represent a network communication link or it may be layered on top of other, lower level channels (e.g. as in a peer-to-peer overlay – see [40]). The interaction between components using the secure channel is analogous to using the original MESSAGE CHANNEL pattern: component A writes a message to component B by calling a “send” operation on the channel, which in turn (logically) handles all protection aspects of the message and delivers it to the recipient B, which can optionally call a “receive” operation. It should be noted that, with respect to implementation, the SECURE MESSAGE CHANNEL functionality may span the participating components as well as the communications infrastructure.

3.1.2 Pattern modifiers

Modifiers are specific types of patterns that can extend, refine or determine a variation of other software patterns (see [37]) when used in combination. They may not and need not present complete design solutions by themselves. Below we outline a collection of pattern modifiers, which can be applied as variants or extensions to determine the protection semantics of a SECURE MESSAGE CHANNEL. The template used is the established simple micro-pattern (textual) template (see Section 3.1), augmented by a Threats mitigated section, which refers to the threat patterns for general distributed systems proposed in [41].

3.1.2.1 Message encryption (security pattern modifier)

Reference patterns: DATA CONFIDENTIALITY [42], INFORMATION SECRECY [43], SYMMETRIC ENCRYPTION [44] and SECURE COMMUNICATION [45] are more detailed versions of this modifier.

Context: When the contents of messages need to be protected against disclosure.

Problem: When messages pass over untrusted networks, their contents can easily be read by unwanted eavesdroppers. How can the message contents be hidden or protected from disclosure?

Solution: Use encryption cryptographic primitives to “scramble” the valuable parts or contents of messages, making them useless for all but the intended recipient(s), which are capable of decrypting them. The encryption algorithms can be either symmetric (see SYMMETRIC ENCRYPTION, as referenced above), or asymmetric, a choice that is determined by performance requirements as well as by how keys will be managed (see the patterns in the Key management pattern family in Section 4.1).

Channel properties affected: secrecy

Threats mitigated: MESSAGE SECRECY VIOLATION (passive eavesdropping).

3.1.2.2 Message tampering protection (security pattern modifier)

Context: When the contents of messages need to be protected against modification by adverse parties.

Problem: When messages pass over untrusted networks – especially via untrusted nodes – their contents can easily be changed by unwanted eavesdroppers. How can the messages be protected from modification?

Solution: Use a cryptographic algorithm with the message contents as input to generate either a piece of data akin to an error detection code, which can be attached to each message; or a transformed message in which modification can easily be detected. Two types of algorithms can be used for this purpose: symmetric (MDCs and MACs) and asymmetric (digital signatures).

In the symmetric case, an MDC (unkeyed hash function) can be used to generate a digest of the contents for each message. To protect the digest itself from tampering, MESSAGE ENCRYPTION must be used in conjunction (applied to the whole message – content and appended digest). Similarly, a MAC (keyed hash function) can be used to generate a hash, which is appended to the message contents. The use of a MAC also requires key management.

In the asymmetric case, the private key of a chosen digital signature scheme is used to sign the messages, and the signature is verified at the receiving end using the corresponding public key.

Channel properties affected: integrity + authenticity + secrecy (using MDCs and MESSAGE ENCRYPTION); or integrity + authenticity (using MACs / digital signatures – see MESSAGE SOURCE AUTHENTICATION below)

Threats mitigated: MESSAGE INTEGRITY VIOLATION (active eavesdropping, modification), MESSAGE AUTHENTICITY VIOLATION (source spoofing).

Relationships: Requires MESSAGE ENCRYPTION if using a symmetric-key solution.

3.1.2.3 Message source authentication (security pattern modifier)

Reference patterns: DATA ORIGIN AUTHENTICATION [42].

Context: When the sender of messages need to be known, and when messages need to be protected against
modification by adverse parties.

**Problem:** When messages are received it is not necessarily possible to identify with absolute certainty whether they came from the original sender, or from an adversarial party. How can the messages be authenticated?

**Solution:** Generate a cryptographic hash of the message using a key; the possession of the key is proof of message authenticity in this case. The rest of the solution is as in the **MESSAGE TAMPERING PROTECTION** pattern.

**Channel properties affected:** as for **MESSAGE TAMPERING PROTECTION**.

**Threats mitigated:** **MESSAGE AUTHENTICITY VIOLATION** (source spoofing), **MESSAGE INTEGRITY VIOLATION** (active eavesdropping, modification)

### 3.1.2.4 Message replay protection (security pattern modifier)

**Reference patterns:** **MESSAGE REPLAY DETECTION** [42]

**Context:** When sending messages over an untrusted network. When messages need to be protected against modification by adverse parties.

**Problem:** Messages are sometimes required to be unique, and not re-sent older messages. This is especially important in the context of security protocols (freshness property), but also in ordinary communication protocols. Messages traveling through an untrusted network can not only be replayed by an adversarial party, but also deleted or re-ordered to cause unwanted effects. There should be some way of enforcing the uniqueness.

**Solution:** Attach one of three time-variant cryptographic primitives to each message: a random value nonce or a timestamp for simple request/reply message exchanges; or a sequence number for stream-based communications.

- **Stream-based communications:** Monotonic numerical sequences create an ordering of all messages on the channel, such that if a message arrives with a lower sequence number than previous messages it can be classified as being out of order. Similarly, if a sequence number is missing (in the case of sequence numbers constructed strictly consecutively), it would indicate a missing message.

- **Request/reply message exchanges:** Nonces and timestamps ensure message freshness, as long as they are truly random or accurate within a given window, respectively. Generating random value nonces requires a good RNG, and potentially buffering at one or two ends of the communication to ensure that values are not repeated (at all or over large time periods). Timestamps require time synchronization between the communicating parties (nodes in the distributed system), e.g. the use of a global timer. Keeping time synchronized in a distributed system is a well-known problem that is not always easy to solve, and must be traded-off against the benefits involved. Some network protocols are dependent on synchronized time, and their integral use could justify the extra effort involved in realizing the timestamp solution.

**Channel properties affected:** integrity (of the message channel, not of the data itself)

**Threats mitigated:** **MESSAGE INTEGRITY VIOLATION** (active eavesdropping).

### 3.1.2.5 Channel liveness detection (security pattern modifier)

**Context:** When sending messages over an untrusted network.

**Problem:** Besides deleting messages sporadically from a communication channel, an adversarial party can also block all messages or launch a denial-of-service attack against one or more message channels in the distributed system, effectively cutting off components from each other. It should be possible to at least detect such cases, and react to them accordingly.

**Solution:** Use a **HEARTBEAT** [46] pattern for sending periodic “are you alive” messages between communicating parties at set time intervals [47], or generate **ACKNOWLEDGEMENTS** [46] at either the message level or higher application levels (e.g. requests, remote function calls etc.). In both cases the solutions of the patterns above, as well as their realizations, remain the same, but are simply carried over into the distributed communication setting. The time interval of the messages will determine for how long a communication channel is allowed to be unavailable undetected, and should be traded-off against the overhead generated by the additional messages over the network.

**Channel properties affected:** integrity, availability

**Threats mitigated:** **MESSAGE INTEGRITY VIOLATION** (active eavesdropping).

### 3.1.3 Solution design level

Once the specifics of a **SECURE MESSAGE CHANNEL** have been established, design decisions must be made regarding how that channel is set up, and where exactly in the architecture it should be incorporated. Patterns encapsulating the latter decisions appear in the next two (pattern family) abstraction levels.

#### 3.1.3.1 Channel setup abstraction level

**Secure communication session (security pattern)**

**Reference patterns:** **CLIENT-SERVER SECURE COMMUNICATION** [48].

**Context:** When realizing a **SECURE MESSAGE CHANNEL**.
Problem: Sometimes applications require both secure and insecure communication over the same connection at different times, e.g. in an e-commerce setting some messages can be sent over an insecure channel, while others – when performing a business transaction – are sensitive and require a secure channel. A closely related problem is when an application or system needs to set up and configure a SECURE MESSAGE CHANNEL between two or more components either at start-up or dynamically, during run-time.

Solution: Create a SECURE MESSAGE CHANNEL between the communicating parties delimited by a notional SESSION instance. Once the session is established, messages can undergo transformations until the session's time limits (pre-set or as a result of cryptographic material updates) have been reached, after which insecure communication can resume, or a new session can be established. The lifetime of the session can also be infinite, in which case all communications will be secured.

As in the CLIENT-SERVER SECURE COMMUNICATION pattern, a SECURE LAYER is used to separate the secure communications functionality from the rest of the system (this design approach can be further refined using the APPLICATION-DRIVEN MESSAGE TRANSFORM and/or INFRASTRUCTURE-DRIVEN MESSAGE TRANSFORM patterns in Section 3.1.3.2). Within this layer, dedicated communication components (nominally Sockets) are tasked with performing transformations on messages according to the protection semantics of the channel. The resulting messages are sent over the network, where corresponding transformations (including verification) are applied.

With respect to dynamics, there are three main steps in realizing the solution, in accord with the general process for communications (cf. [50]):

1. **session establishment**: one party establishes a connection with another party and negotiates any security parameters. Since stronger cryptographic protocols are developed over time, one important parameter should be the cryptographic algorithms used for subsequent operations [18]. To ensure that the connection request is not spurious, the parties should authenticate each other [47] – see the Authentication pattern family in Section 5.1 for different approaches to this. A fresh identifier should also be established for the lifetime of the particular session.

2. **message transfer**: messages are transmitted over the network via the aforementioned SECURE LAYER instance. Any identifiers can be passed along with the messages and protected by the cryptographic primitives employed for the channel.

3. **session release**: the parties close the connection in a secure fashion.

The realization of each of these steps is usually addressed in various protocols, which can be selected or constructed by applying the SECURE PROTOCOL DESIGN process pattern (see Section 3.2).

With respect to the SESSION instance, the actual session data can be transmitted as part of an INVOCATION CONTEXT [51].

### 3.1.3.2 Communication semantics abstraction level

This abstraction level contains two patterns determining how cryptographic functionality will be applied to transform (original) messages to their protected versions.

**Application-driven message transform (security pattern)**

**Context**: When realizing a SECURE MESSAGE CHANNEL in a distributed environment.

**Problem**: For smaller systems, it may be necessary to have full control of the secure communications functionality. At the same time, keeping communications relatively separate from the business logic is an advantage.

**Solution**: Place all secure communications functionality as separate modules or subsystems, which are accessible by all components requiring communications functionality. All security-related transformations on messages require explicit invocation of functions from the dedicated modules or subsystems. The latter can be wrapped using ADAPTERS [52] or more appropriately – especially if implementing a SECURE COMMUNICATIONS SESSION – a WRAPPER FACADE [53], allowing the standard network communications functions to be replaced by name-equivalent secure versions. The STRATEGY pattern [52] can be applied to ensure that the actual cryptographic algorithms used to implement different SECURE MESSAGE CHANNEL features can be chosen dynamically at run-time.

**Infrastructure-driven message transform (security pattern)**

**Context**: When realizing a SECURE MESSAGE CHANNEL in a distributed environment.

**Problem**: System designs that interleave significant secure communications functionality in the business logic are problematic for both maintenance and development. There should be a way to encapsulate the majority of communications-related functions relatively transparently.

**Solution**: Place all secure communications functionality as a set of middleware components, with hooks for applications to either configure or specify certain parameters or extension functions. The latter components can be realized as MESSAGE VALIDATORS [42, 15] alongside, or as, INVOCATION INTERCEPTORS [51]. In the latter case
the interceptors capture all messages and transform them automatically without any intervention from the application itself. Security parameters used for establishing a SECURE COMMUNICATION SESSION can be transferred as part of the INVOCATION CONTEXT [51] for various operations or requests.

3.1.4 Practical implementation details level

3.1.4.1 Key distribution level

In many ways secure communications turns the problem of protecting a communications channel via cryptography to a problem of distributing (and ultimately protecting) cryptographic keys for the corresponding primitives [54]. In this sense, key distribution (understood here to encompass both key agreement and transport [19]) is an essential part of implementing secure communications. For its realization, one can make use of the Key management pattern family from the Security Information Management solution frame, the details of which are presented in Section 4.

3.1.4.2 Technology realization level

In this paper we omit the details of the technology realizations. If they were to be included, this level of abstraction would contain various technology-/implementation-specific patterns, such as XML Encryption and XML Signature for Web-services [44, 55] (see also [56, 15]); as well as technology descriptions as per [30] for the various cryptographic libraries that realize the chosen security protocol(s) and/or algorithms, e.g. OpenSSL (www.openssl.org) or the Java JSSE framework for TLS/SSL. Uniform descriptions of various libraries, algorithms and protocols can be supplemented by descriptions of secure communications technologies onto which the patterns from higher abstraction levels can also be partially or fully mapped, e.g. GNUnet (www.gnunet.org) for secure peer-to-peer communications.

3.2 Security process pattern family

This section describes the micro-process patterns for the Secure communications solution frame.

3.2.1 Protocol design level

This abstraction level (spanning the canonical Solution design and Practical implementation details abstraction levels) contains a single pattern – SECURE PROTOCOL DESIGN – concerned with security protocol design, delineating the activities required of developers to either choose or construct an appropriate protocol for realizing different aspects of any relevant pattern solutions. Since the SECURE PROTOCOL DESIGN pattern is not only applicable to secure communications protocols, it can be re-used across other solution frames and contexts.

3.2.1.1 Secure protocol design (stage micro-process pattern)

Context: When realizing subsystems or modules which use cryptographic primitives.

Problem: Often a number of operations involving cryptography must be performed in a predetermined sequence, such as initializing a connection or negotiating parameters. An ad-hoc use of primitives can be a reasonable solution in some situations, however, this approach provides no guarantees that the sequence of operations cannot be easily exploited by an adversary.

Solution: Choose or construct an appropriate security protocol, or secure an existing communications/business functionality protocol, to encapsulate a given sequence of operations and to specify “how the cryptography should be used” [23]. The protocol must be resilient against a number of attacks, including active and passive eavesdropping on any of the communication links involved, exploitation of special (e.g. initial, final) protocol states, message re-use, and others (for which see [41], especially the threat patterns belonging to the Network protocol attacks class). If an existing protocol is chosen, either from a standard (NIST, ISO/IEC and IETF standards being the most ubiquitous) or some other form of documentation (articles, books), it must be matched against the given context and requirements for suitability. Descriptions of protocols in the literature, such as TLS – described in [57] as a technology-/implementation-specific security pattern, can also guide this latter process.

If a custom protocol is required, or an existing business functionality protocol must be secured, there are several options, the two most popular being the utilization of formal methods (e.g. the use of logics such as BAN or GNY; process algebras such as CSP; or symbolic representations) in conjunction with corresponding software validation tools for model checking or theorem proving (see [23] for an overview); or via careful manual inspection and simulation. Of particular interest are approaches utilizing UML as a means of representing the protocol and model translation tools for verification as, for example, in the work of Smith et al. [58] or Jürjens' UMLsec [2, 59]. The latter approaches are somewhat more “developer-friendly” inasmuch as UML is significantly more familiar to developers than formal languages or process algebras. A case study reporting the use of UMLsec to develop a custom security protocol in an industry scenario can be found in [6]. The precise approach taken to construct the protocol will dictate the concrete steps required to realize the solution of this pattern.
One important point to note when designing a protocol is to avoid introducing custom cryptographic algorithms; unless necessary, it is universally recommended to rely on well researched, peer reviewed, time-proven algorithms from the literature [20, 3].

The threat patterns from [41] can be used to complement the protocol construction process, and can also support the process of implementing the protocol in software (especially the testing and verification stages).

3.2.2 Network configuration level

The single network configuration pattern in this abstraction level (spanning the canonical Solution design and Practical implementation details abstraction levels) is mainly applicable during the implementation and deployment phases of development. Its application can help to ensure that the underlying network setup does not undermine the security measures introduced by patterns instances from the rest of the Secure communications solution frame. The pattern's task granularity reflects the fact that its instance could stand alongside other, related process patterns not addressing communications concerns, as part of an overall deployment “configuration” stage micro-process.

3.2.2.1 Network infrastructure configuration (task micro-process pattern)

Context: Large enterprise (distributed) systems running on or across LANs.

Problem: It is important to ensure that the network (LAN or WLAN) infrastructure underlying the distributed system deployment is not the weakest link in the security chain. Unless taken into account, vulnerabilities arising from the setup of the network can be exploited regardless of how carefully the system's software architecture has been secured.

Solution: Configure the network infrastructure (software on routers, switches and node network configurations) in accord with security policies and to support the security communications architecture of the deployed system. Example aspects of such configuration include setting up the network topology so that external interfaces are clearly separated (e.g. using a de-militarized zone in conjunction with relevant patterns from the Filtering solution frame, which is outside the scope of this paper); creating VLANs and subnets to promote safer routing; configuring ACLs on routers etc.

3.2.3 Implementation workflow for the secure communications solution frame (phase micro-process pattern)

Implementing the secure communications frame in practice can be done by following the steps outlined below (some of which can be performed iteratively). These steps can be interpreted as summarized constituents of a phase micro-process pattern (part of the process pattern family for this solution frame) that makes explicit the refinement process implicit in the frame's structure. Completely analogous patterns – which we have chosen to omit for the sake of brevity – can be made for the other solution frames in the ensuing sections.

1. instantiate the abstract SECURE MESSAGE CHANNEL, together with a set of appropriate modifiers to determine channel's protection semantics;
2. determine how the channel will be set up and where in the software stack (application/middleware) it will be implemented, by instantiating the SECURE COMMUNICATION SESSION and APPLICATION-DRIVEN / INFRASTRUCTURE-DRIVEN MESSAGE TRANSFORM patterns, as required;
3. select appropriate cryptographic algorithms and/or technologies for realizing the SECURE MESSAGE CHANNEL functionality;
4. choose or construct any relevant security protocols, especially SECURE COMMUNICATION SESSION, by applying the SECURE PROTOCOL DESIGN pattern (the latter may interact with the cryptographic algorithms used);
5. determine how cryptographic keys will be handled by instantiating the relevant patterns from the Key management pattern family (Section 4.1);
6. another iteration of SECURE PROTOCOL DESIGN may be necessary to ensure that the communication and key management protocols are adequate and mutually supportive.

4 Security information management solution frame

In this section we present a Key management pattern family, which forms part of a solution frame for managing the information used in a system's security infrastructure. The solution frame is related to the Identity management frame in this paper via a depends on relationship – and realizes a “Manage security data” tactic (this tactic is an addition to the usual list in [25], which, as indicated in [28], should not be taken as definitive and complete).
4.1 Key management pattern family

The Key management pattern family is concerned with managing – i.e. generating, distributing, revoking etc. – the cryptographic keys used in any relevant cryptographic algorithms [60, 61]. The basic management operations referred to above are encapsulated in the abstract Key MANAGEMENT SCHEME pattern, which resides at an abstraction level corresponding to Conceptual Analysis for the whole solution frame. The patterns in subsequent abstraction levels help to determine appropriate protocols and architectures for distributing the keys. Some of these patterns can be seen as adaptations of equivalent patterns for managing security policies, for which see [12], while others are new distilled solutions to the problem at hand. As for all the other solution frames, the encapsulated patterns can be used not only during the design of new systems, but also to analyze existing ones, as well as to help in the understanding and application of standards such as XKMS (http://www.w3.org/TR/xkms).

4.1.1 Conceptual analysis level

4.1.1.1 Key management scheme (abstract security pattern)

Context: Any distributed system which makes use of patterns or other solution constructs relying on keyed cryptographic primitives in the implementation of its security architecture.

Problem: To manage the keying material used in cryptographic primitives.

Solution: Construct a key management scheme, whereby participating components can receive keys from one or more components called collectively the Key manager, which can be thought of (and represented) as a single, abstract component. The Key manager encapsulates and supports a number of abstract operations (see, for example, [62, 63, 61]) – the following descriptions of which are adapted from [62]):

- **key generation**: operations for securely generating keys or pairs of keys, guaranteeing unpredictability and good cryptographic qualities of the generated keys;
- **key distribution**: operations for safely and reliably providing keys to participants which legitimately ask for them;
- **key storage**: operations safely and reliably storing keys for future use by the involved participants;
- **key update**: operations for safely and reliably updating one or more keys of a given participant by replacing the old key(s) with new (fresh) key(s);
- **key destruction**: operations for assuring secure destruction of outdated and no longer needed keys;
- **key archival**: operations for securely archiving keys used in notarization and non-repudiation activities, in case they need to be retrieved later as proofs in possible litigation.

Each of the operations can be realized in a variety of ways, and is the concern of the patterns in the lower levels of abstraction.

There are several types of keys, each of which may require different management strategies. These include data keys – used in performing cryptographic operations on data or communications messages, such as encryption, creation of message digests and others; and key-encryption keys (KEKs) – which, as the term implies, are used to encrypt other cryptographic keys.

4.1.2 Abstract architecture level

The patterns at this level of abstraction seek to establish a base architecture for the whole key management infrastructure, providing a general basis for realizing the individual Key MANAGEMENT SCHEME operations.

4.1.2.1 Basic architectural elements abstraction level

**Trusted Third-Party (security pattern)**

Context and Problem: When establishing keys in a distributed environment, often there is a need additional infrastructure external to the existing system users and/or components to support the implementation of relevant Key MANAGEMENT SCHEME operations.

Solution: Construct or determine a set of components or a whole infrastructure, which all system components can trust for certain key management operations. Such infrastructure can either be in-line, according with the MEDIATOR pattern [52], acting as an intermediary between all the participating components; on-line, accessible and available on the network to all or selected (authorized) components; or off-line, implying some form of out of band access. In the first two cases the component can be part of the system's architecture, while in the third it may be a separate, unrelated piece of software providing services necessary for key distribution.
**Key Certificate (security pattern)**

**Context:** When managing cryptographic keys, especially asymmetric keys.

**Problem:** How can keys be protected so that their authenticity and integrity is assured?

**Solution:** Attach additional information to keys, or encapsulate keys within a specific data structure containing an identifier to which the key is bound, to form a certificate. The resulting certificate can be combined with a digest (using an unkeyed hash function) to detect tampering, then encrypted using another key (often called key wrapping) – which in turn becomes a KEK – and stored publicly. Alternatively, the certificate can be digitally signed by a CERTIFICATION AUTHORITY – an off-line TRUSTED THIRD-PARTY. The latter certificates are usually termed **public key certificates**, while the former are termed **symmetric key certificates**.

**Session Key (security pattern)**

**Context:** When symmetric-keyed cryptographic primitives are required as part of implementing a given security architecture.

**Problem:** To circumscribe the use of cryptographic keys within some kind of limited scope.

**Solution:** Generate and use a symmetric key only within a given SESSION [49], and destroy it immediately afterwards. The key same should not be used across multiple sessions, which may require keeping a history of used keys. A SESSION KEY can also have a pre-set time limit on its use (cryptoperiod), and once this expires, it can be exchanged in the same SESSION using an appropriate implementation of an update / (re-)distribution KEY MANAGEMENT SCHEME operation.

SESSION KEY may be realized either as part of the management functionality or as a **KEY WITH ATTRIBUTES**.

**Key with Attributes (security pattern)**

**Context:** When using cryptographic primitives within a SESSION [49].

**Problem:** To circumscribe the possible uses of a given key.

**Solution:** Attach attributes or properties to a key, such that they (new) key itself becomes a tuple:

\[ \text{Key'} = (\text{Key}, \{\text{attribute set}\}) \]

where the attribute set can determine users or processes allowed to use the key; functionality that they the key can be used for; time-limits on the key etc. Attributes effectively define a set of restrictions, or, in other words, a set of authorization rules, which can be seen as forming a kind of ACCESS CONTROL LIST (ACL – [64]) on the key. Using attributes thus implies the need for constructing a key-management specific (potentially simplified) authorization infrastructure, for which see [12].

### 4.1.2.2 Key management architectural styles abstraction level

**Centralized Key Management Architecture (abstract security pattern)**

**Context:** When the system is based on a client-server distributed architecture, or when the number of components/nodes in the system has fixed or predictable growth.

**Problem:** To manage keys in a distributed environment.

**Solution:** Use an on-line or in-line centralized TRUSTED THIRD-PARTY instance for most or all KEY MANAGEMENT SCHEME operations. Keys are generated either by the TRUSTED THIRD-PARTY (a fully centralized solution), or by the participants (a partially centralized solution), who subsequently send them to the TRUSTED THIRD-PARTY instance. In both cases, the latter transports (or forwards, after verification) the keys to all necessary parties.

**Decentralized Key Management Architecture (abstract security pattern)**

**Context:** When the system is based on a peer-to-peer distributed architecture, when the number of components/nodes in the system is not fixed, or when there are multiple equal entities that partake in key management.

**Problem:** To manage keys in a distributed environment.

**Solution:** Participants either agree on a key (symmetric or asymmetric) by communicating relevant cryptographic, publicly accessible parameters to each other and deriving the key as a function of the latter; or acquire asymmetric keys by generating them and using an on-line or off-line TRUSTED THIRD-PARTY instance to certify them. In the latter case the keys can be made publicly available or given by the parties themselves (with message authenticity guarantees). In the former case some form of mutual authentication or a ONE-TO-ONE SECURE MESSAGE CHANNEL is required between parties to obviate man-in-the-middle attacks.
4.1.3 Solution design level

At this abstraction level more concrete decisions can be made regarding the key management architecture via three basic patterns: CENTRALIZED KEY SERVER (realizing CENTRALIZED KEY MANAGEMENT ARCHITECTURE), PUBLIC KEY INFRASTRUCTURE, and PARTICIPANT-BASED KEY MANAGEMENT (both realizing DECENTRALIZED KEY MANAGEMENT ARCHITECTURE). The latter patterns can be further refined via the use of the PUSH SECURITY INFORMATION and PULL SECURITY INFORMATION modifiers from the Policy management pattern family in [12] from the server / third-party component(s). These two pattern modifiers are briefly summarized in Section 4.1.3.2 (in a separate abstraction level), adapting the original descriptions for key management where appropriate.

4.1.3.1 Basic solution design abstraction level

Centralized Key Server (security pattern)

**Context:** When managing keys in a distributed system, both symmetric and asymmetric.

**Problem:** To manage keys coherently, but without burdening application logic with the actual management functionality.

**Solution:** Create a centralized cluster of dedicated software components to realize an on-line TRUSTED THIRD-PARTY – notionally a server, or Key Distribution Center (KDC) [65] – that together implement all required operations of the KEY MANAGEMENT SCHEME. These operations can be implemented as follows:

- **key generation:** keys are generated in the centralized server component(s). This specifically includes symmetric SESSION KEYS, as per CRYPTOGRAPHIC KEY GENERATION [45] or SESSION KEY GENERATION [48], as well as asymmetric key pairs (see PUBLIC/PRIVATE KEYS GENERATION [48]), which can be used in subsequent cryptographic operations by requesting components.

- **key distribution:** as a first step, all participants must share long-term secret keys with the server component(s), which can be securely distributed off-line, e.g. via off-line administrative means. During run-time, keys are transported by the server to appropriate participants, usually after establishing identity (see the Identity management solution frame in Section 5). The precise details of the key distribution algorithms are left to other patterns (see Section 4.1.4.1).

- **key storage:** keys are stored by the server in a controlled back-end data store, such as a secured SHARED REPOSITORY [66], with access controlled using an appropriate authorization infrastructure (see [12]). A KEY CERTIFICATE can also be used for each participant's key, which is encrypted using a master key belonging to the server component(s), and stored in a public repository. This obviates the need for the server to keep secure copies of all keys, but increases the need to secure the master key.

The update, destruction and archival KEY MANAGEMENT SCHEME operations are likewise handled by the centralized server component(s), acting as a mediator for all key-related operations. A representative architectural model of the solution can be seen in Figure 3.

The server component(s) can be replicated for increased availability, however, each replica server must be physically secure and enforce strong access control for the stored keying material.

**Consequences:**

(+) The server's location – and hence administration – is centralized, allowing for easier setup and configuration.
(+ There are no data consistency requirements, since everything is stored in a single location (or, in the case of replication, is kept consistent by lower level mechanisms).
(-) If the centralized server component – or one of its replicas – is compromised, then the security of all cryptography-based functions in the system is uniformly compromised.
(-) For large systems, the server may become a communication bottleneck if required to perform keying operations frequently.

**Public Key Infrastructure (security pattern)**

**Context:** When managing asymmetric keys, especially across multiple administrative or security domains, or when the system's architecture is decentralized and a corresponding key management strategy is required (for asymmetric keys).

**Problem:** There is a need to manage asymmetric keys coherently so that components on each site in a domain using cryptographic primitives can acquire keys.

**Solution:** Use a **CERTIFICATE AUTHORITY (CA)** – a TRUSTED THIRD-PARTY realization – to produce **KEY CERTIFICATES**, which can then be distributed either on trusted repositories or securely transported (via a ONE-TO-ONE SECURE MESSAGE CHANNEL) by users. The **KEY CERTIFICATES** consist of the public key, an identifier for the system entity bound to the key, the digital signature of the CA. Additional information can be attached either to the key (as per **KEY WITH ATTRIBUTES**) or to the certificate itself. This pattern follows the **CERTIFICATION INFRASTRUCTURE** (Section 5.1.2.3) and **CERTIFICATE-BASED AUTHENTICATION** (Section 5.1.3.3) patterns, since such an authentication approach necessarily implies key distribution.

The **CERTIFICATION AUTHORITY** can be either off-line or on-line. In the latter case, the functionality of the **CERTIFICATE AUTHORITY** can be realized either in the form of a centralized software server, or as part of a number of participating components – i.e. as a decentralized CA (see [67]), wherein the CA’s private key is distributed among participants using a \( (k,n) \)-threshold secret sharing scheme (for which see [68]). A decentralized **CERTIFICATION AUTHORITY** can also be realized by parties digitally signing each other’s keys based on a “web of trust” model (following the OpenPGP approach – www.openpgp.org), in which case the CA private key is notionally the set of all private keys of the signing (trusted) parties.

Besides a **CERTIFICATION AUTHORITY** instance, other representative components of the solution structure include (cf. [19]) – see Figure 4:

- a **Key Generator** component, which relies on **RNG** primitives (see Section 2.1.1) for its functionality;
- a naming or **Identity Allocator** component, which assigns the unique names or identifiers of users, processes etc. in the system (e.g. an instance of the **LOOKUP** pattern in conjunction with **GLOBAL ID** [51]);
- a public **Storage Directory** for certificates.

All the latter components, like the CA instance, can either be centralized or distributed among the participants of the system. Using this solution, the operations of **KEY MANAGEMENT SCHEME** can be realized as follows:

- **key generation**: a public/private key pair is generated by the parties concerned, the CA or another TRUSTED THIRD-PARTY instance, and its public part is signed by the CA.
- **key distribution** and **storage**: keys (within, or accompanied by, certificates) are either available in the **Storage Directory**, or can be obtained from the parties to which the keys are bound. The latter use case realizes the **Storage Directory** component as a local **PUBLIC KEY DATABASE** [45].
- **key update** and **destruction**: if a private key is compromised or a certificate requires update, the CA must revoke the old certificate and issue a new one – see **CERTIFICATE REVOCATION** [45].

![Figure 4: Partial deployment view of Public Key Infrastructure solution](image-url)
As for CENTRALIZED KEY SERVER, the precise realization of the operations above will depend on the protocols chosen (see Section 4.1.4.1).

When a party receives or acquires a KEY CERTIFICATE for another party, it can follow a pre-defined set of steps to verify its authenticity [19]: first the party acquires the public key of the CA (one-time operation for all subsequent verifications) and verifies its authenticity if required; then the party checks the certificate has not expired and is not revoked; and finally verifies the digital signature on the KEY CERTIFICATE using the CA's public key. The key is then ready for use.

In the case of multiple security domains, each domain can be assigned a particular CA, and a hierarchy of CAs can be formed to certify the keys of other CAs [19]. The result is a KEY CERTIFICATE chain – rather than a single KEY CERTIFICATE – which is verified in an analogous fashion to the single CA case (see [19]).

Consequences:

(+ ) There is no single point of failure; each site or security domain manages its own certificates.

(+ ) Once the certificates are generated and made publicly available (or given to each participant), the CA is no longer required.

(-) Certificate revocation can be problematic, since revocation lists (CRLs) may not be distributed sufficiently quickly to avert a misuse.

Participant-based Key Management (security pattern)

Context: When managing a relatively small number of keys using low resources or in highly decentralized settings.

Problem: There is a need to manage keys coherently without any additional infrastructure.

Solution: Use an off-line method between users, such as telephone calls, exchanging information in person etc. to setup an initial symmetric or asymmetric key (see also PUBLIC KEY EXCHANGE in [45]). The manually generated key can then be used to set up a ONE-TO-ONE SECURE MESSAGE CHANNEL allowing for the secure exchange a SESSION KEY or other cryptographic material (cf. SESSION KEY EXCHANGE WITH PUBLIC KEYS [45]). In [19], such a key management approach is termed point-to-point distribution. In this solution the generation, storage and destruction of keys is reliant on each of the participants. The functionality realizing the latter operations can be implemented simply as a separate SECURE LAYER [9] to the rest of the system design.

Consequences:

(+ ) Does not require any additional, third-party infrastructure.

(-) Solution is not scalable.

4.1.3.2 Key management solution refinement abstraction level

Push security information (security pattern modifier)

Context: When wishing to realize the SERVER-BASED KEY MANAGEMENT or CERTIFICATE-BASED KEY MANAGEMENT patterns.

Problem: How should a client component interact with the key management infrastructure.

Solution: The client component (Component A) first contacts the relevant infrastructure components and retrieves the necessary security information (keys, tokens etc.) locally. The infrastructure components can perform any checks to ensure this is a valid, and that the client component can be trusted in some sense. The information can also be validated and/or vouched for as being authentic. Component A can then contact a target component (Component B) and pass the information for the required purpose. Component B can optionally check the validity of the information as needed.

Consequences:

(+ ) A client has already fulfilled everything necessary to obtain the appropriate security information, lessening the burden on the server of verifying it.

Pull security information (security pattern modifier)

Context and Problem: As for PUSH SECURITY INFORMATION.

Solution: The client component (Component A) first initiates an operation on a target component that involves the retrieval of security information, leaving the onus on that component to interact with the relevant security information infrastructure components and “pull” the information. As for the PUSH SECURITY INFORMATION PATTERN, appropriate checks and validation can be performed.
Consequences:
(+ ) A client does not need to verify any security information, since the onus is on the target component to verify it.
(+ ) The target component should not need to check the validity of the information itself, since the key management infrastructure is able to do this.

4.1.4 Practical implementation details level

4.1.4.1 Protocol selection abstraction level

At this level of abstraction the various operations of the KEY MANAGEMENT SCHEME – especially distribution, update and revocation – are realized using a set of corresponding protocols. There may be a unified set of protocols covering all operations, or it may be necessary to select different protocols for different operations in an ad-hoc fashion. In all cases the selection will be dictated by the choices taken for the architecture of the scheme, as well as whether it will be realized using push or pull semantics. Both the CENTRALIZED KEYING PROTOCOL and DECENTRALIZED KEYING PROTOCOL patterns can be applied multiple times for each KEY MANAGEMENT SCHEME operation as required.

This abstraction level also contains several adaptations of the patterns from the (revised, 2002) pattern language of [45], which highlight some example protocol types for the exchange of a SESSION KEY (for secure communication).

Centralized Keying Protocol (security pattern)

Context: When realizing a KEY MANAGEMENT SCHEME operation, especially in a CENTRALIZED KEY SERVER context.

Problem: To select an appropriate protocol for a given operation while relying on centralized components.

Solution: Use a protocol that makes use of TRUSTED THIRD-PARTY components. Both symmetric- and asymmetric-key algorithms can be used in the protocol's design. In the context of the key distribution operation, such protocols are usually termed key transport protocols, a number of which are documented in, for example, [19] or [22]. One prominent example is the Kerberos (modified Needham-Schroeder) protocol, which is based on the use of tickets in a CENTRALIZED KEY SERVER architecture modified by PULL SECURITY INFORMATION. For protocols implementing key generation, distribution and update in the context of group communication (e.g. using key trees), the reader can refer to [69].

Decentralized Keying Protocol (security pattern)

Context: When realizing a KEY MANAGEMENT SCHEME operation, especially in a PUBLIC KEY INFRASTRUCTURE or PARTICIPANT-BASED KEY MANAGEMENT context.

Problem: To select an appropriate protocol for a given operation without relying on centralized components.

Solution: Use a key protocol relying only on the participants, e.g. a Diffie-Hellmann based protocol (again, see [19]), to agree on keys. In the context of the key distribution operation, such protocols are usually termed key agreement protocols, and they can make use of both symmetric- and asymmetric-key algorithms. Since the information passed between the parties may be communicated over the network, it must either be protected using a ONE-TO-ONE SECURE MESSAGE CHANNEL, or some form of authentication (see Section 5) must be used to ensure that man-in-the-middle attacks are averted. The reader can refer, for example, to [70, 19, 22] for documented protocols. For group communication contexts, the reader can also refer to [69], which documents a number of decentralized protocols based on n-party extensions of the original Diffie-Hellmann protocol. Finally, [67] discuss a number of protocols applicable in decentralized peer-to-peer systems.

Session Key Exchange (security pattern)

Description: The establishment of a SESSION KEY, especially for use in the Secure communications solution frame, is one of the central tasks of key management. Therefore, choosing or designing an appropriate protocol for SESSION KEY management is of paramount importance. One particularly noteworthy approach is to use a CENTRALIZED KEYING PROTOCOL to establish an asymmetric key, and then a DECENTRALIZED KEYING PROTOCOL (e.g. some form of Diffie-Hellmann key exchange) for the communicating parties to agree on a SESSION KEY – see SESSION KEY EXCHANGE WITH SERVER-SIDE CERTIFICATES and SESSION KEY EXCHANGE WITH CLIENT- AND SERVER-SIDE CERTIFICATES [45], as well as the identically named (but less general) SESSION KEY EXCHANGE of De Souza and Matwin [48].
4.1.4.2 Technology realization level

The patterns described above can be realized using programmatic technologies such as Java KeyStore objects, or via complete COTS key management systems, such as IBM's Tivoli Key Lifecycle Manager (www.ibm.com/software/tivoli/products/key-lifecycle-mgr), Thales Key Management (http://www.thales-esecurity.com) and others. Technology descriptions at this level of abstraction would include the latter, as well as various libraries and/or stand-alone programs facilitating key generation, distribution etc., such as GnuPG (an OpenPGP implementation – www.openpgp.org), OpenPKI (www.openca.org) and others. A number of these descriptions – especially of general-purpose cryptographic libraries – would be common to both the Secure communications (Section 3) and Identity management (Section 5) solution frames.

5 Identity management solution frame

As a generalization, one can state that for the exception of systems specifically designed with privacy preservation (anonymity) in mind, all entities perform their actions with some form of identity linked to them. This can not only ensure that the actions are accountable, or at least traceable back to some system entity, but can also provide a basis for distinguishing entities from each other. The management of such identities constitutes the subject matter of the Identity management solution frame, which contains a pattern family concerned with (user/process) Authentication, and a pattern family concerned with identity federation (encompassing the patterns of [71]). In this section we present the Authentication pattern family, which is also the more important of the two.

The Identity management solution frame supports (solution relationship) the Secure communications frame from Section 5, and realizes the Authenticate users security tactic from [25].

5.1 Authentication pattern family

Authenticity of users and/or processes (sometimes called “entity authentication” or “peer authentication” [63]) is one of the most basic security properties in a distributed system (cf. [19, 72]). Unlike message source authenticity, the authentication of users or processes guarantees identity in “real-time”, i.e. that a user or process, whether local or remote, possesses the claimed identity at a given point in time, rather than at some time in the past.

Authentication is not only important in itself, but is also a prerequisite for the implementation of a number of other security features, such as secure communications and key management. With respect to the relations of the latter, if secure communications turns a cryptographic problem into one of key management (see Section 4), then key management often turns a key agreement/transport problem into one of verifying identities (see Section 4.1) – i.e. into an authentication problem – to ensure that the parties exchanging or agreeing upon keys are who they claim to be, and/or that any public keys do indeed belong to the relevant parties. Conversely, providing strong authentication guarantees (via cryptography) requires the management of cryptographic keys, which implies that the dependency between authentication and key management – or in other words, between the Authentication and Key management pattern families – is mutual.

Besides supporting key management and secure communications, authentication is often a prerequisite for allowing or denying entities to perform various actions – indeed, many authorization models and/or infrastructures rely explicitly or implicitly on a verified subject identity (see [73, 74]); and is also required for logging and auditing actions made by different users or processes [75].

The Authentication family presented in this sub-section organizes its constituent patterns across all the canonical abstraction levels, beginning with AUTHENTICATOR, which captures the main concepts; followed by three abstract architectures for realizing the AUTHENTICATOR pattern (at the application level, as a server, and as a third-party infrastructure); and finally – at the Solution Design level – patterns for different kinds of authentication strategies.

5.1.1 Conceptual analysis level

5.1.1.1 Authenticator (abstract security pattern)

Description: The most abstract pattern for authentication is the AUTHENTICATOR of Fernandez and Sinibaldi [76]. There are four (representative) participants in the pattern: a Subject, which must be authenticated; an abstract Authenticator component, which performs the authentication procedure; Authentication Information, which is checked when verifying the identity; and Proof of Identity, which is a software token of some type (e.g. a CREDENTIAL – see [77]), which can be presented later as proof of identity. The dynamics of the pattern can be easily discerned from the order of participants outlined above: a Subject contacts the Authenticator component(s), which subsequently establish and verify the Subject's identity and potentially generate(s) a software token, returned to the Subject, for later use.

From the above description it also becomes clear that the Authenticator component may be required to implement several abstract operations, namely: assign identity, establish identity and verify identity, where identity assignment in particular can also be realized in conjunction with IDENTITY FEDERATION [71].
5.1.2 Abstract architecture level

The patterns in this level of abstraction encapsulate different abstract architectures for realizing AUTHENTICATOR. They share the common Context (also as a section of the pattern template) of authenticating users or processes in a networked/distributed setting.

5.1.2.1 Application-driven Authentication (security pattern)

Reference pattern: DIRECT AUTHENTICATION [42, 78].

Problem: To provide application-level authentication for a distributed system.

Solution: Realize the AUTHENTICATOR inside the application logic by dividing the responsibilities or participants among dedicated components, or in each node/software unit. This is particularly useful for application using predictable client-server architectures. In conjunction with the MODEL-VIEW-CONTROLLER (MVC) pattern [11], the identity establishment functionality can be placed at the Views or user interfaces of the server (or application components playing the role of a server in this case), while the verification functionality resides in the Controller. A concrete realization of this pattern is the (distributed) AUTHENTICATOR of Brown et al. [79] (see [10] for a short summary).

Consequences:
(+ ) Applications have full control over the authentication process.
(- ) Applications must handle all authentication functionality, including the verification of identities.

5.1.2.2 Authentication Server (security pattern)

Reference pattern: BROKERED AUTHENTICATION [42], INDIRECT AUTHENTICATION [78]

Problem: To provide authentication for a whole network or distributed system.

Solution: Realize the core component of the AUTHENTICATOR as a centralized (potentially replicated) server, which is responsible assigning, establishing and verifying the identities of all users or processes in the system. Users or processes that must be authenticated interact with the server to acquire the necessary security information as proof of identity, or ask the server to act as an intermediary component during authentication.

One example of an AUTHENTICATION SERVER is the Kerberos system, the protocols for which are also used in Microsoft's Windows 2000 and Active Directory. Kerberos also provides key management functionality (see also CENTRALIZED KEY SERVER in Section 4.1.3.1).

Consequences:
(+ ) Only one component deals with authentication, reducing complexity.
(- ) The central server can become a point of failure, and a compromise can also affect all identity verifications.

5.1.2.3 Certification Infrastructure (security pattern)

Reference pattern: INDIRECT AUTHENTICATION [42], OFF-LINE AUTHENTICATION [78].

Problem: To provide distributed authentication easily across different security or administrative domains.

Solution: Realize the core component of the AUTHENTICATOR as part of an infrastructure external to the application capable of producing signed evidence of identities. In this case this usually implies a Public Key Infrastructure (PKI) – people, processes and a software infrastructure for verifying sensitive information, usually cryptographic keys [54] – but it can also be simply a trusted third party, including specialized system components or roles. In the case of a PKI, the sensitive information is signed (often off-line) and either encapsulated in, or accompanied by, digitally signed certificates. The digital signature is made using the private key of a CERTIFICATION AUTHORITY (CA) [45]. Since the key used by one CA may need to be signed as being authentic by another CA, possibly of a higher rank in some hierarchy, certificates may need to be chained. Verification of the certificates likewise proceeds by verifying the digital signature using the private keys of the CAs in the certificate chain (or single certificate).

Consequences:
(+ ) Distribution of authenticated material is simplified, as it is accompanied or encapsulated in tamper-protected, publicly available certificates and does not require explicit distribution.
(- ) Trust must be placed in a third party capable of vouching for a particular piece of information.

5.1.3 Solution design

The solution design level contains patterns encapsulating various “authentication strategies” for realizing the abstract architectures from the previous abstraction level. The patterns can be applied individually, or combined to make a hybrid strategy. Besides the patterns presented here, the PUSH SECURITY INFORMATION and PULL SECURITY INFORMATION modifiers from the Policy management pattern family in [12], summarized in Section 4.1.3.2 of this paper, are also notionally part of this abstraction level, and can be used to refine the AUTHENTICATION SERVER pattern.
5.1.3.1 Password-based authentication (security pattern)

**Reference patterns:** PASSWORD AUTHENTICATION [80].

**Context:** When wishing to authenticate users as opposed to processes or components.

**Problem:** To provide a basic means of authenticating users.

**Solution:** Users are prompted for a password (a pin or passphrase), which identifies them uniquely. Passwords should be stored in a secure fashion by salting and stretching them (see [65]), and hashing them with a keyed hash function (see [19]). Typically the hash functions used for the purposes of storage are slow, so as to impede any adversaries in brute-force guessing attacks. Similarly, the time interval between incorrect password re-attempts should be sufficiently long, or the number of re-attempts should be limited before some kind of reset, to prevent on-line guessing attacks. In the context of a distributed environment, all passwords in transit should be protected using the ONE-TO-ONE SECURE MESSAGE CHANNEL pattern (see Section 4). This pattern is most appropriate in the context of APPLICATION-DRIVEN AUTHENTICATION or centralized AUTHENTICATION SERVER architectures.

**Variants:** Biometric authentication is possible in addition to, or as a replacement of, the use of passwords, however, biometric security is most suitable on a local scale [78].

**See also:** ACCOUNT LOCKOUT and PASSWORD PROPAGATION [80] for password usage and management strategies, respectively.

**Consequences:**

(+): The password verification process is relatively simple.
(-): Passwords require secure storage, and appropriate access controls.
(-): Users are not good at picking secure passwords resistant to dictionary and other brute-force attacks.

5.1.3.2 Address-based authentication (security pattern)

**Context:** Distributed systems in which source addresses are fixed, or systems spanning large networks in which it is difficult to manipulate routing tables.

**Problem:** To authenticate users or processes based on some software or hardware based characteristic, as opposed to something known or possessed.

**Solution:** Verify identity based on the source address of requests or messages in a communication stream. The most common scenario for this is the use of IP addresses on the Internet to authenticate users as belonging to different domains, institutions etc. IP address ranges are allocated by an external organization, which ensures a relative longevity of addresses. Moreover, spoofing an IP address will route traffic away from the sources machine, and is hence not a significant threat (unless the machine with a valid IP has been compromised). Address based authentication requires that the source address of each message or request is checked individually, or, if better performance is required at the expense of immediacy, per communication session.

This pattern is not a very secure authentication strategy in LAN environments, since it is possible for the routing tables of network elements to be manipulated in such a way that an attacker who has compromised one or more LAN machines can receive messages from a (valid) spoofed source address of another LAN machine. The pattern is also not appropriate when source routing algorithms (return communication path follows initial requested node order) are employed.

**Consequences:**

(+): Verification is relatively simple
(-): Some overhead can be generated from the message/request checking process
(-): The pattern is not appropriate for local networked environments

5.1.3.3 Certificate-based authentication (security pattern)

**Context:** Any networked or distributed system.

**Problem:** To authenticate users or processes without any explicit interaction.

**Solution:** Using a THIRD-PARTY INFRASTRUCTURE, system entities can have their keys or other information stored in certificates, which are digitally signed by a CERTIFICATION AUTHORITY [45] – a trusted third party. Such a certificate thus becomes a CREDENTIAL [77], which can be presented when sending messages or making requests. Certificates can also be seen as tickets in the context of an AUTHENTICATION SERVER architecture, which can be obtained by parties from the central server component(s) as proofs of identity that, in turn, can be presented to other parties when performing some operation.

Certificates must be stored securely, either a dedicated server (e.g. AUTHENTICATION SERVER), using specific certificate/credential management infrastructure (e.g. PERMIS, see also the Policy management pattern family in [12]), or at the node of the particular user/process. An example system using certificates for authentication on LANs is SESAME [81].

**Consequences:**

(+): Certificates offer unforgeable proof of identity, and can also be publicly available.
(-): Requires additional infrastructure for generating and storing certificates.
5.1.3.4 Cryptographic authentication (security pattern)

**Context:** Any networked or distributed system.

**Problem:** To verify the identity of users or processes without explicitly requiring user interaction, while providing strong verification guarantees.

**Solution:** Use a challenge-response protocol, in which one party sends a challenge (e.g. a nonce) to the other party, such that the reply contains unforgeable proof of identity (e.g. the possession of a shared symmetric key). There are a number of challenge-response protocols to choose from (see, for example, [82] and [70]), loosely falling into the following classes:

- **symmetric key:** either using encryption algorithms or keyed hash functions.
- **asymmetric key:** either using encryption, or digital signatures with unkeyed hash functions (see DIGITAL SIGNATURES WITH HASHING in [55]).
- **zero knowledge (ZK):** iterative challenge-response protocols, which allow parties to ascertain identity without revealing a given secret (e.g. secret key). Such protocols can also be based on asymmetric ciphers.

The choice of patterns in higher levels of abstraction in the pattern family will also dictate the choice or design of the authentication protocols, e.g. an AUTHENTICATION SERVER solution will most likely use a server-based symmetric cipher protocol as opposed to a protocol using asymmetric ciphers. In conjunction with PASSWORD-BASED AUTHENTICATION, protocols can also be used when user interactivity is required. One such example is the RADIUS protocol, encapsulated in the REMOTE AUTHENTICATOR/AUTHORIZER pattern [83]. Another example is the Digest Access Authentication protocol for use in HTTP connections (see RFC 2617). A number of (older) dedicated systems for network authentication can also be referred to for concrete implementations of protocols – see, for example, [84] for NetSP, SPX and TESS.

Cryptographic authentication usually implies the establishment of a cryptographic session key in addition to the verification of identity, and is thus also related to the CENTRALIZED KEYING PROTOCOL and DECENTRALIZED KEYING PROTOCOL patterns in Section 4.1.4.1, with respect to the key distribution operation.

**Consequences:**

- (+) Implementations provide authentication with strong guarantees.
- (-) Protocols usually require exchange of cryptographic material, and hence rely on key management, which in turn may rely on other authentication approaches.

5.1.3.5 Mutual authentication (security pattern modifier)

**Context:** Any distributed system where two mutually interacting parties need to be sure of each other's identity.

**Problem:** In a two-way interaction it is sometimes necessary to authenticate not only a client, user or process making some request, but also peer processes/components or servers that handle that request. This ensures that both parties know who they are interacting with in real-time.

**Solution:** Two entities can authenticate mutually by two separate applications of the AUTHENTICATOR pattern, supported by a CERTIFICATION INFRASTRUCTURE as in the MUTUAL AUTHENTICATION pattern of [85]; or in conjunction with either AUTHENTICATION SERVER or APPLICATION-BASED AUTHENTICATION. In this case relevant components, such as the central Authenticator, can be merged as required into a single centralized component. Depending on the chosen authentication strategy pattern, the process for verification may include checking passwords, or, more commonly, a cryptographic challenge-response protocol [82], initiated either by a server or by the other party.

5.1.3.6 Single Sign-On Session (security pattern)

**Description:** Keep track of identities in a time-limited SESSION, so that a user or process does not need to repeatedly authenticate when performing some action. The SESSION pattern can be realized notionally via the use of CREDENTIAL DELEGATION [86], where time-limited proxy certificates are created by an initiating party, signed by that party, and subsequently delegated to other parties for authentication – see the SINGLE SIGN-ON pattern of Lu and Weiss [85]. The “session” in this case is delimited by the expiry of the certificate. [87] outline other approaches, including that used in Kerberos.

5.1.4 Practical implementation details

5.1.4.1 Technology realization level

As with the Secure communications frame, we omit the details of the technical realization level, which would imply describing various libraries, software packages and/or language capabilities, as well as technology-implementation-specific patterns, that would help realize this frame's encapsulated patterns. Some examples of libraries and/or technologies include the Microsoft .NET Windows-based authentication framework and the Java JAAS framework,
as well as the OpenCA framework (www.openca.org). Examples of technology-/implementation-specific patterns include Microsoft's BROKERED AUTHENTICATION: KERBEROS, BROKERED AUTHENTICATION: X.509 PKI and BROKERED AUTHENTICATION: SECURITY TOKEN SERVICE for Web-services in the context of Microsoft's WSE 3.0 [42]; as well as AUTHENTICATED SESSION (for realizing SINGLE-SIGN ON SESSION), PASSWORD AUTHENTICATION and PASSWORD PROPAGATION when taken in their original Web-application-specific context [80].

6 Analysis examples

In this section we illustrate the application of our cryptography-based solution frames in analyzing two existing security architectures: IBM's Lotus Domino/Notes suite (analyzed briefly); and Bell Labs' Plan 9 distributed operating system (analyzed in detail). In both cases our analysis proceeds by identifying which patterns were instantiated from the relevant solution frames and how they were customized, i.e. what design decisions were taken in the construction of the latter architectures. Analyzing the IBM Lotus and Plan 9 security architectures briefly and in detail, respectively, shows how the solution frames can be used to capture relevant (security) design decisions at different levels of granularity and abstraction – providing either an overview or more in-depth description – depending on available documentation and/or purpose of analysis.

A process entirely analogous to the one manifested in the analysis examples would be followed when constructing a new security architecture from scratch, with the exception that one would not analyze existing design decisions but make and realize those decisions by selecting, combining, customizing and instantiating the patterns. In light of this, the brief/detailed analyses in this section indicate how the solution frames can be used analogously to guide design decisions both at a high level and in greater detail during architectural design.

6.1 Brief analysis: IBM Lotus Domino/Notes collaborative system

IBM's Lotus Notes/Domino suite is a client-server software system for intra-organizational collaboration supporting a number of capabilities including e-mail, instant messaging, discussion boards and calendar management. The security features [88, 89] have evolved along with the development of the system, and new features added in each release. In what follows we consider only a subset of relevant security features of IBM Lotus (currently known simply as IBM Domino/Notes) common to releases 5 and 6.

6.1.1 Identity Management (Authentication)

IBM Lotus realizes the AUTHENTICATOR pattern via a CERTIFICATION INFRASTRUCTURE, both for the Notes client as well as for web-based access. PUBLIC-KEY CERTIFICATES are bundled in Notes ID's, which are subsequently used in both native and web-based CERTIFICATE-BASED AUTHENTICATION solution design strategies to support MUTUAL AUTHENTICATION (with both Domino server and Notes client passing each other certificates in a given session/interaction). CRYPTOGRAPHIC AUTHENTICATION (e.g. utilizing Windows NT challenge/response protocols) can also be used for different applications, depending on the requirements. Since Domino server is also an application server, authentication strategies customizing any of the patterns in Section 5.1.3 can be used in conjunction with SECURE PROTOCOL DESIGN. The Domino/Notes suite also supports SINGLE SIGN-ON SESSION with appropriately an set-up Domino Directory.

6.1.2 Security Information Management (Key Management)

IBM Lotus realizes the Key Management Scheme abstract pattern via a DECENTRALIZED KEY MANAGEMENT ARCHITECTURE, which in turn is realized as a PUBLIC KEY INFRASTRUCTURE. KEY CERTIFICATES are managed administratively and using a directory structure (a customized instance of DECENTRALIZED KEYING PROTOCOL).

6.1.3 Secure communications

Specific network ports on both Domino and Notes can be setup to enforce a SECURE MESSAGE CHANNEL with MESSAGE ENCRYPTION, MESSAGE TAMPERING PROTECTION, MESSAGE SOURCE AUTHENTICATION and MESSAGE REPLAY PROTECTION semantics. Secure Communication Sessions are established according to the SSL protocol, which uses X.509 KEY CERTIFICATES distributed as described above. E-mail communications, thought of as a message channel, can also be secured using S/MIME (a design which is an instance of SECURE MESSAGE CHANNEL with INFRASTRUCTURE-DRIVEN MESSAGE TRANSFORM semantics). E-mail signing employs an instance of the SECURE MESSAGE CHANNEL with MESSAGE TAMPERING PROTECTION and MESSAGE SOURCE AUTHENTICATION semantics (using digital signatures). The secure communication strategies are setup and used in conjunction with NETWORK INFRASTRUCTURE CONFIGURATION and hardening of the underlying OS on which the Domino/Notes server and clients are deployed (a description of which is beyond the scope of this paper).
6.2 Detailed analysis: Plan9 distributed operating system

Plan9 from Bell Labs [90, 91] is a distributed operating system inspired by the UNIX operating system family. It takes over a number of distinguishing UNIX features, the most important being the ubiquitous use of file descriptors for accessing resources and interacting with processes. Security is incorporated as an integral part of the system's design, with measures including private process namespaces [92], strong authentication, and protected inter-process communication. As of 2002, of central importance in Plan9's security architecture is the use of factotum components [93], which are effectively pieces of a single distributed process acting as logical PROXY instances (see [11]) on behalf of ordinary users/processes for various security functions. This design ensures that all cryptographic functionality is kept outside applications in an easily verifiable unit, and, if the latter functionality is changed, obviates the need for re-linking factotum-dependent binaries.

In what follows we analyze in some detail how each of the abstraction levels of the solution frames in this paper have been realized in Plan9's security architecture. The discussion attempts to adhere to the solution frame structures closely, however, since the relevant security design decisions are sometimes mixed, we have aimed for coherency over precision.

6.2.1 Identity Management (Authentication)

Conceptual analysis and Abstract architecture: In Plan9, the conceptual AUTHENTICATOR pattern is instantiated for any services or applications requiring identity checking, with the file service that connects resources and processes being the most significant example. Depending on the service type, the AUTHENTICATOR pattern is realized via either an APPLICATION-DRIVEN AUTHENTICATION architecture (standard applications and some special services), or an AUTHENTICATION SERVER (Plan 9 native authentication).

Solution design: The two architectures utilize two main authentication strategy patterns: PASSWORD-BASED AUTHENTICATION and symmetric-key CRYPTOGRAPHIC AUTHENTICATION. The latter two are used in combination (cf. Section 5.1.3.4) at process startup, when the local factotum component employs the PAK protocol to load cryptographic keys from a centralized storage location (if the user has a relevant account), while also realizing SINGLE SIGN-ON SESSION. The AUTHENTICATION SERVER uses pure CRYPTOGRAPHIC AUTHENTICATION with a custom ticket-based authentication protocol (p9sk1 – resulting from the application of SECURE PROTOCOL DESIGN) akin to Kerberos. Tickets can be used as CAPABILITIES [64, 12], which could provide some features of a SINGLE SIGN-ON SESSION.

The factotum components acts as (logical proxy) parties in all authentication protocols, whereby servers requiring the establishment and verification of user/process identity can, via their own factotum components, communicate with corresponding client factotums.

Practical implementation details: A STRATEGY pattern [52] is used by factotum components in the form of a meta-protocol (p9any) to multiplex the different protocols used for CRYPTOGRAPHIC AUTHENTICATION. The authentication protocols are implemented (only) within factotum components, with modules for supporting: "the Plan9 shared key protocol (p9sk1), SSH's RSA authentication, passwords in the clear, APOP, CRAM, PPP's CHAP, Microsoft's PPP's MSCHAP, and VNC's challenge/response" [93].

6.2.2 Security Information Management (Key Management)

Conceptual analysis and Abstract architecture: In Plan9, the KEY MANAGEMENT SCHEME is logically spread across the separate factotum components and the AUTHENTICATION SERVER instance. In realizing the KEY MANAGEMENT SCHEME operations, the overall architecture conforms to a hybrid customized CENTRALIZED KEY MANAGEMENT ARCHITECTURE, with the (notional) server being distributed across (1) the factotum components – together with a centralized secstore subsystem, which serves the purpose of storing long-term keys securely for easy retrieval – in the manner of a DECENTRALIZED KEY MANAGEMENT ARCHITECTURE, and (2) the (truly) centralized AUTHENTICATION SERVER. All keys are stored and used as KEY WITH ATTRIBUTE instances, with attribute pertaining – among others – to corresponding protocol(s) and user(s).

Solution design: The solution design for realizing the key management architecture similarly takes a hybrid approach in employing a CENTRALIZED KEY SERVER pattern using PUSH SECURITY INFORMATION SEMANTICS (doubling its role as AUTHENTICATION SERVER – see Section 6.2.1), which stores SESSION KEY instances; and a distributed PARTICIPANT-BASED KEY MANAGEMENT pattern, with the factotum playing the dual role of a participant on behalf of each user/process, on one hand; and, on the other hand, a (locally) CENTRALIZED KEY SERVER linked with the (truly centralized) secstore (for key storage). This hybrid approach shows that the patterns need not be instantiated in isolation, but can interact in complementary ways.
**Practical implementation details:** Long-term keys are either manually entered / installed by Plan9 users, or retrieved by factotum components from the secstore (see the discussion in Section 6.2.1 on Plan 9 authentication); once loaded into the factotum, they are stored in volatile memory and destroyed when the corresponding process is destroyed. SESSION KEY instances are obtained from the AUTHENTICATION SERVER / CENTRALIZED KEY SERVER instance via a CENTRALIZED KEYING PROTOCOL (in fact, as a side effect of MUTUAL AUTHENTICATION – see the previous section on authentication).

The mechanism for using keys is initiated by client or server applications when – as a result of requiring the use of some cryptographic function – the applications communicate with their local factotums via RPC (Remote Procedure Call, implemented as reads/writes on the factotum file interface), which in turn pass queries for keys and operate with them in protocol runs. The applications simply relay the (security-related) messages to each other, with the actual key-management-related functions and negotiations performed by the respective factotum components, as befits their logical PROXY nature.

### 6.2.3 Secure communications

**Conceptual analysis and Abstract architecture:** Plan 9 uses the SECURE MESSAGE CHANNEL with the three basic modifiers – MESSAGE ENCRYPTION, MESSAGE TAMPERING PROTECTION and MESSAGE SOURCE AUTHENTICATION – to comprehensively protect all communications between system processes.

**Solution design:** The abstract architecture is realized directly according to the TLS protocol [57], by setting up a SECURE COMMUNICATION SESSION with a either a public-key based handshake, or using a pre-shared SESSION KEY (see the discussion on key management above). The communication semantics used conform to the APPLICATION-DRIVEN MESSAGE TRANSFORM pattern, whereby applications are required to call a specific function (pushtls) explicitly for protected message streams.

**Practical implementation details:** Regarding key management: the authentication protocols used by native Plan 9 network services generate a shared SESSION KEY as a side-effect (see the discussion on Plan 9 key management above), which is subsequently used in the APPLICATION-DRIVEN MESSAGE TRANSFORM instance (i.e. as a parameter to pushtls). At the application level, TLS uses a public-key protocol for establishing a SESSION KEY (session establishment step of SECURE COMMUNICATION SESSION), hence requiring applications to acquire public keys. The latter are not managed using a PUBLIC KEY INFRASTRUCTURE, however, but rather, are returned by servers/targets in the form of unverified X.509 certificates used directly as keys (i.e. a DECENTRALIZED KEY MANAGEMENT ARCHITECTURE realized as a PARTICIPANT-BASED KEY MANAGEMENT design) in the TLS handshake.

### 7 Related work

Security solution frames are a relatively recent phenomenon, building on a number of other solution types, both for security and, more generally, for organizing software patterns. The reader is referred to [12] for an in-depth discussion of these.

With respect to (security) patterns for the security concerns addressed in this paper, most, if not all, relevant patterns have been included as part of the Reference patterns sections in the descriptions throughout the paper. Insofar that solution frames can be considered organizational constructs for security patterns, this paper represents, as far as we are aware, the first effort to comprehensively document patterns for secure communications, key management and identity management (more specifically, for authentication), consolidating the previous work of [43], [45] and others (as referenced throughout) in the process. A number of shortcomings in the existing patterns have been rectified where possible, both explicitly – by providing an overall descriptive framework within which the patterns can be used; and implicitly – by providing more accurate descriptions (e.g. the MESSAGE INTEGRITY pattern in the work of Braga et al. [43] promotes the use of an MDC without encryption, which is insufficient to provide integrity – something that is corrected in our MESSAGE TAMPERING PROTECTION pattern). The use of modifiers to determine the protection semantics of a SECURE MESSAGE CHANNEL is an alternative approach to the multiple patterns proposed in [43] for essentially the same purpose.

With respect to related pattern-like security solutions, the parameterized model templates of Rossebo and Bræk [94, 95] for authentication represent a valuable complementary approach. The latter models are organized hierarchically according to the mutuality, number of passes and cryptosystem used for strong authentication, and can be utilized to guide the realization of the different protocols types outlined in our CRYPTOGRAPHIC AUTHENTICATION pattern (see Section 5.1.3.4). In a similar vein, the Generic Security Components and Architectures (GSCs and GSAs, respectively) of Schmidt [26] (e.g. SymmetricEncryptorDecryptor or KeyedHashProcessing) can be used as supplementary designs to guide developers in realizing some of the cryptographic functionality supporting our Secure communication frame (see Section 3).
8 Conclusion and future work

In this paper we presented security solution frames and relevant encapsulated patterns for secure two-party (network) communications; for user/process authentication; and for cryptographic key management, thereby filling a conspicuous gap in the secure software engineering literature. We showed how these solution frames can be used in practice by analyzing the security architectures of the Plan 9 distributed operating system (in detail) and IBM's Lotus Domino/Notes (in brief). The examples also indicated the analogous process – implicit in the solution frame structures, and explicit as a process pattern in each frame's process pattern family – that would be followed when applying the solution frames and/or their patterns to design new security architectures from scratch. The patterns constituting the solution frames can also be used in conjunction with other security patterns and security solution frames (see, e.g., [8] and [12]) to create and/or analyze security architectures for a variety of distributed system types. From a methodological viewpoint, the solution frames in this paper can be seen as generic conceptual artefacts that can be used across a variety of model-based security methodologies (systematic approaches to introducing security during development based on abstract models of some kind) [33].

In our approach to constructing the solution frames, we used implicit knowledge (openly available in the literature) of cryptographic primitives as a basis. Distilling this knowledge in “developer-friendly” patterns, as well as technology descriptions in the spirit of [30], would consolidate the solution frames by providing additional guidance in realizing the more technical aspects of different pattern solutions (e.g. which digital signature scheme to use for MESSAGE SOURCE AUTHENTICATION; what, if any, weak keys different symmetric algorithms have etc.). In this respect the recent work of De Muijnck-Hughes and Duncan [96], which aims to construct a security pattern language for predicate based encryption (PBE) cryptosystems, could be an important supplement to our solution frames (once the latter pattern language is constructed, that is). Following a similar direction by constructing uniform descriptions of cryptographic algorithms in widespread use – such as 3DES-CBC for encryption, HMAC-SHA256 for message authenticity and others – would also allow developers to make informed decisions regarding algorithm suitability more easily.

Continuing with the general direction in this paper of distilling security knowledge in patterns and solution frames implies the construction of new solution frames for other network/distributed system security concerns, such as secure data storage, execution control and secure group communication. Combining such frames into larger structures would lead to comprehensive security solution clusters for general networked and/or distributed systems, or even for different distributed system types. Of especial value would be frames for realizing the mitigating security policies encapsulated in the threat patterns of [41], which would equip developers with a wide range of both offensive and defensive security artefacts.

References

41. Uzunov AV, Fernandez EB. An Extensible Pattern-based Library and Taxonomy of Security Threats for Distributed Systems. Accepted for Computer Standards & Interfaces (Special Issue on Information Security); to appear.

57. kumar a, fernandez eb. a security pattern for the transport layer security (tls) protocol. procs. of the 19th conference on pattern languages of programs (plop), tucson, az: 2012.

58. smith s, beaulieu a, phillips wg. modeling and verifying security protocols using uml 2. procs. of the 2011 ieee international systems conference (syscon), montreal, qc: ieee; 2011, p. 72–9.


60. muftic s, sloman m. security architecture for distributed systems. computer communications 1994; 17: 492–500.


63. muftic s, patel a, sanders p, colon r, heijndjik j, pulkinnen u. security architecture for open distributed systems. 1st ed. john wiley & sons; 1993.

64. delessy n, fernandez eb, larrondo-petrie mm. a pattern language for identity management. procs. of the 2nd ieee int. multiconference on computing in the global information technology (iccg), beijing, china: ieee; 2007, p. 1–6.

65. gasser m. building a secure computer system. van nostrand reinhold; 1988.

66. lalanda p. shared repository pattern. proceedings of the 5th pattern language of programs (plop), 1998.

67. van der merwe j, dawoud d, mcDonald s. a survey on peer-to-peer key management for mobile ad hoc networks. acm computing surveys 2007; 39: 1.

68. stinson dr. an explication of secret sharing schemes. designs, codes and cryptography 1992; 2: 357–90.

69. zou x, ramamurthy b, magliveras ss. secure group communications over data networks. 1st ed. springer; 2010.

70. boyd c, mathuria a. protocols for authentication and key establishment. 1st ed. springer; 2003.

71. delessy n, fernandez eb, larrondo-petrie mm, wu j. patterns for access control in distributed systems. proceedings of the 14th pattern languages of programs conference (plop), monticello, illinois, usa: 2007.


73. de capitani di vimercati s, paraboschi s, samarati p. access control: principles and solutions. software: practice and experience 2003; 33: 397–421.


75. linn j. practical authentication for distributed computing. proceedings of the 1990 ieee computer society symposium on research in security and privacy, ieee; 1990, p. 31–40.

76. fernandez eB, sinibaldi jc. more patterns for operating systems access control. proceedings of the european conference on patterns language of programming (europlop'03), 2003.

77. morrison p, fernandez eb. the credentials pattern. proceedings of the 2006 conference on pattern languages of programs, portland, or: 2006, p. 1–4


79. brown fl, divietri j, de villegas gd, fernandez eb. the authenticator pattern. procs. of plop'99, 1999.


81. ashley pm, vandenwauwer m. practical intranet security: overview of the state of the art and available technologies. 1st ed. kluwer academic; 1999.

82. clark j, jacob j. a survey of authentication protocol literature: version 1.0. 1997.

83. fernandez eb, warrier r. remote authenticator/authorizer. proceedings of the 14th conference on pattern languages of programs (plop 2003), 2003.

84. oppliger r. authentication systems for secure networks. artech house publishers; 1996.

85. lu m, weiss m. patterns for grid security. mini-plop at conference on object-oriented programming systems, languages, and applications (oopsla), 2008.

86. weiss m. credential delegation: towards grid security patterns. 5th nordic pattern languages of programs conference (vikingplop), 2006.

87. pashalidis a, mitchell cj. a taxonomy of single sign-on systems. proceedings of the 8th australasian conference on information security and privacy, perth, western australia: springer-verlag; 2007, p. 249–64.

88. collins f, nielsen sp, dahn f, lüscher m, yamamoto h, denholm b, et al. lotus notes and domino r5.0 security infrastructure revealed. ibm; 1999.

89. tworek w, chiesa g, dahn f, hinkle d, mason a, milza m, et al. lotus security handbook. ibm; 2004.

90. pike r, presotto d, dorward b, flandrena k, thompson k, trickey h, winterbottom p. plan 9 from bell labs. computing systems 1995; 8: 221–54.


92. pike r, presotto d, thompson k, trickey h, winterbottom p. the use of name spaces in plan 9. procs. 5th acm sigops european workshop on models and paradigms for distributed systems structuring, acm; 1992, p. 1–5.

93. cox r, grosse e, pike r, presotto dl, quinlan s. security in plan 9. procs. 11th usenix security symposium, berkeley, ca: usenix association; 2002, p. 3–16.


96. de muijnck-hughes j, duncan i. thinking towards a pattern language for predicate based encryption crypto-systems. procs. 6th ieee int. conference on software security and reliability companion (serc-c), gaithersburg, md: 2012, p. 27–32.
Epilogue

In this chapter we continued and completed (with respect to this thesis) our presentation of the solution frame artefacts – the last of our generic conceptual security artefacts, which we begun presenting in Chapter 4. In the next chapter, we return to our approach to engineering security methodologies (from Chapter 3), and illustrate how some of these generic artefacts – in particular, solution frames – can be introduced into one or more specific security methodologies. The actual use of solution frames was already demonstrated in this chapter and the last, and will be further summarized in Chapter 9.
Chapter 8
(Engineering meta-methodology II)
A framework and process for security methodology quality assessment and improvement

Quality is never an accident; it is always the result of intelligent effort.
– John Ruskin

Prologue

Determining or quantifying the quality of software products is an important software engineering activity. Similarly, determining the quality of a security methodology can be of great importance – not only when faced with adoption – but, to a special measure, in the context of an approach to engineering security methodologies, where one would like to be able to know whether the methodology produced by applying that approach has certain desirable characteristics. Currently there is nothing in the literature concerned with security methodology quality as such. Filling this gap and simultaneously furthering our work begun in Chapter 3, in this chapter we present the final part of our engineering “toolkit”: a comprehensive framework and simple process for assessing and improving the quality of security methodologies. From one point of view, the quality process (in its assessment capacity) is an amplification and detailed specification of the Methodology Verification stage of S-SMEP; from a different point of view, the quality process and framework essentially make up another meta-methodology for security methodology quality control, complementing S-SMEP. From both viewpoints, utilizing the quality framework and improvement/assessment process can help to ensure that a given security methodology engineering approach – and S-SMEP in particular – does not merely provide a means of producing ad-hoc methodologies with random qualities, but a means to systematically produce situationally-suitable methodologies of measurably good quality.

Thus, the quality framework and process presented in this chapter extends – and in many ways completes – the engineering approach from Chapter 3, which is the core of the thesis; and also evaluates the latter, first through providing a means for the approach to inherently evaluate itself, and secondly through illustrating its value and use – both with respect to the generic security artefacts from Chapters 4 to 7 and how they can be used across different methodologies, as well as how S-SMEP can be applied to re-engineer methodologies other than the case study methodology. In this sense, the present chapter forms a final sequel to Chapter 3, and hence to the presentation of our engineering “toolkit” as a whole.
## Statement of Authorship

<table>
<thead>
<tr>
<th>Title of Paper</th>
<th>Assessing and Improving the Quality of Security Methodologies for Distributed Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication Status</td>
<td>□ Published, □ Accepted for Publication, □ Submitted for Publication, □ Publication style</td>
</tr>
</tbody>
</table>

### Author Contributions

By signing the Statement of Authorship, each author certifies that their stated contribution to the publication is accurate and that permission is granted for the publication to be included in the candidate's thesis.

<table>
<thead>
<tr>
<th>Name of Principal Author (Candidate)</th>
<th>Anton V. Uzunov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Conceptualization of work, its realization (research) and documentation (wrote manuscript). Will act as corresponding author.</td>
</tr>
<tr>
<td>Signature</td>
<td>[Signature]</td>
</tr>
<tr>
<td>Date</td>
<td>10/01/2013</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th>K. Falkiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Supervised development of work, helped evaluate and edit the manuscript.</td>
</tr>
<tr>
<td>Signature</td>
<td>[Signature]</td>
</tr>
<tr>
<td>Date</td>
<td>11/10/2013</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th>E.B. Fernandez</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Provided ideas and commented on manuscript.</td>
</tr>
<tr>
<td>Signature</td>
<td>[Signature]</td>
</tr>
<tr>
<td>Date</td>
<td>13/10/2013</td>
</tr>
</tbody>
</table>

| Name of Co-Author | |
|-------------------| |
| Contribution to the Paper | |
| Signature | [Signature] |
| Date | |
Assessing and Improving the Quality of Security Methodologies for Distributed Systems

Anton V. Uzunov*, Katrina Falknera, Eduardo B. Fernandezb

a, School of Computer Science, The University of Adelaide, Adelaide, South Australia, Australia, 5005
b, Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, 777 Glades Rd., Boca Raton, FL 33431

* Corresponding author

Abstract: Security is increasingly becoming one of the most important quality attributes when engineering new software systems, especially complex, distributed systems. It is generally agreed in the secure software engineering community that the introduction of security should proceed by means of a structured, systematic approach. We term the aforementioned systematic approaches, particularly those implying some sort of process aligned with the software development life-cycle, security methodologies. While isolated attempts have been made to demonstrate the value of particular security methodologies, the “quality” of security methodologies, as such, and in general, has never been given due consideration; indeed, it has never been studied as a self-standing topic. The literature therefore entirely lacks supportive artefacts – whether models, frameworks or guidelines – that can provide a basis for assessing, and hence for improving, the quality of security methodologies. The ability to assess and improve security methodology quality is particularly important in the context of methodologies which are constructed “on the fly” to take into account various system specifics and, more generally, for particular project situations. In this paper we fill the aforementioned gap by proposing a comprehensive quality framework and accompanying process, within the context of an existing approach to engineering security methodologies, which can be used for both (bottom-up) quality assessment and (top-down) quality improvement. The main framework elements can be extended and customized to allow an essentially arbitrary range of methodology features to be considered, thus forming a basis for flexible, fine-grained quality control. We demonstrate the bottom-up application of the latter framework and process on three real-life security methodologies for distributed systems, taken as case studies. Based on the assessment results indicating deficiencies, we subsequently show in detail (for one) and briefly discuss (for the remaining set) how the case study methodologies can be re-engineered – incorporating apposite process fragments and subsets of a collection of generic conceptual security artefacts – to improve their quality.

Keywords: security methodology quality, quality framework, quality assessment, quality improvement, quality control, engineering security methodologies

1 Introduction

Security is increasingly becoming one of the most important quality attributes when engineering new software systems, especially complex, distributed systems [Siakas and Georgiadou, 2005]. It is generally agreed in the secure software engineering community that the introduction of security should proceed by means of a structured, systematic approach [Fernandez, 2013, Hafner and Breu, 2009, Howard and Lipner, 2006, Mouratidis et al., 2005, Mouratidis and Giorgini, 2006, Rosado et al., 2008, Schmidt, 2010, Schumacher, 2003, Traore and Woungang, 2013]. We term the aforementioned systematic approaches, particularly those which imply some sort of process aligned with the software development life-cycle, secure software engineering methodologies, or, more simply, security methodologies [Uzunov et al., 2012b].

While the application of a security methodology over the unprincipled, ad-hoc introduction of security is consistently preferable, it is not always clear exactly how “high quality” a given methodology is. In a manner completely analogous to software products – the “inputs” of both development and security methodologies – assessing the quality of security methodologies can help to determine strengths and weaknesses, and hence whether a given methodology meets certain needs; as well as whether certain improvements are needed in order for the methodology to acquire certain desirable characteristics. Quality assessment can also help to differentiate methodologies – and/or to determine suitability for a given project situation – and hence to guide adoption decisions. This is especially important in the context of methodologies constructed “on the fly”, where quality assessment can determine not only suitability for adoption, but also whether the construction or engineering process produced a methodology with (the intended) beneficial characteristics (cf. [McBride and Henderson-Sellers, 2011]).

Despite the value of being able to assess and hence subsequently improve the quality of security methodologies, the literature entirely lacks any supportive artefacts – whether models, frameworks or guidelines – that can provide a basis for the latter tasks. Considerations of (security) methodology quality – and at that, not under the designated term or using the relevant quality-related concepts – can only be found tangentially in the context of literature surveys on security methodologies (e.g. [Jayaram and Mathur, 2005, Uzunov et al., 2012a, Uzunov et al., 2012b]) and a small portion of work concerned with security metrics [Jaatun, 2012, Khan and Zulkernine, 2008], which, in
both cases, do not offer independent or self-contained treatments of the subject. It is not surprising, therefore, that, besides supportive artefacts, the literature also entirely lacks concrete, specifically designed approaches for security methodology assessment and improvement.

In this paper we aim to fill the latter gaps by first defining security methodology quality as a concept and subsequently proposing a comprehensive, extensible quality framework and accompanying process, within the context of a previously proposed approach to engineering security methodologies [Uzunov et al., sub2], that can be used for both (bottom-up) quality assessment – \textit{a posteriori} methodology design – and (top-down) quality improvement – determining requirements and guiding methodology design decisions – in the spirit of software quality engineering (see [Côté et al., 2007; Tian, 2005]). With the use of profiles, the main framework elements can be augmented and customized to allow an essentially arbitrary range of methodology features to be considered for a given project situation, thus forming a basis for flexible, fine-grained quality control.

From the point of view of the aforementioned approach to engineering security methodologies, the proposed quality process (in its assessment capacity) is an amplification and detailed specification of the \textit{Methodology Verification} stage of S-SMEP (Situational Security Methodology Engineering Process) – a stage which received relatively cursory attention in [Uzunov et al., sub2]. Perhaps more importantly, the quality framework and accompanying process presented in this paper can be considered as constituting a stand-alone meta-methodology for security methodology quality control, complementing S-SMEP and indeed any other related meta-methodology or, more generally, any other approach for engineering security methodologies. From both viewpoints, utilizing the quality framework and improvement/assessment process can help to ensure that a given (security) methodology engineering approach does not merely provide a means of producing ad-hoc methodologies with random qualities that “more or less fit the purpose”, but a means to systematically produce sensible methodologies – i.e. methodologies of measurably good quality – suited precisely for the project situation at hand.

We demonstrate the bottom-up application of the latter framework and process on three real-life security methodologies for distributed systems, taken from the literature as case studies. Based on the assessment results indicating deficiencies, we subsequently take one of these case study methodologies and show in detail how it can be re-engineered – incorporating apposite process fragments and subsets of a collection of generic conceptual security artefacts – to improve its quality; and briefly discuss how this can be done for the other two methodologies. In this way we not only explicitly illustrate the use of our quality framework and process for top-down quality improvement, but also implicitly emphasize the benefits of being able to tailor security methodologies “on the fly” using an engineering approach.

The rest of this paper is structured as follows. In Section 2 we provide some background on security methodologies in general, and our approach to engineering security methodologies. In Section 3 we discuss the concept of security methodology quality, and present of overview of our quality framework; the details of all its constituent parts, as well as the accompanying process for using the framework, are presented in Section 4. Taking several existing methodologies as case studies, we apply our process and framework in their quality assessment capacity in Section 5, and subsequently attempt to improve the quality of the methodologies in Section 6. In Section 7 we consider related work, and in Section 8 we conclude and discuss future directions.

2 Background

In this section we provide background on security methodologies and our previously proposed approach to engineering them. Some of the material (especially in Section 2.1) is taken from [Uzunov et al., sub2], with adaptation where appropriate.

2.1 Security methodologies

A security methodology constitutes a “systematic way of doing things in a particular discipline” [Gonzalez-Perez and Henderson-Sellers, 2008], where in this case the discipline is secure software engineering. Security methodologies are strongly related to development methodologies, however, in our work and for the rest of this paper, security methodologies will be considered as separate entities in their own right, distinct from any development methodology to which, or in which, the latter might be related or integrated, respectively. Unlike a development methodology, a security methodology’s primary aim is not to provide guidance for producing particular software artefacts, but rather, for improving the security attributes of a given system, i.e. a security methodology is process-centric. This is not to imply, of course, that security methodologies cannot prescribe the production of various software artefacts also.

Despite their process-centric nature, security methodologies are not simply “bare” processes in the sense of simply being descriptions of activities – a consideration of existing methodologies (see [Uzunov et al., 2012b]) reveals an inherent dichotomy, reflecting the \textit{process-product} dichotomy of development methodologies: that of \textit{process-framework}, which is to say, a security methodology is a process for improving the security attributes of a system via the use of a conceptual framework, consisting of various “security improvement artefacts” (e.g. security solutions and attendant conceptual artefacts – cf. [Schumacher, 2003]). A security methodology \( SM \) can thus be defined more formally as a couple:

\[
SM = (SP, CF)
\]
where $SP$ is a security process – the activities and/or steps taken to secure a software system of some type; and $CF$ is a conceptual security framework, consisting of the conceptual artefacts used by the methodology's process, the number and type of which, in theory at least, can be arbitrary, but as a minimum must include a set of security solutions (defensive artefacts), as well as a set of threats (offensive artefacts) leading to those solutions, whether explicitly defined and cataloged or otherwise. Henceforth we will refer to the two constituents of a security methodology as a security process aspect and a conceptual framework aspect, respectively.

### 2.2 Engineering security methodologies

Over the last two decades, a number of researchers have come to regard fixed, “one-size-fits-all” development methodologies as inadequate and unable to meet the needs of different organizations – a realization that has given impetus to the field of (Situational) Method Engineering – (S)ME [Henderson-Sellers and Ralyte, 2010]. Similar arguments can be adduced with respect to security methodologies (see [Uzunov et al., 2012b]), which has led to the proposal of a comprehensive, pattern-oriented approach to engineering security methodologies in [Uzunov et al., sub2] inspired by a number of SME ideas.

The approach for engineering security methodologies in [Uzunov et al., sub2] is composed of three interconnected parts: a framework of interrelated security process patterns and pattern modifiers ([Kolfschoten et al., 2011]) called SPPF (Security Process Pattern Framework), for engineering the process aspect of a security methodology; a meta-model for engineering a methodology's conceptual security framework aspect; and a unifying meta-methodology (S-SMEP) to guide engineers in using the latter artefacts in a step-wise fashion.

The pattern framework is divided into three levels, only the second of which, strictly speaking, contains process patterns pertaining to the security process aspect of a methodology. Security process patterns are essentially traditional (development-oriented) process patterns [Ambler, 1998] encapsulating re-usable collections of security-related techniques and/or activities in a generic fashion. Patterns are more abstract than detailed process descriptions, and their encapsulated solutions emphasize what should be done in a given context, rather than how – the latter details being specified in the pattern instances. The process patterns can be classified according to their granularity as phase (P), stage (S) and task (T) patterns. A number of the patterns in the first part of the framework with different granularities are also usable as modifiers to other patterns (in the same level), which can result, according to the intents of the methodology engineers, in an interconnected “web” of patterns mutually influencing each other’s solutions.

In contrast to the second framework level where most modifiers are implicit, the first level of the framework contains a collection of explicit pattern modifiers termed generic life-cycle modifiers, which help align particular process patterns to a given development phase/stage, and determine solution variants.

Engineering a security methodology using S-SMEP – which can take on the form of tailoring an existing (base) methodology or constructing a methodology “from scratch” – entails constructing pattern-based methodology models using SPPF and the conceptual framework meta-model, and subsequently specifying the patterns and artefacts. The latter models are represented via a UML-inspired notation.

To give further insight into this approach (represented schematically in Figure 1), we briefly describe the intents (essential solutions) of several of the second-level SPPF patterns below, and duplicate the conceptual framework meta-model diagram from [Uzunov et al., sub2] in Figure 2. In the table presentation of the SPPF patterns, the granularity (Phase/Stage/Task) appears in the column marked “Gr”.

<table>
<thead>
<tr>
<th>Security process pattern</th>
<th>Shorthand name</th>
<th>Gr</th>
<th>Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security Requirements Determination</td>
<td>SecReq</td>
<td>P</td>
<td>Specify both prescriptive and resultant security requirements for the target system.</td>
</tr>
<tr>
<td>Countermeasure Introduction</td>
<td>CounterIntro</td>
<td>P</td>
<td>Introduce security countermeasures by converting problem-space requirements into solution-space goals and policies, and incorporating these into relevant models of the target system.</td>
</tr>
<tr>
<td>Security Implementation</td>
<td>SecImpl</td>
<td>P</td>
<td>Transfer the security-related (standalone or enriched functional) models into concrete software units.</td>
</tr>
<tr>
<td>Security Administration</td>
<td>SecAdmin</td>
<td>P</td>
<td>Configure all component parts of the (implemented) security infrastructure, setting appropriate policies for the individual controls.</td>
</tr>
<tr>
<td>Adversary Modeling</td>
<td>AdvMod</td>
<td>S</td>
<td>Model attackers and their potential attacks (e.g. in the form of threats) to a system.</td>
</tr>
<tr>
<td>Countermeasure Identification</td>
<td>CounterIdentif</td>
<td>S</td>
<td>Map problem-space security requirements to solution-space abstractions.</td>
</tr>
<tr>
<td>Security</td>
<td>SecMod</td>
<td>S</td>
<td>Incorporate security countermeasures (e.g. via the application of security...</td>
</tr>
</tbody>
</table>
Table 1: Summary of a selection of SPPF second-level patterns

<table>
<thead>
<tr>
<th>Security process pattern</th>
<th>Shorthand name</th>
<th>Gr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security Verification</td>
<td>SecVerif</td>
<td>S</td>
</tr>
<tr>
<td>Coding</td>
<td>Cod</td>
<td>S</td>
</tr>
<tr>
<td>Structured countermeasure identification</td>
<td>structured</td>
<td>T</td>
</tr>
<tr>
<td>Instantiation</td>
<td>instantiate</td>
<td>T</td>
</tr>
<tr>
<td>Modeling</td>
<td>model</td>
<td>T</td>
</tr>
<tr>
<td>Semi-automated verification</td>
<td>semi-automated</td>
<td>T</td>
</tr>
<tr>
<td>Attack scenarios</td>
<td>attacks</td>
<td>T</td>
</tr>
<tr>
<td>Refinement</td>
<td>refine</td>
<td>T</td>
</tr>
</tbody>
</table>

Table 1: Summary of a selection of SPPF second-level patterns

Regarding the approach to engineering a methodology's conceptual framework, our meta-model in Figure 2 effectively prescribes a set of constraints on the nature of, and relationships between, the various conceptual artefacts to be used in a security methodology. In particular, the meta-model dictates that all conceptual security frameworks should consist of a set of Conceptual security artefact collections, which aggregate Conceptual security artefacts. Such artefacts can be specified or documented explicitly, thus forming an Explicit artefact collection, which may be structured (e.g. a catalog) or unstructured; or they may be based purely on implicit (expert) knowledge, in which case they form an Implicit artefact collection. In both cases conceptual artefacts may be related, or relatable, to a particular system characterization (e.g. vulnerabilities being related to a network model). The artefacts, their groupings and their instances, can be coarsely categorized into two main classes (not shown in the Figure): offensive artefacts (threat, attack, vulnerability) and defensive artefacts (security solution, security requirement and derivatives). Regarding presentation, the dashed boxes in the figure (Fig. 2) represent elements that cannot be instantiated and are included only for the sake of clarification, while the colours (or shades of gray if not
viewed in colour) are used simply to improve readability and to emphasize the importance of the Conceptual security artefact model element.

![Figure 2: Meta-model for engineering a methodology's conceptual security framework](image)

3 **Methodology quality**

3.1 **General discussion**

Quality can often be an elusive and sometimes ambiguous concept. The etymology of the term “quality” stems from the Latin *qualitas* [Porter, 1913], which designates a characteristic or property of something (first sense); but, the term also implies that something possesses one or more desirable characteristics or properties (second sense), i.e. quality in the second sense is the possession of certain qualities in the first sense.

In the software quality literature it is often customary to adduce several viewpoints to help define what quality is [Kitchenham and Pfleeger, 1996]. The most common of these views refers to quality being “fitness for purpose” [Pressman, 2009, Voas, 2008], which can be interpreted (or re-interpreted) as suitability, or what the user sees or determines as being “good quality”, based on characteristics already present or characteristics deemed desirable in general. Another equally common view is that quality implies the possession of certain inherent characteristics, or more precisely, certain “absolute” features. In both these views, quality can be broken down into a number of directly or indirectly measurable elements equivalent to, or capable of being related to, non-functional requirements: for example, performance, integrity, dependability and others. The precise element semantics and classifications vary between different quality models (see, e.g., [Buglione and Abran, 1999, Côté et al., 2007, Tian, 2005]), but in general these elements can be divided into external, or those distinguishable by users; and internal, or those distinguishable by developers.

While seemingly satisfactory, the latter two views of quality are not the only ones. Another view from the software quality domain essentially stipulates that quality is the result of applying a particular process, i.e. the characteristics of the process reflect themselves in the final characteristics of the product of that process. Strictly speaking, however, this is not a view of quality as such – rather, it is a view of how quality can be achieved [Miller, 2002]. Similarly, the view occasionally advanced (again in the software quality literature) that quality depends on a value attached by stakeholders has little to do with “quality” as understood in either senses of that term, being simply subjective “perceived worth” independent of the characteristics of the target (security methodology in this case), and is thus a separate concept – not a view of quality.

Finally, in the context of general pattern languages, the concept of quality arises in a most interesting way in the work of the architect Christopher Alexander [Alexander, 1979] – the pioneer of patterns for design – who speaks of “the quality without a name”, i.e. a quality possessed by something, which the application of patterns should give rise to. This can be regarded as quality viewed transcendentally, which, as Côté et al. point out [Côté et al., 2007],
can be achieved essentially as the result of achieving quality with respect to the other, more tangible views.

These deliberations indicate that the two preliminary views are indeed the most satisfactory for our context, and lead to the conception that methodology quality is composed of two inter-related aspects: the presence of absolute features inherent in the methodology; and suitability (situational quality) – determined by assigning different weight to certain features and/or by stipulating desirable, situation-specific features. This conception, in turn, leads to a simple, but general, definition of security methodology quality, namely:

the presence of a collection of absolute, desirable and measurable features that render a methodology suitable for a given situation or class of situations.

This definition accords well with the more general ISO definition of software quality [ISO 8402]: “the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs”. Henceforth we will speak interchangeably of security methodology quality (or more simply methodology quality) or the quality of a (security) methodology with our definition from above in mind.

3.2 Quality framework overview

Given our definition of methodology quality, we have chosen to base our framework on the McCall quality model for software [McCall et al., 1977, Cavano and McCall, 1978, McCall, 1994], which employs a configuration of elements breaking “quality per se” into a number of methodology-user-distinguishable (external) elements, which need not or cannot be measured directly; and a number of methodology-intrinsic (internal) elements representing absolute features determining the user-distinguishable elements, each of which is measurable. Following this schema in our framework, methodology quality, as a concept, is broken into a set of contributing quality factors; factors are associated with a collection of quality criteria, which determine the presence of certain features related to a particular factor; and the satisfaction of criteria is measured via a set of metrics (measures of the extent or degree to which something possesses and exhibits a certain quality characteristic [Boehm et al. 1976]). This naturally imparts a two-level structure to the framework, with the first level containing quality factors; and the second level containing criteria and metrics.

In keeping with the McCall model, it is possible for identical criteria to be associated with multiple factors. Metrics, on the other hand, are attached only (and uniquely) to criteria – factors are not measured directly. The relationships discussed thus far are shown more precisely in the conceptual model in Figure 3, which is essentially a “structure model” or meta-model (see [Kitchenham et al., 1997, Wagner and Deissenboeck, 2007]) for the whole quality framework. Factors, criteria and metrics in the figure appear in dark green (or dark grey, if viewing in black and white).

Going beyond the McCall model, the second level of the framework is itself stratified into two sub-levels (represented in the figure by elements with grey and pink [light-grey], respectively), the first of which contains a collection of abstract criteria and abstract metrics, and the second of which contains concrete criteria and concrete metrics, the latter two being instances of the former two. Instantiation implies that the context of, and features encompassed by, a criterion – as well as the indicators aggregated by its metric – become more specific, more tangible, and hence more measurable; it also implies that the properties (class criterion property in the figure) possessed by the criteria become more defined. There are three such properties (the letters in brackets are a short-
hand notation):

- **applicability**: whether the criterion applies to a methodology's security process (SP), conceptual framework (CF), or both (SP, CF);
- **granularity**: whether the criterion is relevant to the whole methodology (w) or part of it (p) – i.e. to groups of elements or individual elements;
- **objectivity**: whether the inherent nature of the encompassed features will result in measures that are predominantly subjective (S), based on judgement or expert opinion, or predominantly objective (O).

A criterion for the same context and features may exist with different properties, which is to say, in more than one variant – for example, a subjective efficiency criterion measured via expert opinion and an objective efficiency criterion measured via number of development artefacts produced.

Besides concrete criteria being instances of abstract criteria, all criteria can be related to each other via four kinds of criterion relationships:

- **(weak) implication (I)**: where the measures of criterion A imply, in general, similar measures of criterion B, e.g. higher artefact specification detail generally implies higher artefact understandability.
- **reverse implication (R)**: where the measures of criterion A imply, in general, the reverse measures of criterion B, i.e. higher measures of A imply lower measures of B.
- **dependency (D)**: where the measure of criterion A is dependent in some fashion on the measure of criterion B.
- **specialization (S)**: where criterion B specializes the context and features of criterion A, and B's associated metric derives certain indicators from the metric of A. Specialization implies that the applicability (security process, conceptual framework, both), granularity (whole or part) and objectivity properties are likewise specialized.

Both abstract and concrete criteria (and metrics, correspondingly) can be specialized as shown in the model.

Regarding metrics, unlike the general view adopted in much of the software quality literature, which emphasizes the measures (see Fig. 3) – i.e. final results – we hold that arriving at these measures inevitably requires some kind of process: for example, when measuring the defect rate one must first collect information about the defects, and then apply some kind of mathematical function with the latter information as its domain. Following this line of thought, a metric can be seen as a function – implicit or explicit – mapping data to numerical or discrete values, in combination with a process that guides the collection of data as well as the application of the latter measuring function itself. This perspective of a metric captures somewhat more rigorously the notion of metrics as “ways of measuring something”, where the “way” aspect is captured as a process and the “measuring” aspect as a function, and is the perspective underlying our definition of the various metrics in our framework. Regarding terminology, we use the somewhat ambiguous concept of an indicator (already referred to above) for the actual measurement points, i.e. characteristics of the methodology that contribute to the final measure (in the form pattern instances, specification of particular patterns etc.); thus in our framework metrics aggregate indicators.

As can be inferred from the preceding discussions, assessing a given security methodology's quality implies assessing each quality factor via a collection of criteria, which in turn are measured in some sense using metrics. In this paper we will concentrate only on qualitative assessments, since imparting a numerical measurement is of less value in comparison to knowing why (a qualitative grade) completeness is lacking. To make the latter “why” more informative, qualitative measures can also be augmented (to form augmented measures) with numerical values or expressions, or lists of satisfying elements, when appropriate. For example, when considering the range of specific conceptual security artefacts the distinction between a "many" and "few" measure can be subjective, hence simple algebraic expressions can be used to convey more concrete information (e.g. "many \{n > 20\}", or "few \{n < 5\}", where n is understood implicitly as being the count of elements). Similarly, lists of activities or artefacts can be enumerated alongside the measure (e.g. "many \{security patterns, threat patterns, semantic analysis patterns\}") or simply "many \{patterns\}"). Appending such information to the final measures facilitates comparison between methodologies, increases precision and promotes greater insight into how and why a given criterion was or was not satisfied.

The (concrete) criteria and associated metrics presented in this paper (Section 4) form a base criteria profile applicable to all general project situations, which represents what we consider to be an “essential” criteria set encompassing a wide range of existing (desirable) methodology features. As discussed previously, quality factors can also be dependent on the specifics of a particular project situation – e.g. the target system to which a methodology will applied, expertise of team members etc.. To give a concrete example for applicability, a methodology can be deemed complete with respect to the base profile, but incomplete with respect to a situation in which a multi-agent system (MAS) must be developed, which would require, as a minimum, specialized solutions to be available for use, as well as more specific modeling constructs. The latter consideration implies that new quality criteria and metrics are required to account for the different circumstances. Such criteria and metrics are included in a situational criteria profile, which augments a customized collection of criteria and metrics (some or all) from the
base profile with ones relevant to a particular project situation, as instances of abstract criteria and/or as subcriteria/specializations of different base criteria. In the case of specializations, it is only necessary to include the final specialized criterion $C^{(n)}$ in a series of specializations $(C, C'', \ldots, C^{(k)}, \ldots, C^{(n)})$, if, for example, the other criteria $C^{(k)}$ (for any $k \neq n$) are deemed too broad or general. In this paper two example situational profiles based on applicability for general distributed and peer-to-peer systems are presented in Section 5.2.

4 Quality framework and assessment/improvement process

Having given a brief overview of the structure of our framework in the previous section, in what follows we present its two levels, beginning with the quality factors (1st level) in the next sub-section, and proceeding with the criteria and metrics (2nd and 3rd levels) for each factor. We conclude the section by presenting the simple assessment process that accompanies the framework and guides its use.

Regarding the construction of the framework, in developing the factors, criteria and metrics we employed two complementary approaches:

- **Deductive**: by using as a starting base the evaluation criteria of [Uzunov et al., 2012b], which represent desirable security methodology features from an industry adoption perspective; and supplementing these by adapting, where appropriate, a selection of the vast range of criteria, attributes, characteristics etc. present in software quality models and throughout the software quality literature in general – e.g. [McCall et al., 1977, McCall, 1994, Hesari et al., 2010, ISO/IEC 9126-1:2001, ISO/IEC 25010:2011, Siakas and Georgiadou, 2005, Tian, 2005].
- **Inductive**: by considering the prominent features of existing security methodologies documented in the literature [Uzunov et al., 2012b] (cf. [Ramsin and Paige, 2008] for development methodologies).

4.1 Methodology quality factors (first framework level)

According to [McCall et al., 1977], a factor is “a condition or characteristic which actively contributes” to quality. Methodology quality can be broken into seven factors, which can be described briefly as follows (the letters in brackets are a short-hand notation):

- **Correctness of construction (R)**: determines whether construction of the methodology – with respect to the order and nature of its activities – is free from any obvious contradictions (e.g. verifying something before designing it).
- **Completeness (C)**: determines whether a methodology possesses certain essential features, e.g. with respect to its activities – their type and nature – or techniques, or (conceptual security) artefacts used.
- **Comprehensiveness (V)**: determines the level of detail and scope of a methodology's constituent parts.
- **Usability (U)**: determines how “easy” it is, from the point of view of developers, to apply a given methodology for developing a secure system.
- **Adaptability (A)**: determines whether the methodology has, or is able to retain, its relevance or applicability (as a whole, or of its constituents), to different situations and/or contexts.
- **Assurance (S)**: determines whether a methodology can provide some kind of guarantees about the enactment of its activities, whether security features are incorporated into a system correctly and/or whether these features provide appropriate protection.
- **Effectiveness (E)**: determines how well a given methodology introduces security into a system, or in other words, how secure the resulting system is after the application of a given methodology in comparison to that methodology not having been applied [Fernandez et al., 2009a].

While these factors do not encompass all the high-level characteristics of a security methodology, they do attempt to capture the main classes contributing to a “good quality” methodology even in an intuitive sense, i.e. a methodology which is well structured, complete, comprehensive, usable and effective.

4.2 Abstract criteria and metrics (second framework level, first sub-level)

In this section we present the abstract criteria and metrics for the quality factors outlined above. The presentation format, employed later for concrete criteria also, uses a single table (see Table 2). The quality factors to which a particular criterion is associated are shown in the left-most column (headed “QF”) with the letters of the short-hand letters from Section 4.1 (as noted earlier, a criterion can be associated with more than one quality factor). Since abstract criteria can be used in a number of contexts, the first column implies possible associations. The second column contains the criteria, with a brief name and explanation. Criterion properties appear in brackets next to the name using the short-hand letters introduced in Section 3.2.

The third column summarizes the metric associated with a given criterion. The abstract metrics, like their concrete counterparts, can be considered check-lists that “grade” the satisfaction of a criterion (see [Cavano and McCall, 1978]) with a discrete set of values, either ranging from “poor” (or equivalent) to “excellent” (or
equivalent) – denoted in the last column of the table by a comma-separated ordered list – or as differentiating classes – denoted by a semi-colon-separated unordered list. Ordered and unordered lists can be combined when, for example, it is envisaged that a criterion may not be applicable in the specified way to a given methodology. Possible values for augmented measures are represented textually in curly braces alongside the relevant qualitative measures.

<table>
<thead>
<tr>
<th>QF</th>
<th>Abstract criterion name and brief description</th>
<th>Abstract metric summary</th>
<th>Possible measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, V</td>
<td>Broadness of scope (SP, CF; w, p; S, O): the spread of activities or scope of activities or (conceptual security) artefacts.</td>
<td>Consider the specifications of the relevant process elements and/or artefacts used in the methodology model.</td>
<td>low, medium, high</td>
</tr>
<tr>
<td>R, U, A, S</td>
<td>Coverage (SP, CF; w, p; S, O): how well certain activities or artefacts cover a given set of cases, scenarios, conditions etc.</td>
<td>Define a set of scenarios or cases for which to evaluate relevant process elements or artefacts, and consider how well each of scenario or case is addressed.</td>
<td>few, many {list}; poor, average, good, excellent {list}</td>
</tr>
<tr>
<td>All but R</td>
<td>Specific activities (SP; w, p; O): whether certain activities are performed.</td>
<td>Based on the methodology model (process aspect) consider the presence or absence of the target activities in relevant pattern instances.</td>
<td>no, partial, yes</td>
</tr>
<tr>
<td>All</td>
<td>Specific activity sequence (SP; w, p; O): whether a certain sequence of activities is performed.</td>
<td>Consider the presence or absence of the sought after activity sequence in the patterns of the methodology model (process aspect).</td>
<td>no, partial, yes</td>
</tr>
<tr>
<td>All but R</td>
<td>Specific activity approach (SP; p; S, O): whether particular activities are performed using some specified approach.</td>
<td>Consider the specifications of the pattern instances in the methodology model that encompass the target activities.</td>
<td>no, partial, yes</td>
</tr>
<tr>
<td>All but R</td>
<td>Specific artefacts (CF; p; O): whether particular artefact types or classes are employed.</td>
<td>Based on the methodology model (conceptual framework aspect) consider the presence or absence of the target artefacts.</td>
<td>no, partial, yes</td>
</tr>
<tr>
<td>C, V</td>
<td>Documentation (SP, CF; w, p; S): whether particular activities or artefacts of a methodology are documented well.</td>
<td>Consider the specifications of the target activities/artefacts, and judge its completeness, comprehensiveness and clarity. Missing documentation is automatically “poor”.</td>
<td>poor, incomplete, sufficient, detailed</td>
</tr>
<tr>
<td>V, A</td>
<td>System applicability (SP, CF; w, p; S, O): whether some activities and/or artefacts of the methodology can be applied in the context of a given type of system(s).</td>
<td>Identical to the metric for Broadness of scope (SP, CF; w, p; S, O).</td>
<td>no, partial, yes</td>
</tr>
<tr>
<td>V, A</td>
<td>Flexibility (SP, CF; w, p; O): with respect to the process aspect, whether the methodology can incorporate extra steps or skip steps; with respect to the conceptual framework aspect, whether the methodology can change artefacts or use a different range or artefacts within the same artefact type.</td>
<td>Consider whether there are flexible process elements or artefacts in the methodology model. Analyze the workflows of the process model to see whether it is possible to perform different activities depending on some criteria. A “conditionally flexible” measure implies a methodology allows variations dependent on given situations; a “tailorable” measure implies a methodology allows dynamic customization inherently.</td>
<td>fixed, conditionally flexible, tailorable</td>
</tr>
<tr>
<td>R, U, A, S</td>
<td>Compatibility (SP, CF; w, p; O): whether given activities and/or artefacts are compatible with some conditions, e.g. with each other, or with development activities.</td>
<td>Depending on the target conditions, consider the methodology elements and their structural relations and/or the specifications of the various elements.</td>
<td>poor, average, good, excellent; incompatible, compatible</td>
</tr>
<tr>
<td>V, U, S</td>
<td>Use of best practices (SP, CF; w, p; S, O): whether activities and/or artefacts employ some kind of widely recognized best practices in some fashion.</td>
<td>Consider the specifications of the target methodology elements, and judge or determine whether practices widely used in real-life projects and/or well documented in the literature are being employed.</td>
<td>no, partial, yes</td>
</tr>
<tr>
<td>QF</td>
<td>Abstract criterion name and brief description</td>
<td>Abstract metric summary</td>
<td>Possible measures</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>V, U, A</td>
<td><strong>Guidance (SP, CF; w, p; S, O):</strong> whether the methodology includes a guided approach for some activity, or whether guidance is inherently provided by some artefacts or the use of some artefacts.</td>
<td>Analyze the specifications of the relevant process elements and/or artefacts. The descriptions should show clear evidence of a guided approach, with one step following another towards some goal (for process elements), or some structure – see the Structure (SP, CF; w, p; S, O) criterion – that lends itself to use in a similar fashion (for artefacts). In all cases guidance is determined from the point of view of enactment.</td>
<td>no, partial, yes</td>
</tr>
<tr>
<td>All but R and E</td>
<td><strong>Structure (SP, CF; w, p; S, O):</strong> to what degree various activities and/or conceptual framework artefacts possess some inherent structure.</td>
<td>Consider the workflow of activities and/or the conceptual framework elements in the methodology model, as well the corresponding specifications. Specific ordering of elements and relationships are some indicators for structure being present. In all cases structure is determined from the point of view of construction.</td>
<td>unstructured, partially structured, structured</td>
</tr>
<tr>
<td>U</td>
<td><strong>Expertise required (SP, CF; w, p; S):</strong> the level of (security) expertise required for a given collection of activities or the use of a given collection of artefacts.</td>
<td>Identical to the metric for Use of best practices (SP, CF; w, p; S, O), except with respect to security expertise.</td>
<td>low, medium, high</td>
</tr>
<tr>
<td>U</td>
<td><strong>Known modeling language (SP, CF; w, p; O):</strong> whether the modeling languages used for various activities and/or artefacts are well-established or not (with the latter implying a need for developer training).</td>
<td>An analysis of relevant process elements and/or artefact specifications can reveal the modeling language. If implicit, the modeling language used by corresponding development approach is assumed and analyzed accordingly (see [Hesari et al., 2010] for criteria for the latter).</td>
<td>non-standard, mixed, standard</td>
</tr>
<tr>
<td>U</td>
<td><strong>Efficiency (SP; w, p; S, O):</strong> how efficient in terms of time required a particular activity, or the methodology as a whole, is.</td>
<td>Metrics for the amount of overhead generated, number of steps used, manual efforts involved, difficulty of activities, number and (relative) complexity of artefacts can be used.</td>
<td>poor, average, good, excellent</td>
</tr>
<tr>
<td>U</td>
<td><strong>Understandability (SP, CF; w, p; S):</strong> the ease with which activity or artefact specifications (if specified explicitly) can be understood, both with respect to security and non-security properties.</td>
<td>Consider the relevant specifications and judge whether they are clear and comprehensible for those unfamiliar with the target elements, especially security non-experts. Formal specifications are a negative indicator, as are unclear descriptions (see also the Documentation (SP, CF; w, p; S) criterion).</td>
<td>poor, average, good, excellent</td>
</tr>
<tr>
<td>U, S, A</td>
<td><strong>Software support (SP; w, p; O):</strong> whether any activities are supported by software tools or frameworks.</td>
<td>Consider the relevant methodology descriptions for when and how any software is used.</td>
<td>none; development-time; run-time</td>
</tr>
<tr>
<td>All but R, S and E</td>
<td><strong>Reusability (SP, CF; w, p; S, O):</strong> whether some activities or artefacts or the results of the application of either, can be used in different or related contexts.</td>
<td>Analyze the descriptions of relevant process elements; consider whether certain conceptual framework elements in the methodology model linked in earlier phases are also used in later phases.</td>
<td>low, medium, high {what is reused}</td>
</tr>
</tbody>
</table>

**Table 2: Abstract quality criteria and metrics**
4.3 Concrete criteria and metrics (second framework level, second sub-level)

In this section we consider the concrete criteria and metrics that together constitute the base criteria profile, grouped by their associated quality factors. The only factor left out of consideration is Effectiveness, with the reason being that attempting to measure or assess methodology effectiveness leads naturally into the area of security metrics (i.e. measuring product security – see [Mellado et al., 2010]), which is outside the scope of this paper.

Regarding the construction of the concrete criteria and metrics, we instantiate the abstract criteria from the first framework level guided by two complementary approaches (see the outset of Section 4; cf. Section 3.2), the first being deductive – where we are led by existing evaluation and (software product / development methodology) quality criteria in the literature; and the second being inductive – where we are led by the prominent features of existing security methodologies that were deemed desirable. The resulting base criteria profile inevitably reflects, however – like any collection of quality criteria – a subjective selection guided not only by existing literature and existing methodologies, but by our expertise and view on what constitutes an “essential” criteria set. Any objections regarding why one abstract criterion was instantiated in one way and not another, or why it was instantiated at all and why another one was not instantiated, can be remedied by the construction of a situational profile (cf. 3.2; see also Section 5.2). Any claim for a “definitive” set of criteria, no matter how objectively it was construed, is destined to be outdated (cf. [McCall et al., 1978] in 1978 vs. [ISO/IEC 9126-1:2001] in 2001 vs. [ISO/IEC 25010:2011] in 2011) and hence, in time, to be considered “less definitive”; therefore, the presence of subjectivity based on expertise in our base criteria profile in no way reduces its validity or value.

The presentation of our base profile follows the tabular form used in Section 4.2, except for those criteria and metrics associated with Correctness of construction, where the presentation is explained in the relevant sub-section. In all cases we use the shorthand notation seen in brackets throughout Sections 2.2 and 3.2 for criterion properties (placed next to the criteria names), relationships and SPPF patterns. Criterion instances can be recognized by the use of brackets after the abstract criterion name, which specify the concrete context and/or target features:

<abstract criterion> (properties p) → <concrete criterion> (context) (defined properties p')

Specialized concrete criteria appear with one or more hyphens after the original context (a convention that holds for the rest of this paper):

 criterions (context) (p) → criterions (context – specialized context1 – sp. cxt. 2 – ...) (p)

Since we have attempted to group the the criteria according to their associated quality factors, and since criteria can be associated with more than one factor, criteria that have been presented earlier appear in the tables in italics with an asterisk in brackets.

The indicators for the metrics in all criteria refer to a methodology's pattern-based model and its accompanying documentation. We should stress that, for the exception of Correctness of construction, these indicators are in general not exhaustive (as befits the “guideline” nature of our metrics), especially for subjective criteria; therefore, it may also be possible to determine the satisfaction of a given criterion from other indicators found in the methodology's descriptions.

4.3.1 Correctness of construction (R)

There are two criteria for the Correctness of construction factor:

- **Specific activity sequence (valid pattern ordering) (SP; w; O):** whether SPPF pattern instances are placed in a sensible order in a given methodology.
- **Compatibility (SPPF patterns) (SP; w; O):** whether SPPF pattern instances are compatible with each other, and with the development life-cycle in which their activities are to be performed.

These two criteria effectively imply constraints on the way SPPF patterns can be arranged in a particular methodology model, which are defined by corresponding metrics attempting to measure two classes of problems for the two criteria (respectively) that should not occur¹: invalid sequential ordering, and inappropriate pattern combinations. Conformance to the constraints defined by these metrics (viz. measures indicating that no problems are present) thus determines structural validity of the security process. This determination is not, however, absolute, since, firstly, the result of a particular metric is also dependent on the specification of the pattern instances in a methodology model – i.e. whether in a given case the activities actually make sense – which should always be taken into account; and secondly, the metrics are not exhaustive, since pattern instances can be arranged in more than just sequential workflows. Despite these points, the metrics – whose (negative) indicators are shown in Table 3 – capture the major indicators of problems with ordering and compatibility, and hence correctness of construction for the

---

¹We should point out that specifying such constraints (and hence criteria) for a methodology's conceptual framework is not necessary, since these are already encapsulated in the framework meta-model (see Section 2.2).
The indicators for the metrics of the two criteria discussed thus far are collectively presented in Table 3. The leftmost column in the table contains a listing of the (currently available) SPPF patterns, with the next column (titled “Gr”) indicating each pattern’s granularity. The third column contains the invalid sequential orderings of the patterns, which define the negative indicators for the metric associated with the ordering quality criterion. For a given pattern (table row), the arrow “←” indicates a “prior to” relationship, and the arrow “→” indicates a “follows after” relationship, either of which can be reversed with any related patterns (i.e. A ← B implies B → A). Because of this symmetry, generally only one of the relationships is listed in the table to save space. The result of the metric takes on the Boolean values “well ordered” (positive measure) or “erroneously ordered” (negative measure).

The fourth and fifth columns list, respectively – for each pattern in a given table row – generic life-cycle modifiers with which the pattern is compatible; as well as inappropriate combinations of the pattern with other SPPF patterns. These two columns thus define the positive (for generic life-cycle modifiers) and negative (for SPPF pattern combinations) indicators for the metric associated with the compatibility quality criterion. The result of the metric takes on the Boolean values “fully compatible” (positive measure) or, if any of the negative indicators are present, “incompatibilities present” (negative measure).

In all cases the shorthand names of the SPPF patterns and life-cycle modifiers are used, whereby conglomerate development phases (ReqAn) imply their encompassing stages (requirements gathering as well as analysis).

<table>
<thead>
<tr>
<th>Security process pattern</th>
<th>Gr</th>
<th>Invalid pattern orderings (sequential)</th>
<th>Inappropriate pattern combinations</th>
<th>Incompatible generic life-cycle modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecReq</td>
<td>P</td>
<td>SecReq ← CounterIntro</td>
<td>Attached Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>CounterIntro</td>
<td>P</td>
<td>SecReq ← CounterIntro</td>
<td>Attached ReqAn or Des</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>SecImpl</td>
<td>P</td>
<td>SecReq ← CounterIntro, ← Impl</td>
<td>Attached ReqAn, Des, Impl</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>SecAdmin</td>
<td>P</td>
<td>SecReq ← CounterIntro</td>
<td>Attached Impl or Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>AdvMod</td>
<td>S</td>
<td>SecMod (when referring to same models)</td>
<td>Attached Impl or Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>CounterIdentif</td>
<td>S</td>
<td>SecMod (when referring to same models)</td>
<td>Attached Impl or Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>SecMod</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SecVerif</td>
<td>S</td>
<td>SecMod ← AdvMod (when referring to same models); ← Cod, ← Map, ← Transform (when referring to same implementation)</td>
<td>Attached Deploy</td>
<td>Attached Deploy</td>
</tr>
<tr>
<td>Cod</td>
<td>S</td>
<td>Outside Impl</td>
<td>Attached ReqAn, Des, Deploy</td>
<td>Attached ReqAn, Deploy</td>
</tr>
<tr>
<td>Map</td>
<td>S</td>
<td>In SecReq</td>
<td>Attached ReqAn, Deploy</td>
<td>Attached ReqAn, Deploy</td>
</tr>
<tr>
<td>Transform</td>
<td>S</td>
<td>In SecReq</td>
<td>Attached ReqAn, Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>assets</td>
<td>T</td>
<td>Outside SecReq</td>
<td>Attached Impl or Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>enumerate</td>
<td>T</td>
<td>Outside SecReq</td>
<td>Attached Impl or Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>risks</td>
<td>T</td>
<td>enumerates</td>
<td>Attached Impl or Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>expert</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>structured</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>specify</td>
<td>T</td>
<td>Outside SecMod</td>
<td>Attached Impl or Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>instantiate</td>
<td>T</td>
<td>Outside SecMod</td>
<td>Attached Impl or Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>model</td>
<td>T</td>
<td>Outside SecMod</td>
<td>Attached Impl or Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>reviews</td>
<td>T</td>
<td>In SecReq</td>
<td>Attached Impl or Deploy</td>
<td>Attached Impl or Deploy</td>
</tr>
<tr>
<td>semi-automated</td>
<td>T</td>
<td>In SecReq; with automated</td>
<td>Attached ReqAn, Deploy</td>
<td>Attached ReqAn, Deploy</td>
</tr>
<tr>
<td>automated</td>
<td>T</td>
<td>In SecReq; with semi-automated</td>
<td>Attached ReqAn, Deploy</td>
<td>Attached ReqAn, Deploy</td>
</tr>
<tr>
<td>Security process pattern</td>
<td>Gr</td>
<td>Invalid pattern orderings (sequential)</td>
<td>Inappropriate pattern combinations</td>
<td>Incompatible generic life-cycle modifiers</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----</td>
<td>----------------------------------------</td>
<td>-----------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>attacks</td>
<td>T</td>
<td></td>
<td>In SecAdmin, unless with automated</td>
<td>attached Deploy</td>
</tr>
<tr>
<td>penetration tests</td>
<td>T</td>
<td></td>
<td>Outside SecVerif</td>
<td>attached ReqAn, Arch</td>
</tr>
<tr>
<td>refine</td>
<td>T</td>
<td>← any pattern instance that is not being refined.</td>
<td>In SecReq or SecMod</td>
<td>attached Deploy</td>
</tr>
</tbody>
</table>

### Table 3: Indicators for ordering and compatibility metrics

#### 4.3.2 Completeness (C)

<table>
<thead>
<tr>
<th>QF</th>
<th>Criterion name and brief description</th>
<th>Brief summary of metric for determining satisfaction</th>
<th>Possible measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Broadness of scope (encompassed development phases) (SP; w; O): to what extent security-related activities are spread across the whole development life-cycle (requirements analysis, design, implementation, testing, deployment, maintenance and deposition).</td>
<td>Consider which generic life-cycle modifiers are used in the methodology model.</td>
<td>single-phase, multi-phase, end-to-end</td>
</tr>
<tr>
<td>C</td>
<td>Coverage (related development activities) (SP; w; S, O): how many standard development activities such as modeling, producing test plans, coding etc. are complemented by security-related activities.</td>
<td>Ascertain that activities exist for each generic life-cycle modifier in the methodology model.</td>
<td>few, many, most {activity list}</td>
</tr>
<tr>
<td>C</td>
<td>Specific activities (early security introduction) (SP; w; O): whether there is an attempt to consider security requirements as early as possible in the development process, i.e. in the requirements analysis phase.</td>
<td>Consider whether the SecReq pattern is part of the methodology model, and if so, which generic life-cycle modifier applies to it.</td>
<td>no, partial {reason}, yes</td>
</tr>
<tr>
<td>C</td>
<td>Specific activities (threat assessment) (SP; w; O): whether threats and/or attacks are explicitly considered as part of the process.</td>
<td>Consider whether the AdvMod pattern is part of the methodology model, and what task modifiers are used.</td>
<td>no, partial {reason}, yes</td>
</tr>
<tr>
<td>C</td>
<td>Specific activities (verification) (SP; w; O): whether some form of assessment or verification is used for ensuring that the introduced security features correctly counter the relevant threats and improve the security of the system as a whole.</td>
<td>Consider the specification of any SecVerif pattern instances, and the relevant task patterns.</td>
<td>none, manual, semi-automated, automated</td>
</tr>
<tr>
<td>C</td>
<td>Documentation (security process) (SP; w; S): whether a methodology is documented well, including all its activities, techniques etc., so that it is possible for developers to make use of it.</td>
<td>Consider how the pattern instances of the methodology model are specified.</td>
<td>poor, incomplete, sufficient, detailed</td>
</tr>
<tr>
<td>C</td>
<td>Specific artefacts (basic security solutions) (CF; p; O): whether the framework contains a range of security countermeasures to cover “common” scenarios in general systems, such as access control and encryption.</td>
<td>Consider the range of the security solution artefact instances.</td>
<td>implicit only, some, many {list}</td>
</tr>
</tbody>
</table>

#### Table 4: Quality criteria and metrics for Completeness (C)

#### 4.3.3 Comprehensiveness (V)

<table>
<thead>
<tr>
<th>QF</th>
<th>Criterion name and brief description</th>
<th>Brief summary of metric for determining satisfaction</th>
<th>Possible measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Documentation (artefacts – offensive) (CF; p; S): how well specified the various offensive artefacts are (i.e. in how much useful detail).</td>
<td>Consider the specifications of the relevant artefact instances and their detail. If they are implicit (reliance placed on experts to define the artefacts during enactment), then by default the</td>
<td>poor, average, good, excellent</td>
</tr>
<tr>
<td>QF</td>
<td>Criterion name and brief description</td>
<td>Brief summary of metric for determining satisfaction</td>
<td>Possible measures</td>
</tr>
<tr>
<td>----</td>
<td>-----------------------------------</td>
<td>------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>V</td>
<td><strong>Documentation (artefacts – defensive)</strong> <em>(CF; p; S)</em>: how well specified the various offensive artefacts are (i.e. in how much useful detail).</td>
<td>Specification detail is poor. When applied to a collection of artefacts, an overall grade (approximately) averaging of all individual grades is taken.</td>
<td>Poor, average, good, excellent</td>
</tr>
<tr>
<td>V</td>
<td><strong>Coverage (artefacts – offensive)</strong> <em>(CF; p; S)</em>: for each offensive artefact type, whether a broad range of scenarios, cases, type-specific properties etc. are covered by corresponding artefact collections.</td>
<td>Analyze the specifications of each artefact type that is being assessed against a set of pre-defined scenarios, cases etc. The final result is measured against these, e.g. poor with respect to inside attacker threats; excellent with respect to low-level web-service vulnerabilities.</td>
<td>Unspecified; poor, average, good, excellent {pre-defined scenarios}</td>
</tr>
<tr>
<td>V</td>
<td><strong>Coverage (artefacts – defensive)</strong> <em>(CF; p; S)</em>: for each defensive artefact type, whether a broad range of scenarios, cases, type-specific properties etc. are covered by corresponding artefact collections.</td>
<td>Identical to the Coverage (offensive artefacts) <em>(CF; p; S)</em> metric, except for defensive artefacts.</td>
<td>Unspecified; poor, average, good, excellent {pre-defined scenarios}</td>
</tr>
<tr>
<td>C, V</td>
<td><strong>Broadness of scope (encompassed development phases)</strong> <em>(SP; w; O)</em> (*).</td>
<td>As for Completeness.</td>
<td></td>
</tr>
<tr>
<td>C, V</td>
<td><strong>Coverage (related development activities)</strong> <em>(SP; w; S, O)</em> (*).</td>
<td>As for Completeness.</td>
<td></td>
</tr>
<tr>
<td>U, V</td>
<td><strong>Structure (artefacts)</strong> <em>(CF; w; O)</em>: to what degree the various conceptual framework artefacts have some inherent structure individually, and when taken together.</td>
<td>Consider the models as well as the specifications of the artefacts in the methodology's conceptual framework; explicit relationships between artefacts and correlations are indicators of strong structure, use of descriptive templates of partial structure. If artefacts are individually structured in some fashion, but are loosely structured taken together, this implies a “partially structured” measure.</td>
<td>Unstructured, partially structured, structured</td>
</tr>
<tr>
<td>U, V</td>
<td><strong>Guidance (identifying countermeasures)</strong> <em>(SP; w; S)</em>: whether the methodology includes a guided approach to identifying appropriate defensive security solutions of various degrees of granularity.</td>
<td>Check the specification of any CounterIdentif instances for the presence of the <em>structured</em> task modifier, as well as for implicit or explicit activities concerned with mapping any set of artefacts to defensive artefacts (e.g. <em>Map</em> pattern).</td>
<td>None, ad-hoc, partially systematic, systematic</td>
</tr>
<tr>
<td>U, V</td>
<td><strong>Guidance (introducing countermeasures)</strong> <em>(SP; w; S)</em>: whether the methodology provides guidance on how and where defensive security solutions should be applied.</td>
<td>Consider the specification of <em>SecMod</em> pattern instances <em>(structured</em> modifier instances being a clear indicator), and implications of the tasks <em>model, specify, instantiate</em>.</td>
<td>None, ad-hoc, partially systematic, systematic</td>
</tr>
<tr>
<td>U, V</td>
<td><strong>Use of best practices (activities)</strong> <em>(SP; w; S)</em>: whether activities that would be considered “best practices” are employed.</td>
<td>Best practices can depend on the specification of activities (purely subjective), but also on the inclusion of task patterns such as <em>penetration tests, enumeration, reviews and risks</em>.</td>
<td>None, some, many {activities}</td>
</tr>
<tr>
<td>U, V</td>
<td><strong>Use of best practices (artefacts)</strong> <em>(CF; w; O)</em>: whether a methodology uses common repositories or catalogs of security knowledge (relating to solutions, threats, vulnerabilities, standards etc.) to increase productivity and/or aid developer training.</td>
<td>Consider the nature of the various artefact instances, and whether they attempt to encapsulate best practices in their specifications (implicit or explicit). Encapsulation in general of security concepts in the particular artefact type is another positive indicator.</td>
<td>None, some, many {list}</td>
</tr>
<tr>
<td>QF</td>
<td>Criterion name and brief description</td>
<td>Brief summary of metric for determining satisfaction</td>
<td>Possible measures</td>
</tr>
<tr>
<td>----</td>
<td>------------------------------------</td>
<td>-------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>U, V</td>
<td>Use of best practices (artefacts – encapsulation) (CF; p; O): whether any particular artefacts encapsulate their security features, and hence distill expert knowledge.</td>
<td>Consider the relevant conceptual security artefact instances – explicit collections indicate encapsulation, the nature and extent of which which can be determined by analyzing the associated specifications in detail.</td>
<td>none, some, many {types}</td>
</tr>
</tbody>
</table>

Table 5: Quality criteria and metrics for Comprehensiveness (V)

### 4.3.4 Usability (U)

<table>
<thead>
<tr>
<th>QF</th>
<th>Criterion name and brief description</th>
<th>Brief summary of metric for determining satisfaction</th>
<th>Possible measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Expertise required (methodology) (SP, CF; w; S): the level of security expertise required to apply the methodology's process.</td>
<td>Lack of AdvMod, the presence of the expert modifier and implicit collections of artefacts indicate a higher level of expertise. The converse can indicate lower levels of expertise.</td>
<td>high, medium, low</td>
</tr>
<tr>
<td>U</td>
<td>Expertise required (methodology – threat modeling) (SP; p; S): the level of security expertise required to apply the methodology's threat modeling/analysis process.</td>
<td>If the AdvMod instance (if present) lacks an attacks modifier, and/or is not linked to explicit offensive artefact collections, then the level is high; otherwise the level is dependent on judging the need for an expert to be involved in the use the latter indicators.</td>
<td>unspecified; high, medium, low</td>
</tr>
<tr>
<td>U</td>
<td>Expertise required (methodology – introducing countermeasures) (SP; w; S): the level of security expertise required to introduce security features (especially security solutions) into a target system.</td>
<td>Consider whether the CounterIntro instance (and associated stages and tasks, especially SecMod) requires the involvement of a security expert, due to implicit (defensive) artefact collections or lack of guidance.</td>
<td>high, medium, low</td>
</tr>
<tr>
<td>U, V</td>
<td>Structure (artefacts) (CF; w; O) (*)</td>
<td>As for Comprehensiveness.</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>Structure (activities – threat modeling) (SP; p; S): whether the threat modeling approach follows some pre-defined conceptual scheme.</td>
<td>Consider whether the structured task is present in AdvMod, and the specification of the enumeration task. Linked artefacts can also contribute to structure and their effect on the activities needs to be judged subjectively.</td>
<td>unspecified; unstructured, partially structured, structured</td>
</tr>
<tr>
<td>U</td>
<td>Structure (activities – introducing countermeasures) (SP; w; S): whether the activities relating to the introduction of defensive security solutions are structured in such as way as to aid developer productivity.</td>
<td>Consider whether pattern instances supporting automation (semi-automated, automated) are present in CounterIntro. Linked artefact collections can also contribute, as well as modifiers such as structured, model, and refine.</td>
<td>unstructured, partially structured, structured</td>
</tr>
<tr>
<td>U</td>
<td>Known modeling language (at ReqAn and Des) (SP; w; O): whether the languages used for various security-related modeling activities during the requirements analysis and design development phases are well-established and widely used.</td>
<td>Consider the modeling languages used in the SecReq and SecMod pattern instances, as well as in the relevant generic life-cycle phases.</td>
<td>non-standard, mixed, standard</td>
</tr>
<tr>
<td>U</td>
<td>Known modeling language (artefacts) (CF; w; O): whether the modeling languages used in the various security artefacts is well-established and widely used.</td>
<td>Consider the explicit specifications of the different artefacts, if any to determine the language used.</td>
<td>unspecified; non-standard, mixed, standard</td>
</tr>
<tr>
<td>U</td>
<td>Understandability (artefacts – offensive) (CF; p; S): the ease with which the offensive artefact specifications, if any, can be understood, both with respect to security and non-security properties.</td>
<td>Specified artefact collections require less expertise than unspecified artefact collections, however, the nature of the artefact is also important and should be considered. Formalized artefacts require more expertise, as do artefact descriptions tailored for</td>
<td>unspecified; low, medium, high</td>
</tr>
<tr>
<td>QF</td>
<td>Criterion name and brief description</td>
<td>Brief summary of metric for determining satisfaction</td>
<td>Possible measures</td>
</tr>
<tr>
<td>----</td>
<td>-------------------------------------</td>
<td>-----------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>U</td>
<td>Understandability (artefacts – defensive) (CF; p; S): the ease with which the defensive artefact specifications, if any, can be understood, both with respect to security and non-security properties.</td>
<td>Identical to the Understandability (offensive artefacts) (CF; p; S) metric, but for defensive artefacts.</td>
<td>unspecified; low, medium, high</td>
</tr>
<tr>
<td>U</td>
<td>Software support (methodology) (SP; w; O): whether the methodology is supported by any tools and/or software frameworks.</td>
<td>Consider the relevant methodology descriptions for when and how any tools and/or frameworks are used, guided by the methodology model (considering patterns with Req, An, Des and/or Impl modifiers attached for development-time tools/frameworks, and with the Deploy modifier attached for run-time tools/frameworks).</td>
<td>none; development-time; run-time {associated activities}</td>
</tr>
<tr>
<td>U</td>
<td>Efficiency (activities) (SP; w; S): whether the methodology generates development overhead (as in the form a large number of software artefacts) in additional to the development process used.</td>
<td>Consider how many manual activities are used, in particular manual inspection tasks (reviews), as well as the use of complicated SecMod instances.</td>
<td>low, medium, high</td>
</tr>
<tr>
<td>U, S, E</td>
<td>Reusability (generated security designs) (SP; w; O): whether the methodology provides any approaches for reusing security features (particularly during design) incorporated in one system across other systems.</td>
<td>Analyze the SecReq and SecMod instances for specific reuse techniques. Consider whether artefacts in the methodology model linked in earlier phases are also used in later phases.</td>
<td>low, medium, high {what is reused}</td>
</tr>
<tr>
<td>U, V</td>
<td>Guidance (identifying countermeasures) (SP; w; S) (*)</td>
<td>As for Comprehensiveness.</td>
<td></td>
</tr>
<tr>
<td>U, V</td>
<td>Guidance (introducing countermeasures) (SP; w; S) (*)</td>
<td>As for Comprehensiveness.</td>
<td></td>
</tr>
<tr>
<td>U, V</td>
<td>Use of best practices (activities) (SP; w; S) (*)</td>
<td>As for Comprehensiveness.</td>
<td></td>
</tr>
<tr>
<td>U, V</td>
<td>Use of best practices (artefacts) (CF; w; O) (*)</td>
<td>As for Comprehensiveness.</td>
<td></td>
</tr>
<tr>
<td>U, V</td>
<td>Use of best practices (artefacts – encapsulation) (CF; p; O) (*)</td>
<td>As for Comprehensiveness.</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Quality criteria and metrics for Usability (U)

### 4.3.5 Assurance (S) and Adaptability (A)

<table>
<thead>
<tr>
<th>QF</th>
<th>Criterion name and brief description</th>
<th>Brief summary of metric for determining satisfaction</th>
<th>Possible measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, S</td>
<td>Specific activities (verification) (SP; w; O) (*)</td>
<td>As for Completeness</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Specific activity approach (formal verification) (SP; p; O): whether verification employs formal methods.</td>
<td>Consider the specification of any relevant SecVerif pattern instances.</td>
<td>none, informal, semi-formal, formal</td>
</tr>
<tr>
<td>S</td>
<td>Software support (verified implementations) (CF; w; O): whether implementation artefacts that are verified (already or otherwise) in some fashion are used integrally by the methodology.</td>
<td>Analyze the conceptual framework artefacts linked to activities with the Impl and Deploy modifiers attached in the methodology model. Verified implementations can be individual software artefacts (e.g. COTS components), or whole frameworks.</td>
<td>none, software artefacts, framework</td>
</tr>
</tbody>
</table>
### Table 7: Quality criteria and metrics for Assurance (S) and Adaptability (A)

<table>
<thead>
<tr>
<th>QF</th>
<th>Criterion name and brief description</th>
<th>Brief summary of metric for determining satisfaction</th>
<th>Possible measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Specific activities (security measurement) (SP; p; O): whether the methodology provides activities or techniques to determine the resulting, post-application level of security of the associated software products.</td>
<td>Check for the presence of explicit measurement activities in any relevant SecVerif pattern instances.</td>
<td>no, partial, yes</td>
</tr>
<tr>
<td>U, S, E</td>
<td>Reusability (generated security designs) (SP; w; O) (*)</td>
<td>As for Usability.</td>
<td>specific, generic, mixed</td>
</tr>
<tr>
<td>A</td>
<td>System applicability (generality) (SP, CF; w; O): whether the methodology can be applied to any type of system, or to a particular class of systems.</td>
<td>Consider the specification of the process model and the range of artefacts available in the conceptual framework. A result of “mixed” indicates a methodology capable of being applied to certain classes of both specific and generic types of systems.</td>
<td>fixed, conditionally flexible, tailorable</td>
</tr>
<tr>
<td>A</td>
<td>Flexibility (methodology) (SP, CF; w; O): whether the methodology can incorporate extra steps to make it richer or skip steps to make it faster or easier to use.</td>
<td>Consider whether there are any flexible process elements or artefacts in the methodology model; also analyze the workflows of the process model to see whether it is possible to perform different activities depending on different conditions (“conditionally flexible”). A “tailorable” methodology allows dynamic customization inherently.</td>
<td>fixed, conditionally flexible, tailorable</td>
</tr>
<tr>
<td>A</td>
<td>Specific activities (trade-off analysis) (SP; p; O): whether the methodology provides guidelines or techniques to balance security with other (software) quality attributes.</td>
<td>Check for the presence of explicit trade-off activities in the CounterIntro pattern instance.</td>
<td>no, partial, yes</td>
</tr>
</tbody>
</table>

### 4.3.6 Base criteria profile relationships

The way the criteria in the base profile are related is shown in Table 8 below, where the column “C.” refers to the relationship type (with the letters as denoted in Section 3.2); the column “T.” refers to the direction of relationship (one way, denoted “→” or “←”; or two-way, denoted “↔”). To save space, some relationships are given as between one criterion and another which has specializations, but is not itself included in the profile – e.g. Understandability (artefacts) (CF; p; S). This is indicated with the text “(specializations)” written next to the relevant criteria, and implies that the relationship applies to the specializations of those criteria (as is the case for all specialized relationships). All specialization relationships are excluded from the table, since they are implicitly included in the criteria names.

<table>
<thead>
<tr>
<th>QF</th>
<th>Criterion A</th>
<th>C. T.</th>
<th>Criterion B</th>
<th>QF</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, V</td>
<td>Broadness of scope (encompassed development phases) (SP; w; O)</td>
<td>I ↔</td>
<td>Coverage (related development activities) (SP; w; S, O)</td>
<td>C, V</td>
</tr>
<tr>
<td>C, V</td>
<td>Broadness of scope (encompassed development phases) (SP; w; O)</td>
<td>D ↔</td>
<td>Coverage (related development activities) (SP; w; S, O)</td>
<td>C, V</td>
</tr>
<tr>
<td>C, V</td>
<td>Broadness of scope (encompassed development phases) (SP; w; O)</td>
<td>I ↔</td>
<td>Specific activities (early security introduction) (SP; w; O)</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>Specific activities (threat assessment) (SP; w; O)</td>
<td>I ↔</td>
<td>Use of best practices (activities) (SP; w; S)</td>
<td>U, V</td>
</tr>
<tr>
<td>C, S</td>
<td>Specific activities (verification) (SP; w; O)</td>
<td>I ↔</td>
<td>Use of best practices (activities) (SP; w; S)</td>
<td>U, V</td>
</tr>
<tr>
<td>S</td>
<td>Specific activity approach (formal verification) (SP; p; O)</td>
<td>D →</td>
<td>Specific activities (verification) (SP; w; O)</td>
<td>C, S</td>
</tr>
<tr>
<td>U, V</td>
<td>Guidance (identifying countermeasures) (SP; w; S)</td>
<td>I ↔</td>
<td>Expertise required (methodology) (SP, CF; w; S)</td>
<td>U</td>
</tr>
<tr>
<td>U, V</td>
<td>Guidance (introducing countermeasures) (SP; w; S)</td>
<td>I ↔</td>
<td>Expertise required (methodology – introducing</td>
<td>U</td>
</tr>
</tbody>
</table>
Table 8: Relationships between the (base) quality criteria

4.4 Quality assessment and improvement process (QAIP)

Having now presented the quality framework, in this sub-section we outline a simple, complementary process for using the framework named QAIP, consisting of two parts: the first for assessing (security methodology) quality, and second for improving quality. QAIP can be used alone, or as part of an encompassing approach to engineering security methodologies, such as in the Verification stage in the S-SMEP meta-methodology in [Uzunov et al., sub2]. In what follows we outline QAIP’s constituent steps.

**Assessment:**

**Step 1:** Creation of a situational criteria profile: create a criteria profile for the given project situation (see end of Section 3.2). The profile will include concrete criteria and metrics belonging to three categories: (1) from the base profile in unmodified form – which are included directly “as is”; (2) from the base profile requiring customization; and (3) new criteria and metrics that are not in the base profile. For the second category, specializations of the base criteria and metrics are created to reflect the new circumstances and then added to the profile. For the third category, criteria and metrics are instantiated from the collection of abstract criteria (first framework level) and added to the profile. To this end the abstract criteria can also be customized prior to instantiation by being specialized (in which case the specialized abstract criterion will be instantiated) or, in more rare cases, augmented to encompass features that it is not currently possible to capture, as required. In all cases the existing criteria and metrics in this paper are treated as “role models” for the construction or specialization of their (new) situational counterparts, which, in general, will refer to indicators present in, or derivable from, a pattern-based methodology model.

Regarding the addition and specialization of criteria, it is worth noting that while new criteria can be added (recursively) as specializations – resulting in arbitrary criteria hierarchies – in our assessment approach we (currently) treat all criteria as a flat hierarchy, measuring each one individually; therefore, specialization implies only a definite relationship between two criteria.

Once all criteria have been added to the profile, criterion relationships are formed between the situational criteria and between the base criteria and the situational criteria, as per Table 8. These relationships are determined by iterating through the list of situational criteria and considering each criterion against all others in a pair-wise fashion.

**Step 2:** Construction of a methodology model: use the Analysis workflow of S-SMEP [Uzunov et al., sub2] (see Section 2.2 for background on the approach) to construct a pattern-based model of the target methodology. First
SPPF patterns are identified in the target methodology, beginning with the phase patterns and proceeding to finer levels of granularity as appropriate, and marked for inclusion in the methodology's process model. A generic life-cycle modifier is attached to each pattern according to the development phases and/or stages in which it is used. Similarly, elements from the (conceptual security framework) meta-model shown in Figure 2 (Section 2.2) are identified in the existing methodology's conceptual framework, and included in the framework model. After identifying each meta-model element (or after the whole conceptual framework model has been completed), the resulting framework model element(s) are linked with corresponding elements from the process model to form the complete methodology model (see [Uzunov et al., sub2] for more details).

Step 3: Measuring criteria: step through each quality factor, beginning with Structural validity, and apply the metrics for each of the criteria in turn. As discussed before, the metrics in Sections 4.3.1 and 4.3.2 provide a way of determining whether a particular criterion is satisfied, and how much. The aforementioned “way” defines a collection of sub-processes, which together form and specify the remainder of the assessment process.

Once a criterion has been measured, consider whether augmenting it with necessary information would be helpful. In theory all measures can be augmented, however, for some it is more useful than others, and one should take care not to over-clutter the presentation.

The relationships between criteria require should be treated in one of two ways:

- either by measuring one criterion in a relationship, followed by its related criterion in order; or
- by checking that the relationships holds and is valid (within what is found reasonable for the actual assessment) once all criteria have been measured – e.g. if criterion A and B are related via reverse implication, and both receive equally poor measures, then they should be re-assessed to ensure that the measures have not been misjudged.

Note that the same criteria can be associated with different factors, therefore, if a criterion has already been measured with respect to one factor it does not need to be measured again when considering another factor.

Step 4: Assessment results: tabulate the measures of the assessment with each criterion clearly related to the corresponding values. All measures should have an indication as to whether they are grades or differentiating classes (e.g. by using colours or different fonts). The results for the criteria should be grouped in some fashion befitting the purpose of assessment – according to factors for quality improvement, according to priority for comparison, and so forth.

Improvement:

Step 5: Determining areas for improvement: based on the assessment results, identify quality factors important for the project situation that have criteria with low grades. These factors constitute “weak spots” requiring improvement. It cannot be expected that all criteria with low grades can or should be improved; therefore, from the collection of lowly-graded criteria, select the most important (again, with respect to the project situation) – these become the primary criteria for improvement. Next, follow the relationships for each selected criterion to determine a set of secondary criteria for improvement. The resulting set of criteria effectively creates an “improvement criteria profile”, i.e. a specific situational criteria profile for the purpose of improving a certain subset of a methodology's features. The aim of the subsequent steps will be to improve the measures on these criteria. As an aside, it is important to note a number of the secondary criteria may already have positive measures, and so do not need to be improved; however, it is necessary for them to be included for consideration in the aforementioned “improvement profile” so that their corresponding positive measures are not somehow lost in the process of re-engineering.

Step 6: Re-engineering the methodology – requirements: apply S-SMEP (Situational Security Methodology Engineering Process – [Uzunov et al., sub2]) in its tailoring workflow, using the given methodology as a base methodology. In doing so, the quality criteria that were chosen in the previous step of QAIP are transformed into requirements (or goals, from which requirements can be derived) for the Methodology Initiation S-SMEP phase that must subsequently be satisfied during the Methodology Design S-SMEP phase (Step 7 below). The specifications of the requirements should stem directly from, or be simple conversions of, the original criteria descriptions, with the desired measures incorporated. In the conversion process, the primary criteria for improvement from Step 5 above become primary requirements, while the secondary criteria are interpreted as secondary requirements according to the nature of their relationships, whereby:

- implication (I) means support (i.e. requirement A → requirement B means that the satisfaction of A will support the satisfaction of B);
- reverse implication (R) means hindrance (i.e. requirement A → requirement B means satisfying A will hinder the satisfaction of B) that should be traded-off;
- specialization (S) and dependency (D) retain their original meaning (i.e. requirement A is a derivative requirement / specialization of requirement B; or requirement A depends on the satisfaction of requirement B for its own satisfaction).

Besides utilizing the criteria in the “improvement profile” referred to in QAIP Step 6, new abstract and/or concrete criteria from the quality framework can be specialized and instantiated as per QAIP Step 1 to create new
requirements, more specific requirements etc. for the Methodology Initiation S-SMEP phase.

**Step 7: Re-engineering the methodology – design and construction:** apply S-SMEP to re-design the concrete methodology model by using the four tailoring operators (Replacement, Deletion, Augmentation, Specialization), which implies re-instantiating particular process pattern and/or conceptual artefacts, changing one or more pattern instances with customized abstract process fragment(s), introducing new process elements or artefacts etc. The metrics for each criterion (requirement) can determine what must be done during this phase – with indicators being re-interpreted as features for satisfying the requirement – in much the same way as the metrics determined the assessment part of QAIP (Step 3). Since satisfying every single requirement may turn out to be too difficult or impossible for the given methodology, it may be necessary to iterate S-SMEP and hence perform QAIP Steps 5 and 6 above again.

**Step 8: Re-assessment:** repeat Steps 3 and 4, to re-assess the resulting methodology. The situational criteria profile that was constructed in Step 1 can be extended by adding any requirements created during Steps 5 and 6 that are, or can be, converted to criteria from the quality framework, and which are not present in the base or original profile. If the updated measures for the criteria identified previously are not satisfactory, then Steps 5 to 7 should be repeated.

## 5 Assessing methodology quality

In this section we apply the assessment steps of QAIP set out above, by first constructing a situational criteria profile defined by a specific target system type, and subsequently using three existing methodologies – namely, the architecture-driven methodology of Ali et al. [Ali et al., 2009]; SERENITY [Maña et al., 2006], and the pattern-driven methodology of Uzunov et al. [Uzunov et al., sub2] (based on [Fernandez et al., 2006]) – as case studies. We should point out that there are no requirements or constraints on the target methodologies to which our quality framework and assessment/improvement process can be applied, and while we have chosen the three aforementioned methodologies as case studies, we could equally well have chosen other existing model-based methodologies, or even code-based methodologies, such as those of [Flechais et al., 2003] or [Breu et al., 2004].

### 5.1 Project situation: a very brief description

For our context, the project situation will be almost entirely circumscribed by the specifics of the target system (described in more detail in [Uzunov et al., sub3]), which is a distributed, collaborative application. Since the application can be realized using either client/server or peer-to-peer architectural styles, we would like the security methodology to be capable of addressing the specifics of both general distributed and peer-to-peer systems. The development approach used for the project has a software architectural focus (with UML used as the architecture description language), hence the methodology should be model-based, with explicit specification of relevant artefacts being of some importance.

### 5.2 Step 1: Creating situational criteria profiles for general and peer-to-peer distributed systems

The aforementioned system- (and project-) specific considerations give rise to two criteria profiles: one for general distributed systems, and one for peer-to-peer systems specifically, with the latter adding several new criteria to the former. For general distributed systems, we posit five additional criteria (and associated metrics) to the base profile; for peer-to-peer distributed systems, we augment the latter profile with five more criteria (total of ten). This is shown in Table 9, where the first three rows constitute the general distributed systems profile, while the whole table – including they grey-background row – constitutes the peer-to-peer profile. The criteria were inspired in part by the evaluation of pattern-driven methodologies in [Uzunov et al., 2012a], which is performed within the scope of security pattern application. They also reflect our view on what constitutes an important set of criteria for the given project situation.

Two remarks are in order regarding the profiles. Firstly, several criteria that may appear to be incorrect specializations (with incompatible properties) are in fact specialized from criteria that were not included in the profile, since the latter were deemed too broad and hence would have been redundant – i.e. the criteria under question are final criteria in a series of specializations, with all other, parent criteria excluded (see the end of Section 3.2). Thus, for example, System applicability (general distributed systems – artefacts – offensive) (CF; w; S) is not a specialization of System applicability (general distributed systems) (SP; p; O) (in the profile), as may appear, but of System applicability (general distributed systems – artefacts) (CF; w; S), which is itself a specialization of System applicability (general distributed systems) (CF; w; S) – neither of which were included in the profile.

As a second remark, it is worth pointing out that both criteria profiles in this section are relatively coarse grained; it is certainly possible for finer-grained criteria to be incorporated (at the expense of additional space in this paper, and additional effort for the methodology assessors), such as the coverage of the artefacts, or their level of documentation, or even whether the analysis stage activities take into account some specific feature of our target system types. We have chosen the current level of granularity as it illustrates the use of situational profiles without
adding additional overhead implied by large numbers of criteria.

<table>
<thead>
<tr>
<th><strong>QF</strong></th>
<th><strong>Conceptual security framework criterion name and brief description</strong></th>
<th><strong>Brief summary of metric for determining satisfaction</strong></th>
<th><strong>Possible values</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>C, A</td>
<td>System applicability (general distributed systems) (SP; p; O): whether processes explicitly taking into account distribution exist.</td>
<td>Analyze each of the activities specified in the pattern instances of the methodology model, as well as the conceptual artefacts to determine whether distribution is accounted for. Consider in particular the scope and nature of design-stage (Des) activities, since this is when distribution-related concerns generally become manifest.</td>
<td>no, partial, yes {activities} {artefacts}</td>
</tr>
<tr>
<td>C, V, A</td>
<td>System applicability (general distributed systems – artefacts – offensive) (CF; w; O): whether there are offensive artefacts taking into account the nature of general distributed systems.</td>
<td>Consider the range of offensive artefacts of the types used in the methodology. The artefacts should take into account both the network and host levels, including consideration of local and remote attacks, protocol stack vulnerabilities and communication threats (as applicable) to be measured as “good”. “Excellent” implies more concerns are addressed, “average” implies less.</td>
<td>unspecified; poor, average, good, excellent</td>
</tr>
<tr>
<td>C, V, A</td>
<td>System applicability (general distributed systems – artefacts – defensive) (CF; w; O): whether there are defensive artefacts taking into account, as a minimum, distributed authentication, authorization as well as secure network communication concerns.</td>
<td>Consider the range of defensive artefacts of the types used in the methodology. A value of “good” implies that the minimum security concerns are covered by relevant artefacts, namely: distributed authentication, distributed authorization as well as secure network communications. Additional solutions for a distributed context result in an “excellent” measure, and “average” when only one or two of the aforementioned concerns are explicitly addressed.</td>
<td>unspecified; poor, average, good, excellent</td>
</tr>
<tr>
<td>V</td>
<td>Broadness of scope (general distributed systems – artefacts – offensive) (CF; p; S): whether the offensive artefacts cover a wide variety of distributed system-specific security concerns.</td>
<td>Consider the specifications of the relevant offensive artefacts, and judge their range in light of the metric for System applicability (general distributed systems – artefacts – defensive) (CF; w; O).</td>
<td>unspecified; poor, average, good, excellent</td>
</tr>
<tr>
<td>V</td>
<td>Broadness of scope (general distributed systems – artefacts – defensive) (CF; p; S): whether the defensive artefacts cover a wide variety of distributed system-specific security concerns.</td>
<td>Consider the specifications of the relevant defensive artefacts, and judge their range in light of the metric for System applicability (general distributed systems – artefacts – defensive) (CF; w; O).</td>
<td>unspecified; poor, average, good, excellent</td>
</tr>
<tr>
<td>C, A</td>
<td>Specific artefacts (peer-to-peer systems – defensive) (CF; w; O): whether there are explicit defensive artefacts for peer-to-peer systems.</td>
<td>Consider the range and nature of relevant defensive artefacts, and determine whether there are any that apply specifically for the peer-to-peer context.</td>
<td>unspecified; none, few, many {list}</td>
</tr>
<tr>
<td>C, A</td>
<td>Specific artefacts (peer-to-peer systems – offensive) (CF; w; O): whether there are explicit offensive artefacts specific to peer-to-peer systems.</td>
<td>Identical to the metric for Specific artefacts (peer-to-peer systems – defensive) (CF; w; O), except for offensive artefacts.</td>
<td>unspecified; none, few, many {list}</td>
</tr>
<tr>
<td>V</td>
<td>Broadness of scope (peer-to-peer systems – artefacts – offensive) (CF; p; S): whether the defensive artefacts cover a wide variety of distributed system-specific security concerns.</td>
<td>Consider the specifications of the relevant offensive artefacts, and judge their range in light of the metric for Specific artefacts (peer-to-peer systems – offensive) (CF; w; O).</td>
<td>unspecified; poor, average, good, excellent</td>
</tr>
<tr>
<td>V</td>
<td>Broadness of scope (peer-to-peer systems – artefacts – defensive) (CF; p; S): whether the defensive artefacts cover a wide variety of distributed system-specific security concerns.</td>
<td>Consider the specifications of the relevant defensive artefacts, and judge their range in light of the metric for Specific artefacts (peer-to-peer systems – defensive) (CF; w; O)</td>
<td>unspecified; poor, average, good, excellent</td>
</tr>
</tbody>
</table>
Table 9: Situational criteria profile for general distributed systems

The relationships between the criteria in the situational profiles are shown in Table 10.

Table 10: Criteria relationships in the distributed and peer-to-peer system-specific situational profiles

5.3 Step 2: Constructing methodology models

In presenting the pattern-based methodology models constructed by following Step 2 of the assessment process we use the format established in [Uzunov et al., sub2], where the model is accompanied by brief descriptions of the process – according to the SPPF phase process patterns – and (major) conceptual model elements. The notation used – based on [Van de Weerd and Brinkkemper 2009] and, ultimately, on UML – is explained in the original paper [Uzunov et al., sub2].
5.3.1 The methodology of Ali et al.

The methodology of Ali et al. [Ali et al., 2009] is an architecture-driven security methodology focusing on SOA-based systems, with strong emphasis on formal verification. We describe its security process and several major conceptual artefacts in Tables 11 and 12 below, which accompany the pattern-based model in Figure 4.

<table>
<thead>
<tr>
<th>Security process pattern (P)</th>
<th>Generic life-cycle modifier</th>
<th>Security process description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecReq</td>
<td>Des</td>
<td>AdvMod (S): after constructing a high-level architectural model of a system with a specific Petri net and temporal logic based ADL (Architecture Description Language), threat modeling is undertaken, where the threats to vulnerable assets and access points (as per the architectural model) are categorized according to Microsoft's STRIDE (see [Swiderski and Snyder, 2004]).</td>
</tr>
<tr>
<td>CounterIntro</td>
<td>Des</td>
<td>CounterIdentif (S): countermeasures are introduced directly, based on expert opinion (as per the task process pattern with the same name), as a result of the previous threat model.</td>
</tr>
<tr>
<td>CounterIntro</td>
<td>Des</td>
<td>SecMod(S): architectural mitigations (as countermeasures), logical constraints and property specifications (using LTL – linear temporal logic – and informal descriptions) are translated to equivalent Petri net behavioural models and added to the architecture model as actual components to elaborate it further. Unless the mitigations are COTS components (modeled as “black-boxes” with just interfaces/ports), the threat modeling (AdvMod pattern instance, as above) is re-iterated to refine their internal structure.</td>
</tr>
<tr>
<td>CounterIntro</td>
<td>Des</td>
<td>SecVerif (S): once the architecture with its components, connectors and constraints is completed, the architectural model is translated into an SMV (Symbolic Model Verifier) ready form. The security properties captured by the threat model (in LTL) are similarly translated to ensure that the properties hold in the presence of the corresponding threats (translated as a set of parameters influencing the model-checker). The counterexamples generated by the SMV model-checker are used to refine the architecture and the threat model iteratively.</td>
</tr>
<tr>
<td>SecImpl</td>
<td>Impl</td>
<td>Cod (S) weakly supported: the architectural models are realized via manual coding, however, there is no guidance on how to best realize the models – which are by necessity formal – so that they adhere to the rigorous specifications. Such guidance would not be required in the case of less formal models, hence the weak support for this stage.</td>
</tr>
</tbody>
</table>

Table 11: Security (phase) process patterns in the methodology of Ali et al.

<table>
<thead>
<tr>
<th>Conceptual security artefact</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRIDE categories</td>
<td>A high level threat schema (see [Swiderski and Snyder, 2004]), which can be used by security-aware developers to classify threats.</td>
</tr>
<tr>
<td>Countermeasure list</td>
<td>An implicit knowledge (reliant on the literature or security experts) of security countermeasures converted to Petri-net based models that can be included in the software architecture.</td>
</tr>
</tbody>
</table>

Table 12: Some major conceptual security artefacts in the methodology of Ali et al.
SERENITY is a methodology for addressing the security and dependability issues peculiar to Ambient Intelligence (AmI) environments, which can be characterized by possessing a high degree of dynamism, heterogeneity and, above all, distribution. At the center of the methodology is a two-part framework providing both development and run-time support for the selection and implementation of security and dependability solutions. The pattern-based model of SERENITY is shown in Figure 5, with the accompanying explanation of the relevant SPPF pattern instances and major conceptual artefacts in Tables 13 and 14.

**Figure 4: Pattern-based model of the methodology of Ali et al.**
During the early development stages, developers identify the necessary security and dependability (S&D) properties from the target application's security goals. A custom language providing simple constructs for expressing, amongst other features, (canonical) S&D properties and contexts is used to specify the security (and dependability) requirements, which are later used as semantic queries in a development framework (the SERENITY development framework or SDF) to determine a set of conforming S&D solutions that can be selected. Requirements are thus prescribed, as opposed to being arrived at by adversary modeling (AdvMod pattern).

**SecMod (S):** a UML profile is used for notating required S&D properties on classes during analysis.

**CounterIdentif (S):** the requirements-as-queries from the earlier stages are used in conjunction with the SDF to determine the set of all available S&D solutions for selection, thereby establishing a conceptual mapping from S&D requirements expressed as required S&D properties to S&D solutions providing those properties. This mapping is realized by the selection of an appropriate solution at some level of abstraction (i.e. Class, Pattern or Implementation). The higher the level, the more choice is left for the SERENITY run-time.
framework algorithms to make the decision as to what security solution is best during runtime. Such an approach to countermeasure identification realizes the structured countermeasure identification task modifier from SPPF [Uzunov et al., sub2].

**CounterIntro** Des **SecMod (S)**: the same UML profile used for notating S&D properties during analysis is also used for representing relevant S&D solutions (as single classes) on UML models during design. This representation is purely to help developers, and does not imply transformations, hence it is a realization of the model task modifier applied to the SecMod stage SPPF pattern.

**SecImpl** Impl **Cod (S)**: During implementation, all applications must use the underlying interfaces and functions defined by the S&D classes and patterns as provided by the SERENITY framework implementation. A Java API exists for this purpose, which provides handlers to applications to make calls on the Executable Components instantiated in the run-time framework instead of communicating with them directly.

**SecAdmin** Deploy The SRF provides automated monitoring capabilities for the security solutions during runtime. When the conditions or environment of the system change, the monitoring rules contained in the S&D solutions can potentially be triggered, leading to a replacement of the solutions with a different one (e.g. a stronger cryptographic algorithm). While not a methodological feature as such, monitoring has implications on how S&D patterns are specified and is connected with the overall aims of SERENITY, making it an essential feature.

**Table 13: Security (phase) process patterns in SERENITY**

<table>
<thead>
<tr>
<th>Conceptual security artefact</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S&amp;D Solutions</td>
<td>Security &amp; Dependability solutions [Gallego-Nicasio et al., 2009; Serrano et al. 2008], are the (abstract) vehicle for security mechanisms in the broadest sense. S&amp;D solutions lie at the center of the SERENITY approach, and can be of several different types, which are structured hierarchically.</td>
</tr>
<tr>
<td>S&amp;D Classes</td>
<td>The highest level of the S&amp;D Solution hierarchy, S&amp;D Classes guarantee particular semantics and a common interface, and abstract the relevant security and dependability properties provided by a collection of S&amp;D Patterns.</td>
</tr>
<tr>
<td>S&amp;D Patterns</td>
<td>S&amp;D Patterns are concrete, individual specifications of solutions with precise formal semantics having multiple possible implementations.</td>
</tr>
<tr>
<td>S&amp;D Implementations</td>
<td>Represent specifications of run-time solutions, which are realized as executable components in the SERENITY run-time framework.</td>
</tr>
<tr>
<td>S&amp;D Requirements</td>
<td>A security and dependability requirement specified in a custom formalized language [Serrano et al., 2009a].</td>
</tr>
<tr>
<td>S&amp;D Property</td>
<td>A class of security and/or dependability policies, e.g. confidentiality, or a combination of such classes.</td>
</tr>
</tbody>
</table>

**Table 14: Some major conceptual security artefacts in SERENITY**

5.3.3 The methodology of Uzunov et al.

The methodology of Uzunov et al. [Uzunov et al., sub2] is a tailored form of the methodology of Fernandez et al. [Fernandez et al., 2006], arrived at by applying our approach to engineering security methodologies. As with the other methodologies, we provide a summary of the security process and conceptual framework aspects in Tables 15 and 16, which accompany the pattern-based model shown in Figure 6.

**Table 15: Security process description**

<table>
<thead>
<tr>
<th>Security process pattern (P)</th>
<th>Generic life-cycle modifier</th>
<th>Security process description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecReq</td>
<td>Req</td>
<td>AdvMod (S): consideration of security starts from the requirements development phase with an analysis of use cases, individually or in sequence (workflows), to determine a set of possible threats and thereby derive a set of security requirements; and to determine a set of minimal rights for each subject (or role) for access control. The threats used to determine how the use case activities can be subverted rely on expertise.</td>
</tr>
</tbody>
</table>
CounterIntro An  

**SecMod (S):** Secure Semantic Analysis Patterns (SAPs [Fernandez and Yuan, 2000]) are used during the analysis phase, where appropriate, to create a secure conceptual model efficiently. The use case analysis results from before are carried over to determine a set of abstract security patterns that collectively embody a subset of the security requirements. These include at least an overall access control model, which is realized by repeated instantiations of a desired (abstract) access control pattern, realizing the authorization rights per subject (or role) determined from before.

SecReq Des  

**AdvMod (S):** security attacks are considered to the software architecture during design as per the process in [Uzunov et al., 2013]. The architecture is decomposed and threats from a threat library and taxonomy are related to appropriate architectural contexts (attacks task pattern), both for general distributed systems and for peer-to-peer systems as needed, systematically generating a set of design-level security requirements (enumerate task pattern). A record is kept of the threat patterns used during this activity.

CounterIntro Des  

**CounterIdentif (S):** the total set of security requirements from both instances of the AdvMod pattern are mapped to appropriate groups of security patterns with the help of a table relating requirement classes to distributed-system-specific concerns (the latter classifying the patterns [Uzunov et al., 2012a]). Additional security patterns can be chosen based on the requirements and/or the abstract security patterns present in the conceptual model from earlier (development) phases.

CounterIntro Des  

**SecMod (S):** guided as appropriate by the architectural decomposition layers, security patterns satisfying the (security) requirements are instantiated into the software architecture to create an initial security architecture, which can be refined by appropriately combining pattern participants, as well as by considering new threats arising from the introduction of the patterns (iteration of previous security stages during Design).

CounterIntro Des  

**SecVerif (S):** after the security architecture has been constructed, it is verified against the initial set of secure use cases, ensuring that each of them is properly addressed.

SecImpl Impl  

**Cod (S) and Map (S):** during security implementation, the necessary countermeasures (as patterns) are realized as software units or mapped to COTS components.

SecImpl Impl  

**SecVerif (S):** once the relevant parts or the whole of the system has been implemented, a set of misuse patterns that realize each threat pattern documented during the Security Requirements Determination phase are collected and used for penetration testing (penetration test task pattern). Combined with developer-driven inspections and reviews (manual reviews task pattern), the verification activity helps to determine whether the countermeasures for each threat have been implemented correctly.

---

Table 15: Security (phase) process patterns in the methodology of Uzunov et al.

<table>
<thead>
<tr>
<th>Conceptual security artefact</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misuse patterns</td>
<td>Misuse patterns [Fernandez et al., 2009b] describe in detail how an attack is performed (including what software components are required) from the attacker's perspective, which patterns can mitigate the attack, and how the attack can be traced once it occurs.</td>
</tr>
<tr>
<td>Security patterns</td>
<td>Security patterns encapsulate countermeasures in a re-ususable format suitable for use during architecture development.</td>
</tr>
<tr>
<td>Threat patterns</td>
<td>Threat patterns [Uzunov and Fernandez, sub1] encapsulate security threats (potential attacks) and allow developers to consider those threats in well-defined architectural contexts.</td>
</tr>
<tr>
<td>Threat taxonomies</td>
<td>The (pattern-based) threat taxonomies of Uzunov and Fernandez [Uzunov and Fernandez, sub1] aggregate structured collections of threat patterns for general distributed systems as well as (derived) patterns for specific distributed system/technology contexts.</td>
</tr>
<tr>
<td>Security architecture</td>
<td>The set of all security measures at the design level together constitute a security architecture, i.e. a slice of the software architecture dealing with security.</td>
</tr>
</tbody>
</table>
Semantic analysis pattern A type of analysis pattern [Blaimer et al., 2010], which combines a ready-made domain/business design with (abstract) security patterns.

Table 16: Some major conceptual security artefacts in the methodology of Uzunov et al.

Figure 6: Pattern-based model of the methodology of Uzunov et al.
5.4 Steps 3 and 4: Measuring criteria and assessment results

We performed step 3 of the assessment process, measuring all the criteria for each quality factor, and subsequently verified that the measures accord with any relevant criteria relationships (where appropriate). The final assessment results appear Table 17. A few remarks are in order regarding the interpretation of these results.

Since the context of some criteria, especially those pertaining to the conceptual framework aspect, is broader than what can be found in a given methodology (e.g. relating to offensive artefacts, when the methodology only employs one type of offensive artefact), we have applied the corresponding metric to a reduced context, which is indicated in the curly braces after the (augmented) measure. We further use the convention of placing a plus “+” after an artefact type to indicate the inclusion of all relevant supportive artefacts (for instances of that artefact type). All instances of the Coverage abstract criterion pertaining to the conceptual framework (CF) have been assessed with respect to an artefact class (either loosely specified or a well-specified meta-model element).

The colours in the table indicate grades, whereby light red [or light grey in black and white] indicates the equivalent of “poor” (lowest grade); yellow [or medium grey] indicates the equivalent of “average” (and sometimes “good” or “excellent” (again, depending on the context); and green [or dark grey] indicates the equivalent of “excellent” (highest grade). The final assessment results appear Table 17. A few remarks are in order regarding the interpretation of these results.

<table>
<thead>
<tr>
<th>QF</th>
<th>Quality criterion</th>
<th>Methodology of Ali et al.</th>
<th>SERENITY</th>
<th>Methodology of Uzunov et al.</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Specific activity sequence (valid pattern ordering) (SP; w; O)</td>
<td>well ordered</td>
<td>well ordered</td>
<td>well ordered</td>
<td>1</td>
</tr>
<tr>
<td>R</td>
<td>Compatibility (SPPF patterns) (SP; w; O)</td>
<td>fully compatible</td>
<td>fully compatible</td>
<td>fully compatible</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>Broadness of scope (encompassed development phases) (SP; w; O)</td>
<td>single-phase</td>
<td>multi-phase</td>
<td>multi-phase</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>Coverage (related development activities) (SP; w; S; O)</td>
<td>few {architectural modeling}</td>
<td>many</td>
<td>many</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>Specific activities (early security introduction) (SP; w; O)</td>
<td>no</td>
<td>partial {weak at Req.4n}</td>
<td>yes</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>Specific activities (threat assessment) (SP; w; O)</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>Specific activities (verification) (SP; w; O)</td>
<td>automated</td>
<td>none</td>
<td>manual</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>Documentation (security process) (SP; w; S)</td>
<td>poor</td>
<td>sufficient</td>
<td>sufficient</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td>Specific artefacts (basic security solutions) (CF; p; O)</td>
<td>implicit only</td>
<td>many {S&amp;D solutions}</td>
<td>many {security patterns}</td>
<td>9</td>
</tr>
<tr>
<td>C</td>
<td>System applicability (general distributed systems) (SP; p; O)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>System applicability (general distributed systems – artefacts – offensive) (CF; w; O)</td>
<td>unspecified</td>
<td>unspecified</td>
<td>yes</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>System applicability (general distributed systems – artefacts – defensive) (CF; w; O)</td>
<td>unspecified</td>
<td>partial</td>
<td>yes</td>
<td>12</td>
</tr>
<tr>
<td>C</td>
<td>Specific artefacts (peer-to-peer systems – offensive) (CF; w; O)</td>
<td>none</td>
<td>none</td>
<td>few</td>
<td>13</td>
</tr>
<tr>
<td>C</td>
<td>Specific artefacts (peer-to-peer systems – defensive) (CF; w; O)</td>
<td>none</td>
<td>none</td>
<td>many</td>
<td>14</td>
</tr>
<tr>
<td>V</td>
<td>Documentation (artefacts – offensive) (CF; p; S)</td>
<td>poor</td>
<td>good</td>
<td>good</td>
<td>15</td>
</tr>
<tr>
<td>V</td>
<td>Documentation (artefacts – defensive) (CF; p; S)</td>
<td>poor</td>
<td>good</td>
<td>good</td>
<td>16</td>
</tr>
<tr>
<td>V</td>
<td>Coverage (artefacts – offensive) (CF; p; S)</td>
<td>unspecified {networked system threats}</td>
<td>unspecified {networked system threats}</td>
<td>excellent {networked system threats}</td>
<td>17</td>
</tr>
<tr>
<td>V</td>
<td>Coverage (artefacts – defensive) (CF; p; S)</td>
<td>unspecified {generic security policies}</td>
<td>excellent {generic security policies}</td>
<td>good {generic security policies}</td>
<td>18</td>
</tr>
<tr>
<td>V</td>
<td>Broadness of scope (encompassed development phases) (SP; w; O) (*)</td>
<td>single-phase</td>
<td>multi-phase</td>
<td>multi-phase</td>
<td>19</td>
</tr>
<tr>
<td>V</td>
<td>Coverage (related development activities) (SP; w; S; O) (*)</td>
<td>few {architectural modeling}</td>
<td>structured {S&amp;D solutions}</td>
<td>partially structured {patterns}</td>
<td>20</td>
</tr>
<tr>
<td>V</td>
<td>Structure (artefacts) (CF; w; O)</td>
<td>partially structured {threat models only}</td>
<td>partial {systematic {SDF}}</td>
<td>partially systematic {mapping table}</td>
<td>21</td>
</tr>
<tr>
<td>V</td>
<td>Guidance (identifying countermeasures) (SP; w; S)</td>
<td>partial {threat models}</td>
<td>systematic {SDF}</td>
<td>partially systematic {mapping table}</td>
<td>22</td>
</tr>
<tr>
<td>V</td>
<td>Guidance (introducing countermeasures) (SP; w; S)</td>
<td>none</td>
<td>ad-hoc {expert-driven}</td>
<td>ad-hoc {pattern}</td>
<td>23</td>
</tr>
</tbody>
</table>

257
<table>
<thead>
<tr>
<th>Use of best practices (activities) (SP; w; S)</th>
<th>some {STRIDE threat modeling}</th>
<th>some {integrated tools}</th>
<th>some {reviews, penetration tests}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of best practices (artefacts) (CF; w; O)</td>
<td>some {threat artefacts}</td>
<td>many {S&amp;D patterns}</td>
<td>many {patterns}</td>
</tr>
<tr>
<td>Use of best practices (artefacts – encapsulation) (CF; p; O)</td>
<td>none</td>
<td>many {S&amp;D patterns}</td>
<td>many {security patterns, threat patterns}</td>
</tr>
<tr>
<td>System applicability (general distributed systems – artefacts – offensive) (CF; w; O) (*)</td>
<td>unspecified</td>
<td>unspecified</td>
<td>yes</td>
</tr>
<tr>
<td>System applicability (general distributed systems – artefacts – defensive) (CF; p; S)</td>
<td>unspecified</td>
<td>unspecified</td>
<td>yes</td>
</tr>
<tr>
<td>Breadth of scope (general distributed systems – artefacts – offensive) (CF; p; S)</td>
<td>unspecified</td>
<td>unspecified</td>
<td>excellent {threats}</td>
</tr>
<tr>
<td>Breadth of scope (general distributed systems – artefacts – defensive) (CF; p; S)</td>
<td>unspecified</td>
<td>unspecified</td>
<td>good {generic security policies}</td>
</tr>
<tr>
<td>Breadth of scope (peer-to-peer systems – artefacts – offensive) (CF; p; S)</td>
<td>unspecified</td>
<td>unspecified</td>
<td>excellent {threats}</td>
</tr>
<tr>
<td>Breadth of scope (peer-to-peer systems – artefacts – defensive) (CF; p; S)</td>
<td>unspecified</td>
<td>unspecified</td>
<td>poor</td>
</tr>
<tr>
<td>Expertise required (methodology) (SP, CF; w; S)</td>
<td>high</td>
<td>high</td>
<td>medium</td>
</tr>
<tr>
<td>Expertise required (methodology – threat modeling) (SP; p; S)</td>
<td>high</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Expertise required (methodology – introducing countermeasures) (SP; w; S)</td>
<td>high</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>Structure (artefacts) (CF; w; O) (*)</td>
<td>partially structured</td>
<td>structured {S&amp;D solutions}</td>
<td>partially structured {patterns}</td>
</tr>
<tr>
<td>Structure (activities – threat modeling) (SP; p; S)</td>
<td>partially structured</td>
<td>unstructured</td>
<td>structural</td>
</tr>
<tr>
<td>Structure (activities – introducing countermeasures) (SP; w; S)</td>
<td>partially structured</td>
<td>non-standard</td>
<td>standard</td>
</tr>
<tr>
<td>Known modeling language (at ReqAn and Des) (SP; w; O)</td>
<td>medium</td>
<td>unspecified</td>
<td>high</td>
</tr>
<tr>
<td>Known modeling language (artefacts) (CF; w; O)</td>
<td>low</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Efficiency (activities) (SP; w; S)</td>
<td>low</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Software support (methodology) (SP; w; O)</td>
<td>low</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>Reusability (generated security designs) (SP; w; O)</td>
<td>none</td>
<td>medium {S&amp;D patterns}</td>
<td>high {pattern-based designs, threats}</td>
</tr>
<tr>
<td>Guidance (identifying countermeasures) (SP; w; S) (*)</td>
<td>partial {threat models}</td>
<td>systematic {SDF}</td>
<td>partially systematic</td>
</tr>
<tr>
<td>Guidance (introducing countermeasures) (SP; w; S) (*)</td>
<td>none</td>
<td>ad-hoc {expert-driven}</td>
<td>ad-hoc {pattern relationships}</td>
</tr>
<tr>
<td>Use of best practices (activities) (SP; w; S) (*)</td>
<td>some {STRIDE threat modeling}</td>
<td>some {integrated tools}</td>
<td>some {reviews, penetration tests}</td>
</tr>
<tr>
<td>Use of best practices (artefacts) (CF; w; O) (*)</td>
<td>some {threat artefacts}</td>
<td>many {S&amp;D patterns}</td>
<td>many {patterns}</td>
</tr>
<tr>
<td>Use of best practices (artefacts – encapsulation) (CF; p; O) (*)</td>
<td>none</td>
<td>many {S&amp;D patterns}</td>
<td>many {security patterns, threat patterns}</td>
</tr>
<tr>
<td>Specific activities (verification) (SP; w; O) (*)</td>
<td>automated</td>
<td>none</td>
<td>manual</td>
</tr>
<tr>
<td>Specific activity approach (formal verification) (SP; p; O)</td>
<td>formal {using SMV checker}</td>
<td>none</td>
<td>informal</td>
</tr>
<tr>
<td>Software support (verified implementations) (CF; w; O)</td>
<td>none</td>
<td>software artefacts {SRF, certified S&amp;D artefacts}</td>
<td>none</td>
</tr>
<tr>
<td>Specific activities (security measurement) (SP; p; O)</td>
<td>no</td>
<td>partial {run-time only, SRF monitoring}</td>
<td>no</td>
</tr>
<tr>
<td>Reusability (generated security designs) (SP; w; O) (*)</td>
<td>low</td>
<td>medium {S&amp;D patterns}</td>
<td>high {pattern-based}</td>
</tr>
</tbody>
</table>
Table 17: Quality assessment results for the three case studies

5.4.1 Comparison and analysis of assessment results

In this sub-section we briefly illustrate how the methodologies can be compared based on the assessment results for each quality factor.

Correctness of construction (R): All the methodologies demonstrated correct construction, with no observable problems as per the relevant metrics.

Completeness (C): In comparison to the other two methodologies, the methodology of Ali et al. is least complete, in part because there is only a single publication dedicated to it, which does not offer extensive insight into the approach; and also because it only covers one development phase: design. It does, however, provide threat analysis and automated verification. None of the methodologies can be considered end-to-end, which has negative ramifications for the development activities covered, and hence for the overall completeness.

Comprehensiveness (V): Considering the comprehensiveness factor, the methodology of Ali et al. measures uniformly low, due to its complete reliance on implicit artefact specifications (indeed, any methodology that does not specify or explain its defensive artefacts would measure similarly). The provision of guidance in identifying and introducing countermeasures in the other two methodologies, as well as the use of best practices, lead to a higher comprehensiveness factor.

Usability (U): The high expertise requirements – due again to the lack of artefact specifications, and hence reliance on dedicated security experts – and non-standard (i.e. not widely used) modeling language employed by the methodology of Ali et al. considerably lower its usability factor. Distillation of best practices and expert knowledge in patterns in SERENITY and the methodology of Uzunov et al., on the other hand, provides improved usability. SERENITY further provides tool support, which has a positive bearing on efficiency. Efficiency in SERENITY is offset negatively, however, by the complete lack of guidance and structure in the threat modeling process (and hence reliance on expertise).

Adaptability (A): Only the methodology of Uzunov et al. provided any adaptability features, being conditionally flexible with respect to the (distributed) system type.

Assurance (S): SERENITY provides a reasonable degree of assurance, especially through tool and runtime framework support. The methodology of Ali et al., on the other hand, attempts to provide assurance through its formal verification features. Least assurance is provided by the methodology of Uzunov et al., which relies solely on the re-use of evaluated designs.

5.4.2 Discussion

The assessment results clearly indicate that each methodology has strengths and weaknesses that should be considered prior to using a given methodology on a real project. Determining the weight of each criteria is dependent on the situation at hand. For example, the lack of tool support in the methodology of Uzunov et al. may be seen a deficiency that would render the methodology less desirable (especially in industry settings) than the benefits gained by other features (cf. [Uzunov et al., 2012b]). Similarly, a situation requiring a methodology applicable to Ambient Intelligence would certainly give greater weight to the positive features of SERENITY as opposed to the features of the other methodologies; and a project utilizing Petri nets as the main architectural formalism would benefit more from the methodology of Ali et al. (although the low usability of the latter methodology will be an inhibiting factor in all situations). These observations also support our view that assigning quantitative measures to quality is of less value than qualitative grades, which form a better basis for comparing and evaluating methodologies (cf. Section 3.2).
With respect to our project situation – outlined in Section 5.1, and further defined by the situational criteria profiles in Section 5.2 – the methodology of Uzunov et al. is the most suitable of the three case studies chosen. The other two methodologies are specific for particular types of distributed systems (Adaptability factor); and either imply high expertise requirements (hence low usability) with non-standard modeling languages (the methodology of Ali et al.) or have strong dependency on implementation frameworks (SERENITY). Of course, it is important to note that the methodology of Uzunov et al. was constructed specifically for our project situation, hence its suitability is an inherent feature. This fact does not actually diminish the importance of the assessment results, but rather, confirms their importance inasmuch as they indicate a methodology constructed to be suitable is indeed suitable (and, in fact, is the most suitable of the methodologies assessed).

6 Improving methodology quality

In the previous section we presented our approach for assessing methodology quality, and applied it on three security methodologies taken from the literature, which were shown to be deficient with respect to one or more quality factors. In this section we show how these quality factors can be improved by supplementing the methodologies with a collection of generic conceptual security artefacts, some of which have already been encountered in the methodology of Uzunov et al. in Section 5.3.3, and by incorporating relevant abstract process fragments. We begin by summarizing the latter artefacts and fragments below in Sections 6.1 and 6.2 respectively, after which we follow the quality improvement steps of QAIP (Steps 5 to 8) for the methodology of Uzunov et al. in detail (Section 6.3); and briefly mention how QAIP can be applied to the other two methodologies in Section 6.4.

6.1 Generic conceptual security artefacts

The generic artefacts summarized in this section aim to cover two main elements of a conceptual security framework for any methodology: threats and their dual solutions (countermeasures).

**Architectural decomposition framework** [Uzunov et al., 2013]: architectural decomposition is a form of characterization, which, when complemented by an analysis process, provides structure for guiding the incorporation of early non-functional (in our case security) requirements and the determination of new requirements. The framework itself resides at a complementary level of abstraction to the software architectural models, and is stratified into three layers, with the first containing a conceptual model of distributed systems, usable as a meta-model for creating derived architectural models; the second containing a simple, layered architectural meta-model, which allows separation of the architecture according to areas of functionality (functionality decomposition layers); and the third containing a set of meta-modeling elements (technical realization abstractions) that detail the decomposition.

**Threat library and taxonomy** [Uzunov and Fernandez, sub1]: a two-level collection of threat patterns for general distributed systems, encompassing both system threats and meta-security threats (threats to the security infrastructure itself). The base taxonomy for distributed systems can be extended by specializing individual patterns for different architectural contexts to produce system–technology-specific taxonomies. These taxonomies (like the base taxonomy itself) combine the benefits of a structured library and a STRIDE-like classification scheme, and support threat modeling by allowing non-expert developers to consider a wide variety of threats (threat pattern instances) tailored to particular applications. The patterns constituting threat taxonomies can be used to complement any of the existing threat and/or attack modeling approaches.

**Security solution frames** [Uzunov et al., sub1, Uzunov and Fernandez, sub2]: are solution structures that encapsulate and organize security patterns into different levels of abstraction (vertically) and different facets of a root security policy (horizontally), thereby providing guidance in realizing security policies from an abstract conceptual to a concrete design level. Solution frames exist for authorization [Uzunov et al., sub1], supporting the construction of customized conceptual authorization models and complementary enforcement architectures; as well as for the design of authorization policy/rule management infrastructures [Uzunov et al., sub1], secure two-party communications, identity management (authentication only), and cryptographic key management [Uzunov and Fernandez, sub2]. The security patterns encompassed by the solution frames can be replaced by, or related to, other conceptual security artefacts – e.g. security aspects [Georg et al., 2009] – while retaining the overall intrinsic structure of solutions (as specific security policy encapsulations), making solution frames, like the other generic artefacts, methodology agnostic.

6.2 Abstract process fragments

Conceptually, abstract process fragments are specializations of specific macro-process patterns and concrete process fragments extracted from existing security methodologies or parts thereof. They can be stored in a repository and are particularly useful for realizing SPPF patterns during methodology construction. Below we briefly outline the solutions of four such fragments, with the granularity and references to where the fragment was generalized from appearing in brackets.

**Decomposition-based threat modeling** (Stage granularity, [Uzunov et al., 2013]): Using the architectural decomposition framework (generic artefact) implies a decomposition process, together with an accompanying
analysis process. The process of decomposition follows the structure of the framework: first a derived model is (or at least can be – it is optional) created using the provided abstractions; the different parts of the derived model (or actual architecture, if a derived model was not created) are subsequently mapped to the functionality decomposition layers, following the pre-defined meta-model (essentially mapping model elements to meta-model elements); and finally, the decomposition is refined by mapping derived model (or architecture) elements to the technical realization abstractions. The threat analysis process proposed in [Uzunov et al., 2013] attempts to exploit the features of the decomposition by considering security threats at each functionality layer and for all the realization abstractions, thus providing a whole threat modeling approach.

**Table-based policy mapping (Task granularity, [Rosado et al., 2008, Delessy and Fernandez, 2008]):** Security requirements (in the problem-space) or policies (in the solution-space) are mapped to countermeasures using a table of pre-defined pairs (requirement/policy class, countermeasure class). Finer-grained requirement/policy mappings are also possible, whereby a table maps or translates one set of requirements/policies or requirement/policy classes to another, thus creating a kind of “adapter”. All the mapping pairs in the table must be mined from the literature.

**Utilizing a development artefact repository (Task granularity, [Rosado et al., 2008, Gutiérrez et al., 2006]):** Applying a security methodology means that a number of security-related development artefacts are generated in conjunction with the development process. These artefacts can be model instances, structured collections of patterns, security configurations and others. Using this fragment, all relevant development artefacts are stored in a repository for re-use on later projects. A context or other meta-information is attached to each artefact so that it can be identified appropriately and used in the correct situations. Prior to performing an activity that will use or produce artefacts of the types stored in the repository, the repository is queried and the relevant artefacts extracted for consideration.

**Guided security solution refinement (Task granularity, [Uzunov et al., sub1]):** Using the solution frame artefacts allows the selection of progressively more concrete security patterns across the development life-cycle. This implies a process for each solution instance: first abstract patterns are selected, which define the concepts of the solution and an abstract architecture; then the abstract architecture is refined in successive levels of concreteness, leading to a detailed design; and finally, a number of technology realizations are chosen for the design to be transferred to implementation. In this way developers are guided in introducing particular security solutions in a step-wise fashion. The latter approach is encapsulated in micro-process patterns in each solution frame, and the **Guided security solution refinement** fragment in turn encapsulates the task of selecting these patterns and instantiating them into the flow of relevant (countermeasure introduction) activities.

### 6.3 The methodology of Uzunov et al.

#### 6.3.1 QAIP Step 5: Determining areas for improvement

The quality factors most important for our project situation are completeness and comprehensiveness (especially with respect to general distributed systems); usability (for non-expert developers); and, to a lesser degree, adaptability. Considering the assessment results in Table 17, the criteria we will strive to improve in these factors are Nos. 21/36, 22/46, 23/47, 30, 35 and 38 (see last column). Other criteria, such as Nos. 3, 4, 13 and 44 are also important, but require extensions to the methodology that are beyond the scope of this paper, and hence will not be considered here (in this context it is also worth pointing out that it is not an absolute necessity for every single criterion to obtain an “excellent” grade in order for the methodology to be considered “good quality”). The criteria chosen for improvement become our primary criteria. Considering the criteria relationships (Sections 4.3.6 and 5.2) adds criteria Nos. 12, 18 and 56 (referring to Table 17) as secondary criteria for improvement – which in turn add criteria Nos. 26/50, 40, 41 and 42 (again, as secondary criteria). Because the implication relationship between the criteria Nos. 35 and 47 in Table 17 is two-way, we retain both criteria as primary.

#### 6.3.2 QAIP Step 6: Re-engineering the methodology – requirements

Having obtained a set of criteria, we can now convert them to requirements as part of enacting the stages of the S-SMEP Methodology Initiation phase.

#### 6.3.2.1 S-SMEP Requirements Elicitation and Analysis stage

The converted criteria result in the following requirements for re-engineering the methodology (cf. the original criteria descriptions in Section 4.3). Besides the initial converted criteria, we have added four finer-grained requirements relating to the methodology's conceptual framework to better measure the value of the re-engineering modifications. Two of the added requirements (Req10 and Req11) are specializations, and two (Req6 and Req7) are related via a dependency relationship with respect to the original criteria-turned-requirements; in all cases the newly created relationships appear in brackets next to the requirement number using the same notation seen in Section 4.3.6. The quality factor which the requirement is related to is shown in square brackets prior to it. The letters in brackets after the requirement number indicate whether the requirement is primary (p) or secondary (s). As per QAIP, Req8, while well satisfied by the methodology already, is included to ensure that re-engineering does affect
the corresponding features adversely.

Req1 (p): [V, U] Guidance (introducing countermeasures) (SP; w; S): the methodology should provide systematic guidance on how and where defensive security solutions should be applied.

Req2 (p): [V, U] Structure (activities – introducing countermeasures) (SP; w; S): Structure (activities – introducing countermeasures) (SP; w; S): the methodology's activities relating to the introduction of defensive security solutions should be fully structured in such a way as to aid developer productivity.

Req3 (s): [C, V, A] System applicability (general distributed systems – artefacts – defensive) (CF; w; O): the defensive artefacts employed by the methodology employs should take into account, as a minimum, distributed authentication, authorization as well as secure network communication concerns.

Req4 (p): [V] Broadness of scope (general distributed systems – artefacts – defensive) (CF; p; S): the defensive artefacts employed by the methodology should address a variety of different (specific) sub-concerns within each security concern outlined in Req3.

Req5 (s): [V] Coverage (artefacts – defensive) (CF; p; S): for each defensive artefact type, ensure that a broad range of scenarios or cases, stemming from generic security policies, are covered by corresponding artefact collections.

Req6 (s): (D → Req4, I → Req5): [V] Coverage (general distributed systems – artefacts – security solutions) (CF; p; S): for each security solution artefact type, ensure that a broad range of specific security policies are covered.

Req7 (s): (D → Req4): [V] Coverage (general distributed systems – artefacts – threats) (CF; p; S): for each threat artefact type, ensure that a broad range of specific security policies are covered.

Req8 (s): [A] System applicability (generality) (SP, CF; w; O): the methodology as a whole should have mixed applicability to both general distributed systems and peer-to-peer systems.

Req9 (p): [V, U] Structure (artefacts) (CF; w; O): the various conceptual framework artefacts should have inherent structure individually, as well as when taken together.

Req10 (s): (S → Req9): [V, U] Structure (artefacts – offensive artefacts) (CF; p; O): all offensive conceptual framework artefacts should have inherent structure individually, and when taken together (relationships).

Req11 (s): (S → Req9): [V, U] Structure (artefacts – defensive artefacts) (CF; p; O): all defensive conceptual framework artefacts should have inherent structure individually, and when taken together (relationships).

Req12 (p): [U] Expertise required (introducing countermeasures) (SP; w; S): the level of security expertise required to introduce security features (especially security solutions) into a target system should be low.

Req13 (p): [V, U] Guidance (identifying countermeasures) (SP; w; S): the methodology should include a fully systematic approach to identifying appropriate defensive security solutions of various degrees of granularity.

Req14 (s): [U] Expertise required (methodology) (SP, CF; w; S): the overall level of security expertise required to apply the methodology – its activities and related artefacts – should be as low as possible.

Req15 (s): [U] Understandability (artefacts – offensive) (CF; p; S): the ease with which the offensive artefact specifications can be understood, both with respect to security and non-security properties, should be high.

Req16 (s): [U] Understandability (artefacts – defensive) (CF; p; S): the ease with which the defensive artefact specifications can be understood, both with respect to security and non-security properties, should be high.

Req17 (s): [U] Known modeling language (artefacts) (CF; w; O): the modeling languages used in the various security artefacts should be well-established and widely used.

Req18 (s): [V, U] Use of best practices (artefacts – encapsulation) (CF; p; O): artefacts should as far as possible encapsulate their security features, distilling expert knowledge.

6.3.3 QAIP Step 7: Re-engineering the methodology – design and construction

In this step we use the metrics for the criteria corresponding to requirements Req1 to Req12 to guide the S-SMEP Design/Modeling and Construction phases.

6.3.3.1 Methodology Design/Modeling phase

We have already performed the relevant stages and tasks of S-SMEP to construct the initial methodology model, namely, the Selecting and organizing SPPF patterns stage with the Identifying patterns in base methodology task and Constructing an initial security process model task; as well as the Instantiating the conceptual framework meta-model stage. Below we describe the enactment of the remaining stages in the tailoring workflow.
Selecting existing methodology elements for tailoring (stage)

We apply the four S-SMEP generic operators to satisfy our requirements as described below. The actual realization of the tailoring changes (modification of the models, re-linking of elements etc.) is left to the Methodology Construction S-SMEP phase. We concentrate on SPPF patterns and conceptual framework artefacts used at the Design (des) phase, since this is where the effects of distribution are (fully) manifest.

Req3 to Req6 (applicability and broadness of defensive artefacts for distributed systems):

Solution frames (Section 6.1) are inherently applicable to general distributed systems, and those currently available in the literature ([Uzunov et al., sub1, Uzunov and Fernandez, sub2]) encompass generic security policies for all the concerns outlined in Req3. They also cover a wide range of specific policies within each of the latter concerns, thereby providing the necessary breadth of scope. Therefore, we apply Augmentation (operator) to the conceptual framework for the addition of the security solution frame generic artefact.

Req1, Req2 (guidance and structure of countermeasure introduction):

Solution frames contain micro-process patterns that closely follow the (frame) abstraction-level structure (Req2). The very structure of the solution frames guides developers from concepts to designs for a given security policy (Req1). To tag the inclusion of the corresponding micro-process patterns, we apply the Augmentation (operator) to the SecReq stage pattern at Design (des).

Req2, Req7, Req13 (structure of countermeasure introduction activities and threat coverage, guidance for countermeasure identification):

The above changes for Req1 and Req2 are focused on introducing countermeasures intro a target system's architecture; the actual mapping between threats and countermeasures is not considered. Currently this mapping is realized via a table of pairs – resulting from the Table-based policy mapping fragment (Section 6.2) – mapping security requirement classes to distributed-system-specific concerns, which in turn map to security patterns (the Distribution-aware classification scheme artefact in Figure 6 – see Section 5.3.3). The actual mapping from the threats to security concerns is left to expert/contextual judgement; moreover, the generic mitigation policies encapsulated in threat patterns are never utilized. Besides the lack of structure in this respect, the existing mapping activity and its corresponding table do not currently take into account the solution frames introduced by the previous changes (for Req3 to Req6). To address the aforementioned problems we apply Replacement and Augmentation (operators) to the Distribution-aware classification scheme artefact, indicating the addition of a table mapping requirements classes to threat mitigation policies (i.e. generic security policies, which are in turn linked to the solution frames); and the addition of another table mapping general policies to distributed-system-specific security concerns. In a similar vein, we apply Augmentation (operator) to the solution frame collection and threat taxonomy artefacts, indicating the introduction of yet another mapping table relating these artefacts in reverse, i.e. mapping a given solution frame (or its root policy – encapsulated in a security tactic) to a threat class or to a collection of individual patterns from the threat taxonomy. Finally, we apply Specialization (operator) to the CounterIdentif stage pattern at design (des), to indicate an updated mapping activity using the latter tables.

Req9 to Req11, Req13 (structure and broadness of artefacts):

The artefacts employed by the methodology (patterns, frames) are already well structured individually; when taken together, the patterns encompassed by solution frames, as well as the patterns classified according to the (to be introduced) mapping tables are similarly well structured. The previous changes for Req2 / Req7 / Req13 aim to increase the structure between threat patterns and security solutions, and thus also help to ensure the broadness of defensive artefacts available for use. What remains is the relationships between the (early) requirements and those stemming from the threat patterns, i.e. from the Adversary Modeling stage performed during Design (des) (see Fig 6). For this we apply Augmentation (operator) on the CounterIdentif stage, further to Specialization (operator) from before, so that it employs the table artefacts stemming from the Req2 / Req7 / Req13 tailoring changes for all (not just design-level) requirements. This new mapping activity would require that the requirements are “adapted” (as per the advantages of using the Table-based policy mapping fragment) to mitigating policies, which can then be used to determine appropriate security solutions. For this purpose we also apply Augmentation (operator) to the SecReq phase at Design (des) to indicate that all security requirements should be converted to an appropriate form prior to their use in the CounterIdentif activities.

Req12 (low security expertise for countermeasure introduction):

The addition of solution frames, which distill expertise, cover a wide range of security policies, and also provide guidance in introducing concrete security policies; and the introduction of additional structure to ensure cohesiveness between different artefacts help to lower the expertise required for introducing countermeasures, and hence contribute towards satisfying Req12. Therefore, we do not need to select any elements for tailoring beyond those already selected for Req1 to Req11.

Req14 (low security expertise overall)
As for as for Req12, the changes indicated thus far, especially for identifying and introducing countermeasures, should help towards a better satisfaction of this requirement, and do not imply any further tailoring.

Req8, Req15 to Req17 are already fully satisfied, and none of the modifications indicated thus far will adversely affect them. The addition of solution frames positively affects Req18 (already well satisfied in the original methodology), since they encapsulate patterns, which in turn encapsulate their security features.

6.3.3.2 Methodology Construction phase

At this phase we implement the tailoring operators from the Methodology Design/Modeling S-SMEP phase for the SPPF patterns and conceptual framework elements.

**Instantiating SPPF patterns (stage) and Realizing conceptual framework model elements (stage)**

– Augmentation (operator) to the conceptual framework for security solution frames: implies the simple addition of the relevant model element as an explicit structured security artefact collection instance. The solution structures are already specified in detail in the literature (hence this particular tailoring change can be classed as being fully implemented).

– Augmentation (operator) on the SecMod pattern at Design (des): we instantiate another SPPF instantiate task modifier and add it to the SecMod stage in parallel workflow with the existing task (thereby specializing SecMod). This added task will be realized by the Guided security solution refinement fragment (Section 6.2) – i.e. its sub-activities will use the micro-process patterns from relevant security solution frames, which in turn guide the instantiation of the solution frame's constituent security patterns.

– Replacement (operator) on the Distribution-aware classification scheme artefact: since the root policy of a given security solution frame is encapsulated in a related security tactic, we construct the first of our mapping tables with (requirement class, generic security policy) pairs – the Requirement class to policies table supportive artefact, linked to the Design-level security requirements artefact – where the requirement classes are taken from the old Distribution-aware classification scheme artefact, and the generic security policies are the mitigating policies of the threat patterns from the Threat taxonomy artefact.

– Augmentation (operator) on the Distribution-aware classification scheme artefact: we instantiate a new conceptual framework model element – Policies to distributed-system-specific concerns table – for the second table with (generic security policy, distributed-system-specific security concern) pairs, which we subsequently construct and specify. Acting as an adapter in combination with the Requirement class to policies table, this table allows for requirements to be mapped to security patterns via concerns through generic policies – hence giving developers the option to use security solution frames when available, optionally in tandem with standalone security patterns; only (standalone) security patterns, when a frame for a given policy is not available; or even custom, on-the-fly, expert-created countermeasures, if a given specific security policy is not addressed by either (standalone) patterns or patterns constituting the solution frames.

– Augmentation (operator) on the Solution frame collection and Threat taxonomy artefacts: we instantiate another new conceptual framework model element – Policies to threats table – giving us reverse relations between the mitigating policies encapsulated by the threat patterns and the pattern's encompassing threat classes.

– Augmentation (operator) on the SecReq at Design (des) phase: as a prior step to identifying countermeasures in CounterIdentif, all requirements are first converted to design-level equivalent requirements. This instigates the instantiation of several conceptual framework model elements: Early requirements – a “dual” of the abstract security patterns used by the Countermeasure Introduction activities at Analysis (an), complementing the Design-level requirements artefact; as well as Technical requirements – an artefact specializing Design-level requirements, and to which all other requirements will be converted. Further model elements are added to encapsulate the collections in which these artefacts will reside. Relating the technical requirements to both threats and their encapsulated mitigating security policies, we instantiate the SecMod SPPF pattern together with specify and refine task patterns in a cyclical workflow, to capture the activity of specifying requirements in the appropriate form: “protect from X”, where X is a threat pattern; or “need Y”, where Y is a generic security policy; and refining these to a final set. The latter form essentially defines the Technical requirements artefact. The conversion activity utilizes the Requirement class to policies table artefact, to which it is duly linked, as a “sieve” for early requirements – i.e. those stemming from the AdvMod instance at Analysis (an) – which can be mapped back and forth between threat patterns and policies and combined/merged (in the refine task) with related design-level requirements, until a final set of technical requirements is reached.

– Augmentation and Specialization (operators) to the CounterIdentif stage: the activities are updated and re-specified to reflect the new approach of mapping requirements to security solutions. Since all the requirements are in the form of Technical requirements (as per the updated SecReq phase), they can be immediately mapped to a corresponding policy, and from there to either a security solution frame (stemming from a tactic which encapsulates that policy)
and thence to more specific policies addressed by the patterns constituting that solution frame; or to separate
security patterns from the Distributed security pattern catalog artefact via the Policies to distributed-system-specific
cconcerns table artefact. This provides a structured and systematic approach to identifying countermeasures
promoting traceability between security solutions, threats, policies and requirements.

The final methodology model is shown partially (cropped to fit the page) in Figure 7. The colours (or shades of
grey) imply elements that have been replaced (green), specialized (yellow) or added (cyan – standard elements; pink
– supportive artefacts). The CounterIdentif stage pattern appears in black background since it is realized using an
abstract process fragment; the foreground colour is yellow to indicate that it has been specialized. We omit the usual
accompanying descriptions of the pattern instances and/or artefacts, as we have attempted to delineate these in the
presentation of the tailoring changes above.

6.3.4 QAIP Step 8: Re-assessment

Now that we have shown how the methodology of Uzunov et al. can be re-engineered, it remains to be seen whether
the re-engineered version – under the assumption that all artefacts and process elements are fully implemented and
specified – provides the desired quality improvements. To this end we perform the QAIP assessment steps as in
Section 5, and tabulate the new measures for the criteria that were chosen for improvement (see the right-most
column of Table 17) in the fourth column of Table 18, alongside the old measures in the third column. We have
added all the new criteria that were used as requirements (indicated by black background cells in the left-most,
quality factor column), as well as several other related criteria to give better reflect the changes to the methodology.

The assessment results indicate that the re-engineering process achieved the desired results, namely, to improve
the initially chosen quality criteria, and hence the corresponding quality factors. Perhaps equally importantly, the
measures of all criteria that were positive in the original methodology have been retained. This last point is
something that cannot always be anticipated, since re-engineering in general implies making trade-offs. As an aside,
we should point out the special importance of the Correctness of construction criteria in this context, which indicate
whether the tailoring changes gave rise to any inconsistencies. Although not shown in Table 18, these criteria were
(positively) satisfied by the re-engineered methodology.

In re-engineering this case study we focused exclusively on improvements for a general distributed systems
context; in other words, we used only the general distributed systems criteria profile. The methodology of Uzunov et
al. thus remains somewhat incomplete with respect to the peer-to-peer systems quality profile, as can be seen from
the criterion Specific artefacts (peer-to-peer systems – defensive) (CF; w; O), as well as the related Broadness of
scope (peer-to-peer systems – artefacts – defensive) (CF; p; S). This is especially notable given that threats for peer-
to-peer systems, which, as indicated by the relevant criteria, possess a broad scope, would have no corresponding
countermeasures. Improving this particular methodology feature implies the distillation of relevant security
solutions as patterns in various security solution frames.

6.4 Applying QAIP to SERENITY and the methodology of Ali et al.

In much the same way as shown for the methodology of Uzunov et al., it is possible to apply QAIP to improve the
quality of SERENITY and the methodology of Ali et al. After identifying relevant criteria for improvement, for
SERENITY the ensuing re-engineering process would imply removing some of the reliance on software tool support
to increase SERENITY’s applicability to general distributed systems; as well as the use of the Decomposition-based
threat modeling abstract process fragment (utilizing the distributed architecture decomposition and threat
taxonomy/library generic artefacts – see Section 6.2) to improve threat-modeling associated criteria, which, as can
be seen from Table 17 (criteria Nos. 5, 6, 34), received uniformly low measures. The methodology of Ali et al.
received low measures on many of the quality criteria in general, but especially those related to the Usability factor;
therefore re-engineering would imply the use of the Utilizing a development artefact repository abstract process
fragment (Section 6.2) for increasing petri-net-based model re-use across projects; as well the integration of security
solution frames with appropriate petri-net models for each encapsulated security pattern (cf. [Horvath and Dörges,
2008]). The latter improvement would naturally require a significant amount of work in specifying the relevant
artefacts.

6.5 Discussion

As was shown in the detailed application of QAIP to the methodology of Uzunov et al., the re-engineering process
achieved the desired results, namely, to improve the initially chosen quality criteria, and hence the corresponding
quality factors. The changes resulting from re-engineering were largely in keeping with the semantics of the original
methodology. More drastic changes are also possible, as was proposed in the case of SERENITY (dispensation of
run-time tool support). While such changes could adversely affect some quality criteria (e.g. the latter change for
SERENITY would cause some Usability criteria to receive lower measures), they could also be used to combine
elements of different methodologies. For example, the methodology of Ali et al. can be combined with the
methodology of Uzunov et al. such that the former methodology is employed for certain critical software
components during design, while the latter is used otherwise. This would combine formal guarantees (for the critical components) with ease of use and comprehensiveness (for the rest of the system), resulting in a rich, flexible methodology.

Although not explicitly stated as part of QAIP, it would be possible to use the “from scratch” construction workflow of S-SMEP instead of the tailoring workflow, to create a completely fit-to-purpose methodology by instantiating SPPF patterns and realizing them via process fragments generalized from different methodologies, guided throughout (in the same fashion as QAIP Steps 5 and 6) by a collection of quality criteria-turned-improvement.

<table>
<thead>
<tr>
<th>QF</th>
<th>Quality criterion</th>
<th>Measures (methodology of Uzunov et al.): Before re-engineering</th>
<th>Measures (methodology of Uzunov et al.): After re-engineering</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Specific artefacts (basic security solutions) (CF; p; O)</td>
<td>many {security patterns}</td>
<td>many {security solution frames, security patterns}</td>
<td>9</td>
</tr>
<tr>
<td>C</td>
<td>System applicability (general distributed systems – artefacts – defensive) (CF; w; O)</td>
<td>yes</td>
<td>yes</td>
<td>12</td>
</tr>
<tr>
<td>V</td>
<td>Coverage (artefacts – offensive) (CF; p; S)</td>
<td>excellent {networked system threats}</td>
<td>excellent {networked system threats}</td>
<td>17</td>
</tr>
<tr>
<td>V</td>
<td>Coverage (artefacts – defensive) (CF; p; S)</td>
<td>good {generic security policies}</td>
<td>excellent {generic security policies}</td>
<td>18</td>
</tr>
<tr>
<td>V</td>
<td>Structure (artefacts) (CF; w; O)</td>
<td>partially structured {patterns}</td>
<td>structured</td>
<td>21</td>
</tr>
<tr>
<td>V</td>
<td>Guidance (identifying countermeasures) (SP; w; S)</td>
<td>partially systematic {mapping table}</td>
<td>systematic {threats, requirement/policy mappings}</td>
<td>22</td>
</tr>
<tr>
<td>V</td>
<td>Guidance (introducing countermeasures) (SP; w; S)</td>
<td>ad-hoc {pattern relationships}</td>
<td>systematic {solution frame micro-patterns}</td>
<td>23</td>
</tr>
<tr>
<td>V</td>
<td>Use of best practices (artefacts – encapsulation) (CF; p; O)</td>
<td>many {security patterns, threat patterns}</td>
<td>many {solution frames, security patterns, threat patterns}</td>
<td>26</td>
</tr>
<tr>
<td>V</td>
<td>Broadness of scope (general distributed systems – artefacts – defensive) (CF; p; S)</td>
<td>good {generic security policies}</td>
<td>excellent {generic security policies}</td>
<td>30</td>
</tr>
<tr>
<td>V</td>
<td>Coverage (general distributed systems – artefacts – security solutions) (CF; p; S)</td>
<td>average {specific security policies}</td>
<td>good {specific security policies}</td>
<td>-</td>
</tr>
<tr>
<td>V</td>
<td>Coverage (general distributed systems – artefacts – threats) (CF; p; S)</td>
<td>good {mitigating security policies}</td>
<td>good {mitigating security policies}</td>
<td>-</td>
</tr>
<tr>
<td>V</td>
<td>Structure (artefacts – offensive artefacts) (CF; p; O)</td>
<td>structured</td>
<td>structured</td>
<td>-</td>
</tr>
<tr>
<td>V</td>
<td>Structure (artefacts – defensive artefacts) (CF; p; O)</td>
<td>partially structured {patterns}</td>
<td>partially structured {patterns}; structured {solution frames}</td>
<td>-</td>
</tr>
<tr>
<td>U</td>
<td>Expertise required (methodology – introducing countermeasures) (SP; w; S)</td>
<td>medium</td>
<td>low</td>
<td>35</td>
</tr>
<tr>
<td>U</td>
<td>Structure (artefacts) (CF; w; O) (*)</td>
<td>partially structured {patterns}</td>
<td>structured</td>
<td>36</td>
</tr>
<tr>
<td>U</td>
<td>Structure (activities – introducing countermeasures) (SP; w; S)</td>
<td>partially structured</td>
<td>structured</td>
<td>38</td>
</tr>
<tr>
<td>U</td>
<td>Expertise required (methodology) (SP; CF; w; S)</td>
<td>medium</td>
<td>low</td>
<td>32</td>
</tr>
<tr>
<td>U</td>
<td>Understandability (artefacts – offensive) (CF; p; S)</td>
<td>high</td>
<td>high</td>
<td>40</td>
</tr>
<tr>
<td>U</td>
<td>Understandability (artefacts – defensive) (CF; p; S)</td>
<td>high</td>
<td>high</td>
<td>41</td>
</tr>
<tr>
<td>U</td>
<td>Known modeling language (artefacts) (CF; w; O)</td>
<td>partially systematic {mapping table}</td>
<td>systematic {threats, requirement/policy mappings}</td>
<td>46</td>
</tr>
<tr>
<td>U</td>
<td>Guidance (identifying countermeasures) (SP; w; S)</td>
<td>ad-hoc {pattern relationships}</td>
<td>systematic {solution frame micro-patterns}</td>
<td>47</td>
</tr>
<tr>
<td>U</td>
<td>Guidance (introducing countermeasures) (SP; w; S)</td>
<td>many {security patterns, threat patterns}</td>
<td>many {solution frames, security patterns, threat patterns}</td>
<td>49</td>
</tr>
<tr>
<td>A</td>
<td>System applicability (generality) (SP, CF; w; O)</td>
<td>mixed</td>
<td>mixed</td>
<td>56</td>
</tr>
<tr>
<td>A</td>
<td>Flexibility (methodology) (SP, CF; w; O)</td>
<td>conditionally flexible</td>
<td>conditionally flexible</td>
<td>57</td>
</tr>
<tr>
<td>A</td>
<td>System applicability (general distributed systems – artefacts – defensive) (CF; w; O)</td>
<td>yes</td>
<td>yes</td>
<td>61</td>
</tr>
<tr>
<td>A</td>
<td>Flexibility (general distributed and peer-to-peer-systems) (SP, CF; w; O)</td>
<td>yes</td>
<td>yes</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 18: Quality assessment of the re-engineered methodology of Uzunov et al.
Figure 7: Pattern-based model of the re-engineered methodology of Uzunov et al.
7 Related work

As we remarked in the Introduction (Section 1), there is very little work on quality assessment of security methodologies, and, in fact, this paper represents (to the best of our knowledge) the first attempt to provide a comprehensive approach for this purpose. This is not to say, however, that quality assessment of (general) processes, as such, is novel in any way; or that the contiguous concepts of assessing the quality of development methodologies, or of the security activities within an organization, have not been explored before.

For development methodologies in particular, related ideas can be found in the form of the SEI Capability Maturity Model (CMM) (see, for example, [Henderson-Sellers et al., 2004] for a relevant discussion of CMM in the context of SME), and also under the broad term of “process improvement”, with the import that an improved process can lead to an improved product (cf. the views of software quality discussed in Section 3.1). Similar initiatives within security, such as SSE-CMM, BSIMM, OpenSAMM and others encompass a similar range of ideas [Fonseca and Vieira, 2013].

BSIMM [McGraw et al., 2012], for example, is an attempt to measure and/or encourage incorporation of a range of security-related activities across the SDLC. BSIMM distributes these activities – of which there are over one-hundred in total in BSIMM4 [McGraw et al., 2012] – across a framework of twelve security-related practices, subdivided into four domains: Governance (with the practices Strategy and metrics, Compliance and policy and Training); Intelligence (practices Attack models, Security features and design and Standards and requirements); SSDL Touchpoints (practices Architecture analysis, Code review and Security testing) and Deployment (practices Penetration testing, Software environment and Configuration management and vulnerability management). The presence or absence of the activities within each of these practices in the development process essentially determines the relevant (security) maturity level. Example activities include Standardize architectural descriptions (“the organization uses an agreed-upon format for describing architecture, including a means for representing data flow” [McGraw et al., 2012]) or Have software architects lead review efforts (“software architects throughout the organization lead the architecture analysis process most of the time” [McGraw et al., 2012]). In all cases the activities are distilled versions taken from the development processes of existing organizations, i.e. they reflect industry best practice with respect to security incorporation throughout the SDLC.

OpenSAMM 1.0 [OpenSAMM, 2009], currently an OWASP project, is very similar to BSIMM, with a division of four business functions (Governance, Construction, Verification and Deployment) subdivided into four practices, each practice containing a collection of relevant activities. The essential difference between OpenSAMM and BSIMM lies in the range of activities and their greater technical emphasis – especially in the Construction-related activities, which in OpenSAMM are noteworthy for their software architectural focus (e.g. suggesting the distillation of security patterns, more specific details on threat modeling etc.). The latter aspect in particular positions OpenSAMM much more closely to our proposal in this paper, inasmuch as a number of the OpenSAMM activities can be considered to be specific process patterns specializing the general process patterns of our SPPF (cf. Section 2.2). Maturity levels in OpenSAMM are determined according to questionnaires that ultimately posit, as in BSIMM, the presence or absence of certain security activities in a given development process.

Despite the activity relation and superficial similarities between the two security capability maturity models outlined so far and our own work in this paper, there are a number of important differences that set our work apart. The most significant of these is that both security CMMs apply to what in BSIMM is termed a Secure Software Development Lifecycle (SSDL) – “any SDLC with integrated software security checkpoints and activities” – i.e. development methodologies with integrated security-related activities, as opposed to a dedicated, self-contained security process. Strictly speaking, CMM assessment is not an assessment of security methodologies at all, but of development methodologies, and, more specifically, of the development process – including all managerial aspects – of an organization, with the aim of indirectly measuring – qualitatively – the security of the final software product [Jaatin, 2012]. This leads to the second important difference: security CMMs in general adopt a fundamentally organization-centric viewpoint; our approach, on the other hand, adopts a methodology-centric viewpoint, focusing exclusively on the construction – with the implied technical aspects – of a given security methodology. We do not attempt to determine a “maturity level” either for an organization or a methodology; instead, the measures of the individual quality criteria in a criteria profile determine the final quality of a given methodology.

Other security CMMs used in practice besides the two mentioned above, such as SAFECode, Securiosis and SSE-CMM (for which see [Fonseca and Vieira, 2013] and [Kara et al., 2012]) are either more specialized, emphasize predominantly implementation activities, or focus almost exclusively on higher-level project management aspects – all characteristics that further distance the CMMs from our work in this paper.

Outside of the field of security and aside from CMMs, development methodology quality is considered centrally in the general evaluation framework of [Hesari et al., 2010], which proposes a comprehensive set of quality features and corresponding criteria structured in hierarchical fashion. The framework acts as a substratum for evaluating the quality of development methodologies of any paradigm, although no specific metrics or assessment processes are presented. The essential contribution of the framework is thus the collection of features and criteria and their thorough evaluation against existing frameworks for object-oriented and agent-oriented methodologies. In a sense, the framework of [Hesari et al., 2010] can be seen as simpler non-security analogue of our own quality framework, that could be used to assess the encompassing development methodology into which a security methodology is to be integrated.
A rather different and more detailed approach for evaluating the quality of development methodologies can be found in the work of Välimäki et al. [Välimäki et al. 2009], who propose an assessment technique for process pattern languages – named Q-PAM – inspired by the Architecture Trade-off Analysis Method (ATAM) [Bass et al., 2003]. Besides the shared characteristic of our proposal in this paper in drawing inspiration from the domain of software quality, Q-PAM represents an orthogonal assessment approach. Q-PAM proceeds by defining a quality profile consisting of (process) non-functional requirements from existing software quality standards; creating scenarios, which are possible situations; and assessing whether given patterns in some process pattern language can support the scenarios. There are several drawbacks for each of the three aforementioned steps: for the first step, the process of taking non-functional requirements from software quality standards – which by necessity require adaptation for the process context – is arbitrary, lacking in guiding criteria; for the second step, the construction of scenarios intrinsically leads to an ad-hoc approach, in which the evaluation is only as good as the generated scenarios, which rely purely on the participant's intuition; and for the third step, the analysis applies to patterns, not processes which are “patternized”, as the authors state (this last drawback being inherent in the approach).

The latter proposals for development methodologies, while not directly applicable in our security context, have served as partial inspiration for our own approach, influencing in particular the construction of our criteria profiles.

8 Conclusion and future work

Just as assessing the quality of software products can help to determine what improvements are required and/or whether a particular product is suitable for a particular purpose, so assessing the quality of security methodologies can help to determine suitability for a given project situation; whether a given methodology meets certain needs; and whether certain improvements are needed in order for the methodology to acquire certain desirable characteristics. Quality assessment can also help to differentiate methodologies and hence to guide adoption decisions. Our quality framework and accompanying process, presented in Section 4 and demonstrated in Section 5, were created with the latter aims in mind. Once the framework and process were applied for the assessment of our case study methodologies, we showed how the methodologies can be re-engineered to improve their quality (Section 6) – thereby illustrating the dual use of our quality framework for top-down quality improvement. In both cases the use of situational profiles in the form of customized criteria and metrics (for assessment), or criteria directly converted into requirements (for improvement), allow a consideration of project-relevant methodology features of any granularity – from those pertaining to the whole methodology at a high-level, to those pertaining to specific artefact or process element characteristics – thus promoting flexible, fine-grained quality control.

Our work in this paper can be extended in a number of directions. At the moment, a number of the criteria in the base profile (approximately half overall), especially for usability, are subjective (i.e. their measures are expected to be predominantly based on expert judgements). Of course, subjectivity is and will always remain an important element of quality assessment; however, in certain situations it may be desirable that a given measure be obtained via (relatively) objective means. An important avenue for future work in this regard is the development of objective counterparts to a selection of the subjective criteria, with appropriate metrics that would reduce the need for expert judgements.

Regarding top-down quality improvement, the proposed framework could be incorporated as an essential part of S-SMEP to guide the selection of SPFF patterns or abstract process fragments from existing methodologies (especially in the “from scratch” engineering strategy), ensuring a certain degree of quality a priori. Following the same direction, employing a tactics-driven approach akin to that of [Zhu and Staples, 2007] (which is analogous to the use of tactics in software architecture) is a further possibility, whereby appropriate tactics could be developed to guide the instantiation of given SPFF patterns targeting a given quality factor or criterion. The tactics would thus reside on a complementary level of abstraction to the patterns, giving additional conceptual tools for security methodology engineers, and could, when extended even further, lead to meta-patterns in the spirit of [Rolland and Plihon, 1996] and [Denecké, 2002]. Attaching criteria to each abstract process fragment or even to each pattern instance in the security processes of different methodologies (i.e. the latter supporting the extraction of abstract fragments) is another valuable extension that would not only complement the latter ideas, but also help in general with the selection of appropriate process elements to replace or supplement given pattern instances in an existing (base) methodology.

In this paper we have considered six of the seven proposed quality factors – effectiveness was purposefully excluded, as explained in Section 4.1. The construction of appropriate criteria and metrics for effectiveness, based on, for example, the work of [Khan and Zulkernine, 2008] among others, is thus another important future direction. In a similar vein, integrating documented approaches for assessing the quality of various process elements or artefacts of a methodology individually – e.g. the approaches of Halkidis et al. [Halkidis et al., 2006] for security patterns (Security solution artefact instance), or of Mouratidis et al. [Mouratidis et al., 2013] for selecting cloud providers (Map SPPF pattern instance) – would be a further extension to the framework and assessment/improvement process.

As we pointed out in Section 7, security capability maturity models such as BSIMM and OpenSAMM are related to methodology quality, even though in a development methodology context, and from an organization-centric viewpoint. Aligning the activities in such maturity models with the security process patterns from SPFF, and thereby correlating the quality framework in this paper (which uses pattern-based models as its basis) is an interesting future
direction that will also give the work a stronger industry focus. Especially important in this context is the supplementation of the technical assessment of a security methodology provided by our framework with the organization-centric, management activities provided by the maturity models, which are development process driven.

A final direction that would be worth exploring is the addition of supporting processes or techniques, such as AHP (Analytic Hierarchical Process – [Saaty, 2000]) for guiding the comparison of methodologies based on quality measurements.

References


Uzunov A V, Fernandez EB and Falkner K (n.d.) An Extensible Pattern-based Library and Taxonomy of Security Threats for Distributed Systems. accepted for Computer Standards & Interfaces (Special Issue on Information Security)


Epilogue

In this chapter, we presented a framework for security methodology quality accompanied by a process utilizing S-SMEP from Chapter 3, for methodology quality assessment and improvement. While not, strictly speaking, “proving” or validating (as in an industry trial run) the value of our methodology engineering approach begun in Chapter 3, or its constituent parts, the chapter allows that approach to validate itself with respect to its products – i.e. the methodologies produced – which is, in effect, the most one can do for evaluating work residing on the meta-modeling or meta-methodology level. The chapter thus served the dual purpose of an evaluation and at the same time extension of S-SMEP and the rest of the engineering “toolkit” presented in this thesis. The chapter also fulfilled two other important goals: to illustrate how the generic conceptual security artefacts from Chapters 4 to 7 can be “deployed” in different methodologies; and to fully develop – i.e. engineer – the case study methodology for this thesis. With this we have completed the presentation of the thesis from the methodology-engineering-centric viewpoint: begun with surveying the area of security methodologies and security patterns (as the most promising artefact type); then the presentation of the S-SMEP meta-methodology in Chapter 3 and a collection of generic conceptual security artefacts in Chapters 4 to 7 that can be “deployed” across different methodologies during the application of S-SMEP. Throughout we have applied the “tools of the toolkit” in the engineering of a specific security methodology. In the next, final chapter, we indicate the realism of this case study methodology and show how it can be used in practice.
Chapter 9
(Case study methodology)
A comprehensive security methodology for distributed systems

In theory, theory is the same as practice, but not in practice.
– Fnord Bjørnberger

... it is not my design to teach the method that everyone must follow in order to use his reason properly, but only to show the way in which I have tried to use my own.
– René Descartes (Discourse on Method, 1637)

Prologue

In the previous chapters we presented our engineering “toolkit” with all its constituent parts: frameworks, artefacts, meta-methodologies. Throughout Chapters 3 and 8 in particular, we applied this “toolkit” to engineer a specific pattern-driven methodology for distributed systems, which also utilizes all out generic artefacts. It this chapter we aim to summarize this methodology in a single place in slightly more detail, and to indicate its realism by showing how it can be used in practice on a project concerned with the development of a secure collaborative, distributed system. The latter system was in fact first introduced briefly in Chapter 6 for the authorization artefact; some of those details, and in particular the construction of a conceptual authorization architecture and supporting infrastructure are repeated here. What makes even that discussion unique is that the details are placed in a more holistic context, showing an interplay of the generic artefacts – in this particular methodology. The aim is thus illustration, as opposed to attempting to “prove” that the case study methodology is effective, or more effective than other methodologies, in producing a secure system. The quality of the case study methodology with respect to factors other than quality was considered already in Chapter 8.

If a security methodology is considered a solution to the problem of introducing security systematically, then the approach encapsulated in the “toolkit” for engineering security methodologies can be seen as a “meta-solution” capable of generating particular solutions. In this light, our case study methodology, which was used as an illustrative example for evaluation throughout the thesis, is one such solution.
# Statement of Authorship

<table>
<thead>
<tr>
<th>Title of Paper</th>
<th>A Comprehensive Pattern-Driven Security Methodology for Distributed Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication Status</td>
<td>○ Published, ○ Accepted for Publication, ○ Submitted for Publication, ○ Publication style</td>
</tr>
</tbody>
</table>

## Author Contributions

By signing the Statement of Authorship, each author certifies that their stated contribution to the publication is accurate and that permission is granted for the publication to be included in the candidate's thesis.

<table>
<thead>
<tr>
<th>Name of Principal Author (Candidate)</th>
<th>Anton V. Uzunov</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Conceptualization of work, its realization (research) and documentation (wrote manuscript). Will act as corresponding author.</td>
</tr>
<tr>
<td>Signature</td>
<td>Date</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th>Eduardo B. Fernandez</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Provided ideas, references, and comments.</td>
</tr>
<tr>
<td>Signature</td>
<td>Date</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of Co-Author</th>
<th>Katrina Falkner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution to the Paper</td>
<td>Supervised development of work, helped to evaluate and edit the manuscript.</td>
</tr>
<tr>
<td>Signature</td>
<td>Date</td>
</tr>
</tbody>
</table>
A Comprehensive Pattern-Driven Security Methodology for Distributed Systems

Anton V. Uzunov\textsuperscript{a,*}, Eduardo B. Fernandez\textsuperscript{b}, Katrina Falkner\textsuperscript{a}

\textsuperscript{a}, School of Computer Science, The University of Adelaide, Adelaide, South Australia, Australia, 5005
\textsuperscript{b}, Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, 777 Glades Rd., Boca Raton, FL 33431

* Corresponding author

E-mail addresses: \{anton.uzunov, katrina.falkner\}@adelaide.edu.au (Anton V. Uzunov, Katrina Falkner), ed@cse.fau.edu (Eduardo B. Fernandez)

\textbf{Abstract:} Incorporating security features is one of the most important and challenging tasks in designing distributed systems. Over the last decade, researchers and practitioners have come to recognize that the incorporation of security features should proceed by means of a structured, systematic approach, combining principles from both software and security engineering. Such systematic approaches, particularly those implying some sort of process aligned with the development life-cycle, are termed security methodologies. There are a number of security methodologies in the literature, of which the most flexible and, according to a recent survey, most satisfactory from an industry- adoption viewpoint are methodologies that encapsulate their security solutions in some fashion, especially via the use of security patterns. While the literature presents several mature pattern-driven security methodologies with either a general or a highly specific system applicability, there are currently no (pattern-driven) security methodologies specifically designed for general distributed systems. Going further, there are also currently no methodologies with mixed specific applicability, e.g. for both general and peer-to-peer distributed systems. In this paper we aim to fill these gaps by presenting a comprehensive pattern-driven security methodology – arrived at by applying a previously devised approach to engineering security methodologies – specifically designed for general distributed systems, which is also capable of taking into account the specifics of peer-to-peer systems as needed. Our methodology takes the principle of encapsulation several steps further, by employing patterns not only for the incorporation of security features (via security solution frames), but also for the modeling of threats, and even as part of its process. We illustrate and evaluate the presented methodology in detail via a realistic example – the development of a distributed system for file sharing and collaborative editing. In both the presentation of the methodology and example our focus is on the early life-cycle phases (analysis and design).

\textbf{Keywords:} secure software engineering, security methodologies, distributed systems security, security patterns, threat patterns, security solution frames

\section{1 Introduction}

Incorporating security features is one of the most important and also one of the most challenging tasks in designing distributed systems \cite{Belapurkar et al., 2009, Bidan and Issarny, 1997}. Over the last decade, researchers and practitioners have come to recognize that the incorporation of security features should proceed by means of a structured, systematic approach, combining principles from both software and security engineering \cite{Davis et al., 2004, Mouratidis and Giorgini, 2006, Rosado et al., 2010, Shumacher, 2003, Van Wyk and McGraw, 2005}. Such systematic approaches, particularly those implying some sort of process aligned with the software development life-cycle, are termed security methodologies \cite{Uzunov et al., 2012b}. There are a number of security methodologies in the literature, of which the most flexible and most satisfactory from an industry-adoption viewpoint are methodologies that encapsulate their security solutions in some fashion (see \cite{Uzunov et al., 2012b}), especially via the use of security patterns \cite{Schumacher et al., 2006, Fernandez, 2013}. While the literature presents over a dozen such pattern-driven security methodologies, both young and mature \cite{Uzunov et al., 2012b, Uzunov et al., 2012a} – possessing a range of valuable and beneficial features – with respect to system applicability, these methodologies are uncomfortably positioned at two extremes of a spectrum: either they are highly specific, or highly generic. This makes such methodologies inadequate for project situations requiring the development of general distributed systems, since the methodologies will either lack provisions for the specific security concerns of general distributed systems or different types of distributed systems (too generic) \cite{Rosado et al., 2010}; or they will be incompatible with the features of the target system (too specific) \cite{Uzunov et al., 2012a} – whether because of the processes involved (e.g. PWSsec \cite{Gutiérrez et al., 2009}); or because of the conceptual artefacts used (e.g. the methodology of Delessy and Fernandez \cite{Delessy and Fernandez, 2008}).

At present, there are no pattern-driven security methodologies specifically designed for general distributed systems – i.e. positioned somewhere in the middle of the specificity-generality spectrum referred to above (we are considering here exclusively methodologies using security patterns, not patterns interpreted as architectures or components as in the work of \cite{Schmidt et al., 2011}). Going further, there are also currently no methodologies in the
literature with mixed specific applicability [Uzunov et al., 2012a] – for example, for both general and peer-to-peer distributed systems; or for general and web-based applications.

In this paper we aim to fill the latter gaps, by presenting a comprehensive pattern-driven security methodology specifically designed for general distributed systems, named ASE, which is also capable of taking into account the specifics of peer-to-peer systems as needed. ASE emphasizes the early life-cycle phases (analysis and design) – since this is where all security countermeasures are planned [Bidan and Issarny, 1997], as well as where, according to Jaquith [Jaquith, 2002], approximately half of all major security flaws can be prevented (cf. [Hoglund and McGraw, 2004]); and takes the principle of solution encapsulation several steps further, by employing patterns not only for the incorporation of specific security attributes (via security solution frames), but also for the modeling of threats, and even as part of its process. Through its distributed-systems-specific set of conceptual artefacts and its comprehensive process, ASE is capable of addressing all or most of the core distributed systems security concerns, providing developers with detailed guidance on how and where to introduce relevant security features into a system's architecture during development.

Besides being a self-contained security methodology, ASE is also an example of a re-engineered methodology, arrived at by applying the approach to engineering security methodologies presented in [Uzunov et al., sub2] in its tailoring capacity – with the methodology of Fernandez et al. [Fernandez et al., 2006] as a base methodology. From the latter standpoint, if a security methodology is considered a solution to the problem of introducing security into software systematically, and the aforementioned engineering approach is seen as a meta-solution capable of generating solutions, then ASE can be seen an example of just one particular solution, with its own set of beneficial features.

The rest of this paper is structured as follows. In Section 2 we provide some brief background on security methodologies as well as an overview of the construction of ASE. We describe ASE in depth in Sections 3 and 4. In Section 5 we illustrate and evaluate ASE in detail via a realistic example, namely, the development of a distributed system for file sharing and collaborative editing. Finally, in Section 6 we conclude and discuss future directions.

(We should point out that there is no separate section on related work – the reader is referred to [Uzunov et al., 2012b, Uzunov et al., 2012a] for a comprehensive overview.)

2 Background

2.1 Security methodologies

A security methodology constitutes a “systematic way of doing things in a particular discipline” [Gonzalez-Perez and Henderson-Sellers, 2008], where in this case the discipline is secure software engineering. Security methodologies are strongly related to development methodologies, however, in our work and for the rest of this paper, security methodologies will be considered as separate entities in their own right, distinct from any development methodology to which, or in which, the latter might be related or integrated, respectively. Unlike a development methodology, a security methodology's primary aim is not to provide guidance for producing particular software artefacts, but rather, for improving the security attributes of a given system, i.e. a security methodology is process-centric. This is not to imply, of course, that security methodologies cannot prescribe the production of various software artefacts also.

Despite their process-centric nature, security methodologies are not simply “bare” processes in the sense of simply being descriptions of activities – a consideration of existing methodologies (see [Uzunov et al., 2012b]) reveals an inherent dichotomy, reflecting the process-product dichotomy of development methodologies: that of process-framework, which is to say, a security methodology is a process for improving the security attributes of a system via the use of a conceptual framework, consisting of various “security improvement artefacts” (e.g. security solutions and attendant conceptual artefacts – cf. [Schumacher, 2003]). A security methodology $SM$ can thus be defined more formally as a couple:

$$SM = (SP, CF)$$

where $SP$ is a security process – the activities and/or steps taken to secure a software system of some type; and $CF$ is a conceptual security framework, consisting of the conceptual artefacts used by the methodology's process, the number and type of which, in theory at least, can be arbitrary, but as a minimum must include a set of security solutions (defensive artefacts), as well as a set of threats (offensive artefacts) leading to those solutions, whether explicitly defined and cataloged or otherwise. Henceforth we will refer to the two constituents of a security methodology as a security process aspect and a conceptual framework aspect, respectively.

2.2 Construction and overview of ASE

ASE can be seen as an evolution of the security methodology of Fernandez et al. [Fernandez et al., 2006] (see also [Uzunov et al., 2012b] for an overview), arrived at by applying the systematic approach to engineering security methodologies presented in [Uzunov et al., sub2].

In its overall structure, ASE follows the order of development phases for a generic software development life-cycle [Uzunov et al., 2012b; Uzunov et al., sub2] (the short-hand notation in brackets will be used throughout the
Figure 1: Pattern-based model of ASE
rest of this paper): Requirements analysis (ReqAn), encompassing requirements elicitation/gathering (Req) and analysis (An) activities; Design (Des), encompassing architecture development (Arch) and detailed design (DetDes) activities; Implementation (Impl), encompassing coding (Cod) and testing (Test) activities; and Deployment (Deploy), encompassing administration and installation activities.

In essence, ASE begins by considering the use cases of a system during Requirements analysis, and how such use cases can be thwarted either by insiders or outsiders (Security Requirements Determination (SecReq) phase, Adversary Modeling (AdvMod) stage). This leads to an identification of countermeasures appropriate at this early stage of development and their modeling into the system's conceptual model. A similar sequence of activities is performed during Design, this time with respect to the system's architecture, considering in detail areas of functionality, threats, appropriate countermeasures, and then modeling the countermeasures to produce a secure software architecture. The remaining activities are concerned with verification (at different stages) and security implementation. A pattern-based model of ASE is shown in Figure 1, with the left-hand side containing ASE's security process aspect, and the right-hand side containing ASE's conceptual security framework aspect. The reader is referred to [Uzunov et al., sub2] for details on the notation and the semantics of such models – for this paper the figure can be taken simply as a schematic view of the flow of activities and the conceptual artefacts used.

In the next two sections (Section 3 and 4) we describe the two aspects of the methodology in turn.

3 ASE: conceptual security framework aspect

The elements constituting ASE's conceptual security framework adhere closely to the original classes of elements in the domain-specific meta-model of [Uzunov et al., sub2]. Referring to the right-hand side of Figure 1, besides generic artefacts connected with object-oriented use cases – employed in the early phases of the security process – there are both offensive and defensive artefacts (Threat and several Security solution element instances), as well as a number of supportive artefacts in the form of various mapping tables. Below we briefly describe the features of three core artefacts from the aforementioned collection used specifically throughout the Design-phase parts of ASE's security process. More details, as well as finer-grained models of these artefacts that allow a fuller version of ASE's conceptual security framework to be reconstructed can be seen in the relevant paper cited next to each artefact's name below. We mention other relevant artefacts (in italics font) as part of the descriptions with reference to the model in Figure 1.

Distributed architecture decomposition framework [Uzunov et al., 2013]: architectural decomposition is a form of system characterization, which, when complemented by an analysis process, provides structure for guiding the incorporation of early non-functional (in our case security) requirements and the determination of new requirements. The architectural decomposition framework artefact resides at a complementary level of abstraction to the software architectural models, and is stratified into three layers, with the first containing a conceptual model of distributed systems, usable as a meta-model for creating derived architectural models; the second containing a simple, layered architectural meta-model allowing for the separation of the architecture according to areas of functionality (functionality decomposition layers); and the third containing a set of meta-modeling elements (technical realization abstractions) that detail the decomposition.

Threat taxonomy (and library) [Uzunov and Fernandez, sub1]: a two-level collection of threat patterns – a type of abstract software pattern encapsulating threats, architectural contexts and mitigating security policies – for general distributed systems, encompassing both system threats and meta-security threats (threats to the security infrastructure itself). The base taxonomy for general distributed systems can be extended by specializing individual patterns for different architectural contexts to produce system-/technology-specific taxonomies. These taxonomies (like the base taxonomy itself) combine the benefits of a structured library and a STRIDE-like classification scheme, and support threat modeling by allowing non-expert developers to consider a wide variety of threats (threat pattern instances) tailored to particular applications. The threat taxonomy artefact is linked with the distributed architecture decomposition framework artefact, as the latter defines the architectural contexts for the individual threat patterns.

Security solution frame collection [Uzunov et al., sub1, Uzunov and Fernandez, sub2]: solution structures that encapsulate and organize security patterns (see [Schumacher et al., 2006, Fernandez, 2013]) into different levels of abstraction (vertically) and different facets of a root security policy (horizontally), providing guidance in realizing the security patterns from an abstract conceptual to a concrete design level, and simultaneously across the development life-cycle. Security solution frames also form a bridge between security tactics [Bass et al., 2003, Ryoo et al., 2010], which, in our interpretation, embody a single generic security policy, and patterns, which address more specific security policies. Neither tactics nor security policies are shown in Figure 1 for the sake of simplicity – their relationships can be seen in [Uzunov et al., sub1, Uzunov and Fernandez, sub2].

The literature presents frames for the most essential distributed systems security concerns (cf. [Uzunov et al., 2012a]), though, at the time of this writing, not for all (see Section 6 on future work). We describe the current solution frames together with projected frames for addressing other major security concerns, along the tactics they realize, in Table 1 (an asterisk next to the name of a frame indicates the description is a future projection). In Table 2 we show the relationships between the solution frames (see [Uzunov et al., sub1, Uzunov and Fernandez, sub2] for
details), which are also valid for the identically named generic security policies encapsulated in the threat patterns as mitigating policies – and mirrored in the relationships of the tactics embodying those policies (which we omit for the sake of brevity). The relationships are represented in the table using the following letters: requires (not necessarily strictly) (R), supports (or is beneficial to) (S) and depends on (the way one or more patterns are realized in the target frame) (D).

<table>
<thead>
<tr>
<th>Security solution frame</th>
<th>Description and security tactics realized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization</td>
<td>Encapsulates security patterns allowing for custom conceptual authorization models to be built and realized using different enforcement architectures [Uzunov et al., sub1]. <em>Tactic realized</em>: Authorize Users.</td>
</tr>
<tr>
<td>Identity management</td>
<td>Encapsulates patterns concerned with managing (assigning, establishing, validating) identities of users and/or processes in a distributed system (partially in [Uzunov and Fernandez, sub2]). <em>Tactic realized</em>: Authenticate Users.</td>
</tr>
<tr>
<td>Secure communication</td>
<td>Encapsulates security patterns for enabling two or more parties to communicate securely over a message channel [Uzunov and Fernandez, sub2]. <em>Tactics realized</em>: Protect Communications (Maintain Data Confidentiality &amp; Maintain Integrity in a communications context).</td>
</tr>
<tr>
<td>Filtering (*)</td>
<td>Shall encapsulate patterns for network- and application-level filtering of information, including firewalls and custom data filters. <em>Tactic realized</em>: Limit Exposure.</td>
</tr>
<tr>
<td>Storage security (*)</td>
<td>Shall encapsulate patterns for secure storage of information, including patterns for storing passwords, information dispersal, database security and others. <em>Tactics realized</em>: Keep Data Secure (Maintain Data Confidentiality &amp; Maintain Integrity in a storage context).</td>
</tr>
<tr>
<td>Logging and monitoring (*)</td>
<td>Shall encapsulate patterns for logging and monitoring events in a distributed system, ranging from creating simple log files to the deployment of intrusion-detection systems (network monitoring). <em>Tactics realized</em>: Intrusion Detection &amp; Audit Trail.</td>
</tr>
<tr>
<td>Execution control (*)</td>
<td>Shall encapsulate patterns for controlling and managing the execution of processes or, more generally, execution abstractions in a distributed system, including for process serialization / concurrency management, safe handling of mobile code, process isolation, improved resource availability and others. <em>Tactics realized</em>: Limit Access &amp; Safeguard Process (*).</td>
</tr>
<tr>
<td>Security information management</td>
<td>Encapsulates patterns concerned with the management of security information such as policies, credentials, cryptographic keys etc. (partially in [Uzunov and Fernandez, sub2] and in [Uzunov et al., sub1]). <em>Tactic realized</em>: Manage Security Information.</td>
</tr>
</tbody>
</table>

Table 1: Security solution frame descriptions

<table>
<thead>
<tr>
<th>Security solution frames</th>
<th>Authorization</th>
<th>Identity management</th>
<th>Secure communication</th>
<th>Filtering (*)</th>
<th>Storage security (*)</th>
<th>Logging &amp; monitoring (*)</th>
<th>Execution control (*)</th>
<th>Security info. management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization</td>
<td>R</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identity management</td>
<td>S</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure communication</td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filtering (*)</td>
<td></td>
<td></td>
<td>SD</td>
<td>D</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage security (*)</td>
<td>D</td>
<td>S</td>
<td>D</td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logging and monitoring (*)</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution control (*)</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security information management</td>
<td>D</td>
<td>D</td>
<td>S</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Relationships between security solution frames
Although in ASE we employ the solution frames without modifications/customizations, their encapsulated patterns can be replaced by, or related to, other conceptual security artefacts not shown in Figure 1 – e.g. security aspects [Georg et al., 2009] – while retaining the overall intrinsic structure of solutions (as specific security policy encapsulations).

4 ASE: security process aspect

In this section we outline ASE’s security process aspect following the sequence of phases for the generic development life-cycle, describing in each development phase the relevant SPPF process pattern instance(s) (see Section 2). Since our emphasis is on the improvements and novel contributions of the methodology over its antecedent (base methodology), we focus predominantly on activities performed during the design development phase, describing activities performed during other phases only briefly.

4.1 Requirements analysis (ReqAn)

ASE’s activities performed during the requirements analysis development phase essentially take a “system use security” viewpoint, concerned with the (external) interactions with the system to be developed.

The early phases of ASE are for the most part identical to those of its antecedent methodology [Fernandez et al., 2006, Braz et al., 2008] (see also [Uzunov et al., 2012b] for an overview), with the creation of a set of security use cases according to a set of standard (object-oriented) use cases. Each use case is analyzed for how it can be thwarted, considering four main security attributes: confidentiality, integrity, availability and accountability [Braz et al., 2008]. This activity is in all respects an instance of the Security Requirements Determination (SecReq) process pattern in [Uzunov et al., sub2], with the main goal being to determine a collection of (early) security requirements, which can be carried over into later phases of the security process. The requirements can (optionally) be related to a collection of encompassing requirements classes from the left-most column of Table 5, to ease the transition into the next ASE stages.

Secure Semantic Analysis Patterns (SAPs [Fernandez and Yuan, 2000]) are used during the analysis phase, where appropriate, to create a secure conceptual model efficiently. The use case analysis results from before are carried over to determine a set of abstract security patterns [Fernandez et al., 2008] that collectively embody, or, more precisely, are interpreted as embodying, a subset of the early security requirements. The latter patterns aim to guide the solution and are not requirements per se (in contrast to the original methodology of Fernandez et al., where they essentially take on that role). These patterns can be chosen from, or be related to, abstract patterns belonging to one or more pattern families from appropriate security solution frames (at their Conceptual analysis abstraction level), according to corresponding generic security policies, and instantiated in the conceptual model. In this case the mappings in Table 5 are employed to determine the relevant solution frames.

4.2 Design (Des)

In this sub-section we describe ASE’s activities performed during the design development phase, namely, the determination of design-level security requirements; the identification of appropriate security countermeasures satisfying those requirements; and the modeling of the identified countermeasures in the system’s software architecture. In all cases the activities take a “system architecture security” viewpoint.

4.2.1 Security Requirements Determination (SecReq) phase

4.2.1.1 Adversary Modeling (AdvMod) stage

To determine a set of design-level security requirements, we perform a detailed threat modeling process (instantiating the Adversary Modeling stage process pattern from [Uzunov et al., sub2]) that relates various threats to different parts of a target system's software architecture in a systematic, structured fashion. The first step in the latter process is to create a well-defined system characterization: in our case, this is an architectural decomposition as per [Uzunov et al., 2013]. The decomposition forms the basis for the analysis of threats, which are uniformly selected from the collection of threat patterns in [Uzunov and Fernandez, sub1]. Below we describe the steps of the threat modeling process, omitting the details of the (sub-)process for architectural decomposition, which is presented in [Uzunov et al., 2013], and focusing on the (sub-)process for analyzing threats and determining security requirements. For the sake of consistency, we should point out that the latter threat analysis is based on a customization of the related security analysis process presented in [Uzunov et al., 2013]. From the viewpoint of the latter paper, threat analysis is but one particular process for security which can be attached to the architectural decomposition framework.

Decomposition steps 1 to 3: (1) Using the meta-model from the framework's first level, create a derived architectural model representing a high-level decomposition into nodes, components, associations, domains etc. (2) Identify/assign elements of the model to the decomposition framework's (second level) functionality layers. (3)
Finally, for each functionality decomposition layer, map the corresponding elements of the derived model to technical realization abstractions (from the third level of the framework) as appropriate. The process of architectural decomposition should not be confused with the similarly termed functional decomposition (an architecture development technique), which is concerned with partitioning the system into logical units according to requirements – analogous to constructing a chart of a country. Architectural decomposition is concerned with the creation of a meta-level representation of the system's architecture according to a fixed, invariant set of models and abstractions (in the case of [Uzunov et al., 2013], structured into a framework) to which existing architectural elements can be mapped – analogous to overlaying an atlas “grid” on top of an existing chart.

**Threat analysis preparatory step (optional):** The rest of the adversary modeling process relies on the use of the threat patterns from the threat taxonomy and library of Uzunov and Fernandez [Uzunov and Fernandez, sub1] (see Section 3). In a number of situations the base taxonomy for general distributed systems in the latter reference may provide a sufficient number of threats for consideration, however, in many situations it may be advantageous to specialize this base taxonomy to a specific system/technology context (e.g. when developing web applications, multi-agent systems, systems using J2EE and so forth). The details of the specialization process, which, when applied, produces a derived, application-specific taxonomy, are described in [Uzunov and Fernandez, sub1]. For the rest of this paper we will assume that the base threat taxonomy and the peer-to-peer system-specific taxonomy presented in [Uzunov and Fernandez, sub1] are sufficient for the situation at hand.

**Threat analysis Step 1:** Once an architectural decomposition has been obtained, threat analysis can be performed via two complementary approaches, in both cases utilizing either the base or an application-specific threat taxonomy constructed in the previous, preparatory step.

The first approach (*context-driven*) relies on considering each functionality decomposition layer (in no particular order) and selecting appropriate threat classes containing patterns with matching architectural context(s) defined by the decomposition (see Table 3). Patterns for related technical realization abstractions are chosen in like manner (see Table 4). In this way, threats can be (1) mapped or related to the elements corresponding to the high-level modeling abstractions of the decomposition; (2) considered with respect to the different areas of functionality of the system, e.g. communication, and all related elements; and (3) to particular realizations of the functionality, e.g. particular network protocols, messages (integrity, confidentiality) and so on.

The second approach (*threat-driven*) relies on selecting threat patterns in some order – either predetermined or arbitrary – and relating them, via their architectural contexts, to the different parts of the architecture. This approach is more suited for developers with security expertise (unless every single pattern in the threat taxonomy is considered iteratively, in which case little expertise is required).

Regardless of the approach taken, associations between software components (the terminology pertains to the architectural decomposition) play an important role in this process step inasmuch as they help to determine component relationships and hence the possible vulnerabilities that can be exploited.

Once a threat pattern has been selected, it must be instantiated for the given architecture in order to determine the exact threat. Every threat is also analyzed with respect to the whole set of decomposition layers, considering its repercussions vertically down the layers, one by one.

<table>
<thead>
<tr>
<th>Functionality decomposition layer</th>
<th>Relevant threat classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>User interaction</td>
<td>Identity attacks, Passing illegal data, Remote information inference, Repudiation, Uncontrolled operations</td>
</tr>
<tr>
<td>Data / storage management</td>
<td>Passing illegal data, Stored data attacks, Remote information inference, Uncontrolled operations</td>
</tr>
<tr>
<td>Resource management</td>
<td>Uncontrolled operations</td>
</tr>
<tr>
<td>Distribution control</td>
<td>Identity attacks, Passing illegal data, Remote information inference, Uncontrolled operations</td>
</tr>
<tr>
<td>Communication</td>
<td>Network communication attacks, Network protocol attacks, Repudiation</td>
</tr>
<tr>
<td>Addressing</td>
<td>Network communication attacks, Network protocol attacks, Repudiation</td>
</tr>
</tbody>
</table>

**Table 3:** Mappings between functionality decomposition layers and threat taxonomy's (first level) threat classes

<table>
<thead>
<tr>
<th>Technical realization abstraction</th>
<th>Relevant (first level) threat patterns from Uzunov and Fernandez's base threat taxonomy [Uzunov and Fernandez, sub1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>User interaction</td>
<td>Identity spoofing, Advantageous identity allocation, Injection, Repudiation, Unauthorized access, Spoofing privileged processes, Exploitation of tight component coupling, Process</td>
</tr>
<tr>
<td>Technical Realization Abstraction</td>
<td>Threat Patterns</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Output ports</td>
<td>Output information disclosure, Data inference</td>
</tr>
<tr>
<td>Data / storage management</td>
<td>Injection, Corruption, Data inference, Session state poisoning, Process overflow attack</td>
</tr>
<tr>
<td>Data structures</td>
<td>Injection, Corruption, Data inference, Unauthorized access, Resource exhaustion</td>
</tr>
<tr>
<td>Storage abstractions</td>
<td>Injection, Corruption, Data inference, Unauthorized access, Resource exhaustion</td>
</tr>
<tr>
<td>Database systems</td>
<td>Injection, Corruption, Track erasing, Data inference, Unauthorized access, Resource exhaustion</td>
</tr>
<tr>
<td>File systems</td>
<td>Injection, Corruption, Data inference, Track erasing, Unauthorized access, Resource exhaustion</td>
</tr>
<tr>
<td>Resource management</td>
<td>Unauthorized access, Invoking unauthorized operations, Resource exhaustion, Targeted process crashing, Covert network channels</td>
</tr>
<tr>
<td>Resources</td>
<td>Invoking unauthorized operations, Spoofing privileged processes, Exploitation of tight component coupling, Exploiting concurrency flaws, Resource exhaustion</td>
</tr>
<tr>
<td>Algorithms</td>
<td>Identity spoofing, Injection, Probing, Output information disclosure, Invoking unauthorized operations, Spoofing privileged processes, Exploitation of tight component coupling, Process overflow attack, Exploiting concurrency flaws, Targeted process crashing</td>
</tr>
<tr>
<td>Software component interfaces</td>
<td>Advantageous identity allocation, Output information disclosure, Invoking unauthorized operations, Exploitation of tight component coupling, Exploiting concurrency flaws</td>
</tr>
<tr>
<td>Operations</td>
<td>Identity spoofing, Injection, Probing, Output information disclosure, Unauthorized access, Invoking unauthorized operations, Spoofing privileged processes, Unsafe code execution, Exploitation of tight component coupling, Process overflow attack, Exploiting concurrency flaws, Resource exhaustion, Targeted process crashing</td>
</tr>
<tr>
<td>Execution abstractions</td>
<td>Identity spoofing, Injection, Probing, Output information disclosure, Unauthorized access, Invoking unauthorized operations, Spoofing privileged processes, Unsafe code execution, Exploitation of tight component coupling, Process overflow attack, Exploiting concurrency flaws, Resource exhaustion, Targeted process crashing</td>
</tr>
<tr>
<td>Processes (OS)</td>
<td>Identity spoofing, Injection, Probing, Output information disclosure, Unauthorized access, Invoking unauthorized operations, Spoofing privileged processes, Unsafe code execution, Exploitation of tight component coupling, Process overflow attack, Exploiting concurrency flaws, Resource exhaustion, Targeted process crashing</td>
</tr>
<tr>
<td>Communication</td>
<td>Message secrecy violation, Message integrity violation, Message authenticity violation, Injection, Repudiation, Covert network channels</td>
</tr>
<tr>
<td>Messages</td>
<td>Message secrecy violation, Message integrity violation, Message authenticity violation, Traffic analysis, Session hijacking, Session state poisoning, Message flooding, Scanning, Probing, Covert network channels</td>
</tr>
<tr>
<td>Message channels</td>
<td>Message secrecy violation, Message integrity violation, Message authenticity violation, Traffic analysis, Session hijacking, Session state poisoning, Message flooding, Scanning, Probing, Covert network channels</td>
</tr>
<tr>
<td>Protocols</td>
<td>Message replay, Message re-use, Protocol field modification, Use of abnormal packet sizes, Use of abnormal packet sequencing, Use of reserved protocol packets, Protocol initial/end state exploitation, Data inference, Repudiation</td>
</tr>
<tr>
<td>Networking infrastructure (S/W)</td>
<td>All the threat patterns in the Network communication attacks and Network protocol attacks classes are relevant, considered this time at the lower network layers.</td>
</tr>
<tr>
<td>Addressing</td>
<td>Identity spoofing, Advantageous identity allocation</td>
</tr>
<tr>
<td>Addresses / identifiers</td>
<td>Identity spoofing, Advantageous identity allocation, Route poisoning, Message replay, Message re-use, Protocol field modification, Repudiation</td>
</tr>
<tr>
<td>Protocols / algorithms</td>
<td>Identity spoofing, Advantageous identity allocation, Route poisoning</td>
</tr>
<tr>
<td>Routing data structures (tables)</td>
<td>Identity spoofing, Advantageous identity allocation, Route poisoning</td>
</tr>
</tbody>
</table>

*Table 4: Mappings between technical realization abstractions and threat patterns from the (base) threat taxonomy's first level*
Threat analysis step 2: Once a list of threats has been compiled, the importance and possibility of each threat can be considered and ranked appropriately (i.e. some form of risk assessment can be performed).

Threat analysis step 3: When a particular requirement or policy has been determined, it can be “passed” vertically down the decomposition layers (as was done with the threat) to determine applicable enforcement strategies and/or further requirements. Note that this step applies to the incorporation of security requirements from previous development stages as well. It should be remarked that the combined decomposition and threat modeling process is iterative in nature, i.e. the steps above may need to be repeated as the architecture is being created. An additional step concerned with a consideration of meta-security threats from the second level of the threat taxonomy in [Uzunov and Fernandez, sub1] is described later, in Section 4.2.2.4.

4.2.1.2 Security Modeling (SecMod) stage

Either following, or in parallel with, the threat analysis process, corresponding technical (design-level) security requirements are determined for every important threat. These technical requirements are specified in one of two (canonical) forms:

- negative: \(<protect from X>\), where \(X\) is a given threat pattern instance; or
- positive: \(<enforce Y>\), where \(Y\) is the corresponding general (or specific) security policy encapsulated in the threat pattern encapsulating threat \(X\).

This style of specifying requirements reflects the duality of threat patterns/security solutions constituting the conceptual security framework for the methodology, as explained in Section 3, which is exploited in the next phase of the methodology to determine the appropriate countermeasures mitigating or stopping the threats.

The above specification of requirements also applies to the early (high-level) security requirements determined at the previous phase of the methodology (and previous development phase: Analysis), which are converted to design-level requirements via the aid of two mapping tables. The first of these tables – Table 5 (the Requirement class to policies table CF element in Figure 1) – maps general security requirement classes (first column) to generic security policies (third column). The requirements classes are described in the second column, with the descriptions taken from Firesmith [Firesmith, 2003] (often verbatim, and sometimes with adaptations). The second table – Table 6 (the Policies to threats table CF element in Figure 1) – maps generic security policies to threat classes and threat patterns within those classes from the base threat taxonomy (for general distributed systems) of Uzunov and Fernandez [Uzunov and Fernandez, sub1]. In this table, the rows with italics are the meta-security threat classes (differentiated further by the use of an asterisk in brackets) and patterns. It should be noted that while threat classes and patterns in the table pertain to the base threat taxonomy (for general distributed systems), the mappings also apply to specializations of the base patterns (the exceptions being patterns that augment or alter the base version of their encapsulated mitigating security policies), and hence for derived taxonomies.

The mappings in both tables (which we do not claim are exhaustive), should be applied during the analysis of different software components at different levels of granularity, including the security infrastructure itself – e.g. by considering the security information management infrastructure as a system in its own right, decomposing it, and considering threat patterns for the resulting architectural contexts.

Regarding threats that are linked together, if a given threat \(T\) contains in its description threat pattern instances \(TP_{n_1}, \ldots, TP_{n_k}\), then the set of technical security requirements can be expressed simply by using \(<protect from TP_{n_i}>\) clauses for each threat, but grouped under the single requirement. This provides traceability between the requirements and threats found during the threat analysis process. If each requirements were to refer to only one threat pattern instance at a time the traceability would be harder to keep.

Once specified in canonical form, the requirements can be used in the threat analysis process along with all others (see Threat analysis step 3).

In all cases the requirements are refined iteratively along with the threat analysis process, until a finalized set of specified requirements is obtained.

With respect to the construction of the methodology (see Section 2.2), this activity as a whole is an instance of the Security Modeling (SecMod) SPPF stage pattern, modified, however, by the encompassing Security Requirements Determination (SecReq), which implies that not countermeasures, but requirements, are being specified (according to the specify SPPF task pattern).
<table>
<thead>
<tr>
<th>Requirement Class (adapted from [Firesmith, 2003; Rosado et al., 2011])</th>
<th>Description of requirement class (verbatim from Firesmith [Firesmith, 2003]; with some adaptations)</th>
<th>Generic security policies (from [Uzunov and Fernandez, sub1])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification &amp; Authentication Requirements</td>
<td>Any security requirement that specifies the extent to which a business, application or component shall establish and verify the identity of its externals (e.g., human actors and external applications) before interacting with them.</td>
<td>Identity management</td>
</tr>
<tr>
<td>Authorization &amp; Trust Requirements</td>
<td>Any security requirement that specifies the trust, access and/or usage privileges of (in most cases authenticated) users and client applications.</td>
<td>Authorization, Execution control</td>
</tr>
<tr>
<td>Confidentiality Requirements</td>
<td>Any security requirement that specifies the extent to which an application or component shall ensure that its data and communications are not seen or accessed internally or externally by unauthorized parties.</td>
<td>Secure communications, Identity management, Authorization, Storage security</td>
</tr>
<tr>
<td>Integrity Requirements</td>
<td>Any security requirement that specifies the extent to which an application or component shall ensure that its data and communications are not intentionally corrupted via unauthorized creation, modification, or deletion.</td>
<td>Secure communications, Authorization, Execution control, Storage security</td>
</tr>
<tr>
<td>Immunity Requirements</td>
<td>Any security requirement that specifies the extent to which an application or component shall protect itself from infection by unauthorized undesirable programs (e.g., computer viruses, worms, and Trojan horses).</td>
<td>Execution control, Filtering</td>
</tr>
<tr>
<td>Exportability (Code mobility) Requirements</td>
<td>Any security requirement that specifies constraints on code requiring to be exportable and executable across different machines [Rosado et al., 2011].</td>
<td>Authorization, Execution control</td>
</tr>
<tr>
<td>Privacy Requirements</td>
<td>Any security requirement that specifies the extent to which a business, application or component shall keep its sensitive data and communications private from unauthorized individuals and programs.</td>
<td>Secure communications, Identity management</td>
</tr>
<tr>
<td>Nonrepudiation Requirements</td>
<td>Any security requirement that specifies the extent to which a business, application or component shall prevent a party to one of its interactions (e.g., message, transaction) from denying having participated in all or part of the interaction.</td>
<td>Secure communications, Identity management, Storage security, Logging and monitoring</td>
</tr>
<tr>
<td>Security Auditing Requirements</td>
<td>Any security requirement that specifies the extent to which a business, application or component shall enable security personnel to audit the status and use of its security mechanisms.</td>
<td>Logging and monitoring</td>
</tr>
<tr>
<td>Credentials &amp; Security Information Requirements</td>
<td>Any security requirement pertaining directly to the expression or management of the rights or identities of users or resources (e.g. for interdomain access) [Rosado et al., 2011], or to the expression or management of any security-related information.</td>
<td>Security information management</td>
</tr>
<tr>
<td>Intrusion Detection Requirements</td>
<td>Any security requirement that specifies the extent to which an application or component shall detect and record attempted access or modification by unauthorized individuals.</td>
<td>Logging and monitoring, Filtering, Execution control</td>
</tr>
<tr>
<td>Availability &amp; Survivability Requirements</td>
<td>Any security requirement that specifies the extent to which an application shall survive the intentional loss or destruction of a component, or that specifies the extent to which a constituent component or communications stream shall be available and fully operational.</td>
<td>Storage security, Logging and monitoring, Secure communications, Execution control</td>
</tr>
<tr>
<td>System Maintenance Security Requirements</td>
<td>Any security requirement that specifies the extent to which an application or component shall prevent unauthorized modifications (e.g., defect fixes, enhancements, updates) from accidentally defeating its security mechanisms.</td>
<td>Logging and monitoring, Authorization, Secure communications</td>
</tr>
</tbody>
</table>

Table 5: Mappings between security requirement classes and generic security policies

285
<table>
<thead>
<tr>
<th>Generic security policy</th>
<th>Threat classes (from base taxonomy)</th>
<th>Threat patterns (from base taxonomy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization</td>
<td>Uncontrolled operations</td>
<td>Unauthorized access, Invoking unauthorized operations, Spoofing privileged processes (transitive actions), Unsafe code execution, Exploitation of tight component coupling</td>
</tr>
<tr>
<td></td>
<td>Remote information inference</td>
<td>Data inference</td>
</tr>
<tr>
<td></td>
<td>Network communications attacks</td>
<td>Session state poisoning</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Countermeasure design (*)</td>
<td>Use of default credentials, Bypassing controls, Leveraging authorization model</td>
</tr>
<tr>
<td></td>
<td>Configuration / administration (*)</td>
<td>Exploiting bad policies, Unauthorized modification of rights (meta-authorization policies)</td>
</tr>
<tr>
<td>Identity management</td>
<td>Identity spoofing</td>
<td>Identity spoofing, Advantageous identity allocation</td>
</tr>
<tr>
<td></td>
<td>Cryptography attacks (*)</td>
<td>Forging cryptographic credentials, Abuse of weak algorithm, Exploiting vulnerable security protocol, Password attacks (guessing, brute force, rainbow tables etc.)</td>
</tr>
<tr>
<td></td>
<td>Countermeasure design (*)</td>
<td>Use of default credentials, Bypassing controls</td>
</tr>
<tr>
<td>Secure communication</td>
<td>Network communications attacks</td>
<td>Message secrecy violation (passive eavesdropping, reading plaintext message), Message integrity violation (active eavesdropping, modification), Message authenticity violation (source spoofing), Traffic analysis / protocol sniffing, Covert network channel, Session hijacking, Session state poisoning, Route poisoning, Message flooding</td>
</tr>
<tr>
<td></td>
<td>Network protocol attacks</td>
<td>Message replay, Message re-use, Protocol field modification, Use of abnormal packet sizes, Use of abnormal packet sequencing (re-ordering), Use of reserved protocol packets, Protocol initial/end state exploitation</td>
</tr>
<tr>
<td></td>
<td>Loss of accountability</td>
<td>Repudiation</td>
</tr>
<tr>
<td></td>
<td>Cryptography attacks (*)</td>
<td>Abuse of weak algorithm, Exploiting vulnerable security protocol</td>
</tr>
<tr>
<td>Filtering</td>
<td>Passing illegal data</td>
<td>Injection</td>
</tr>
<tr>
<td></td>
<td>Remote information inference</td>
<td>Scanning (information gathering), Probing (vulnerability checking), Output information disclosure</td>
</tr>
<tr>
<td></td>
<td>Network communications attacks</td>
<td>Message flooding</td>
</tr>
<tr>
<td></td>
<td>Network protocol attacks</td>
<td>Protocol field modification, Use of abnormal packet sizes, Use of abnormal packet sequencing (re-ordering), Use of reserved protocol packets</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Configuration / administration (*)</td>
<td>Exploiting bad policies</td>
</tr>
<tr>
<td>Storage security</td>
<td>Stored data attacks</td>
<td>Corruption</td>
</tr>
<tr>
<td></td>
<td>Remote information inference</td>
<td>Data inference</td>
</tr>
<tr>
<td></td>
<td>Passing illegal data</td>
<td>Injection</td>
</tr>
<tr>
<td></td>
<td>Cryptography attacks (*)</td>
<td>Abuse of weak algorithm, Exploiting vulnerable security protocol</td>
</tr>
</tbody>
</table>
Logging and monitoring | Loss of accountability | Track erasing, Repudiation  
| Network communication attacks | Covert network channel, Message flooding  
| Remote information inference | Scanning (information gathering), Probing (vulnerability checking)  

Execution control | Uncontrolled operations | Unsafe code execution, Exploitation of tight component coupling, Process overflow attack (buffer / integer overflows), Exploiting concurrency flaws (TOCTOU), Resource exhaustion  

Security information management | Cryptography attacks (*) | Exploiting vulnerable security protocol  

| Table 6: Mappings between generic security policies to (general distributed systems) threat classes and patterns |

4.2.2 Countermeasure Introduction (CounterIntro) phase

4.2.2.1 Countermeasure Identification (CounterIdentif) stage

Once requirements are specified in the canonical forms as per the Requirements Determination phase, they must all be mapped to concrete countermeasures. As with the threat modeling, there are two approaches to achieving this. In the first approach, countermeasures are determined from the generic policies encapsulated by the threat patterns according to a given threat (in the threat enumeration). In the second approach, tactics are directly used to improve certain (security) quality attributes, such as confidentiality, integrity etc. as per the traditional usage of tactics (see [Bass et al., 2003]). The duality of threats-solutions allows an immediate evaluation of whether a give tactic is needed by considering the relevant threat patterns and their applicability in a given context (as per the threat modeling phase in Section 4.2.1 above). Below we describe the first approach in more detail.

Requirement to countermeasure mapping:

As a first step to determining countermeasures from security requirements, all negative requirements are converted into their (dual) positive form on the basis of the generic security policies encapsulated in the relevant threat patterns, i.e.:

\[
<\text{protect from } X> \rightarrow <\text{enforce [generic security policy encapsulated by } X]> 
\]

In this way all requirements are converted into a set of policies requiring enforcement. Those requirements already in positive form referring to specific policies are converted to requirements for enforcing generic policies with the mapped to requirements references alongside (e.g. in brackets, or whatever other notation is deemed suitable):

\[
<\text{enforce } Y> \rightarrow <\text{enforce [generic security policy } Y^* \text{ of which } Y \text{ is a refinement]} (Y)> 
\]

The list of these requirements in positive form are subsequently mapped to concrete countermeasures at various levels of abstraction and granularity (as required) in one or more of the following ways:

1. to security tactics embodying the corresponding generic policies (see Section 3), which in turn are realized by security solution frames, and, subsequently, by relevant security pattern families within those frames.
2. directly to security solution frames and, subsequently, to relevant security pattern families within those frames. Since the names of the generic policies are identical to the names of the solution frames, this mapping is very straightforward. Following the organization of the relevant families, requirements can also be mapped to the encapsulated patterns which address more specific policies.
3. to standalone or compound security patterns – especially for a distributed systems context (see [Uzunov et al., 2012a, Fernandez and Uzunov, 2012, Fernandez and Larrondo-Petric, 2007]) – that are not part of any solution frame. This particular mapping is realized with the help of another table – Table 7 – which, acting as an adapter, maps requirements to security patterns via distributed systems security concerns through generic policies. This gives developers the option to use standalone security patterns when a frame or pattern within a frame for a given generic or specific policy, respectively, is not available and/or not appropriate. For the concerns in the table we follow the scheme used in [Uzunov et al., 2012a]; the last column lists some example security patterns (see [Fernandez, 2013] and [Uzunov et al., 2012a] for descriptions and references).
4. if a given specific security policy is not addressed by either standalone patterns or patterns constituting the solution frames, then developers can resort to using custom, on-the-fly, expert-created countermeasures to realize a given policy.
In the case of mappings 1 and 2, any defined solution relationships should be followed to ensure that the selected solutions are not incompatible, including the relationships between security frames (as in Table 2), pattern families and individual patterns within, and across, other families – as well as between standalone patterns where appropriate (as in [Yskout et al., 2006]; see also [Fernandez, 2013]). Following these relationships not only allows for a more complete coverage of the security requirements, but can also help to cover unexpected requirements arising from the selection of particular solutions (cf. [Hatebur et al., 2008]).

Collectively, the four mappings outlined above promote flexibility in employing any combination of security tactics, security solution frames (and their encapsulated patterns) and standalone security patterns – as well as custom-made solutions – in realizing the policies resulting from the security requirement conversion. The use of the mapping tables (Tables 3 to 7) provides strong traceability between security solutions, threats – as well the architectural contexts, and the components therein, to which the threats apply – policies and requirements.

<table>
<thead>
<tr>
<th>Generic security policy (from [Uzunov and Fernandez, sub1])</th>
<th>Distributed systems security concern (following [Uzunov et al., 2012a])</th>
<th>Example security patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization</td>
<td>distributed authorization and access control</td>
<td>Remote Authenticator/Authorizer, Bodyguard</td>
</tr>
<tr>
<td>Identity management</td>
<td>distributed authentication, identity management,</td>
<td>Identity Federation, Privacy pattern language [Hafiz, 2013]</td>
</tr>
<tr>
<td>Secure communication</td>
<td>secure communication</td>
<td>VPN, Secure Client/Server Communication, Secure Distributed Publish/Subscribe</td>
</tr>
<tr>
<td>Filtering</td>
<td>data filtering</td>
<td>Object Filter Architecture, XML Firewall</td>
</tr>
<tr>
<td>Storage security</td>
<td>distributed authorization and access control</td>
<td>OFAC Framework, Data Integrity in P2P Systems [Schleinzer and Yoshioka, 2010], Encrypted storage [Kienzle et al., 2002]</td>
</tr>
<tr>
<td>Execution control</td>
<td>(secure) distribution control, (secure) processing</td>
<td>Secure Broker, Object Mediation, Secure Pipes and Filters, Server Sandbox [Kienzle et al., 2002]</td>
</tr>
<tr>
<td>Logging and monitoring</td>
<td>monitoring</td>
<td>IDS</td>
</tr>
<tr>
<td>Security management</td>
<td>information security contexts</td>
<td>Credential, Security Policy [Kienzle et al., 2002]</td>
</tr>
</tbody>
</table>

**Table 7: Mappings between generic security policies and distributed systems security concerns**

### 4.2.2.2 Security Modeling (SecMod) stage

When appropriate countermeasures have identified, their encapsulated solutions can be realized in the system's software architecture. Regardless of whether one started with security tactics to arrive at solution frames, or directly selected relevant frames, the process for incorporating the encapsulated solutions proceeds by following the abstraction levels of the pattern family structures as per the implicit or explicit implementation micro-process pattern contained in the frame. More specifically, this implies that first abstract patterns are selected from a given family, which define the concepts of the solution and an abstract architecture; then the abstract architecture is refined in successive levels of concreteness, leading to a detailed design; and finally, a number of technology realizations are chosen for the design to be transferred to (code-level) implementation. In this way developers are guided in introducing particular security solutions in a step-wise fashion.

Throughout this process, the patterns are related to the system characterization – namely, the original architectural decomposition from the earlier phases – beginning with the highest layer of the functionality decomposition layers and proceedings downwards, relating the patterns to the abstractions (and the corresponding parts of the actual software architecture) in each succeeding layer as was done in the Adversary Modeling stage during Design (see Section 4.2.1.1). This extends the similar process in the methodology of Fernandez and colleagues.

A given pattern or collection of patterns (for a given iteration, since this activity is done several times) is instantiated into the software architecture as per the standard pattern instantiation approach (mapping participants to components, or realizing them with collections of components – see [Buschmann et al., 1996, Mouratidis et al., 2006, Uzunov et al., 2012a]) to produce a secure software architecture. The security slice of this (i.e. the security architecture proper) consists of all the pattern instances and their relationships, and, due to the very nature of patterns, is itself re-usable. The following strategies – adapted from the work of [Kamal et al., 2011] – are used to resolve pattern participants during the instantiation process:
• absorb participants: the functionality of a given participant in one pattern is absorbed in the components of another pattern instance.
• merge participants: the functionality two or more participants in several patterns is combined and realized in a single, multi-functional component.
• employ another pattern: another pattern is used to realize the participants of a given pattern. Note that this can be recursive, leading to multiple pattern instances (usually of non-security architectural or design patterns) for a given security pattern.

UML is used as a modeling language as per the original methodology of Fernandez and colleagues.

With respect to specific micro-process patterns that can be employed during this stage, special attention should be given to Secure Protocol Design from the Secure communications frame [Uzunov and Fernandez, sub2] whenever protocols of any kind are designed as part of the standard development activities (usually at detailed design). Such protocols could be of a general nature, “for describing how components in a distributed system synchronize, exchange messages and share resources” [Brooke and Paige, 2008]; or for security-related tasks, such as how cryptographic keys are distributed.

4.2.2.3 Security Verification (SecVerif) stage

The next step in the process is to verify that the security architecture resulting from instantiating the different patterns does indeed satisfy the requirements, and, in particular, mitigates the threats enumerated earlier. This is accomplished by tracing the solutions to the original threat patterns and set of security use cases constructed during the design and analysis development stages, respectively, and inspecting whether they are addressed appropriately in the security architecture. In terms of the SPPF framework of [Uzunov et al., sub2], this process is essentially an instance of the Manual inspection task pattern.

4.2.2.4 Re-iteration of the Security Requirements Determination phase

Threat analysis step 4 (on re-iteration): Since software components can either relate to the system's business (functional) concerns or, in this case, to security (see [Uzunov et al., 2013]), it is necessary to repeat the decomposition and threat analysis described in Section 4.2.1.1 for each newly introduced security component, protocol etc. to protect the security infrastructure itself from threats. To do this, the threat analysis steps 1 to 3 can be repeated, this time selecting appropriate threat patterns from the meta-security threats class (see [Uzunov and Fernandez, sub1] for details).

4.3 Implementation (Impl)

Although our focus is on design-level security, for the sake of completeness in this section we briefly describe ASE’s activities performed during the implementation development phase. We do not consider any activities aligning with the deployment development phase, even though some solution frames provide patterns that are specifically applicable then (e.g. for secure network configuration [Uzunov and Fernandez, sub2]).

4.3.1 Security Implementation (SecImpl) phase

4.3.1.1 Code and Map stages

During this phase, the security pattern instance that collectively form the security architecture are coded together with the rest of the system. Secure coding practices are used throughout (see [Dowd et al., 2007, McConnell, 2004]), with appropriate language idioms (e.g. [Walker et al., 2012] for C constructs) employed where appropriate. Static analysis tools can be used during this stage to increase code quality and hence to reduce the number of vulnerabilities arising (purely) from the implementation [Chess and West, 2007, Chahar et al., 2012].

There may also be a need to map certain features or components of the architecture to COTS components (as per the Map SPPF stage pattern in [Uzunov et al., sub2]), in which case the WRAPPER FACADE pattern [Schmidt et al., 2000] can be used to ensure appropriate security functionality if it is not already present in the target component(s).

4.3.1.2 Security Verification (SecVerif) stage

Once all iterations of the security implementation stage are completed, the resulting software system must be carefully verified as to whether it really does satisfy the security architecture specifications. This is accomplished by considering misuse and attack pattern realizations (see [Fernandez et al., 2009] and [Hoglund and McGraw, 2004], respectively) of each of the threat patterns found during the design development phase, and performing penetration testing on the software system (cf. [Uzunov and Fernandez, sub1]), as per the Penetration test SPPF task pattern. This helps to not only verify that a particular countermeasure has been implemented correctly, but to also determine whether that countermeasure is effective against (corresponding) representative attacks.
5 Applying ASE in practice

In this section we demonstrate how ASE is applied in practice to the development of an architecturally non-trivial, secure distributed system. In keeping with our focus on design-level security, we apply ASE up to, but excluding, implementation (hence we do not demonstrate ASE’s implementation stages, which were described very briefly in Section 4.3). We omit a number of the details in our descriptions of how ASE is applied during requirements analysis, since they are well documented in [Braz et al., 2008] and [Fernandez et al., 2006, Fernandez, 2013] (cf. Section 4.1). This allows us to focus on the novel contributions of ASE during design, as we did in the description of ASE itself (cf. Section 4.2).

As remarked earlier, security methodologies are entities independent of, but related to, development methodologies, within which they must be integrated. Unlike the methodology of Fernandez et al., which is integrated or at least aligned to a generic object-oriented methodology, we do not wish to advocate any particular paradigm or development methodology. We thus take a development approach that can be termed loosely “architecture-oriented”. The SDLC for this approach is aligned with the generic life-cycle in [Uzunov et al., sub2], and begins with a requirements analysis phase encompassing the familiar requirements elicitation/analysis activities (including, in particular, the creation of a domain-driven conceptual model, as per the DOMAIN MODEL pattern of [Buschmann et al., 2007]). Transitioning from the problem to the solution space, development subsequently focuses on the design of software architectural models via the creation of manager components (akin to the generic services of [Chang et al., 1987]), which encapsulate the requirements and provide some piece of functionality at a logical level; as well as via the application of software patterns – both architectural [Buschmann et al., 1996] and domain-specific [Avergiou and Tandler, 2006, Guerrero and Fuller, 2001, Lukosch and Schümmer, 2004]. The resulting models are separated into several architectural views (mixing the available views from the literature, both older and more recent [Jain and Parao, 1991, Taylor et al., 2010]): logical, execution and deployment (physical). The models are refined and elaborated and then transitioned into implementation. Regardless of our brief, explicit description of our espoused development approach above (purposefully not a full development methodology per se), we must emphasize that it is by no means the only possibility, and that other development approaches with different paradigms can also be employed in conjunction with ASE.

In the ensuing presentation we follow the order development phases, as we did when describing ASE, and within this the order of ASE’s phases and stages, with the shorthand notation from [Uzunov et al., sub2] in brackets next to each phase or stage name.

5.1 Requirements analysis (ReqAn)

5.1.1 Outline of system functionality

The purpose of our example system, called SHARED (SHAring and collaboRative EDiting environment), is to offer intra- and inter-organizational file sharing and multi-user, collaborative document editing capabilities. Each user has a single “Sharespace” containing graphical representations of the files that are shared for a given company project: documents, supporting multimedia files and executables. Files can be downloaded locally, and new ones can be shared with other team members/collaborators through the same interface. Each users can edit document files collaboratively via a graphical user interface integrated with their personal Sharespace interface (on their local machine), based on a partially synchronous paradigm. During editing, an authoring user selects a section to edit (or end-of-file to augment the document), which causes it to be locked, and after editing makes an explicit update, which causes an event to be broadcast to other participants.

Figure 3. Users can thus work on a document simultaneously while all beings online in a collaborative session, as well as off-line, whereby changes are propagated and synchronized at startup. While all the participants are part of the same organization (or a linked group), external reviewers can also take part in viewing and commenting on the documents. A conceptual model of SHARED is shown in Figure 3. The model contains a (customized and adapted) instance of the COMMUNICATIONS ARCHITECTURE FOR GROUPWARE SYSTEMS pattern from [Ochoa et al., 2002] to capture a number of the generic functional aspects. A collection of use cases for SHARED is shown in Figure 2. The most notable ones being Share file (with Share document – whereby a document is created and shared with other users or with the world – as a derivative) and Edit document. We omit the descriptions of these use cases for the sake of space, and also because they are largely self-explanatory and/or well known from similar existing (collaborative) systems.
5.1.2 ASE's Security Requirements Determination (SecReq at Req) and Countermeasure Introduction (CounterIntro at Analysis) phases

Applying ASE’s early phases (at Requirements analysis – see Figure 1), we consider how the system uses can be thwarted by an insider (user/author) or an external attacker, with respect to confidentiality, integrity, availability and accountability. In performing this task we employ the threat patterns in an ad-hoc fashion to provide additional guidance beyond our own expertise. The partial results are tabulated in Table 8, with the first two columns referring to the use case and the remaining columns describing the possible misuse activities. The latter misuse activities give rise to a set of early security requirements, several of which we make explicit and summarize below (with encompassing security requirement class from Table 5 in brackets, and relevant misuse activities from Table 8 after the arrow):

- **SR1**: only authorized users with appropriate rights should be able to view documents *(Authorization & Trust Requirements, Confidentiality Requirements)* ← M6, M8
- **SR2**: only authorized users with appropriate rights should be able to edit documents *(Authorization & Trust Requirements, Integrity Requirements)* ← M4, M9
- **SR3**: files sharing and download should be restricted to a subset of users (e.g. team/project) according to appropriate rules *(Authorization & Trust Requirements)* ← M1, M3, M7
- **SR4**: shared files should be “sanity checked” or adhere to some kind of rules *(Immunity Requirements, Exportability (Code mobility) Requirements)* ← M2
- **SR5**: users should be accountable for their document editing changes, and should not be able to deny they have made a change *(Nonrepudiation Requirements, Integrity Requirements)* ← M5

![Figure 3: Conceptual model of SHARED](image-url)
When complete, the latter list forms part of the early requirements of SHARED (along with any prescribed requirements, which we have consciously omitted). Carrying the system uses into ASE's Security Modeling (SecMod at ReqAn) stage, we can consider abstract security patterns embodying the early security requirements or forming an initial solution for a given problem. Using Table 5 in the process, we determine to use the Authorization security solution frame for setting up appropriate authorization models in response to SR1, SR2 and SR3. Since there are effectively two main sets of functionality in SHARED – file sharing and document editing – each part requires a different authorization model. We begin by instantiating the AUTHORIZATION pattern from the frame's Conceptual analysis abstraction level twice, with appropriate merging of elements, and subsequently add pattern modifiers for the specifics of file sharing:

- the GROUP modifier, to group together authors of a given document;
- and a second instance of the GROUP modifier, to group together files based on their properties (ATTRIBUTE modifier);

as well as for the specifics of document editing:

- a ROLE modifier, whereby each user can be an Author, a Co-author or a Reviewer;
- FINE GRANULARITY, so that not only individual documents, but also the editing actions over parts or paragraphs of a document can be controlled;
- SECURITY SESSION, so that editing actions need only be authorized once per editing/collaborative session (for a given ROLE or a given author);
- USER-DRIVEN AUTHORITY, whereby the Author ROLE has control over the file, and other roles can have ownership rights.

The last modifier was used as a result of considering the modifier relationships in [Uzunov et al., sub1] for the given context. With respect to the early security requirements, the two instances of the abstract pattern AUTHORIZATION can be thought as embodying SR1/SR2 and SR3, respectively (i.e. the further development of the solution is required to take into account the model), while the modifiers give initial solutions to the problem of authorizing users, which is worked out during design.

At this point we also consider the possible actions (at an abstract, conceptual level) that can be performed in file sharing and in editing and link the rights associated with the subject-object pairs (resulting from ABSTRACT AUTHORIZATION) accordingly.

- Sharing environment actions (initial): add file, share file, download file, open document
- Editing actions (initial): read document, update document, read paragraph, update paragraph, modify document, modify paragraph

The refinement and determination of the specifics of these actions and rights is left for the design development phase (Section 5.2).

The "secured" conceptual model resulting from the pattern instantiations is shown in Figure 4, with green elements (or light gray, if viewing in black and white) being those that have been modified; gray elements are those arising from the file sharing (conceptual) authorization model; and black elements are those arising from the document editing authorization model.
In similar fashion, we can consider abstract security patterns for the other requirements and instantiate them into the conceptual model to guide the development of the pertinent solutions.

5.2 Design (Des)

5.2.1 Outline of SHARED's design

SHARED's initial, “insecure” architecture is constructed by instantiating a number of (non-security) architectural patterns and applying appropriate partitioning (see [Sardjono and Simons, 2004]). Following the patterns of [Avgeriou and Tandler, 2006], the resulting architectural design can be described briefly as follows:

- DISTRIBUTION MODEL: We employ the DISTRIBUTED APPLICATION KERNEL client/server pattern from [Renzel and Keller, 1997] for the overall distributed architecture.
- MESSAGE EXCHANGE MODEL: The messaging infrastructure is realized using BROKER for the client/server communication, PUBLISH/SUBSCRIBE (see [Buschmann et al., 1996] for both patterns) for notifying users of editing updates. From a higher-level perspective, the communications infrastructure forms a single unit spanning both the client and server.
- FUNCTIONAL DECOMPOSITION MODEL: The core functionality is contained in the application server, which

![Figure 4: Secured conceptual model with authorization patterns applied](image-url)
manages the sharing of files as well as all editing sessions. All components are structured into five LAYERS [Buschmann et al., 1996], addressing concerns in (ordered highest to lowest): session management; event management; file and document management; notification management; and communications management. The client functionality is also stratified according to the LAYERS pattern in a similar fashion, closely mirroring the structure of the server, with a top-level layer addressing user interface concerns and subsequent layers encapsulating editing functionality, event management, document management and synchronization and communications management. The overall structure of the clients and server also accords with the three-layered re-usable architecture proposed by Neyem et al. [Neyem et al., 2008] (with the Collaboration layer in the latter containing all user interfaces/editing and sharing functionality; the Coordination layer containing all session, event and data management details; and the Communications layer containing the messaging infrastructure).

On the client side, the editing functionality is realized using the MODEL-VIEW-CONTROLLER (MVC) pattern [Buschmann et al., 1996], with the model being the (local, cached) document, the view contained in the user interface, and the controller residing partially in the user interface and partially in the editing management components.

– SHARING MODEL: The data shared (from a purely functional viewpoint) includes the files, of which documents are a subset, accompanying (static) meta-information (INFORMATION OBJECT pattern [Guerrero and Fuller, 2001]), as well as editing information such as the editing state and session information (cf. [Avgeriou and Tandler, 2006]). All files have full persistence (PERSISTENCY pattern [Avgeriou and Tandler, 2006]) and are stored by the server (CENTRALIZED OBJECTS pattern [Lukosch and Schümmer, 2004]), which in turn uses a SHARED REPOSITORY [Lalanda, 1998] for back-end data storage, thus off-loading some of the data management burdens. Server and repository replicas can also be employed, however, we do not consider this explicitly, since it complicates the example.

During editing, documents are downloaded by each (authorized) user to their machine and are managed by a local cache, in addition to being stored on the server, implying a semi-replicated (DATA) DISTRIBUTION SCHEME [Avgeriou and Tandler, 2006].

– CONCURRENCY MODEL: The REACTOR pattern [Schmidt et al., 2000] is used in the server's event management layer, to demultiplex events and create appropriate handlers for both file sharing and document editing activities. The use of PUBLISH/SUBSCRIBE introduces event asynchrony, hence HALF-SYNC/HALF-ASYNC [Schmidt et al., 2000] is used across the event management and editing layer components for handling (incoming) notification events and (outgoing) editing events.

– SYNCHRONIZATION MODEL: Non-document files do not need to be synchronized, since sharing takes on a global namespace (everything being controlled by the server first in this initial design). Documents, however, do need synchronization. This is implemented in the session management components, where users choose to edit particular sections of a document, which in turn are locked (DON'T TRUST YOUR FRIENDS [Lukosch and Schümmer, 2004]). This also implies a turn-taking FLOOR CONTROL approach (see [Avgeriou and Tandler, 2006]) to editing document sections (though not whole documents). When a user updates the document, the change is sent to the server and again passed on to the session management layer, and from there to document management components in lower layers where the changes are handled. Notifications are subsequently sent to all users in a given session, or set as pending for when the authors are online (MEDIATED UPDATES pattern [Lukosch and Schümmer, 2004]).

A model of the resulting architecture can be seen in Figure 5. This model attempts to capture several architectural views: the colours indicate the different layers that are part of the FUNCTIONAL DECOMPOSITION MODEL, with their major constituent components, some of which are shown in more detail with their internal composition, in boxes (logical view); arrows indicate the flow of control (execution view); and the dashed borders indicate machine boundaries (deployment view).
In what follows we describe the application of ASE during the Design (development) phase.

5.2.2 ASE Security Requirements Determination (SecReq at Des) phase

5.2.2.1 Adversary Modeling (AdvMod at Des) stage

Taking the architectural model in Figure 5 (together with the descriptions in Section 5.2.1) as a starting point, we use the distributed architecture decomposition framework (see Section 3) to create a derived architectural model, partially shown in Figure 6. For the derived model we have used a UML-like notation, where the green (or gray, if viewing in black and white) associations are compound from the existing associations (which are modeled as UML associations, without associations classes for the sake of saving space). The dashed one-way arrows indicate that particular software components reside on a given physical node; dashed two-way arrows between components is a shorthand notation for indicating that two components reside on the same node. Regarding the level of detail, making this model (or others like it) more detailed, by, for example, considering the sub-components of the major components, will provide greater breadth for the threat analysis; however, in this case this would also lead to unnecessary complexity and make the example less tractable.
We assign each of the components in the derived model to functionality decomposition layers. This is shown partially in Figure 7, where most components and some associations have been assigned to appropriate layers (note that a given component and/or association can straddle multiple layers, and can also be reassigned if needed). In the figure letter are used as a shorthand notation for the layers: User interaction (U), Resource management (R), Data/Storage management (S), Distribution control (D), Communications (C) and Addressing (A). Since at this point in the development, and at this level of abstraction, we do not know how all the architectural elements will be realized (e.g. the Broker may be realized using CORBA, or may be realized using a custom, simplified RPC-based infrastructure), we make conjectures regarding the technical realization abstractions as required.

We next perform the threat analysis by considering threat patterns from the base taxonomy as per Tables 3 and 4,
starting with the User Interaction decomposition layer (and the relevant abstractions there) and proceeding downwards. We analyze both the client and the server together, and also separately. Once a threat was determined to be likely and selected for inclusion in our list, we passed it down the decomposition layers to consider it against other architectural contexts and abstractions (and hence components to which the latter correspond) as appropriate. To complement this, we also consider threats from the taxonomy individually, and match them to appropriate contexts. We group together threats (i.e. threat pattern instances) that are linked or result from the realization of one or more other threats.

Below we briefly outline the resulting list of threats (at the application level – we do not consider threats to the underlying infrastructure for this example), focusing on those with the greatest likelihood. We continue to use the shorthand notation for the decomposition layers in the brackets describing the architectural context.

- **T1**: Identity spoofing (D – all <maps to: client-side UI>) via Identity spoofing (U – input ports <maps to: Sharespace interface>): users can attempt to gain unauthorized access to files and documents if there is no mechanism to check their identities.

- **T2**: Repudiation (U – input ports <maps to: editing interface>) via Track erasing (S – data structures <maps to: client document management components>) and/or Track erasing (S – file systems): users can make changes to a document and subsequently manipulate the local data stores prior to synchronization so that updates are not detected properly, leading to erasure of changes.

- **T3**: Data inference (S – storage abstractions <maps to: documents local to Client>) linked to Data inference (U – output ports <maps to: editing interface>): users can infer additional data/sections from a file, which they would otherwise not be able to.

- **T4**: Corruption (S – <user meta-data on Client>) linked with Corruption (S – <user meta-data on Server>): users can manipulate any number of abstractions to change their local meta-data.

- **T5**: Unauthorized access (R – resources & S – storage abstractions <maps to: file store on Server>) via Invoking unauthorized operations (D – operations <maps to: Server event handler>): malicious requests could result in files being downloaded without appropriate rights.

- **T6**: Resource exhaustion (R – resources & S – database systems <maps to: file store on Server>): purposefully sharing a large number of files or files with a large size could cause depletion of space on the server storage back-end.

- **T7**: Session state poisoning (S – data structures <maps to: client data store>): editing session information carried through the communications medium can be modified by corrupting session information (see also T6).

- **T8**: Exploiting concurrency flaws (D – all <maps to: event handling mechanisms>): purposeful or accidental requests can be made that affect the asynchronous handling of editing (especially update) events, causing clashes and/or inconsistencies of document data.

- **T9**: Exploiting concurrency flaws (R – algorithms <maps to: document management>): it may be possible to exploit the algorithms for updating documents to either erase tracks or cause corruption (see also T2, T4).

- **T10**: Message secrecy violation (C – all abstraction <maps to: communications infrastructure on clients and server>): attackers can eavesdrop on the communications medium to obtain document and/or file information.

- **T11**: Message integrity violation (C – all <maps to: comm. infr. on clients and server>): attackers can eavesdrop on the communications medium to change document and/or file information, potentially without detection.

- **T12**: Message authenticity violation (C – messages and message channels <maps to: comm. infr. on clients and server>): attackers or users can change the source of their message to interfere with the collaborations or obtain information.

Some threats were deemed feasible, but unlikely, e.g. Injection (S – database systems <maps to: Server database>) via Injection (U – input ports <maps to: editor interface>), which although theoretically possible, would be made difficult since the flow of control to the database back-end is interrupted by a number of other components that transform the requests into various formats.

### 5.2.2.2 Security Modeling (SecMod at Des) stage

Following (or simultaneously with) the Adversary Modeling (at Design) stage, we form a list of canonicalized security requirements (refer to Figure 1 for the flow of activities) based on the threats from the analysis process as well as the early security requirements. More specifically, the early requirements are converted to technical requirements using Table 5 as follows:

- **TSR1**: <enforce Authorization (for document viewing and editing)> ← SR1, SR2
- **TSR2**: <enforce Authorization (for file sharing)> ← SR3
- **TSR3**: <enforce Identity management (for users)> ← SR1, SR5
• TSR4: <enforce Storage security (documents)> ← SR1, SR2
• TSR5: <enforce Storage security (shared files)> ← SR3
• TSR6: <enforce Storage security (ID binding and hashing of documents)> ← SR5
• TSR7: <enforce Execution control (for shared files)> ← SR1, SR2
• TSR8: <enforce Logging and monitoring (document changes)> ← SR5

These technical requirements are relatively specific, since the mapping of early requirement classes to generic policies was guided by our expertise. If developers are not experienced, however, then the more generic forms of, for example, “<enforce Storage security> ← SR1, SR2, SR3, SR5” should be sufficient, and can be refined during later ASE stages.

Regarding the technical security requirements arising from the threat analysis, some concrete examples in summarized form are:

• TSR9: <protect from Identity spoofing (D – all)>, <protect from Identity spoofing (U – input ports)>
• TSR13: <protect from Unauthorized access (R – resources & S – storage abstractions)>, <protect from Invoking unauthorized operations (D – operations)>
• TSR15: <protect from Session state poisoning (S – data structures)>
• TSR17: <protect from Exploiting concurrency flaws (D – all)>
• TSR18: <protect from Message secrecy violation (C – all)>

5.2.3 ASE Countermeasure Introduction (CounterIntro at Des) phase

5.2.3.1 Countermeasure Identification (CounterIdentif at Des) stage

Having enumerated all the technical requirements, we convert them all into positive form to determine the precise realizations, thereby extending and consolidating the previous Countermeasure Identification stage performed during Analysis (Section 5.1.2). The reference to architectural contexts is kept to give guidance during the modeling of corresponding solutions. Besides TSR1 to TSR8 (above), which are already in positive form, some example conversions (briefly summarized) are:

• TSR9 → <enforce Identity management> at the Client user interfaces
• TSR2, TSR13 → <enforce Authorization> for file sharing, at the Server-end
• TSR15 → <enforce Storage security> on the Client-end meta-data
• TSR16 → <enforce Execution control (process serialization)>
• TSR18 → <enforce Secure communications> between the Client and Server components

As shown above, some requirement mappings may overlap or come from early requirements. Putting all the requirements together and using the four mappings from Section 4.2.2.1, allows us to arrive at the following countermeasures (we have decided not to use tactics directly in this example):

Solution frames:

• Authorization solution frame (as already chosen in Section 5.1.2), for controlling sharing and editing actions ← TSR1, TSR2, TSR13
• Identity management solution frame (Authentication pattern family) ← TSR3, TSR9
• Secure communications solution frame ← TSR18 to TSR22
• Security information management solution frame (Policy management pattern family) ← as a result of using Table 1 (solution frame relationships) for Authorization
• Security information management solution frame (Key management pattern family) ← as a result of using Table 1 (solution frame relationships) for Secure communications

Appropriate patterns can be selected in accord with the structure of the solution frames to develop the solution for a set of given requirements. In this sense, a further, finer-grained identification of countermeasures is obtained by following the solution frame structures.

Stand-alone and/or compound security patterns:

Since solution frames are not available for some requirements (e.g TSR15 from T7, or TSR16 from T8), we use Table 7 and the patterns catalog in [Uzunov et al., 2012a], to determine appropriate stand-alone or compound patterns. We omit the details of this step.

5.2.3.2 Security Modeling (SecMod at Des) stage

At this stage we design the security architecture of SHARED using the stand-alone patterns and the patterns
encapsulated in the solution frames identified previously. Below we describe in some detail the design of the enforcement architectures for the two conceptual authorization models (document editing and file sharing) from Section 5.2.2.2, and in less detail the design of the secure communications and authentication infrastructures—all using the previously identified solution frames. For the sake of simplicity, in our descriptions we do not always strictly follow the progressive refinement process inherent in the pattern family structures, but discuss only the final results, after the design decisions have been made.

**Secure communications infrastructure:**
From the secure communications solution frame (Secure two-party communications pattern family) we first instantiate SECURE MESSAGE CHANNEL between the clients and the server, which will be placed inside the Communications Infrastructure components (see bottom part of Figure 5). In satisfying the security properties implied by requirements TSR18 to TSR22 we also add to the SECURE MESSAGE CHANNEL the modifiers MESSAGE ENCRYPTION, MESSAGE TAMPERING PROTECTION and MESSAGE SOURCE AUTHENTICATION. We apply SECURE COMMUNICATIONS SESSION across the Session Layer of the server and the Communications Infrastructure components, such that session initiation is begun, managed and terminated by the (server-side) Session Manager component (Fig. 5). We omit describing the details of the actual protocols. Assuming that the Communications Infrastructure will be realized using a COTS middleware solution, we instantiate the INFRASTRUCTURE-DRIVEN MESSAGE TRANSFORM. This implies that we rely on the necessary algorithms and protocols (e.g. SSL/TLS) to have been implemented as part of the middleware infrastructure. We omit the details of key management for the sake of space and simplicity.

**Authentication infrastructure:**
From the Identity management solution frame (Authentication pattern family), we instantiate the AUTHENTICATOR pattern across the clients and server architectures, and realize it as an AUTHENTICATION SERVER (centralized) on the server-side. We subsequently use the PASSWORD-BASED AUTHENTICATION and CRYPTOGRAPHIC AUTHENTICATION solution design strategies for allowing users to login with a password, sent to the AUTHENTICATION SERVER via a protected communications channel (unilateral authentication). User IDs are assigned administratively and encapsulated (together with certain authorization rights, as discussed below) in CREDENTIALS stored on the clients for later use. Since users only need to login once for a given collaborative SESSION (whether sharing or editing), the SINGLE SIGN ON pattern is realized as a side-effect.

**Authorization infrastructure:**
From the Authorization solution frame we first instantiate ABSTRACT AUTHORIZATION ARCHITECTURE twice for the two conceptual authorization models.

**File sharing:** For file sharing, all of the ABSTRACT AUTHORIZATION ARCHITECTURE’S participants – PEP (Policy Enforcement Point), PDP (Policy Decision Point), PIP (Policy Information Point) and PAP (Policy Administration Point) reside on the server. The ABSTRACT AUTHORIZATION ARCHITECTURE is realized using an INTERCEPTOR connected to the Event Manager. When users share a file, the PDP components must check the file group as well as the allowable user groups with which the file can be shared; when users wish to download a file, only the user group is checked. Using PULL AUTHORIZATION STRATEGY, the rules associated with the sharing operations are encapsulated in ACLs (ACCESS CONTROL LIST pattern) attached to each file. These rules are determined in part administratively (group assignment), and in part by users (allowable operations based on groups) via the client-side Sharespace interface. The ACLs are thus stored according to the CENTRALIZED POLICY MANGER from the Security information management solution frame (Policy management pattern family) with PULL SECURITY INFORMATION semantics, where the central administrative interface participant residing on the server is accessible to the client Sharespace UI.

**Document Editing:** For document editing, we choose to span the PEP and PDP participants (from the second ABSTRACT AUTHORIZATION ARCHITECTURE instance) across both the clients and server, to reduce the performance bottlenecks associated with realizing the FINE GRANULARITY authorization modifier (Fig 4). Following this decision, editing operations are separated into two categories: viewing and modification operations. To enforce viewing operations using the PEP on the server-side (and hence avoid placing overhead on the client), we apply the VIEW pattern [Guerrero and Fuller, 2001], whereby after a document is requested by a user and after the relevant File Sharing model ACLs are checked—only those document parts are sent which the user is allowed to view and/or modify. Besides the document, a CREDENTIAL encapsulating the user’s editing rights is sent to the client, which can be presented again in subsequent modification operations.

Modification operations are enforced via the client-side PEP, which is linked to the server PDP. The server PDP works in conjunction with the PIP participant (which gathers the necessary meta-data from the document and user, combined with the user CREDENTIALS) and passes the client-side PEP the result. Both client- and server-side PEPs from the ABSTRACT AUTHORIZATION ARCHITECTURE are realized as INTERCEPTORS playing the role of REFERENCE MONITORS, in the client case connected to the Editing Controller and in the server case connected to the Session Manager (Fig. 5).
Given our use of CREDENTIALS, we are in effect realizing a PUSH AUTHORIZATION STRATEGY for all modification operations, with the CREDENTIALS acting as CAPABILITIES, which encapsulate the authorization rules. A DE-CENTRALIZED POLICY MANAGEMENT realization of the POLICY MANAGER pattern is used to manage all the user-related security information on the client (including the CREDENTIALS), also requiring an addition of another, secondary store local to each client.

Since the majority of both the File Sharing and Document Editing enforcement architecture functionality resides on the server, some participants (e.g. the PDP) can be merged as per the pattern participant resolution strategies in Section 4.2.2.2.

Naturally, the descriptions above are hardly sufficient to capture the whole design of the security architecture for SHARED. For example, to satisfy TSR15 in conjunction with the SECURITY SESSION modifier from the authorization model (see Figure 4), implies the use of the SESSION pattern in the Session Management layer (Server-side), using, for example, Session Objects which are bound and cryptographically hashed with the identities of users (cf. Identity management solution frame) and used as part of, or in conjunction with, the CREDENTIALS during editing updates. A number of such fine-grained detailed design decisions must be considered and refined, perhaps iteratively, together with the (initial functional) architecture from Section 5.1.1 to obtain the final secure software architecture.

5.2.3.3 Security Verification (SecVerify at Des) stage

During this stage we check all the traceability links to make sure that the countermeasures we have incorporated into SHARED's architecture really do protect from all the threats and satisfy all the necessary requirements.

5.2.3.4 Re-iteration of ASE's Security Requirements Determination phase

Having designed SHARED's security architecture, we perform the threat analysis process with the meta-security threat patterns. This allows to determine, for example, that, with respect to the authorization infrastructure, messages sent between the client-side PEP (connected to the Editing Controller) and the server-side PDP realizations can be eavesdropped (considering patterns from the Network communication attacks threat class); or that the authorization rules for file sharing, which are determined administratively, can be subverted (Unauthorized modification of rights threat pattern).

Mitigation of these threats follows the same stages as on the initial iteration, i.e. transitioning into ASE's Countermeasure Introduction stage. For example, with respect to the second example threat above, an application of the META-AUTHORIZATION CONTROL pattern from the Authorization solution frame would be identified, which in turn implies the creation of an additional conceptual authorization model and enforcement architecture (in this case we would choose to use the existing REFERENCE MONITOR based infrastructure to avoid duplication of functionality).

5.3 Additional details

5.3.1 Alternative design considerations

Below we discuss some alternative design decision that could have been taken, and how ASE can be applied for them.

5.3.1.1 Peer-to-peer architectural style

If during the initial architecture development phase it was decided that instead of the client-server architectural style a peer-to-peer style would be used for file sharing, or editing, or both, then the threat analysis process of ASE (Security Requirements Determination at Design stage) would have been performed using threat patterns from the peer-to-peer taxonomy (which itself encompasses the general distributed systems threat taxonomy) in [Uzunov and Fernandez, sub1]. Such a decision during later development would simply imply re-iterating the relevant ASE stage to consider the appropriate threats. In both cases this would reveal additional, system-specific threats, for example:

- Churn (R – <maps to: file manager components>) (a variation of T8) potentially in conjunction with Query flooding (C – message channels <maps to: comm. infr. and also Notification Layer on Server>): users share a large number of files and/or cause a large number of notifications to be broadcast.

- Unauthorized operation on data (S – <maps to: document manager and client document store>): depending on the peer-storage scheme, users may be able to manipulate local documents to cause changes to propagate to other users.

Mapping to concrete solutions would be done as before, with the difference being that there are no patterns specific
to peer-to-peer systems in any of the solution frames (at the time of this writing) and only one such independent
security pattern (DATA INTEGRITY IN P2P SYSTEMS [Schleinzer and Yoshioka, 2010]). This implies that custom-
made solutions will need to be created, as per mapping number 4 in Section 4.2.2.1, thereby reducing the ease of use
and hence usability of ASE for the Security Modeling (at Design) stage.

Regarding the existing solutions that are already modeled (or will be modeled, if the peer-to-peer style supplants
the client/server style early on), some pattern instances would require reconsideration. For example, with respect to
the patterns for the authorization infrastructure, the ABSTRACT AUTHORIZATION ARCHITECTURE’s PEP will have to
span a single peer, and a VIEW of a given document will have to be generated individually. The peer-to-peer style
similarly impacts the management of policies as well as cryptographic keys; in most of these cases the existing
security patterns, or customizations thereof, should continue to be applicable and could be re-instantiated.

5.3.1.2 Fully synchronous, real-time editing

Another highly relevant design decision with impact on the security architecture is the choice of fully synchronous,
real-time editing. In this case the patterns BELIEVE IN YOUR GROUP and DETECT A CONFLICTING CHANGE
[Lukosch and Schümmer, 2004] would be used in the CONCURRENCY MODEL realization, and locking would no
longer be required. This would also lead to a consideration of data consistency issues (considering threats with
respect to the Distribution Control decomposition layer, the Event Manager and Session Manager components etc.,
not only in the actual documents, but also between the authorization rules and the document updates (see [Cherif et
al., 2013]). Thus the meta-security iteration of ASE (Section 5.2.3.4) would gain greater importance. As for the peer-
to-peer decision above, appropriate countermeasures would include process serialization, which would be part of the
Execution Control solution frame (currently not specified in the literature), hence custom-made solutions based on
synchronization patterns in, e.g. [Schmidt et al., 2000] (see [Avgeriou and Tandler, 2006]) or [Silva, 1997] would
have to be used.

5.3.2 Security Implementation (SecImpl) phase

Implementing the security architecture consists in mapping some parts to COTS components (as in the secure
communications protocols to an underlying middleware platform supporting the relevant features), and manually
coding the rest (e.g. the authorization infrastructure) as separate modules that are part of the system implementation
itself. Good coding practices, static tools etc. can be used to reduce the number and severity of code-related
vulnerabilities (which would be introduced solely at this phase). The WRAPPER FACADE pattern [Schmidt et al.,
2000] (cf. [Silva et al., 1996]) and its secured counterpart SECURE WRAPPER FACADE [FERNANDEZ AND UZUNOV,
2013], as well as other adaptation patterns such as ADAPTER, PROXY etc. [Buschmann et al., 1996, Gamma et al.,
1995], can be particularly useful when interfacing with COTS components and/or legacy code [Becker et al., 2006]
(see also [Fernandez and Larrodno-Petrie, 2007]).

6 Conclusion and future work

In this paper we presented ASE – a comprehensive pattern-driven security methodology designed specifically for
(general) distributed systems. Through its distributed-systems-specific set of conceptual artefacts and its
comprehensive process, ASE is capable of addressing all or most of the core distributed systems security concerns,
providing developers with detailed guidance on how and where to introduce relevant security features into a
system's architecture during development. After presenting ASE’s conceptual framework and security process
aspects (Sections 3 and 4, respectively), we showed in detail how ASE can be applied in practice via the
development (up to and including design) of a realistic case study – the SHARED distributed collaborative system
(Section 5).

ASE inherently possesses beneficial features that allow it to fully satisfy a number of the core security
methodology evaluation criteria for industry adoption in [Uzunov et al., 2012b], which in turn can be seen as
elementary metrics for methodology quality (in what follows the corresponding criterion number from the latter
reference appears in brackets):

- the provision of guidance on what security measures should be applied and where they should be applied;
as well as the essential use of threat modeling (criterion No. 2);
- sensitivity to the system or application, with security measures specific to the nature of the application, with
a wide range of solutions (for distributed systems) (criterion No. 3);
- low security expertise requirements, with strong reliance on the existing software engineering skillsets of
developers (criterion No. 4).

Other beneficial features aligned with the aforementioned evaluation criteria of secondary importance include (cf.
[Uzunov et al., 2012b]):
• use of established modeling languages, such as UML, to reduce developer training and effort (criterion No. 6);
• presence of assessment and verification activities to ensure the introduced security solutions correctly counter the relevant threats (criterion No. 8);
• manageable overhead in proportion with the complexity of the target system (criterion No. 9);
• use of common repositories or catalogs of security knowledge relating to solutions (security patterns, solution frames) and threats (threat patterns) to encourage the application of best practices, increase productivity and aid developer training (criterion No. 10).

6.1 Flexibility and range of solutions
Besides being applicable to general distributed systems, ASE is also capable of addressing the specifics of peer-to-peer systems (Section 5.3.1.1), making it a first example of “a methodology that makes provisions for both specific distributed system types and general distributed systems” [Uzunov et al., 2012a], and hence a flexible security methodology in the technical sense of [Uzunov et al., sub2]. With respect to this flexibility, it is important to note that ASE is currently limited in practice by the lack of security solution artefacts specific to peer-to-peer systems that would correspond to the relevant patterns from the peer-to-peer-specific threat taxonomy/library of Uzunov and Fernandez [Uzunov and Fernandez, sub1] – cf. Section 5.3.1.1. The construction of a range of such solutions in the form of security patterns encapsulated in the various solution frames thus forms an important task for the future. A related and similarly important task is the specification of the “missing” solution frames that were marked with an asterisk in Tables 1 and 2.

6.2 Extending and assessing the effectiveness of ASE
In this paper we have focused almost exclusively on the early development life-cycle phases (analysis and design), with little consideration given to ASE’s Security Implementation phases. Detailing and evaluating the latter stages forms another important future task. A particularly worthwhile pursuit towards this end would be the automatic or semi-automatic generation of test cases from the threat patterns utilized during ASE’s Adversary Modeling stage, which could build on existing approaches for other threat artefacts – such as that of Marback et al. for threat trees [Marback et al., 2009]. Continuing the focus on the testing development stage, the integration and use of security test patterns [Smith and Williams, 2012] for penetration testing, potentially as a complement to the aforementioned test cases, would be another valuable addition to ASE.

Following on from the latter implementation-oriented direction is the construction of appropriate security metrics for measuring the effectiveness of ASE, i.e. whether a given degree of security, in some sense, has been achieved. This in turn could lead to a consideration of measuring the effectiveness of security methodologies in general – a subject which has received very little attention.

6.3 Practical details: tool support
With respect to practical application in industry scenarios, one area where ASE is lacking is the provision of software tool support. While, as illustrated in our case study in this paper, the development overhead is not unmanageable, a number of the tasks could be made more efficient via the use of tools, even of an elementary nature (e.g. showing developers a list of available patterns with descriptions for a given solution frame). More advanced tools that can “recommend” solutions (cf. [Serrano et al., 2009]) or manage the traceability links are also an option.

6.4 Development methodology integration
While in our illustration of ASE we espoused a (purposefully) loose “architecture-oriented” development approach, ASE is not tied to this approach or to any specific development methodology. Incorporating an object-oriented-like analysis stage is certainly helpful for ASE’s early phases, however, it is perfectly feasible to employ an approach (or strict methodology) with an altogether different paradigm for the other stages and phases, e.g. the architecture-centric Attribute-Driven Design (ADD – see [Bass et al., 2003]) for the design phase. A good example of such a hybrid-paradigm methodology can be found in [Sangawan et al., 2008]. More generally, the nature and paradigm of the development methodology used along with a given security methodology is not without importance, and an investigation of the precise compatibility and interplay between development and security methodologies constitutes yet another important research direction.

6.5 Re-engineering ASE and transferring features to other methodologies
As final, concluding remark, it is worth pointing out that most of the future directions outlined thus far that imply some kind of improvement to ASE can be realized not only in an ad-hoc fashion, but also by systematically re-engineering ASE. As mentioned in Section 2.2, ASE itself is the result of such a re-engineering endeavour, using the approach presented in [Uzunov et al., sub2]. This also accentuates the point that the beneficial features of ASE
(especially the distributed-systems-specific artefacts, which are themselves generic) can also be engineered into other security methodologies, just as other features of other methodologies can be engineered into ASE. In this sense, ASE can indeed be seen – as remarked in the Introduction – as just one particular solution (with its own beneficial, self-contained features) generated by a meta-solution to the problem of systematically introducing security into software.

References


Epilogue

In this chapter, we summarized our case study methodology and showed via a realistic example – the development of a secure collaborative, distributed software system – how it can be used in practice. With this we have completed the presentation of the thesis from the example-methodology-centric viewpoint: begun with surveying the area of security methodologies and security patterns (as the most promising artefact type); then the engineering of the case study methodology in Chapter 3, which included the use of two generic conceptual artefacts; and the improvement of its quality in Chapter 8 – with the use of the generic artefacts throughout the engineering process. In the latter context all the artefacts can be seen as the generic artefacts which they are, or as illustrative examples of what developers need to do to specify particular artefacts for a methodology they are engineering.

Although the case study methodology was pattern-driven, our engineering approach – which includes the solution frames as generic artefacts – is applicable to methodologies of any paradigm. Of course, it may not be (and is not, as shown in Chapter 8) possible to tailor all methodologies as “smoothly” as the case study methodology; in some cases the methodology has to be simply pulled apart and pieced back together again in order to incorporate solution frames, for example. In all cases a certain degree of creativity is required to link the new and old artefacts, and the new and old process elements; however, this is completely in accord with the use of patterns and with design in general: engineering security methodologies using our approach is an inherently “human-driven” activity.
Conclusion and Future Directions

But the chief ground of my satisfaction with this method, was the assurance I had of thereby exercising my reason in all matters, if not with absolute perfection, at least with the greatest attainable by me: besides, I was conscious that by its use my mind was becoming gradually habituated to clearer and more distinct conceptions of its objects; and I hoped also, from not having restricted this method to any particular matter, to apply it to the difficulties of the other sciences, with not less success than to those of algebra.

– René Descartes (Discourse on Method, 1637)

Achievement of Research Objectives and Contributions

At the outset of this thesis (see Introduction), we posed a particular research question: “how can suitable, high quality security methodologies with relevance to distributed systems be designed and constructed for particular project situations delineated by a collection of security methodology requirements?” This thesis has attempted to answer the latter question by proposing a comprehensive conceptual “toolkit” for engineering and re-engineering security methodologies on-the-fly, from requirements expressed as quality criteria, to design, construction, and finally verification and quality improvement. The proposed parts of the toolkit achieved the research objectives set out in the Introduction as follows:

O1: The creation of artefacts and/or processes addressing the construction, tailoring and/or combination of security methodologies.

The S-SMEP meta-methodology from Chapter 3 embodies a systematic approach both for tailoring and for constructing security methodologies from scratch based on security process patterns and meta-modeling. By applying S-SMEP and employing abstract process fragments (distilled via the S-SMEP Generalization stage), features from different methodologies can be combined and mixed at will. The tailoring aspect of S-SMEP was illustrated via the case study methodology (Chapters 3, 8 and 9); the construction from scratch approach for instantiating meta-model elements was illustrated in the form of the (re-interpreted) generic conceptual artefacts, for the System characterization, Security solution and Threat meta-model elements.

O2: The creation of a collection of complementary artefacts and/or processes for supporting the introduction and/or analysis of security features – with relevance to distributed systems – across a range of methodologies.

The collection of generic conceptual security artefacts from Chapters 4, 5, 6 and 7 address a broad range of core networked and distributed systems security features: authorization, authentication, secure network communications and security information management (rule/policy and cryptographic key management). These generic artefacts are essentially methodology-agnostic, and can be “deployed” via S-SMEP independently or as groups to replace and/or supplement existing conceptual framework artefacts for the same or similar purposes (according to the elements of the meta-model from Chapter 3).

O3: The creation of artefacts and/or processes for qualitatively determining or measuring the suitability or quality of security methodologies resulting from construction / tailoring / combination efforts.

The quality framework and QAIP process from Chapter 8 essentially constitute a meta-methodology for assessing and improving security methodologies. When used as part of the S-SMEP Verification stage, the latter framework and process help to ensure that all
construction/tailoring efforts are sensible – i.e. of measurably good quality – and that the final methodology satisfies all the project-specific (security methodological) requirements. The base quality profile for the framework can be extended and/or customized via the creation of situational profiles for methodology features desirable on different project situations, allowing for fine-grained quality control. The application of the quality framework and accompanying process was illustrated on the case study methodology in detail, and briefly on two other real-life security methodologies.

Besides being part of an overall approach, each of part of the toolkit for engineering security methodologies makes its own set of unique, self-contained contributions to the area of secure software engineering, as outlined in detail in the corresponding Chapters 3 to 8. While S-SMEP is essentially complete in itself, a number of the other “tools in the toolkit”, such as the quality framework (Chapter 8), the pattern-based threat library/taxonomy (Chapter 5) and even the conceptual framework meta-model (Chapter 3), and SPPF framework (Chapter 3), can be extended / supplemented / specialized to produce either engineering (meta-)artefacts or methodology artefacts that are suited to particular project situations. Security solution frames can also be extended by adding new patterns or by linking patterns with various realizations. This provides developers, architects, methodology engineers and other team members with a high degree of freedom and flexibility in applying the approach.

Future Research Directions

The work presented as part of this thesis can be extended in a number of directions, a great many of which have already been discussed at length at the end of each paper constituting Chapters 3 to 8, and also in Chapter 9. Some important points from these, as well as overall directions not included in the foregoing discussions include:

- The creation of a searchable database (methodbase) of abstract process fragments, indexed by different criteria, such as the quality criteria form Chapter 8, would be a valuable aid in the construction and combination of security methodologies. The first step in this is the extraction of such fragments, and the attachment of the criteria as meta-information. Such fragments would fit into the third SPPF level (Chapter 3).

- In a similar vein, the development of CAME (computer-aided methodology engineering) tool support will be an important addition to improving the ease of use and efficacy of applying S-SMEP.

- Tool support can also be developed for applying the generic conceptual artefacts (e.g. a simple GUI with search capabilities for matching attack patterns, or for security patterns realizations). In practical terms, this could be achieved as part of developing modular tool support for the case study methodology, which utilizes all the generic artefacts (Chapter 9).

- For the architectural decomposition generic artefact (Chapter 4), constructing derivative functional decomposition layers and/or technical realization abstractions for specific system types would allow finer-grained or more specialized architectural information to be captured. This would turn the decomposition framework into a flexible artefact.

- Specializing the existing pattern-based threat library/taxonomy (Chapter 5) for several varieties of widely used system-/technology-specific contexts, such as web-based systems, would give developers “ready-made” libraries/taxonomies they can immediately use on relevant projects, thereby reducing time and effort requirements. Another
important direction is the determination of (optimal) linkages between individual threat patterns and misuse and attack patterns, which would provide stronger support for security testing (including penetration testing) and verification activities.

- With respect to security solution frames (Chapters 6 and 7), an immediate practical extension is the addition and/or linking via relationships of security patterns from the literature, e.g. specific to Web-services, for the concerns addressed in this thesis (authorization, authentication and others). Another important extension is the integration of compound security patterns (see [Fernandez and Uzunov, 2012]), either as part of the frames themselves, as related (external) supplements, or even via the construction of new, compound solution structures.

- The development of new solution frames reflecting the mitigating security policies encapsulated in threat patterns – especially Execution Control, Storage Security (including encrypting stored data and distributed data dispersal) and Filtering – would continue the work begun in this thesis for constructing a comprehensive, generic, developer-friendly collection of security solutions. Distilling patterns for those pattern families which were omitted from consideration – such as secure routing, secure group communication, identity federation and others – could also form part of the latter work. The availability of frames – and/or compound structures as mentioned above – addressing a wide variety of security concerns, would allow for an exploration of the possibility to customize and combine frames into larger structures as suggested in Chapter 9 (security solution clusters) for different system types.

- Taken together, the latter four directions could lead to an exploration of how to design entire re-usable conceptual security frameworks that could be customized for different methodologies as per the conceptual framework meta-model.

- Constructing and analyzing combined, “hybrid” methodologies, whereby one methodology is used for some aspect of a system and another for another aspect, is yet another worthwhile pursuit. An example would be the case study methodology in this thesis being used for non-critical parts of a system, and a methodology such as that of Ali and colleagues (see Chapters 3 and 8) employing formal verification being used for critical system components.

- Further detailed case studies akin to the methodology in this thesis may provide additional insights leading to refinements of various parts of the proposed engineering toolkit. Perhaps more importantly than refinements, however, additional case studies would allow for a study of the combinability and transferability of features across methodologies (via, for example, abstract process fragments). Not all methodologies are compatible with a given range of features, and determining rules and/or heuristics for when a given methodology is or is not compatible would be a valuable extension of this research.
Bibliography

(For Introduction and Conclusion and Future Directions)


