Methods for Two-Party Privacy-Preserving Linear Programming

Alice Bednarz

Thesis submitted for the degree of
Doctor of Philosophy
in
Applied Mathematics
at
The University of Adelaide

Discipline of Applied Mathematics
School of Mathematical Sciences

January 22, 2012
Contents

Abstract ........................................................................................................ vii
Declaration ............................................................................................. ix
Acknowledgments ................................................................................... xi

1 Introduction ............................................................................................... 1
1.1 Cooperating with people you don’t trust ........................................... 1
1.2 Collaboration in the business world ................................................. 1
1.2.1 Benefits of collaboration .......................................................... 2
1.2.2 Barriers to collaboration .......................................................... 7
1.2.3 A solution: secure multiparty computation ................................ 9
1.3 Summary of this thesis ........................................................................ 10

2 Background ............................................................................................... 13
2.1 Secure multiparty computation ....................................................... 13
2.1.1 Brief history .............................................................................. 13
2.1.2 Problem formulation .................................................................. 14
2.1.3 What do we mean by ‘secure’? .................................................. 14
2.1.4 Core concepts ........................................................................... 18
2.1.5 Secure building blocks ............................................................. 20
2.1.6 Protocol composition ............................................................... 23
2.1.7 Application areas ....................................................................... 24
2.1.8 Challenges: theory to practice .................................................. 25
2.1.9 Related secure optimisation work ............................................. 26
2.2 Linear programming ............................................................................ 28
4.4.2 Restoring security ........................................... 70
4.5 The generalised-permutation algorithms ....................... 74
  4.5.1 Base protocol: two-party protocol ....................... 74
  4.5.2 Sub-protocol: secure evaluation of \( \hat{c} \) ................. 75
  4.5.3 Variation 1 — semi-trusted third party ................... 76
  4.5.4 Variation 2 — additively-split output .................... 77
4.6 Conclusion .................................................. 77

5 Practical Security Analysis ........................................ 79
  5.1 The acceptable security paradigm ............................. 79
  5.2 Partial exposure of private data ............................. 80
    5.2.1 Preliminary example ..................................... 81
    5.2.2 Alice’s probability of revealing \( k \) correct columns of \( M \) .... 82
    5.2.3 Repeated instances of similar LPs ....................... 86
  5.3 Other side issues to consider to ensure the claimed security-level .... 89
    5.3.1 Distribution of the random elements ..................... 90
    5.3.2 Preliminary constraint rescalings ........................ 91
    5.3.3 Features that cannot be disguised ....................... 91
    5.3.4 Attacks Bob can mount against Alice .................... 92
  5.4 Conclusions ............................................... 93

6 Transformation Method for Row-Partitioned LPs .................... 95
  6.1 Background ................................................ 95
  6.2 Problem definition ......................................... 96
  6.3 Solution approach ......................................... 97
    6.3.1 A second layer of disguise: matrix \( K \) ................. 100
    6.3.2 The transformation method ................................ 101
  6.4 Complexity ............................................... 104
  6.5 Detailed protocols ........................................ 105
Contents

6.5.1 Secure evaluation of $\hat{N}$ .............................................. 105
6.5.2 Secure evaluation of $\hat{b}$ .............................................. 107
6.5.3 Secure evaluation of $\hat{d}$ .............................................. 107
6.6 Conclusion ................................................................. 108

7 Security Analysis: Attacks and Workloads .................................. 109

7.1 Introduction ................................................................... 109
7.2 Case 1: Alice does not receive $x^*$ ................................... 110
7.2.1 Information deducible from the transformation .................. 111
7.3 Non-linear-algebra attack .................................................. 112
7.3.1 Detailed calculations .................................................... 114
7.3.2 Summary of the non-linear-algebra attack ....................... 116
7.4 Consistency-check attack ................................................... 117
7.4.1 Demonstration ............................................................ 118
7.5 Observation that reduces Alice’s workload .......................... 128
7.5.1 The row-switching phenomenon .................................... 129
7.6 Case 2: Alice does receive $x^*$ ....................................... 134
7.6.1 Permutation-reduction attack ........................................ 134
7.7 Security level .................................................................. 135
7.8 Conclusion and future work .............................................. 138
7.9 Additional calculations: deriving the formula for $R_{n,m}$ ........... 139

8 A Privacy-Preserving Interior Point Method ................................. 143

8.1 Motivation ................................................................. 143
8.2 Background and related work .......................................... 145
8.2.1 Simplex method vs interior point methods ..................... 145
8.3 Traditional primal affine-scaling method ............................ 146
8.3.1 How interior point methods work .................................. 146
8.3.2 Scaling transformation ................................................ 147
8.4 Security model and secure building blocks .......................... 149
B  Regular LPs in Canonical Form 186

B.1  Standard conversion procedures for regular LPs 186

B.1.1  Constraint conversions 186

B.1.2  Objective function conversions 187

B.1.3  Variable conversions 187

References 188
Imagine two companies who each manage part of a delivery network. Suppose these companies are considering the benefits of cooperation — would they be able to deliver packages faster and more cheaply if they could share each other’s networks? Answering this question would usually mean sharing private company information with each other, such as network topologies, internal capacities and budgets. Companies are reluctant to share such information, for fear of losing their competitive advantage, fear of revealing sensitive company information, or fear of breaching anti-trust laws. However, collaboration could bring mutual gain. What if it were possible to collaborate without sharing sensitive information?

Fortunately, a methodology called secure multiparty computation can alleviate these privacy concerns. Developed in the 1980s, it allows parties who don’t trust each other to compute functions involving their private data amongst themselves, whilst keeping their private information hidden. This method eliminates the need to find a trusted third party.

Linear programs are one of the most common types of optimisation problems, occurring widely in industry. For example, supply chain planning problems, delivery problems and scheduling problems can all be modelled as linear programs.

Recently, the idea of optimisation with privacy concerns has been studied in the form of privacy-preserving linear programming. The aim is to solve a linear program jointly-defined by two or more parties, who wish to solve the joint program for mutual benefit, yet are unwilling to share their data. In this thesis we address the two-party case, where each party owns a subset of the linear program’s components.

Solution methods for privacy-preserving linear programs can be grouped into two classes: cryptographic methods, and transformation methods. Cryptographic methods implement a privacy-preserving version of the Simplex method, while transformation methods ‘disguise’ the linear program using random matrices, allowing one to use any standard optimisation software to solve the linear program. The transformation methods are more efficient, but only provide ‘heuristic’ security guarantees; whereas the cryptographic methods have robust security guarantees, but suffer from slow performance.
This research investigates new variants of both methods. We correct a flaw in two of the early transformation methods, and subsequently develop two new transformation methods, each with a quantifiable worst-case security level. We also propose the first cryptographic method to be based on an interior point method.

We explain why the notion of a single ‘canonical’ form, to which all methods can be applied, does not exist for privacy-preserving linear programs. In particular, the transformation methods are extremely sensitive to the way the data is partitioned between the parties involved, and the type of constraints (inequality, equality) and variables (non-negative, free) in the linear program. To make it clear which scenarios a transformation method applies to, we develop a new notation system, in the spirit of Kendall’s queueing notation, to uniquely specify a privacy-preserving linear program. Our notation incorporates aspects which were previously overlooked.

Throughout the thesis, we explain why careful attention must always be paid to the output of a secure computation, and in the final chapter we present an attack on privacy-preserving linear programming that is entirely dependent on knowing the output (the optimal solution).

Although more work is needed to create commercially viable solution methods, privacy-preserving linear programming may one day allow companies to realise the potential benefits of collaboration, without the associated risks.
Declaration

I, Alice Bednarz, certify that this work contains no material which has been accepted for
the award of any other degree or diploma 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.

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.

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 catalogue and also
through web search engines, unless permission has been granted by the University to
restrict access for a period of time.

Signature..................................   Date..............
Acknowledgments

I would like to thank my supervisors, Associate Professor Matthew Roughan and Professor Nigel Bean, for their guidance, expertise, encouragement, and for always being so giving of their time. I feel very lucky to have had such wonderful supervisors. Whenever I was stuck or needed advice they always managed to help me find a solution; they were always very discerning and thorough proof-readers, and it was a pleasure to work with them.

Thanks also to everyone at the School of Mathematical Sciences for their friendship and support, especially my officemates for making the PhD a fun experience, and for all the enjoyable morning coffees!

There’s no way I could have finished this thesis without the support of my family and friends. Thanks to my sister Jana for all the little surprises, the home-cooked dinners, the interesting conversations, and for keeping me entertained with a steady supply of CDs and DVDs.

Thanks to Mum and Dad for their never-ending love and support; for all the dinners and rides to the bus station, and for putting up with me being stressed sometimes!

Thanks to Nanna and Papa for your delicious choc-bits and baking.

Thanks to the Hawkes’ for all the fun Wednesday night dinners and catch ups — they always helped take my mind off my thesis and de-stress!

A big thanks to Bernie for proof-reading some chapters for me, and for your helpful feedback and suggestions.

And finally thanks to Nathan for motivating me to keep going and keeping my spirits up — there’s no way I could have finished this if it wasn’t for your continual encouragement and pep talks — and for all your love and support, and for always making me laugh.
Chapter 1

Introduction

1.1 Cooperating with people you don’t trust

Suppose two rival millionaires meet on the street. For their own peace of mind, they want to know which of them is richer after the global financial crisis, without revealing their own personal wealth. Could they manage to achieve this by working together?

Let’s extend this idea to something a little more substantial, by considering an example involving the Internet. Different parties manage different segments of the Internet. Any improvement to the current performance of the network would benefit all parties. However, an innate distrust of competitors leads to each party making decisions in isolation. The net result of this approach is a poorly optimised global network. Is this the best they can do? Or can they somehow collaborate, improving the overall performance, whilst maintaining their own company secrets?

In response to such problems, the field of secure multiparty computation has evolved. A tool-kit of techniques exists, and the challenge now is to apply these tools to real-world problems. In this thesis we will look at some examples in the field of optimisation. The term ‘optimisation’ refers to the process of minimising the cost (or maximising the profit) of some operation, such as the delivery of goods, or the manufacture of products.

1.2 Collaboration in the business world

Optimisation problems occur frequently in the business world, and the processes to be optimised often involve more than one company — for example, optimising the flow of goods in a supply chain requires collaboration between suppliers, distributors and retailers. Collaborative optimisation, when companies work together to optimise a joint operation, has the potential for huge efficiency improvements. For example, to save on fuel costs, packaged goods companies can share trucking space for goods bound for the same destination. Similarly, delay of traffic through a shared network can be reduced.
if the network operators cooperate. Unfortunately, though, optimisation normally requires complete data sharing, and such cooperative ventures rarely occur due to privacy concerns.

After all, in today’s business world, who can you trust? Corporations are generally unwilling to share private data with each other for fear of:

- revealing company secrets;
- revealing their financial strategies, or their financial health;
- breaching privacy or anti-trust legislation;
- exploitation – fear that their data may be used against them;
- loss of control – who knows where their data might end up?; or simply,
- fear of embarrassment.

Yet sharing data could bring mutual gain. What if it were possible to collaborate without risking sensitive information?

The obvious way to mitigate privacy concerns is to introduce a trusted third party. A trusted third party is an external body, such as a consultancy house, whom the parties enlist to conduct the joint computation. However, these bring their own set of problems, as we discuss in Section 1.2.2.1; another solution is needed.

How to conduct optimisation without sharing private data and without engaging a trusted third party is the focus of this thesis. In particular, we focus on linear programming, a common type of optimisation strategy which is simple, tractable and applicable to a wide range of problems.

We first consider some examples from industry to demonstrate the benefits of collaboration. We then discuss the barriers to collaboration, the main barrier being privacy. Current options for achieving collaborative optimisation require a high level of trust and disclosure of sensitive information. As such, they are not being adopted in practice as often as they could be. However, techniques from the field of secure multiparty computation can help overcome the privacy challenges.

1.2.1 Benefits of collaboration

Prime candidates for collaborative optimisation are in the area of network management. Let us consider three examples: reducing delay in a jointly-managed network; combining trucking routes via collaborative logistics; and supply chain management.
1.2.1.1 Delivery network optimisation

Consider two companies, A and B, who each manage part of a delivery network to deliver packages to customers. The situation is illustrated in Figure 1.1. To begin with, consider the top half of the figure. The package to be delivered is represented by an envelope, while the destinations are represented by mail boxes. Company A manages the left-hand-side of the network, and Company B manages the right-hand-side. Each network has three entry points, and three exit points, and the two networks are connected by three ‘bridges’.

Suppose a package arrives at the first entry point of the network owned by Company A, and is destined for the mailbox circled in the top half of Figure 1.1. As far as Company A is concerned, it only has to deliver the package as far as Company B’s network, and B will do the rest and deliver the package to the customer. Company A has a choice of three paths from the entry point, with transport times of 2 minutes, 5 minutes, and 7 minutes respectively. Naturally, Company A will choose the fastest route to Company B, and send the packet along the first bridge, for a total travel time of 2 minutes. However, notice that B is now stuck with using a 10-minute route to reach the destination, for a total delivery time of 12 minutes.

Now consider the second half of Figure 1.1, which shows the same scenario if the delivery decision had been made together, considering the network as a whole. Had the companies been aware of each other’s internal networks, Company A could have used the second path, with a time of 5 minutes, allowing B to use its 2-minute path, for a total delivery time of only 7 minutes. (Although the total workload for Company A has increased under the collaborate scheme, this is only a very small example: in a real-life scenario, over many deliveries and with two-way traffic, any workload increases would balance out.) The incentive to cooperate is that both parties want to deliver the packet to the customer as quickly as possible. The roadblock is privacy: neither company wishes to reveal their network layouts and delivery times to the other party.

Broadly speaking, the Internet can be thought of as a delivery network, where packets of data are delivered to the end-users. The real Internet involves many more parties than the network considered here — there are approximately 30 000 individually-managed networks [10]. Each of these make their own internal decisions about how packets move through their networks. There could potentially be thousands of cases like the one demonstrated where performance could be improved by working together. However, a reluctance to share internal network data prevents such optimisation from occurring. How can these gains be realised?
Figure 1.1: If Companies A and B make decisions in isolation, and A chooses the locally-optimal path, the delivery time is 12 minutes. If they shared all their data and worked together, the delivery time is reduced to 7 minutes.
1.2.1.2 Collaborative logistics optimisation

Collaborative optimisation has great potential for savings in the packaged goods industry. General Mills and Land O’Lakes are two producers of packaged food in the United States. With fuel costs on the rise, companies such as these are always looking for ways to save on their transportation costs. One of the big problems they face is ‘empty miles’. These are the miles travelled by empty trucks, in between deliveries. At any given moment, it is estimated that approximately one quarter of all trucks in the U.S. are travelling empty [97]. These empty trips are responsible for huge inefficiencies, not only in a monetary sense, but also in an environmental sense. In the U.S. alone, around 29 million trucks deliver over 10 billion tonnes of goods annually [69], and in 2007, their fuel consumption was estimated at 55 billion gallons [69].

One solution to reduce the number of empty miles is collaborative logistics. If companies could somehow ‘team up’, and combine routes to reduce empty miles, huge savings could be realised. This can be achieved by sharing trucking routes.

To see the benefits in action, consider the following real-life scenario from [22]. To start with, a General Mills truck departs from a warehouse in New York, to deliver pizza rolls to New Jersey. The truck then moves on to other General Mills warehouses in other states, before making its last General Mills delivery in Ohio. Normally, the remainder of the journey back to New York would be empty miles. Instead, the General Mills truck stops at a Land O’Lakes warehouse in Kent, Ohio, and picks up a delivery bound for New Jersey. It makes this delivery on behalf of Land O’Lakes on its way back to New York, substantially reducing the time the truck is empty and therefore a ‘wasted resource’. This cooperation saves Land O’Lakes having to ‘dead haul’ a truck to Kent to pick up the goods. Not only does this kind of collaboration save money – Land O’Lakes saves $200 per week from this route alone [22] – it also reduces carbon emissions and saves fuel.

The collaboration described is illustrated in Figure 1.2: the top part of the figure shows the dead miles if there is no cooperation; while the bottom part of the figure shows how the dead miles can be reduced when Land O’Lakes shares a segment of General Mills’ route. This type of route-sharing has reduced Land O’Lakes’ annual freight costs by 15% [22]. In a similar story, Georgia-Pacific, another large food producer, decreased their empty truck movements from 18% to 3%, thanks to collaborative logistics, for a saving of $11 million a year [118]. However, the downside is that the companies must rely on a trusted-third party to find routes they can share; this is discussed further in Section 1.2.2.
Chapter 1. Introduction

General Mills makes its last delivery here. The remainder of the journey is a "dead haul" back to NY.

General Mills delivers the Land O'Lakes load to NJ on its way back to NY.

(1) No collaboration

General Mills makes its last delivery here. The remainder of the journey is a "dead haul" back to NY.

(2) With collaboration

This segment of General Mills' route is 'shared' by Land O'Lakes.

The General Mills truck picks up a load from a Land O'Lakes warehouse.

General Mills delivers the Land O'Lakes load to NJ on its way back to NY.

Figure 1.2: Land O'Lakes shares part of General Mills' route, reducing the number of miles driven by an empty truck for both companies. Land O'Lakes doesn't have to drive an empty truck to the warehouse in Ohio, and General Mills can reduce the amount of time its truck is empty on its return trip to New York.
1.2.1.3 The collaborative supply chain

The examples considered so far are both examples of two-party collaboration. There are also real-world problems involving more than two parties. Although we don’t provide methods for the multiparty case in this thesis, we mention an example here to give readers the big picture. One example is supply chain collaboration. A supply chain is made up of manufacturers, distributors and retailers. Cooperation can help manufacturers and retailers effectively match supply with demand, with benefits including shorter delivery times, less excess stock being held in inventories, and fewer out-of-stock situations [115].

However, there is mutual distrust among supply chain members [115], and in order to make globally optimal decisions, the members would need to share information about their production capabilities, production schedules, production and storage costs, promotion plans, and on-hand inventory [5, 9, 74]. This is not something they are usually willing to do, for fear it could reduce their bargaining power [116]. For example, monetary data such as a product’s holding cost or the penalty costs for late delivery of a product are considered sensitive information since they could be exploited by another member to the owner’s detriment [116]. As Simatupang and Tayur [115] explain,

‘Supply chain members often do not wish to share their private information completely and faithfully with all other members due to the economic value of that information (actual or perceived). As a result, the supply chain suffers from sub-optimal decisions and opportunistic behaviour.’ [115, p.17]

So despite the promise of mutual benefit for all parties, a reluctance to share information prevents it from being realised.

1.2.2 Barriers to collaboration

What are the current options for collaborative optimisation? For example, how can two companies discover that they share part of a route and ‘team up’? The answer is that they must both be members of a collaborative logistics service, run by a trusted third party. For example, in 2009, Lean Logistics introduced a service called ‘GreenLanes’ [85]. The idea is to pair up companies who use similar routes so that the number of empty miles travelled can be reduced. Similar products are offered by the VICS\footnote{Voluntary Interindustry Commerce Solutions Association} Empty Miles Service [127], and Sterling Commerce’s ‘Transportation Management System’ (formerly Nistevo) [120].

1.2.2.1 Problems with using a third party

Using a trusted third party has a number of drawbacks. First, you must entrust them with all of your sensitive data. Even though you’re not sharing data directly with
your competitors, you still have to share it with the third party. As Logistics Manager Patrick Johnson from Land O’Lakes explains [22]:

‘Land O’Lakes routes are not shown to General Mills [a competitor], for example, but Nistevo sees where scheduling can occur, and does all the matching.’

Although only Nistevo sees the whole picture, the fact that anyone sees the whole picture is cause for concern. For instance, it is likely that the third party will be able to tell when your promotions are going to occur — how can you be sure they won’t sell this information to your competitors? Even if the logistics provider is reputable and seemingly has no motive to become corrupt, there are other issues to consider. For example, what if at some stage in the future you pull out of the service? How can you be sure that the third party will delete all of your data from their system? You must also ensure that the data storage system the third party is using is secure [31].

Second, using a third party can be costly. Land O’Lakes paid $250,000 to subscribe to the collaborative logistics service [22], and despite this cost being recouped within six months, it is still an outlay that could constitute a barrier to collaboration.

Third, there is no guarantee of optimality. With VICS Empty Miles, for example, users are required to do the majority of the searching and matching of routes themselves; there is no automated optimisation software. It is very unlikely that users would find the optimal solution ‘by hand’, so this approach is still going to be sub-optimal. In addition, users can pick and choose how many of their delivery routes to input to the site, so they’re not even searching the full space of routes.

Fourth, you must trust the third party to behave fairly [74]. That is, you must trust that they won’t secretly favour one company over another. How do you know your competitor is not getting a ‘better deal’ than you? You have no way of checking this for yourself. As you don’t have access to all the data, you have no way of verifying that the third party really is finding the optimal solution, or whether the solution benefits all parties equally.

Even those using a third party are not off the hook when it comes to revealing private information to competitors. For example, with VICS Empty Miles, once a user finds a matching route, they then have to negotiate with the potential partner ‘off-line’. It is likely that then you will eventually have to reveal some private data in order to broker a deal with them.

1.2.2.2 Sharing private information

If parties cooperate directly with one another, rather than going through a third party, there are even more problems. For starters, companies do not want to reveal sensitive
company information, for fear that it will be exploited [77]. Privacy is still cited as one of the major reasons for not engaging in collaborative optimisation. As Land O’Lakes logistics manager Patrick Johnson explains, collaborative logistics is difficult to execute, because:

‘You’re having to share stuff, and that’s never easy in a competitive environment that’s made up of companies with different cultures and different objectives and different metrics and performance measurements.’ [22]

Johnson adds:

‘It’s hard enough sometimes to team with the people inside your company, let alone those outside your company. [...] You’ve got to develop a certain amount of trust.’ [117]

Bruce Richardson of AMR Research Inc (now part of Gartner, Inc), a research firm focusing on the global supply chain, says [22]:

‘Teaming with others inside your industry, some of whom are considered the enemy on other fronts, still isn’t a simple concept for a lot of people.’

In addition to privacy concerns, companies also have to consider anti-trust regulations, and ensure that their data won’t be leaked to someone other than the party they’re cooperating with [77]. In short, the fact that private information is involved makes collaborative optimisation very difficult to implement.

1.2.3 A solution: secure multiparty computation

A field called secure multiparty computation can help overcome these privacy concerns, and allow the potential benefits of collaborative optimisation to become a reality. Also known as secure distributed computing, this technology does away with the need for a trusted third party altogether. It allows a group of parties who don’t trust each other to compute functions involving their private data amongst themselves, while keeping their private information hidden. No party learns any more than they would have if a trusted third party conducted the computations [31]. In short, secure multiparty computation allows you to have your cake, and eat it too.

At the time of its inception in the 1980s, secure multiparty computation was of largely theoretical interest. Since then, increased computing power and improved algorithms mean that the technique is now moving from theory to practice. In this thesis, we will focus primarily on two-party secure computation. The benefits to companies are (i) opportunities for collaboration in situations where privacy is a barrier; (ii) a reduction in collaboration costs in cases where a third party is currently used; and (iii) assurance that the computation is correct. We consider this technology further in the next chapter.
As demonstrated by the examples considered here, there are many opportunities for companies to collaborate with one another for mutual gain, but these opportunities are not always taken up due to privacy concerns. Our work using secure computation can result in a collaborative bottom line that can be competitive with the use of a trusted third party, but without the associated risks.

1.3 Summary of this thesis

Linear programming is one of the cornerstones of the optimisation field, and the idea of joint optimisation naturally leads one to consider whether a secure distributed version of linear programming could be devised. This is the focus of this thesis.

In Chapter 2, we introduce secure multiparty computation and discuss its main results. Following a brief overview of linear programming theory, we introduce the notion of a ‘secure’ or ‘privacy-preserving’ linear program. Here, the components are partitioned between two parties, who don’t wish to share their data. Solution methods for such programs can be divided into two classes: transformation methods, which ‘disguise’ the linear program, and cryptographic methods, which implement a secure version of the Simplex method. We discuss the advantages and limitations of each approach.

In Chapter 3, we demonstrate that the traditional ‘canonical form’ of linear programming problems fails in the case of privacy-preserving linear programs, and motivate the need for a new notation system for classifying privacy-preserving linear programs. We develop a seven-factor notation system inspired by Kendall’s queueing notation [73]. The notation can precisely and concisely represent the data-sharing scenarios and restrictions implicit in a solution method. Work in this chapter has been submitted for publication [13].

In Chapter 4, we expose a flaw in an existing transformation method, and show how to correct it, for the case when one party owns the objective function, and the other owns the constraints. The core of this work has been previously published [11].

In Chapter 5, we analyse the security of the corrected method more deeply, highlighting potential security risks, such as the potential to learn partial information about another party’s private data, and the danger of multiple runs of similar linear programs.

In Chapters 6 and 7, we present a transformation method for a row-partitioned constraint set with non-negativity constraints and inequality constraints. This is a common case in practice which was not solvable with any of the existing transformation methods. Further, this is the first transformation method with a quantifiable worst-case security level.
In Chapter 8, we develop the first cryptographic method to be based on an interior point method. All other cryptographic methods have been based on the Simplex method. Since interior point methods are known to offer superior performance over Simplex for certain types of problems, it is important that users are not restricted to using Simplex. Work in this chapter has been submitted for publication [14].

Finally, in Chapter 9, we present an attack on privacy-preserving linear programming, which is undetectable and able to be executed no matter which solution method is selected. Our aim in describing this attack is to show that no matter how ‘secure’ a method is, there may still be ways for a malicious adversary to extract as much information as possible from the output. Work in this chapter has been submitted for publication [12].

The thesis is concluded in Chapter 10, followed by Appendices A and B, which provide a library of secure building blocks, and some basic linear programming theory, respectively.
Chapter 2

Background

This chapter introduces the field of secure multiparty computation – what it is, what it’s used for, and where the field is today. One application of secure computation is solving optimisation problems which combine the private data of two or more parties. Linear programs are one common type of optimisation problem that occur widely in industry. A brief overview of linear programming theory is given, before considering the addition of privacy concerns. We survey the available solution techniques for these so-called ‘privacy-preserving linear programs’, before motivating the need for new methods.

2.1 Secure multiparty computation

Imagine a group of mutually distrustful players, who nonetheless wish to cooperate to compute a function of their private inputs. For example, a roomful of executives may wish to calculate their average salary, but of course do not wish to disclose their individual salaries to their colleagues. This is essentially the problem of secure multiparty computation. This seemingly impossible task, of facilitating joint computation while maintaining the privacy of the individual inputs, has been the subject of much research over the past 30 years.

2.1.1 Brief history

The idea of secure multiparty computation was introduced by Yao in the 1980s [132,133]. He considered two parties, one with private data $x$, and the other with private data $y$, who wish to jointly compute $f(x,y)$, such that only the output is public, while the private data remains hidden from the other person. Yao presented general results, for the two-party case, showing that any polynomial-time function $f(x,y)$ that could be represented as a Boolean circuit could be computed in a secure fashion [133]. Yao’s work was soon generalized to the $n$-party case by Goldreich, Micali and Wigderson [64]. Together, these papers proved the remarkable result that any polynomial-time function could be computed in a secure fashion.
These generic solutions require the desired function to be represented as a Boolean circuit, composed of a series of gates, which is then evaluated securely gate-by-gate\(^1\). Unfortunately, for the majority of non-trivial computations, such an approach leads to a very large circuit, rendering the protocol inefficient [37]. As a result of the inefficiency problems of the generic approach, much research has been devoted to efficiency improvements, and developing efficient protocols for specific tasks. For example, see Naor and Pinkas [93], Hirt et al. [67], and Damgard and Nielsen [38].

More recently, research has turned to developing protocols for specific applications; for instance in the areas of electronic voting [36], secure auctions [95], statistical analysis on shared data [52], and privacy-preserving data mining [82].

### 2.1.2 Problem formulation

In its most general form, the problem of secure multiparty computation is as follows. There are \(N\) parties labelled \(P_1, P_2, \ldots, P_N\) on a network, where party \(P_i, i = 1, \ldots, N\), has private data \(x_i\), and they wish to compute a joint function \(f(x_1, x_2, \ldots, x_N) = (y_1, y_2, \ldots, y_N)\). The goal is to compute \(f\) without \(P_i\) learning anything about \(x_j\) or \(y_j\) for \(j \neq i\), other than what can be inferred from their own data \(x_i\) and their output \(y_i\). In some cases, all parties may receive the same output (i.e., all values for \(y_i\) are equal), but this depends on the situation. In this thesis, we focus on the two-party case.

It is not assumed that all parties are honest, and hence the computation needs to occur while possibly under attack by an adversary, who may wish to learn the data of honest parties, or cause the calculation to be incorrect. Thus there are two important requirements of a multiparty computation protocol – privacy and correctness [80]. The privacy requirement maintains that parties should learn their output and nothing else, while the correctness requirement ensures the validity of the final output.

Of course, the trivial solution to all multiparty computation problems is simply to have each party hand their input to a trusted third party, who computes the desired function and hands the answer back to everyone. Alas, in reality, such trusted third parties rarely exist, or are expensive and inflexible to employ.

### 2.1.3 What do we mean by ‘secure’?

Defining what it means for a computation to be secure proved quite a challenging task, and although intuitive definitions were around from the beginning, it was several years before the field arrived at a precise definition. Today, the standard definition is due to Canetti [23], building on the work of Micali and Rogaway [89], among others, and is known as the ideal/real simulation paradigm.

\(^1\)See Lindell and Pinkas [83] for a detailed description of Yao’s circuit protocol.
The ‘ideal’ model is one in which the parties send their inputs to a trusted party, who computes the function and sends back the outputs. The ‘real’ model, on the other hand, is one in which the parties run some protocol amongst themselves with no trusted help. Loosely speaking, the ideal/real simulation paradigm states that a protocol is secure if it emulates the ‘ideal’ scenario [80]. In the ideal model, the parties learn nothing about each other’s data aside from what can be inferred from their input and the output. A computation is said to be secure if the parties learn nothing more about each other’s data during the execution of the real protocol than they would have in the ideal model [63]. As illustrated in Figure 2.1, performing the protocol must be equivalent to what would happen if a trusted third party existed. Implicit in this definition are a variety of security guarantees including privacy of inputs and correctness of outputs [80].

Proving that a protocol is secure under the ideal/real paradigm requires one to show that the intermediate results of a protocol can be simulated based on a party’s input and output only [81]. We do not go into the formalisations of these requirements here; see Goldreich [63, Ch.7] for details.

Figure 2.1: The ideal/real paradigm states that a protocol is secure if performing the real protocol ‘emulates’ the ideal protocol, in that no more information is disclosed.

Such security guarantees must hold in the presence of ‘cheating’ by one or more of the parties involved. Usually, cheating is realised in the form of an adversary who is ‘honest-but-curious’ (also referred to as ‘semi-honest’ or ‘passive’). This means that they will follow the protocol correctly, but will conduct extra calculations ‘on the side’ on any intermediate output, to try to discover whatever information they can about another party’s data [64]. (Intermediate output refers to any output other than the function result which is received by the parties during the calculation. For example, intermediate output might be the encryption of another party’s data.) We can regard a passive adversary as a machine that follows the protocol in terms of what it sends to other players, but may compute more than required [64]. One can also assume ‘malicious’ adversaries, who may deviate from the prescribed protocol however they choose [64]. They may deliberately not follow instructions correctly, in order to cause
the final output to be incorrect, or to uncover the private data of other parties. In this thesis we adopt the ‘honest-but-curious’ model, since we assume that the solution to the joint optimisation problem is of mutual benefit to all parties, so no-one has an incentive to deliberately sabotage the protocol.

2.1.3.1 Computational security vs Information-theoretic security

There are two very different models for secure computation — the computational model, and the information-theoretic model [35]. This distinction is important. The difference lies in the type of security they provide. The computational model (also called the cryptographic model) relies on the existence of one-way trapdoor functions, introduced by Diffie and Hellman [45], which are functions whose inverse is difficult to compute without knowledge of a secret key (the so-called ‘trapdoor’ information). Essentially, the computational model assumes that public key cryptography is possible. Although it is theoretically possible to discover the key, computational security assumes that it would be impossible for an adversary to do so in a reasonable amount of time. Hence, in the computational model, the adversary is assumed to be computationally bounded. But as Kilian [75] points out, cryptographers rarely sleep soundly, since if someone devises a way to solve the so-called intractable problems that underlie cryptography (such as factoring large integers), all security will be lost.

In the information-theoretic model, on the other hand, security is assured regardless of how much computing power an adversary has available to them [28]. For this reason, information-theoretic security is also called ‘perfect’ or ‘unconditional’ security. There is no code to break here; rather, all intermediate results are shared in such a way that with less than a certain fraction of the shares it is absolutely impossible to gain access to the private information of others. However, once more than a certain fraction of the players collude, it is possible to access private information. As long as the fraction of corrupt players is below a certain threshold (see Table 2.1), the results in this model hold unconditionally, without the need for any cryptographic assumptions.

Due to the different types of security offered — computational security versus information-theoretic security — the two paradigms have different communication models. In the computational model, it is assumed that communication channels are authenticated, meaning that an adversary can eavesdrop, but cannot modify messages sent from one party to another [84]. Note that eavesdropping does no harm since all messages are encrypted. In the information-theoretic model, on the other hand, it is assumed that the communication channels are private, meaning that an adversary cannot eavesdrop on messages sent from one party to another [84]. This assumption is necessary because data is not otherwise encrypted in the information-theoretic setting, though obviously communication channels can be made private using encryption.
2.1.3.2 Feasibility results

A (very simplified) summary of the feasibility results for the two models of secure computation, assuming semi-honest adversaries, is given in Table 2.1. The results in the computational setting are due to Yao [133] and Goldreich, Micali and Wigderson [64], while the results in the information-theoretic setting are due to Ben-Or, Goldwasser and Wigderson [15] and Chaum, Crepeau and Damgard [27]. Surveys of all the general results can be found in [35, 63, 84].

The main point to notice is that the information-theoretic approach fails when more than half of the parties are corrupted. Therefore, the computational model is the only option for two-party protocols.

<table>
<thead>
<tr>
<th>Max. no. of parties which can be corrupted</th>
<th>Is Secure Computation possible?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Information-Theoretic Setting</td>
</tr>
<tr>
<td>&lt; $\frac{1}{2}$</td>
<td>✓</td>
</tr>
<tr>
<td>$\geq \frac{1}{2}$</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 2.1: Feasibility results for the two models of secure multiparty computation, assuming semi-honest adversaries. Note that the information-theoretic approach cannot be used once at least half of the parties are corrupted.

2.1.3.3 Alternative definitions of security

Although the ideal/real paradigm elegantly captures security, several authors have questioned this approach, and its possible restrictions in practice. For example, Du and Zhan [53] state that ‘achieving [ideal security] is not difficult, but achieving it efficiently is’. As an alternative, they introduce the notion of ‘acceptable security’ — a tradeoff between perfect security and an efficient method. The model allows partial information disclosure, up to some ‘acceptable level’ in favour of better performance. The authors point out that, ‘if the ideal security is too expensive to achieve, people might prefer low-cost solutions that can achieve security at an ‘acceptable’ level’ [53, p.128]. These sentiments are echoed by Chiang et al. [29], who note that the ideal/real paradigm provides no means to quantify security.

Taking these concepts further, in this thesis our focus is on quantifying the probability that an adversary may uncover a party’s private data. For a protocol to be secure, the probability of guessing must be specified and low. The ideal is for this security level to be adjustable as well, but one can often reach a point where the probability is so low
that being able to adjust it is irrelevant, as we will see in Chapters 5 and 7. We define the particular security model used in later chapters where appropriate.

Consideration of the output

Another feature not covered by the ideal/real definition is the potential for the output to reveal private information. Although secure computation guarantees that the intermediate results won’t disclose information, it makes no guarantees that the final output itself won’t destroy security [84]. In response to this realisation, Atallah et al. [5] and Pibernik et al. [100] introduce the concept of ‘inverse optimisation’, which refers to the ability of a party to learn information about another party’s data based on the input and the output of the protocol. Indeed, we’ve encountered problems in this thesis where the answer gives everything away. In Chapters 5 and 7, the potential for the output to undo the security of a method will be explained in detail, while in Chapter 9 we present an output-based attack that is independent of the solution method.

Our approach to security in this thesis is to care about the ‘holistic’ problem, encompassing all facets including the protocol itself, the output, and side issues that may impact security (e.g. multiple runs of the same program). In particular, we adopt this approach in Chapters 5 and 7, when we analyse the security of two transformation methods.

2.1.4 Core concepts

The protocols of secure multiparty computation rely on some clever core concepts, including oblivious transfer, secret-sharing schemes and homomorphic encryption. These are discussed in more detail below.

Oblivious Transfer

Oblivious transfer is one of the cornerstones of cryptography (see Kilian [75]). Suppose Party A owns an array of data, and Party B wishes to access the data at a certain cell of the array. The catch is that Party B does not want A to know which cell it accessed. Oblivious transfer makes such a transaction possible — B will obtain access to precisely the element it desires, and no others, while A remains ‘oblivious’ as to which element it transferred to B.

Rabin [101] first developed the idea of oblivious transfer in the context of secret exchange, by presenting a scenario where a party named Alice has some secret information which she transfers to another party named Bob with probability 1/2, but remains oblivious as to whether the information was received. Rabin’s idea was put into a more useful form, called ‘1-out-of-2 oblivious transfer’, by Even, Goldreich and Lempel [54]. Here, Alice has two messages, \( M_0 \) and \( M_1 \), and Bob has a desired index \( q \in \{0,1\} \),
corresponding to the message he wishes to obtain. Bob can now choose which message
he wants to receive. The protocol works by encrypting Alice’s messages in such a way
that in the final step, the only message Bob is able to decrypt is the one corresponding
to his chosen index (i.e., $M_q$). A simple depiction is shown in Figure 2.2. The full
protocol is provided for interested readers in Appendix A.

The 1-out-of-2 oblivious transfer protocol is easily extended to 1-out-of-$n$ oblivious
transfer, where Alice has $n$ messages [94] (first suggested in [19]), as well as $k$-out-of-$n$
oblivious transfer, where Bob obtains $k$ of Alice’s $n$ messages (see [91]).

![1-out-of-2 Oblivious Transfer](image)

**Figure 2.2:** The start and end states of Alice and Bob in 1-out-of-2 oblivious transfer.

**Secret Sharing Schemes**

Shamir’s [114] ingenious secret sharing scheme splits a secret between a group of peo-
ple such that only certain subgroups together can retrieve the secret. The scheme is
based on polynomial interpolation. Briefly, the idea is for the secret value to be the
constant term of a $t$-degree polynomial. Each party’s ‘share’ of the secret is a point on
the polynomial, and the only way to reconstruct the whole polynomial (and hence the
secret value) is to have $t + 1$ points. So long as fewer than $t + 1$ parties are corrupt, the
secret is safe.

Using Shamir’s secret sharing scheme, it is possible to add and multiply secret values
in the shared space. Addition of secret values is very simple. Suppose each party holds
a point on a polynomial $f(x)$, and a point on a polynomial $g(x)$; then they can locally
add the points together to obtain a point on $f + g$. If the point on $f$ was a share of secret
value $s_1$, and the point on $g$ represented a share of the secret value $s_2$, then the point on
$f + g$ represents a share of the sum ($s_1 + s_2$). Multiplication is more complicated and we
refer the reader to [15, 35] for an explanation. Any method that uses secret sharing as
its base is information-theoretically secure — that is, without access to enough shares,
it is impossible for a corrupt party to reconstruct the secret value. An analogous scheme
in the computational model is called a homomorphic encryption scheme.
Homomorphic Encryption Scheme

A homomorphic encryption scheme is a special type of public-key cryptosystem [45,104] with the desirable property that one can carry out some basic arithmetic operations like addition and multiplication while in the encrypted domain. That is, given $E(x)$ and $E(y)$, one can compute $E(x \perp y)$ without decrypting $x$, $y$ for some operation $\perp$ [103]. (We use the notation $E(x)$ to indicate that a secret value $x$ has been encrypted with public key $E$.) One example is the Paillier cryptosystem, invented by Pascal Paillier in 1999, which is an additive homomorphic cryptosystem [98]. Given only the public-key and the encryption of messages $m_1$ and $m_2$, one can compute the encryption of $m_1 + m_2$. That is,

$$E(m_1 + m_2) = E(m_1) \cdot E(m_2).$$

In other words, multiplying the encryptions of $m_1$ and $m_2$ results in an encryption of $m_1 + m_2$ [103].

One drawback is that homomorphic encryption can only be performed on elements from a finite field. For example, Paillier’s scheme takes inputs from $[1, N]$, where $N$ is an RSA modulus, and encrypts them using exponentiations modulo $N^2$ [84]. Thus if we wish to use homomorphic encryptions on real values, we will have to use some kind of discrete approximation to those values, which can lead to a loss of accuracy. We will return to this idea in the final section of this chapter.

Prior to 2009, homomorphic cryptosystems either allowed addition operations or multiplication operations to be carried out in the encrypted domain, but not both. In 2009, Craig Gentry generated much excitement with his development of the first fully-homomorphic (allowing both addition and multiplication) encryption scheme as part of his PhD thesis at Stanford University [59, 60]. This means that a computer can perform a series of calculations on encrypted data without ever needing to decrypt it, making it possible to outsource private computations. However, Gentry’s scheme is still far from being practical for real-world applications due to the computational effort involved [112].

2.1.5 Secure building blocks

A number of basic protocols exist which allow common operations to be performed in a secure manner. These can serve as useful building blocks in the development of customised protocols for specific applications. Some basic protocols are described below, however, this list is not exhaustive. We will explain the first protocol in detail and list a few others briefly.
Secure Summation

Secure summation is one of the simplest protocols and will be explained here in detail. The goal is to add up a set of private values, each held by individual parties.

More formally, $k$ parties each have a secret value $v_i$, and they wish to know the total sum of their values $\sum_{i=1}^{k} v_i$. We assume that the total sum lies in the range $[0, n]$. The protocol begins by designating one party the ‘master party’, numbered 1. The other parties are numbered 2, \ldots, $k$. Party 1 generates a random number $R \sim U(0, n)$. It then computes $R + v_1 \pmod{n}$, and sends the sum, call it $q$, to Party 2. All arithmetic operations are performed mod $n$ to ensure that all intermediate values remain uniformly distributed on $[0, n]$ and thus reveal no information about the $v_i$ values. Party 2 computes $q + v_2 \pmod{n}$. The result is passed to Party 3, who similarly adds $v_3$ to it and passes the result to the next party. This process continues until the result is passed back to Party 1, who subtracts off the original random number $R$ to obtain $\sum_{i=1}^{k} v_i$, which it then distributes to the other parties, or goes on to use in subsequent calculations. See [33] for a more complete discussion.

A simple example of the process is given in Figure 2.3. Here, $n = 20$, Alice is the master party, and her secret value is $v_1 = 8$. Bob’s secret value is 5, and Carol’s is 3. Thus the algorithm needs to return 16, which it does. The numbers shown in bold alongside the arrows represent the values passed from one party to the next.

![Figure 2.3: Secure computation of a sum between 3 parties.](image)
Secure Comparison

This protocol was first presented as the ‘Millionaire’s Protocol’ in Yao’s seminal paper [132]. The original scenario involved two millionaires, Alice and Bob, who wish to determine who is richer, without disclosing their own wealth. If Alice has $x$ million and Bob has $y$ million, the algorithm simply returns 1 if $x > y$ and 0 otherwise. The secure comparison protocol can be used as a basis to find $\arg\min_i \{x_i\}$, $i = 1, \ldots, n$, where $x_i$ are secret values held by Parties $P_1$ to $P_n$.

Secure Scalar Product

Computing the scalar product is very useful in many applications. Alice has a vector $x$, Bob has a vector $y$, and we need to use the value $x \cdot y$. The complication is that knowing $x \cdot y$ can sometimes reveal parts of $x$ or $y$. For example, if Alice has $x = (1, 0, 0)$ and Bob has $y = (6, 7, 7)$. Then $x \cdot y = 6 + 0 + 0 = 6$. Thus Alice will know that Bob’s first element is 6.

To avoid such a situation, no party is allowed to know the whole of $x \cdot y$. We accomplish this by splitting the solution into 2 shares, $v_A$ and $v_B$ such that

$$v_A + v_B = x \cdot y.$$ 

Several protocols for securely computing the scalar product are outlined in [7, 61, 123]. Wang et al. [129] provide an empirical analysis to compare the run-time of several of the available scalar product protocols.

The idea of splitting the output is referred to as ‘additive splitting’ [79]. We say that a data value $x$ is ‘additively split’, if Alice owns a share $x_A$, and Bob owns a share $x_B$, such that $x = x_A + x_B$. Note that $x_A$ and $x_B$ can be large, and can be either positive or negative, so that a share does not allow inferences to be easily made about $x$ [8]. To be information-theoretically secure, splitting should be performed modulo some number $N$ [5]. That is, $x = x_A + x_B \mod N$. If arithmetic is not modular, the value of $x$ is still secure in a ‘practical’ sense [5], in that the chance of information being revealed about $x$ is negligible. A value can also be XOR split, where $x = x_A \oplus x_B$ [58]. In this thesis, we will only use additively split data.

Set Operations

Algorithms have been developed for set intersection and set union (eg. see [21, 33]). For example, suppose Party A owns a set $S_a$ and Party B owns $S_b$. They can calculate the union $S_a \cup S_b$ without revealing the complete contents of their individual sets, only what is revealed by the output. That is, they don’t need to reveal $S_a \cap S_b$. 


2.1.6 Protocol composition

Given that these building blocks exist, it would be desirable to have a design methodology that guaranteed that when we used the blocks in combination, the overall protocol would be secure. What are the security guarantees of this kind of protocol composition?

Protocol composition is a non-trivial task. There are different types of composition. For instance, we can have sequential composition, where protocols are executed one after another, or concurrent composition, where protocols are run in parallel. We must also consider multiple runs of the same protocol. The currently available theorems regarding protocol composition are due to Canetti [23], who developed the notion of universally composable security [24].

Not all types of composition have been resolved. There are theorems for sequential composition, which is the situation we face in Chapter 8 of this thesis. The basic idea behind the sequential composition theorem is that we can design a protocol that uses a secure building block as a subroutine, and then analyse the security of the larger protocol by assuming that a trusted third party computes the subroutine [84].

To understand this more fully, assume that we have designed a ‘high-level’ protocol that uses a secure building block, such as secure scalar product, as a subroutine. First, we must prove the security of the secure scalar product protocol for the case when it is executed as a stand alone protocol. If we are using an existing secure building block, this has been done for us. Next, we consider the security of the ‘high-level’ protocol and assume that a trusted third party will step in to help compute the scalar product [23]. In other words, the call to the secure subroutine is replaced by an ‘ideal call’ to a trusted third party.

The composition theorem states that when the ‘ideal calls’ to the trusted third party to compute the subroutine are replaced by real executions of a secure protocol computing the subroutine, the protocol remains secure [84]. This equivalence is depicted in Figure 2.4. More formally, suppose we have a function \( g \) that is computed by invoking functions \( f_1, \ldots, f_n \) in sequence. Canetti’s composition theorem [23] states that if a function \( g \) is proven secure when these functions are computed by a trusted third party, then \( g \) will remain secure when the calls to the trusted third party are replaced by secure protocols which compute \( f_1, \ldots, f_n \). A consequence of this theorem is that as long as all intermediate output (the output from the secure building blocks \( f_1, \ldots, f_n \)) in the high-level protocol is shared (e.g. additively split), the overall protocol \( g \) will be secure in the semi-honest model [5].
2.1.7 Application areas

At the time of its inception, secure multiparty computation was of largely theoretical interest. However, the development of more efficient algorithms and increased computing power has led to a number of real-world applications. These include privacy-preserving data mining [82, 124], secure voting [36], and secure auctions [18, 95]. We describe some of these application areas below, although we stress that most are still at the ‘hypothetical’ stage, and have yet to see practical use.

Privacy-Preserving Data Mining: The goal in privacy-preserving data mining is to allow several database owners to compute queries on their joint databases, without disclosing their individual (private) records. For example, two medical research institutes may wish to compute aggregate statistics over private medical records stored in their respective databases [84]. In one of the seminal papers, Lindell and Pinkas [82] show how to perform a decision tree construction algorithm on a joint database. Privacy-preserving data mining is discussed comprehensively in [124].

Collaborative Intelligence Gathering: This is another real-world application of privacy-preserving data mining. Detecting terrorist threats, disease outbreaks, or credit card fraud is often improved if one has access to global, rather than only local, information [32]. However, parties are often reluctant to share their data due to conflicts of inter-
Electronic Voting: A challenge in electronic voting systems is security. Secure multiparty computation can allow voters to ensure that the votes are valid, and are tallied correctly, without revealing individual votes [36].

Defence: Roughan and Arnold [105] show how one could combine private target information from several sensors owned by allies (who may not necessarily be allies in the future) to obtain an improved estimate of the target’s location than could be achieved by one sensor alone.

Scientific and Statistical Computations: Du and Atallah [7, 48–51] examined a range of two-party applications, from least-squares regression to a variety of geometric problems, such as finding the closest pair of points belonging to sets owned by different parties. However these studies usually stopped short of taking the protocol all the way to a real application. In other words, protocols were developed as workable building blocks, but were not actually implemented in any real-life situations.

E-commerce: Catrina and Kerschbaum [26] examine how to help foster the uptake of secure computation in the area of e-commerce. They note that e-commerce has a strong emphasis on privacy, and that applications such as benchmarking (comparing your company’s performance indicators against those of your competitors), auctions and collaborative supply chain management could all benefit from secure multiparty computation. Atallah et al. devise protocols for secure benchmarking [6].

Auctions: Sealed-bid auctions place a great deal of faith in the auctioneer. In a Vickrey auction, where the winner pays the second-highest bid, the bidders must trust the dealer to report the second-highest bid correctly. For instance, the dealer might ask the auction winner to pay $1000, when in fact the second-highest bid was only $800, and pocket the difference [95]. Secure computation can bypass these problems and greatly reduce the amount of trust one needs to place in the dealer (see [95] for details). In fact, the first practical use of secure computation was an auction involving sugar beet licences [17], discussed in the next section.

2.1.8 Challenges: theory to practice

Although the theoretical development of secure multiparty computation has undergone significant advancement, the development of practical applications has been slow [28]. It is not clear why this is so. It could be that it was important to thoroughly understand the theoretical basis before attempting to apply the protocols in practice. Alternatively,
since the applications abound in such diverse disciplines, it may be that increased exposure of secure computation is needed across disciplines in order to spark the interest of other researchers who may then be able to see applications in their disciplines. One group of researchers suggest that the lack of commercial uptake could also be due to ‘a general lack of understanding in the general public of the potential of the technology’ [17, p.329].

The first large-scale commercial application of secure multiparty computation took place in 2008, in the form of an online double-auction of sugar-beet licenses in Denmark [17]. (A double-auction is used to find a fair market price for a commodity given the existing supply and demand in the market [37].) The aim was to reallocate sugar-beet production licences to where they would be most profitable. The auction was successful, involving 1200 bidders and resulting in 25,000 tonnes of production rights changing hands. Not only did the use of secure computation bypass any lengthy security discussions, it meant that all bidders could specify how much they would buy or sell at a certain price with the knowledge that all information would remain private, and thus they were in no danger of jeopardising their positions. Secure multiparty computation provided a low-cost electronic solution. In a survey conducted after the auction, 80% of participants said that the confidentiality of their bids was important to them [17].

A number of challenges remain before secure computation can be implemented in everyday life. The first is efficiency — e.g., the Danish protocol took up to 1 minute to encrypt and submit a bid, and the entire joint calculation took 30 minutes [17]; this timeframe may be unacceptable for some applications. Another challenge is raising awareness of the technology. There are probably a host of applications in other disciplines, but the researchers there don’t know that privacy-preserving computation exists. Cheung and Nguyen [28] note that, with demand for secure and privacy-enhancing applications growing rapidly, the time is ripe for researchers in diverse disciplines outside of cryptography to ‘understand the concepts of SMC [Secure Multiparty Computation] and to develop practical SMC protocols for their respective applications’ (p. 9). In particular, there is still plenty of scope for applications in the area of optimisation; we intend to help fill this gap.

2.1.9 Related secure optimisation work

Several papers have examined optimisation problems in the context of secure multiparty computation. Brickell and Shmatikov [21] showed how shortest path algorithms could be applied in a privacy-preserving manner to a joint graph with pieces owned by competing companies. A motivating application was a potential merger between two Internet service providers, who wish to see how efficient the resulting joint network would be without revealing the details of their existing networks.
Sakuma and Kobayashi [109] were the first to consider the travelling salesman problem, an NP-hard (non-deterministic polynomial-time hard) combinatorial optimisation problem, in a privacy-preserving setting. They looked at a case where a client wants freight delivered to a variety of cities and is trying to decide between two shipping companies, A and B. The client cannot reveal the delivery points to either company before contracting them. Similarly, the delivery costs between any two delivery points are confidential and cannot be revealed to the client. In general, we have a ‘server’, possessing the costs between all cities, and a ‘searcher’, who has list of cities that need to be visited and would like to know the cost of the optimal tour. This is illustrated in Figure 2.5. The authors solve this problem using a genetic algorithm in combination with privacy-preserving protocols [109].

Roughan, Shen, Sang, Zhang and colleagues have considered a variety of traffic management problems, including privacy-preserving performance measurement and secure data aggregation [106, 107, 111]. They have also considered inter-domain routing — finding shortest paths across computer networks owned by mistrusting parties with minimal sharing of information [108].

Supply chain management is another area that can benefit from privacy-preserving optimisation [5, 8, 99, 113]. Companies wish to optimise the movement of goods through the supply chain without needing to reveal cost or capacity data (which could sacrifice their competitive advantage or breach anti-trust laws [32, p. 20]). Atallah et al. [5] consider determining order quantities based on joint-forecasts to minimise the supply-
chain expected cost; while Pibernik et al. [100] consider the joint economic lot-sizing model, which determines the optimum inventory size for the buyer and the supplier.

A final type of secure optimisation problem that has been considered is secure linear programming [25, 43, 48], also called privacy-preserving linear programming [11, 86, 121], which is the focus of this thesis. We first define regular linear programming before considering the addition of privacy concerns.

2.2 Linear programming

Linear programming is a technique used to solve a large class of operations research problems in which the objective function and constraints are linear. Operations research is the study of how to improve the efficiency of business and industry operations. These types of problems include:

- transportation problems – delivering goods for the minimum cost while satisfying supply/demand constraints;
- scheduling problems, such as the scheduling of buses; and
- resource allocation problems – deciding how many of each product to make to maximise profit while satisfying resource (raw material and labour) constraints.

These problems occur in a very wide range of industries, including agriculture, economics, transportation, and the military.

Each of these types of problems can be formulated as a linear program. A linear program is an optimisation problem in which the goal is to maximise or minimise a linear objective function of decision variables subject to a set of linear constraints. For instance, the objective might be to maximise profit, while the decision variables might be how many of each product to manufacture, subject to resource constraints. Each constraint is either an equality or an inequality associated with some linear combination of the decision variables.

One particular formulation of the linear programming problem, known as standard form, is:

$$\max \quad f(x) = c^T x$$
$$\text{subject to} \quad Mx = b$$
$$x \geq 0,$$  \hspace{1cm} (2.2.1)

where $c \in \mathbb{R}^n$ is a vector of objective function coefficients, $M \in \mathbb{R}^{m \times n}$ is a constraint matrix, $b \in \mathbb{R}^m$ is a vector of constraint values, and $x \in \mathbb{R}^n$ is a vector of decision variables. Throughout this thesis, all vectors are lower case and bold, while all matrices are written in upper case and are not bold.
Several variations of this form are possible. For example, we can have minimisation
instead of maximisation, or inequality constraints instead of (or in addition to) equality
constraints. However, the different variations can all be written in the standard form
shown in (2.2.1) [125]. The conversion process is discussed in more detail in Chapter 3,
where we will consider whether such conversions can be allowed in a secure setting.

2.2.1 Definitions and basic theory

A ‘feasible solution’ to a linear program is a set of values for the decision variables which
satisfies all of the constraints. Every linear program also has an associated ‘feasible re-

gion’, which is the set of all such feasible solutions [30]. More formally, the feasible
region is the convex set $C = \{ \mathbf{x} \in \mathbb{R}^n | M \mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0 \}$. To find the optimal solution
to the linear program, we need to find the vector(s) $\mathbf{x} = (x_1, \ldots, x_n)^T \in C$ for which
$f(x_1, \ldots, x_n)$ takes its maximum (or minimum) value. The optimal solution is denoted
by $\mathbf{x}^*$.

The fundamental theorem of linear programming states that the optimal solution, if it
exists\(^2\), always lies on one of the vertices (corner points) of the feasible region [102].

We illustrate this idea via an example.

Consider the following simple example of a linear program:

$$
\begin{align*}
\text{max} & \quad z = 5x_1 + x_2 \\
\text{s.t.} & \quad 2x_1 + x_2 \leq 8 \\
& \quad 2x_1 + 3x_2 \leq 12 \\
& \quad x_1, x_2 \geq 0.
\end{align*}
$$

(2.2.2)

Its associated feasible region is shown in Figure 2.6. The optimal solution, if it exists,
will occur at one of the vertices of the feasible region. In this example, the vertices
are $A(0,0)$, $B(0,4)$, $C(3,2)$ and $D(4,0)$, as shown in Figure 2.6. One way to find the
optimal solution is to evaluate the objective function at each of these vertices. If we
do this, we will see that the optimal solution (maximum value of $z$) occurs at vertex
D, when $x_1 = 4$ and $x_2 = 0$. This is also the point at which the objective function last
‘touches’ the feasible region if we consider moving the objective line up the objective
surface across the feasible region, from left to right in this case, as in Figure 2.7. Later
on, in Chapter 9, we will see how this relationship between the objective function and
the optimal solution can form the basis of an attack on privacy-preserving linear pro-
gramming. The corresponding optimal value is $5(4) + 0 = 20$.

How can we find the vertices algebraically? A necessary and sufficient condition for a
point $\mathbf{x}$ to be a vertex of the feasible region is that $\mathbf{x}$ is a ‘basic feasible solution’. We can

\(^2\)Sometimes, a linear program is infeasible, meaning that it has no feasible solutions, or is unbounded
[102]. In these cases, no optimal solution exists.
Figure 2.6: The feasible region for the example linear program in (2.2.2). The optimal solution, if it exists, will occur at one of the vertices of the feasible region. In this example, the vertices are A(0,0), B(0,4), C(3,2) and D(4,0). One way to find the optimal solution is to evaluate the objective function at each of these vertices.

Figure 2.7: The optimal solution occurs at vertex D, when \( x_1 = 4 \) and \( x_2 = 0 \). This is the point at which the objective function last ‘touches’ the feasible region. The maximum value of the objective function, \( z \), is 20.
find these basic feasible solutions algebraically as follows. First suppose we have a linear program in standard form which has \( m \) constraint equations in \( n \) unknowns, where \( n > m \). A ‘basic solution’ is obtained by setting \( n - m \) variables equal to 0 and solving the system of \( m \) equations for the remaining \( m \) variables (some of these systems may not be consistent). The solution to the system will have \( m \) so-called ‘basic variables’ and \( n - m \) ‘non-basic variables’ (equal to 0). If all variables are non-negative, the solution is called a ‘basic feasible solution’ and corresponds to a vertex of the feasible region.

Normally there are too many basic feasible solutions to apply the ‘brute-force approach’ and try each vertex in turn. For example, if \( n = 50, m = 20 \), there are potentially \( \binom{50}{20} \approx 10^{13} \) basic feasible solutions — far too many to enumerate by hand! Instead, more efficient methods have been developed.

### 2.2.2 Brief history

In the late 1930s, Kantorovich devised solutions to problems concerning transportation and production planning, and Koopmans made a substantial contribution to the solution of transportation problems in World War 2 [30].

Some time later, in 1947, Dantzig developed a new method for solving linear programs, called the Simplex method [39]. The method arose out of work he did on planning problems for the U.S. Air Force [40]. His algorithm is considered one of the ten algorithms with the greatest influence on the development and practice of science and engineering in the twentieth century [47].

The Simplex method exploits the property that the optimal solution will occur at a vertex. The algorithm jumps from vertex to vertex, in such a way that the objective function will be no worse at the next vertex than it is at the current vertex. Simplex accomplishes these jumps through a pivoting operation on a matrix. The matrix is called the ‘Simplex tableau’. We will not go into the precise details of the pivoting procedure here, but interested readers are referred to Vanderbei [125] for a comprehensive treatment.

Although it is elegant and efficient, the Simplex algorithm has the undesirable property of having an exponential worst-case performance — the number of steps grows exponentially with the number of variables. An example problem displaying the worst-case performance was given by Klee and Minty [76].
This realisation sparked the search for an algorithm with polynomial-time\(^3\) complexity \[30\]. The first polynomial-time algorithm for linear programming, called the ellipsoid method, was discovered by Khachiyan in 1979 \[125\], but despite being better in theory, it could not compete with the Simplex algorithm in practice.

In 1984, Karmarkar \[72\] developed a different polynomial-time algorithm, known as an interior point method, and claimed that it was faster than the Simplex method \[125\]. While his claim was not universally true in practice, interior point methods are competitive with the Simplex method and usually give superior performance for very large problems \[4,125\]. Karmarkar’s work led to the development of other interior point methods, and it is still an active research area today. In practice, depending on the problem to be solved, it will sometimes be preferable to use Simplex, and other times an interior point method. We come back to this point when we motivate the need for an interior-point based method for privacy-preserving linear programming in Chapter 8.

### 2.3 Privacy-preserving linear programming

Since Dantzig’s invention of the Simplex method, many years of effort have gone into deriving fast, computationally efficient methods for solving linear programs. These solutions assume, however, that all components of the linear program (the objective function and constraints) are completely known.

In privacy-preserving linear programming, the difficulty is that the components are not completely known by a single party. Instead, two or more parties each contribute parts of the objective function and/or constraints, which must be kept secret from the other parties. Only the optimal solution (or part of the optimal solution) is allowed to be made public. Such a formulation is useful when mutually-distrusting parties wish to perform global optimisation and solve a joint-program for mutual benefit. A two-party privacy-preserving linear program can be expressed in the following general form:

\[
\begin{align*}
\text{max} & \quad (c_1 + c_2)^T x \\
\text{s.t.} & \quad (M_1 + M_2)x \quad \leq \quad (b_1 + b_2) \\
& \quad x \quad \geq \quad 0,
\end{align*}
\]  

(2.3.1)

where \(c_1 + c_2\) is a vector in \(\mathbb{R}^n\), \(b_1 + b_2\) is a vector in \(\mathbb{R}^m\) and \(M_1 + M_2\) is an \(m \times n\) matrix of constraint coefficients. Throughout this thesis, the two parties will be named Alice and Bob. The components \(\{c_1, M_1, b_1\}\) are owned by Alice, while \(\{c_2, M_2, b_2\}\) are owned by Bob. Precisely which of these components are private and the form they take depends on the situation, as we will see in the following examples.

\[^3\text{This means that the solution can be found in an amount of time that is bounded by a polynomial in } n, \text{ the number of variables } [30].\]
Privacy-preserving linear programming is a relatively new research area. The problem itself was first posed by Du in 2001, as part of his PhD thesis [48]. One of the complicating factors of this area is that privacy-preserving linear programming brings together two different fields: security and optimisation. It is not common for researchers to have expertise in both of these areas.

Another difficulty is that optimisation problems often involve real numbers, whereas cryptography is normally restricted to a finite field. There is something of a ‘culture clash’ when the culture of linear algebra comes up against the cryptographic culture of finite fields. For example, when using finite fields, one has to make the finite field bigger than the maximum value to be represented (in order to simulate integer arithmetic without wraparound occurring). However, this type of requirement can be difficult to satisfy in optimisation, because one cannot predict in advance how big the optimum will be. We first present some examples of privacy-preserving linear programs before considering these issues further.

2.3.1 Examples

We present two real-world examples to show how privacy-preserving linear programming could be useful.

Example 2.1. (Production Problem)

The first (non-cooperative) part of this problem was adapted from an example in [3]. Gary’s Golf produces regular and deluxe golf bags. Production consists of four stages:

1. Cutting and dyeing the material
2. Sewing
3. Finishing
4. Inspection and Packing

There are a limited number of hours available for each task. Table 2.2 lists the number of hours each bag needs per department, as well as the total available hours in each department over the coming 3-month period.

Gary’s Golf makes a profit of $10 per regular bag and $15 per deluxe bag. The company would like to determine how many of each type of bag to make in order to maximise their profit (assuming all bags made will be bought).
Let $R$ be the number of regular bags to make and $D$ be the number of deluxe bags to make. The production problem is formulated as a linear program below:

$$
\begin{align*}
\text{max } z &= 10R + 15D \\
\text{s.t. } & \frac{7}{10}R + 1D \leq 630 \\
& \frac{1}{2}R + \frac{5}{6}D \leq 600 \\
& 1R + \frac{2}{3}D \leq 708 \\
& \frac{1}{10}R + \frac{1}{4}D \leq 150 \\
& R, D \geq 0
\end{align*}
$$

The optimal solution is $(R^*, D^*) = (100, 560)$, giving a total profit of $z^* = $9400.

Now suppose that Gary’s Golf would like to shut down their cutting and dyeing department and outsource this stage to another company. However, they only wish to do this if their profits will improve. Suppose Gary’s Golf starts by approaching Company B. Suppose Company B has 600 spare hours of capacity for cutting and dyeing, and that they can cut and dye a regular bag in $\frac{4}{10}$ of an hour, and a deluxe bag in $\frac{7}{10}$ of an hour. However, Company B does not want to disclose its capacity or production times to Gary’s Golf, just as Gary’s Golf doesn’t want to reveal its production times, capacity or profit margins to Company B, in case the deal does not go ahead. All that Company B is willing to reveal is that it will charge $0.80$ (over the existing costs) to cut and dye a regular bag and $1$ (over the existing costs) to cut and dye a premium bag. How can Gary’s Golf decide if the deal is worthwhile?

To assess the viability of the deal, the companies can formulate the joint problem as a privacy-preserving linear program. Here, Company B owns the first constraint (which must be kept private from Gary’s Golf), while Gary’s Golf owns the remaining constraints together with the objective function (which must be kept private from Company B), as shown below. Note that the profit values in the objective function have been adjusted to account for Company B’s charges.
Privacy-Preserving Linear Program — with outsourcing to Company B

\[
\begin{align*}
\text{max} \quad & z = 9.2R + 14D & \quad \text{(Gary’s Golf)} \\
\text{s.t.} \quad & \frac{4}{10}R + \frac{7}{10}D \leq 600 & \quad \text{(Company B)} \\
& \frac{1}{2}R + \frac{2}{6}D \leq 600 & \quad \text{(Gary’s Golf)} \\
& 1R + \frac{2}{3}D \leq 708 & \quad \text{(Gary’s Golf)} \\
& \frac{1}{10}R + \frac{1}{4}D \leq 150 & \quad \text{(Gary’s Golf)} \\
& R, D \geq 0
\end{align*}
\]

Assuming a solution procedure for this joint-program exists, the optimal solution under this outsourcing scheme is \((R^*, D^*) = (420, 432)\), with a total profit of \(z^* = 9912\). Thus Gary’s Golf would be $512 better off under this scheme.

The joint-LP is not straightforward to solve, since neither party owns all of the components. The purpose of this example is merely to show how a jointly-defined linear program might arise in practice, but we will not discuss how to solve it until Section 2.3.2. The goal is for Gary’s Golf and Company B to jointly solve this problem (without the use of a trusted third party) such that Gary’s Golf doesn’t find out Company B’s data and vice versa, and only Gary’s Golf learns the optimal solution.

Gary’s Golf could run a similar linear program with any other contractors under consideration and then broker a deal with whichever one results in the highest net profit. The beauty is that, should a deal not go ahead, neither company is any the wiser about the other’s capacity or constraints. In this way, privacy-preserving linear programming allows businesses to test the benefits of a partnership with minimal risk.

Example 2.2. (Transportation Problem)

This example has been adapted from Vaidya [122]. A winemaking company grows grapes in three locations, and then sends them to different wineries to produce wine. The company grows two different varieties of grapes, called ‘grape 1’ and ‘grape 2’, which are needed by the wineries in varying amounts.

Suppose the company decides to outsource the delivery of the grapes, so that it can focus its attention on growing the grapes. They wish to hire a contractor to do the deliveries for them, and naturally want to choose the contractor who will be able to deliver the required grapes for the lowest cost. However, the contractors are not willing to reveal their delivery costs to the winemaker. Similarly, the winemaker would rather not reveal their supply and demand information, which constitute private company information, in case the deal doesn’t eventuate.
This transportation problem is another example of a secure linear program — the contractor holds the objective function, which must be kept private from the winemaker, while the winemaker holds all of the supply and demand information, which must be kept private from the contractor. Figure 2.8 summarises the supply and demand information, while the transportation costs per kilogram of grapes are shown in Table 2.3. Note that this table of delivery costs represents private data for the contractor.

![Figure 2.8: Summary of the grape transportation problem. The figures alongside the vineyards represent the supply of each type of grape available from each vineyard, and the figures alongside the wineries represent the demand for each type of grape at each winery.](image)

<table>
<thead>
<tr>
<th>Vineyards</th>
<th>Wineries</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1: 20 tonnes G2: 10 tonnes</td>
<td>G1: 20 tonnes G2: 15 tonnes</td>
</tr>
<tr>
<td>G1: 10 tonnes G2: 0 tonnes</td>
<td>G1: 20 tonnes G2: 5 tonnes</td>
</tr>
<tr>
<td>G1: 10 tonnes G2: 10 tonnes</td>
<td></td>
</tr>
</tbody>
</table>

\[ c_{ij} = \text{cost ($/kg) of transporting grapes (of any type) from vineyard } i \text{ to winery } j, \quad i = 1, 2, 3; \ j = 1, 2. \]

**Figure 2.8:** Summary of the grape transportation problem. The figures alongside the vineyards represent the supply of each type of grape available from each vineyard, and the figures alongside the wineries represent the demand for each type of grape at each winery.

<table>
<thead>
<tr>
<th>Winery 1</th>
<th>Winery 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vineyard 1</td>
<td>10</td>
</tr>
<tr>
<td>Vineyard 2</td>
<td>12</td>
</tr>
<tr>
<td>Vineyard 3</td>
<td>13</td>
</tr>
</tbody>
</table>

**Table 2.3:** Transportation costs per kg of grapes. The cost per kg is the same for both types of grapes. These costs represent private data for the contractor.

We now define the decision variables. Let

\[ g_{ij} = \text{number of kg of grape 1 to deliver from vineyard } i \text{ to winery } j, \]  
\[ h_{ij} = \text{number of kg of grape 2 to deliver from vineyard } i \text{ to winery } j, \]

where \( i \in \{1, 2, 3\} \) and \( j \in \{1, 2\} \). The objective is to determine the values of these variables that will minimise the total transportation cost while satisfying the demand
and supply restrictions. We formulate the linear program as follows, with the supply and demand values converted to kilograms:

**Privacy-Preserving Linear Program (with objective function owned by contractor and constraints owned by client)**

Minimise \( z \) (total delivery cost)

\[
\begin{align*}
\text{s.t.} & \\
g_{11} + g_{21} + g_{31} & \geq 20000 \quad \text{(demand for G1 at winery 1)} \\
g_{12} + g_{22} + g_{32} & \geq 20000 \quad \text{(demand for G1 at winery 2)} \\
h_{11} + h_{12} + h_{31} & \geq 15000 \quad \text{(demand for G2 at winery 1)} \\
h_{12} + h_{22} + h_{32} & \geq 5000 \quad \text{(demand for G2 at winery 2)} \\
g_{11} + g_{12} & \leq 20000 \quad \text{(supply of G1 at vineyard 1)} \\
g_{21} + g_{22} & \leq 10000 \quad \text{(supply of G1 at vineyard 2)} \\
g_{31} + g_{32} & \leq 10000 \quad \text{(supply of G1 at vineyard 3)} \\
h_{11} + h_{12} & \leq 10000 \quad \text{(supply of G2 at vineyard 1)} \\
h_{21} + h_{22} & \leq 0 \quad \text{(supply of G2 at vineyard 2)} \\
h_{31} + h_{32} & \leq 10000 \quad \text{(supply of G2 at vineyard 3)} \\
g_{ij}, h_{ij} & \geq 0 \quad \text{for } i = 1, 2, 3; j = 1, 2,
\end{align*}
\]

where

\[
z = 10(g_{11} + h_{11}) + 26(g_{12} + h_{12}) + 12(g_{21} + h_{21}) + 13(g_{22} + h_{22}) + 13(g_{31} + h_{31}) + 8(g_{32} + h_{32}).
\]

To write the problem in matrix-vector form, let

\[
x = [g_{11} \ g_{12} \ g_{21} \ g_{22} \ g_{31} \ g_{32} \ h_{11} \ h_{12} \ h_{21} \ h_{22} \ h_{31} \ h_{32}].
\]

In matrix-vector form, the problem is written as:

Min \( z = [10 \ 26 \ 12 \ 13 \ 13 \ 8 \ 10 \ 26 \ 12 \ 13 \ 13 \ 8]^T x \)

\[
\begin{bmatrix}
-1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & -1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{bmatrix} x \leq \begin{bmatrix}
-20,000 \\
-20,000 \\
-15,000 \\
-5,000 \\
20,000 \\
10,000 \\
10,000 \\
10,000 \\
0 \\
10,000
\end{bmatrix}
\]

\[
x \geq 0.
\]
Note that the first four constraints have been multiplied by $-1$ to make them $\leq$ constraints. Here, the cost vector $c$ represents private data for the contractor, while the RHS vector $b$, containing the supply and demand information, represents private data for the winemaking company. The constraint matrix $M$ does not contain any sensitive information and is allowed to be public.

Notice that this example has a different partitioning of the components than the golf-bag production example — here, one party holds $c$ and the other holds $b$; while $M$ is public. In the production example, on the other hand, one party held $c$ and part of $M$, while the other party held the rest of $M$. The fact that there are many different partitioning options is an important feature of privacy-preserving linear programs, which will be discussed further in Chapter 3.

2.3.2 Existing solution methods

Several different methods for privacy-preserving linear programming have been developed. These can be divided into two classes: cryptographic methods \cite{25,79,119,122} and transformation-based methods \cite{11,48,86,87,121}. Cryptographic methods implement a privacy-preserving version of the Simplex method, while the transformation-based methods ‘disguise’ the original linear program by way of random matrices. We discuss each class, and their advantages and limitations, in turn. One of the things we’ll see as we progress is that the different techniques make sense in different contexts.

Cryptographic Methods

The cryptographic methods (that existed prior to our work) invoke formal cryptographic techniques \cite{25,79,119,122} in order to implement a privacy-preserving version of the Simplex method. These approaches hide the private data by using cryptographic techniques at each step in the algorithm. For example, Li and Atallah \cite{79} use homomorphic encryption and blind-and-permute procedures to permute the tableau at each iteration, while Toft \cite{119} uses secret-sharing with shared matrix indexes.

Most cryptography is carried out over a finite field and thus is constrained to integer data values. For example, some of the Simplex-based methods \cite{119} impose integer restrictions on the objective function coefficients, and constraint coefficients. This restriction is a direct result of the cryptographic tools being used. Toft’s method \cite{119} must be instantiated using tools such as secret sharing (eg. Shamir sharing over $\mathbb{Z}_q$) or threshold homomorphic public key cryptography (eg. Paillier encryption), both of which can only be used for integer values.

In addition, the finite-field requirement means that only integer-variants of Simplex can be used. This presents a problem, since, as Catrina and de Hoogh \cite{25} point out, solving non-trivial linear programs using these algorithms leads to computation with
integers potentially involving thousands of bits. One iteration of Li and Atallah’s [79] method can potentially double the bit-length of the values in the tableau [119], leading to an explosion in bit length. As Toft explains:

“As the computation is secure, we must not learn the actual bit-lengths, implying that we must always work with full-size numbers. Starting with 32-bit numbers, after ten iterations these have grown to 32-thousand bits. After twenty they have increased to 32 million. Thus, even for small inputs their basic operations soon become modular exponentiations with a million-bit modulus.” [119, p. 3]

An improved version by Toft [119] has a linear growth, but the bit-length can still reach thousands of bits, which limits the practical applicability [25]. Catrina and de Hoogh [25] alleviate the integer restriction by devising a secure Simplex variant which can handle fixed-point rational numbers. Although this provides substantial efficiency gains, they still face a bottleneck due to the secure comparison operations. As a result, none of the existing cryptographic methods are suitable for large optimisation problems.

Deitos and Kerschbaum [44] improve the efficiency of Li and Atallah’s technique [79] by introducing a probabilistic element which reduces the number of secure permutations required. Another avenue for increasing efficiency is to introduce parallel processing [43]. Although there are avenues for improving the efficiency of these algorithms, the bottom line is that they require all iterations to be done within an encrypted domain, which results in a very large computational overhead.

In summary, although these methods offer strong security guarantees, they have efficiency problems. Can we trade some sensible amount of security in order to get an efficiency boost? This is the idea behind the transformation methods.

Transformation Methods

In the transformation methods, rather than trying to secure each step of an existing LP algorithm, all the secrecy-related work happens before the linear program is solved. Methods of this type ‘disguise’ the linear program using random matrices, and then solve the linear program in the disguised domain. It is important to point out that the transformation methods often still employ some cryptographic sub-protocols in order to apply the disguise, but the big difference is that these protocols are only performed once at the start of the algorithm, rather than at each step.

In terms of efficiency, applying the disguise and then solving the disguised problem is more work than solving the original linear program, since one still has to solve the linear program once it is disguised. However, this is less work than performing encryptions/decryptions at every step of the algorithm, as done in the cryptographic methods.
The first transformation-based approach was presented by Du [48]. Du’s idea was to convert the original LP into an equivalent system in which the constraints and objective function had been ‘disguised’ by random matrices. One party solved the disguised problem, and then the solution was converted back to the original domain. (Du’s method will be explained more fully in Chapter 4.) The appeal of such an approach is its simplicity, efficiency and the freedom to apply any LP solver once the problem is disguised. However, Du’s method was flawed [11, 121]. It was later corrected by Bednarz et al. [11] (also discussed in Chapter 4), but only for a special case where Alice owns the objective function and Bob owns the constraints. Mangasarian [86,87] recently developed the first transformation method for the multi-party case, and for vertically-partitioned data [86]. Moreover, Mangasarian was the first to propose a jointly-chosen disguise. An important consequence is that no cryptographic operations whatsoever are required to apply the disguise, making Mangasarian’s methods the most efficient to date.

Recently, Wang et al. [128] proposed a new type of transformation method for a cloud computing scenario. In this case, the user owns the complete linear programming problem, but has insufficient computing resources to solve it. The user decides to “outsource” the problem to “the cloud”. However, the user is unwilling to reveal their private data to the cloud, so employs a transformation method to disguise it. The transformation applies an affine mapping, consisting of a matrix multiplication plus a translation. Wang et al.’s method also has no reliance on cryptography to apply the disguise — all matrix multiplications can be performed locally by the user. The cost of solving the encrypted LP versus solving the original LP is found to be minimal. In addition, Wang et al. cleverly employ duality theory as a means of allowing the user to verify that the computation performed by the cloud is correct. See [128] for more details.

The transformation approaches offer several advantages over the cryptographic methods. The primary advantage is efficiency — once the problem is disguised, it can be solved locally by one of the parties with no need for further cryptographic operations. The second advantage is that users are not bound to the Simplex method, but are free to take advantage of the last half-century of advances in linear programming techniques. The third advantage is that we avoid the problem of how to perform cryptographic operations on non-integer numbers, and are free to use the full power of floating-point arithmetic. Finally, the algorithms are conceptually simpler and do not require specialised optimisation software.

Yet the transformation methods are not without their disadvantages. A criticism of these methods is that they only provide ‘heuristic’ security guarantees [79, 121]. Indeed, without the cryptographic protocols at each step, security is more challenging to prove. In this thesis we adopt (and extend) the ‘acceptable security’ paradigm of Du and Zhan [53], and quantify the probability of a security breach, which is so small as to be considered negligible for large problems. This is done in Chapters 5 and 7. We
Chapter 2. Background

will see that large problems actually strengthen the security guarantees of the transformation methods, which, in conjunction with their high efficiency, makes them ideal for large problems.

Another limitation of the transformation methods is that they are heavily dependent on the way the data is partitioned between the participants; the same disguise will not work for any type of partitioning. For example, the methods of Vaidya [122] and Bednarz et al. [11] apply only to the case when one party owns the objective function, and the other owns the constraint matrix. This lack of flexibility can be considered a drawback, at least until such time that we have a method to cover every possible partitioning scenario. There is also no standard way of defining the subset of LPs to which a transformation applies. This is a serious problem, since applying a transformation method to the wrong type of LP can void its security guarantees; we discuss this problem further in Chapter 3, and propose a new notation system to classify LPs.

Similarly, each transformation method is restricted to LPs with certain types of constraints and variables. For example, Mangasarian’s method [87] is designed for LPs which only have equality constraints — inequality constraints which have been converted to equality constraints via slack variables are not permitted. It would seem that converting an LP to standard form (using slack variables) would be equivalent to having strict equality constraints, but this equivalence does not hold in the context of privacy-preserving LPs. We will see why in the next chapter.

The cryptographic methods, on the other hand, are far more generic. To the best of our knowledge, they are more robust to problem structure and the types of constraints and variables. Most methods assume that all values of the constraint matrix, objective function, and RHS values are secret-shared between the participants (e.g. [25,119]).

It is important to point out that the transformation methods can also face problems due to integer restrictions, depending on the cryptographic tools used to apply the disguise, although to a lesser degree (as the cryptographic tools are only needed for the transformation stage, not for the solution of the LP). For example, Vaidya [121] notes that his secure transformation procedure will only operate on integers in a finite field, due to the use of homomorphic encryptions. The ideal is to have transformation methods that don’t require any cryptographic tools with integer restrictions, like Mangasarian’s method. The transformation methods that we will apply in this work don’t require homomorphic encryption and hence can be applied to floating point numbers.

Which approach is better?

There are advantages and limitations associated with both techniques, so it is important not to discount either. A comparison of the two approaches is given in Table 2.4.
Chapter 2. Background

<table>
<thead>
<tr>
<th>Transformation Methods</th>
<th>Cryptographic Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADVANTAGES</strong></td>
<td><strong>DISADVANTAGES</strong></td>
</tr>
<tr>
<td>efficient</td>
<td>inefficient</td>
</tr>
<tr>
<td>free to use any LP solver</td>
<td>tied to a particular LP solver</td>
</tr>
<tr>
<td>can use floating-point arithmetic</td>
<td>finite field restrictions</td>
</tr>
<tr>
<td>good for large problems</td>
<td>impractical for large problems</td>
</tr>
<tr>
<td><strong>DISADVANTAGES</strong></td>
<td><strong>ADVANTAGES</strong></td>
</tr>
<tr>
<td>heuristic security guarantees</td>
<td>robust security guarantees</td>
</tr>
<tr>
<td>very sensitive to problem structure</td>
<td>robust to problem structure</td>
</tr>
<tr>
<td>unsuitable for small problems</td>
<td>able to be used for small problems</td>
</tr>
</tbody>
</table>

Table 2.4: Summary of the advantages and disadvantages of each type of method.

In reality, practical problems will sometimes be better suited to cryptographic approaches, and sometimes to transformation-based approaches. As a rule of thumb, the transformation methods are well-suited to large problems. Not only do large problems strengthen the security of these methods (as we will see in Chapters 5 and 7), but attempting to use a cryptographic method would simply be too slow. For example, an implementation of Toft’s method by Wibmer *et al.* [130] with 5-6 parties took days to run [130, p.686]. For a $100 \times 100$ matrix (5 parties involved), Catrina and de Hoogh’s method takes approximately 8 seconds *per iteration*. For Vaidya’s revised Simplex method (2 parties involved), for an LP with 25 rows and 35 columns, the cost per iteration would be 23 seconds, and for a larger problem with 800 rows and 1300 columns, one iteration takes 40 minutes [122]. The dominating factor in Vaidya’s method is the scalar product computation, while in Catrina and de Hoogh’s method it is the secure comparison operations.

Mangasarian’s transformation method, by contrast, for an LP with 1000 variables and 600 constraints (with 3 parties involved), had a total computation time of only 11.8 seconds. Similarly, an LP with 1000 variables and 1000 constraints took 14 seconds [87]. These are obviously crude comparisons, but at a broad level they illustrate the differing levels of efficiency of the cryptographic and transformation-based approaches.

For small problems, on the other hand, transformation methods may not be a safe option. As we will see in Chapters 4, 5 and 7, for very small problems (< 5 variables and < 5 constraints), the disguise methods do not offer satisfactory security.

One reason for their faster solution times is that the transformation methods don’t face any problems due to bit-length or secure comparisons. Once a suitable bit-length
is chosen to represent the input values, it remains constant throughout the protocol. Unlike the methods in [79,119], the bit-length will not grow to thousands of bits. Also, no expensive secure comparisons are required as in [25], since Simplex is not being performed in an encrypted domain. The disguised problem can be solved locally, leading to substantial efficiency gains.

In this thesis, we develop methods of both types. We first turn our attention to the transformation methods, and show how to correct a flaw in some existing methods (Chapters 4 and 5). Next, we consider a different type of transformation method that can be used to solve a wide class of practical problems (Chapters 6 and 7). In Chapter 8, we consider a new type of cryptographic method; namely, one based on an interior point method, rather than the Simplex method. Finally, we consider an attack that can be performed on any of the methods, both in the literature and in this thesis, and extend a word of caution about the use of secure linear programming.

Notational Difficulties

Due to the vast array of possible formulations of privacy-preserving LPs, there have been difficulties in formulating problems. For starters, there is no common terminology for such programs, so researchers have often had to define their own terminology to describe partitioning (e.g. row-partitioned, horizontally-partitioned, arbitrary partitioning, etc). Sometimes, the type of partitioning has not been made clear (e.g. [48]). Given that the transformation methods (and possibly some of the cryptographic methods) are restricted to certain types of problems structures, we need a way of defining these problem structures. A lack of precision can spell disaster from a security standpoint, since it is impossible to analyse security unless it is very clear who owns what.

Other subtleties, such as the information contained in the output, haven’t always been looked at as carefully as they should have been. This is partly because there was no systematic notation system requiring the specification of all these features. Given these problems, which largely stem from a lack of standard notation, we propose a new notation system for privacy-preserving LPs in Chapter 3. The new system is a way to concisely represent the restrictions inherent in a method.
Chapter 3

A Classification System for Privacy-Preserving LPs

As we saw in Chapter 2, an exciting new suite of optimisation algorithms allows a group of participants to jointly optimise without revealing their private data. However, these algorithms, particularly those that are transformation-based, introduce security considerations that depend on the way the problem is formulated. For example, some methods apply only to row-partitioned problems, while others only apply when one party owns the objective function and the other owns the constraint set. As well, some methods are sensitive to the types of constraints and variables, and so the standard conversion procedures for regular linear programs can no longer be naively applied. We can sum up these issues by saying that the concept of a canonical form ‘fails’ for privacy-preserving LPs.

Despite this sensitivity to problem form, there is currently no standard way of defining which types of privacy-preserving linear programs a method applies to. If we try to define privacy-preserving linear programs using the same notation as regular linear programs, we quickly see that this does not readily allow the subtleties of each problem to be described. To remedy this, we introduce a simple, flexible classification system in the spirit of Kendall’s queueing notation [73].

3.1 Failure of canonical form

We have long known that there are multiple ways of specifying a linear program, for instance with inequality or equality constraints, but that these are equivalent and that it is easy to transform between the different problem definitions (see Appendix B for details). This is no longer the case in privacy-preserving linear programming, due to security considerations.
Even though two privacy-preserving linear programs might be *mathematically equivalent*, a difference in data partitioning, or the type of constraints and variables, changes the way they must be solved. One cannot convert a row-partitioned LP into a column-partitioned LP, say, because these are fundamentally different problems. Also, although the regular constraint and variable conversions are still *mathematically* valid for privacy-preserving LPs, applying them may destroy the security properties of the method; some examples will be given in Section 3.4.

This is a blow for solution development. In regular linear programming, the beauty is that any LP can be converted into ‘canonical form’ (using the procedures outlined in Appendix B), so that a method designed for canonical form can in fact be applied to any LP. By contrast, the existing transformation methods, and at least one cryptographic method [122], are designed to handle only a *subset* of the possible problem-types.

The different types of linear programming problems in this secure context have just begun to be understood [121]. A few papers have started to define the problem they work on [11, 86, 87, 121, 122], though rarely using common notation. Other papers are vague or imprecise when they describe the problem to which their algorithm applies. A lack of standard notation leads to authors using different terms to describe the same problem, or it results in important features of the problem formulation being overlooked.

To remedy this, we introduce a new seven-factor notation system to precisely and concisely classify the different types of secure linear programs. The system is inspired by Kendall’s notation for classifying queueing systems [73]. In our notation system, a secure linear program is described in shorthand notation by $A/B/C/D/E/F/G$, where:

- **A** = Objective Function Partitioning;
- **B** = Constraint Partitioning
- **C** = RHS Vector Partitioning
- **D** = Allowable Constraint Types
- **E** = Allowable Variable Types
- **F** = Optimal Solution Partitioning
- **G** = Optimal Value Partitioning

We now discuss each of these factors in turn.

### 3.2 Defining the data partitioning

One of the first complications with privacy-preserving linear programs is how to describe the way the private data is partitioned between the parties. No longer can we express
all constraint sets as, say, $Mx = b$, because this doesn’t provide enough detail as to how $M$ and $b$ are shared. Are $M$ and $b$ known to one party only? Is $M$ row-partitioned? Column-partitioned? Instead, we need to express $M$ and $b$ as the sum of components, one held by Alice, and the other by Bob. We write:

$$M = M_1 + M_2,$$
$$b = b_1 + b_2,$$

where $M_1$ and $b_1$ are held by Alice (and must be kept secret from Bob), and $M_2$ and $b_2$ are held by Bob (and must be kept secret from Alice). Even this formulation still doesn’t tell us enough detail about the type of partitioning at work. We can go a step further and try to write out the matrix structures explicitly, as shown in (3.2.1),

$$\begin{align*}
\max f(x) &= c^T x \\
\text{s.t.} \quad \left( \begin{array}{c}
M_1 \\
0
\end{array} \right) + \left( \begin{array}{c}
0 \\
M_2
\end{array} \right) x &\leq b \\
x &\geq 0,
\end{align*}
$$

but how do we say that $c$ is public, or that $b$ is owned entirely by Alice, without resorting to imprecise and possibly confusing descriptions?

In order to adequately describe the partitioning, the new notation will have to precisely specify how the components $c$, $M$ and $b$ are shared. Before we develop the notation system further, we first define the different types of partitioning scenarios.

### 3.2.1 Types of data partitioning

Here we outline the types of data partitioning possible for the constraint matrix, the RHS vector, and the objective function.

#### 3.2.1.1 Constraint Matrix

The constraint matrix $M = (M_1 + M_2)$ can be either:

1. **Public**: $M$ is known in full by both parties. Implicitly, neither $M_1$ nor $M_2$ is hidden from the other party.

2. **Owned by one party**: $M$ is known in full by only one of the parties (which means either $M_1 = 0$ or $M_2 = 0$);

3. **Partitioned**: known in part by each of the parties, such that each party knows only a subset of the total constraint coefficients. In this case the constraint matrix can either be:
Chapter 3. A Classification System for Privacy-Preserving LPs

(a) **Row-Partitioned**: each party contributes entire rows of the constraint matrix. Thus $M_1 + M_2$ is row partitioned, i.e.,

$$M_1 + M_2 = \begin{bmatrix} \overrightarrow{M_1} \\ \overrightarrow{0} \end{bmatrix} + \begin{bmatrix} \overrightarrow{0} \\ \overrightarrow{M_2} \end{bmatrix}.$$  

This case corresponds to having a set of shared variables, but constraints that express private information.

(b) **Column-Partitioned**: each party contributes entire columns of the constraint matrix. Thus $M_1 + M_2$ is column-partitioned, i.e.,

$$M_1 + M_2 = \begin{bmatrix} \overrightarrow{M_1};0 \\ 0;\overrightarrow{M_2} \end{bmatrix}.$$  

Here each party controls a subset of private variables, with collaborative constraints.

(4) **Shared**: each party contributes to all coefficients of $M$. In other words, $M_1$ and $M_2$ have no structurally-known zeros. This is not to say that $M_1$ and $M_2$ cannot contain zeros, but rather that any zeros cannot be in locations which are known to the other party. By definition this case excludes the previous three cases. This case covers additive sharing, used in [79], and secret sharing, used in [25, 119].

Of the above cases, it may seem that case (4), arbitrary sharing, would be the hardest case to handle. It may also be tempting to think that since the shared case (4) is the most general, any solution method for case (4) would also be secure if applied to case (3), as suggested in [121]. These assumptions are false. A method for case (4) cannot be assumed to remain secure for row-partitioning, for example, since the row-partitioned case has a publicly-known structure which is open to exploitation by an adversary. In fact, it can be easier to design privacy-preserving linear programming methods for case (4) than for the other cases, since there is no known structure for an adversary to exploit in $M_1$ and $M_2$. Therefore it is important to highlight that the cases must be treated separately. Similar partitioning options apply for the RHS vector and the objective vector; these are listed below.

### 3.2.1.2 Objective Function Partitioning

The objective function coefficients $c^T = c_1^T + c_2^T$ form a single row vector, which can be either (1) public, (2) owned by a single party, meaning either $c_1^T = 0$ or $c_2^T = 0$, (3) partitioned, or (4) shared. In the case of (3), the only possibility is column-partitioning, since $c_1^T + c_2^T$ is a single row vector. In this case we have:

$$c_1^T + c_2^T = \begin{bmatrix} \overrightarrow{c_1^T};0 \\ 0;\overrightarrow{c_2^T} \end{bmatrix},$$

and the interpretation might be that each party controls a subset of private variables.
### 3.2.1.3 RHS Vector $b$ Partitioning

It may seem natural to assume that the RHS vector $b = b_1 + b_2$ will be partitioned in the same manner as $M$. For example, if $M$ is public, then $b$ would be public also. However, this assumption does not allow enough flexibility to cover all cases — for example, what if $M$ is public, but $b$ is shared? An example of this case would be a resource allocation problem where the parties make their resource requirements for the different products publicly available, but keep their amount of each resource (e.g., raw materials, labour) private. In addition, it is sometimes impossible for $b$ to be partitioned in the same manner as $M_1 + M_2$. For example, if $M_1$ and $M_2$ are column partitioned, then it is *impossible* for the RHS vector to be column-partitioned also, since it is a single column vector. Hence, in order to allow enough flexibility when specifying a linear program, and to avoid impossible partitionings, we need a separate factor to specify how $b$ is shared. We note that a previous description of the types of sharing, [121], did not consider $b$ explicitly.

The RHS vector $b = b_1 + b_2$ can be any of cases (1)–(4), but in the case of (3), the only possibility is row-partitioning, since $b_1 + b_2$ is a single column vector.

We note that although our notation system allows different sharing options to be selected for $M$, $b$ and $c^T$, this flexibility must be used with care. If we consider all combinations of sharings of $M$, $b$ and $c^T$, of which there are 80 ($5 \times 4 \times 4$), we see that there are some nonsensical combinations. For example, a row-partitioned $M$ and $b$ may not make sense with a column-partitioned $c^T$, even though it is possible to specify such a program using our notation system. Thus, we wish to stress that not every combination of factors will make sense in practice.

### 3.2.2 Notation for data-partitioning

We recommend classifying privacy-preserving linear programs using a notation similar to Kendall’s queueing notation. In total, we will have seven factors. The first three factors will spell out exactly how $c$, $M$, and $b$ are partitioned. That is, we will say that an LP is an A/B/C linear program, where $A$, $B$ and $C$ represent codes that correspond to the partitioning-definitions introduced previously, and describe how $c$, $M$ and $b$ are distributed, respectively. These codes are listed in Table 3.1.

For instance, the classification P/R/R means that the objective function is public, and $M$ and $b$ are row-partitioned. When the objective function is owned by one party, say, Alice, we use the notation O(A) to stand for ‘owned by Alice’. Equivalently, the objective could be O(B) (‘owned by Bob’), or O(C) (‘owned by Party C’), etc., depending on how many parties are involved in the computation.
However, defining the partitioning alone is not enough. The transformation methods are also extremely sensitive to the types of constraints (inequality, equality) and variables (non-negative, free) present in the LP. Also, both cryptographic methods and transformation methods may place restrictions on which parties can receive the output of the LP. We consider these issues in the following section.

### 3.3 Beyond data-partitioning

Most transformation method descriptions carefully describe the partitioning scenario for which they are designed, and stop there. We argue that detailing the partitioning alone is not enough — the existing transformation methods are sensitive to more than just data partitioning: they are also sensitive to the types of constraints (inequality, equality) and variables (non-negative, free). Tying this back to the failure of the canonical form idea, we can say that the usual constraint and variable conversion procedures ‘fail’ for privacy-preserving LPs, in the sense that they cannot be naively applied. To the best of our knowledge, constraint conversion procedures do not affect the cryptographic methods. However, both cryptographic and transformation methods are sensitive to the form of the output of the LP. Some examples of these additional restrictions (together with the aforementioned data-partitioning restrictions) are provided in Table 3.3.

The sensitivity to these additional features is not something that has been emphasised in any of the existing papers. Rather, the constraint and variable restrictions have been

<table>
<thead>
<tr>
<th>Objective Function Partitioning ($c$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P  Public</td>
</tr>
<tr>
<td>O($X$) Owned by party ‘$X$’, e.g., O(A)</td>
</tr>
<tr>
<td>C  Column-partitioned</td>
</tr>
<tr>
<td>S  Shared (no known zeros)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint Partitioning ($M$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P  Public</td>
</tr>
<tr>
<td>O($X$) Owned by party ‘$X$’, e.g., O(A)</td>
</tr>
<tr>
<td>R  Row-partitioned</td>
</tr>
<tr>
<td>C  Column-partitioned</td>
</tr>
<tr>
<td>S  Shared (no known zeros)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RHS Vector Partitioning ($b$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P  Public</td>
</tr>
<tr>
<td>O($X$) Owned by party ‘$X$’, e.g., O(A)</td>
</tr>
<tr>
<td>R  Row-partitioned</td>
</tr>
<tr>
<td>S  Shared (no known zeros)</td>
</tr>
</tbody>
</table>

Table 3.1: Classification codes for data partitioning
### Method Restrictions

<table>
<thead>
<tr>
<th>Method</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mangasarian [87]</td>
<td>Objective function is publicly known.</td>
</tr>
<tr>
<td></td>
<td>Constraint matrix and RHS vector are row-partitioned.</td>
</tr>
<tr>
<td></td>
<td>All constraints are equalities (cannot be the result of slack or surplus variables; must be equalities from the beginning).</td>
</tr>
<tr>
<td>Mangasarian [86]</td>
<td>Objective function is column-partitioned across the parties.</td>
</tr>
<tr>
<td></td>
<td>Constraint matrix is column-partitioned across the parties.</td>
</tr>
<tr>
<td></td>
<td>RHS vector is publicly known.</td>
</tr>
<tr>
<td></td>
<td>All constraints are inequalities to begin with.</td>
</tr>
<tr>
<td></td>
<td>All variables must be free.</td>
</tr>
<tr>
<td></td>
<td>Bob owns the constraint matrix and RHS vector.</td>
</tr>
<tr>
<td></td>
<td>All constraints must be inequalities.</td>
</tr>
<tr>
<td></td>
<td>Only Bob is allowed to receive (x^*).</td>
</tr>
<tr>
<td></td>
<td>Both parties are allowed to know (f^*).</td>
</tr>
<tr>
<td>Vaidya [122]</td>
<td>Alice owns the objective function.</td>
</tr>
<tr>
<td></td>
<td>Bob owns the constraint matrix.</td>
</tr>
<tr>
<td></td>
<td>The RHS vector is publicly known.</td>
</tr>
<tr>
<td></td>
<td>Bob alone knows (x^*).</td>
</tr>
<tr>
<td></td>
<td>No one receives (f^*).</td>
</tr>
</tbody>
</table>

| Table 3.2: Restrictions inherent in existing methods |

implicit in the program listing. This can lead to problems when the algorithms are executed in practice, since users may unintentionally violate one of the assumptions of the method, thereby voiding the method’s security guarantees. Therefore our notation system needs to precisely convey these restrictions.

### 3.4 Failure of constraint and variable conversions

Let us consider two examples to show how applying the normal conversions that work for regular LPs can inadvertently affect security.

#### Example 1: Deviating from ‘only inequality constraints’ restriction

Assume we have the following privacy-preserving linear program:

\[
\begin{align*}
\text{max } & \quad f(x) = c^T x \\
\text{s.t. } & \quad (M_1 + M_2)x \leq b \\
& \quad x \geq 0, 
\end{align*}
\]

where \(M = (M_1 + M_2)\) and \(b = b_1 + b_2\) are row-partitioned, and \(c\) is public. Suppose we attempt to solve it using Mangasarian’s method [87], since we know this method is suitable for row-partitioned LPs with public objective functions. However, we notice
that the method requires the LP to have equality constraints. No problem, we say, I’ll just convert my inequality constraints to equality constraints. However, this will result in a critical violation of security.

If we convert all constraints to equalities by adding slack variables, we are implicitly changing $M$ to $\overline{M} = [M; I]$. Attempting to disguise these equality constraints by pre-multiplying by a random matrix $P$ (as in [87]) will give \( \hat{M} = \begin{bmatrix} P(M_1 + M_2) \end{bmatrix} \), exposing $P$ in full! Thus the disguise is useless.

**Example 2: Deviating from ‘free variables only’ restriction**

Again assume we have the linear programming problem shown in (3.4.1), except this time $M = (M_1 + M_2)$ and $c = c_1 + c_2$ are column-partitioned. Suppose we attempt to solve it using Mangasarian’s method [86], since we know this method is suitable for LPs in which the objective function and constraint matrix are column-partitioned. However, we notice that our program has non-negativity constraints, whereas the problem considered by Mangasarian’s method does not. No problem, we say, I’ll just apply the method anyway. However, the presence of the non-negativity constraints will expose the random matrix used in the transformation, rendering the disguise useless. Let’s see how.

Mangasarian’s [86] disguise is based on the following equivalence relationship. The linear program

$$
\begin{align*}
\text{max} & \quad f(x) = c^T x \\
\text{s.t.} & \quad Mx \leq b,
\end{align*}
$$

has the same solution as the following ‘transformed’ linear program:

$$
\begin{align*}
\text{max} & \quad f(x) = c^T B^T u \\
\text{s.t.} & \quad MB^T u \leq b.
\end{align*}
$$

where $u = B(B^T B)^{-1} x$ and $B \in \mathbb{R}^{k \times n}$ is a random matrix. Security stems from the fact that no party knows all of matrix $B$. If we append non-negativity constraints $x \geq 0$ to (3.4.2), we will accordingly have to append the equivalent constraint $B^T u \geq 0$ to the transformed LP (3.4.3). The transformation matrix will then be public knowledge! The private constraints $M$ can then be easily recovered using $M = (MB^T)(B[B^T B]^{-1})$, and all security is lost.

As these examples demonstrate, it is critical that any constraint and variable restrictions inherent in a method are adhered to by users. New terminology is therefore needed to more accurately specify the flexibility of a method. In the following sections we outline our proposed classifications for the constraint types and variable types.
3.4.1 Constraint type classifications

We have seen that one cannot naively convert inequalities to equalities in the transformation-based methods, as this can cause the disguising matrix to be revealed in full following the transformation. Hence we introduce the descriptor ‘pure equality constraints’ to mean that the linear program has not had any slack variables added in order to make the constraints equalities — all constraints must be equalities from the beginning.

We know we cannot safely convert inequalities to equalities. Whether you can safely convert equalities to inequalities is an open problem. Expressing an equality as a pair of inequalities gives rise to repeated rows in $M$, except for uniform sign differences. The security implications of this are unclear, but repeated rows introduce a risk because known patterns in $M$ may be exploitable by an adversary. As such we have taken a precautionary approach and included separate classifications for each constraint type.

In our notation system, the labels given to the factor ‘constraint type’ are:

1. **Pure Equalities (\(=\))**: All constraints are pure equalities. Inequality constraints which have been converted to equalities by adding slacks are not permitted.

2. **Pure Inequalities (\(\leq\))**: All constraints are pure inequalities. Equality constraints which have been converted into two inequality constraints are not permitted.

3. **Mixed (\(\leq, =\))**: The constraints are a mixture of inequalities and equalities. Though as before, no conversion is allowed.

When we have mixed constraints, there are two constraint matrices involved. We assume the same type of sharing applies to both matrices, as having different sharing for each does not make sense in practice. With the inequality constraints stored in $M$ and $b$, and the equality constraints stored in $N$ and $d$, we express the full constraint set as:

$$Mx \leq b; \quad Nx = d.$$  

3.4.2 Variable type classifications

The transformation methods are also sensitive to the form of the variables. For instance, non-negative variables can present a problem in the transformation methods, unless a method has been specifically designed to handle non-negativity. Including non-negativity constraints can reveal a disguising matrix in full, as shown in Example 2. Alternatively, non-negativity constraints can be disrupted by the random matrices used in the disguise (which can lead to infeasible solutions being returned), unless these matrices are carefully chosen [11]; this will be discussed in Chapter 4.

For methods which can handle non-negative variables, it is an open problem as to whether they can be used for programs with free variables. Converting free variables
to be the difference of two non-negative variables will mean that the constraint matrix will have repeated columns of coefficients (except for a uniform change in signs). As with repeated rows, this may compromise security. Thus three classification levels are needed to distinguish between methods which can handle (i) free variables only, (ii) non-negative variables only, and (iii) a mixture of the two.

In our notation system, the labels given to this factor are:

1. **Free Variables Only (F):** All variables are free.
2. **Non-negative Variables Only (N):** All variables are non-negative. Free variables which have been re-written as the difference of two non-negative variables are not permitted.
3. **Mixed (N, F):** The variables can include both free and non-negative variables.

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq$</td>
<td>Pure Inequality (only inequality constraints)</td>
</tr>
<tr>
<td>$=$</td>
<td>Pure Equality (only equality constraints)</td>
</tr>
<tr>
<td>$\leq, =$</td>
<td>Mixed (both inequality and equality constraints)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Non-negative variables only</td>
</tr>
<tr>
<td>F</td>
<td>Free variables only</td>
</tr>
<tr>
<td>N, F</td>
<td>Mixture of free and non-negative variables</td>
</tr>
</tbody>
</table>

Table 3.3: Classification codes for constraint and variable types

### 3.5 Output considerations

Regular linear programs have two associated outputs: an optimal solution $x^*$ and an optimal objective value $f^*$. In regular linear programs, $f^*$ is considered “optional output”, since it can be easily worked out by substituting $x^*$ into the objective function. Such a calculation is simple because the objective function is completely known. In privacy-preserving linear programs, however, we must consider both outputs separately, since it may no longer be a simple matter to substitute $x^*$ into a possibly private objective function.

Unlike regular linear programming, great care must be taken to ensure that the output of a privacy-preserving linear program does not destroy security. As with any type of secure computation, the output contains information, and making the output known to all parties could jeopardise security [84]. For example, knowledge of $x^*$ can be used
to attack a transformation method [11]. Although public output is usually the most
desirable case, sometimes it may only be safe for one party to know \( x^* \) [122], or for
\( x^* \) to be split into shares [119]. Similar considerations apply for the optimal value. In
short, specifying who may know the output is as important as the body of the method
itself. Thus we advocate that the type of output partitioning be part of the program
specification.

In many optimisation situations, not all of the outputs are needed. For example, in the
grape delivery example (see Chapter 2, Example 2.2) it would be pointless to reveal
\( x^* \) in addition to \( f^* \), since \( x^* \) would give away all the supply and demand information,
which is exactly what we were trying to protect! The contractor could simply add up
the appropriate values in \( x^* \) to infer the demand at each winery and the (minimum)
available supply at each vineyard. The value of \( f^* \) is all the winemaking company needs
to know in order to establish the viability of the deal; thus that is all the protocol should
return. Revealing \( f^* \) does not jeopardise the private data of the contractor, since there
are infinitely many cost vectors \( c \) that satisfy the equation \( c^T x^* = f^* \).

Ideally, the classification system should include any output restrictions, to force devel-
opers of a method to make any restrictions regarding who may know the output explicit
to users.

### 3.5.1 Specifying the output partitioning

In terms of our notation system, bearing in mind the output’s potential to affect security,
we need to introduce classifications for how the output is partitioned. Below we list the
partitioning options for \( x^* \) and \( f^* \), using the definitions introduced in Section 3.2.1.

#### 3.5.1.1 Partitioning of \( x^* \)

The optimal solution \( x^* \) can be (i) public, (ii) known by a single party, (iii) partitioned,
where the only option is row-partitioning since \( x^* \) is a single column vector, (iv) shared
or (v) known to no one. A public optimal solution is usually the most desirable option
for a stand-alone program from a usefulness perspective, but it is not always possible
due to security concerns. If the secure linear program is to form part of a larger protocol,
then shared output may be desirable. We introduce an extra category for the case when
no one receives the optimal solution. This is because there could conceivably be times
when the parties simply wish to know \( f^* \) only: e.g., when they need to assess which
supplier will be the cheapest.

#### 3.5.1.2 Partitioning of \( f^* \)

There are several options for the optimal value \( f^* = c^T x^* \). As \( f^* \) is a scalar, row and
column partitioning are not possible. However an extra category is required for the case
when no one is told \( f^* \), as in Vaidya’s revised Simplex method [122]. So in summary the optimal value can be (i) public, (ii) known by a single party, (iii) shared, or (iv) known to no one.

<table>
<thead>
<tr>
<th>\textbf{Optimal Solution Partitioning ((x^*))}</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
<tr>
<td>O(X)</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>\emptyset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\textbf{Objective Value Partitioning ((f^*))}</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
<tr>
<td>O(X)</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>\emptyset</td>
</tr>
</tbody>
</table>

Table 3.4: Classification codes for the output

The corresponding classification codes are provided in Table 3.4. Having now considered all seven factors separately, we are ready to bring them together into one notation system.

### 3.6 Full notation system

We propose a notation system to precisely and concisely classify the different types of secure linear programs. Moreover, the idea can be extended to other types of secure optimisation problems, such as quadratic programming and non-linear programming, although we leave this to be developed as needed. Inspired by Kendall’s notation for classifying queueing systems [73], seven factors are used to uniquely classify an LP. A secure linear program is described in shorthand notation by A/B/C/D/E/F/G, where:

- A = Objective Function Partitioning (c)
- B = Constraint Partitioning (M)
- C = RHS Vector Partitioning (b)
- D = Allowable Constraint Types
- E = Allowable Variable Types
- F = Optimal Solution Partitioning (x*)
- G = Optimal Value Partitioning (f*).
The possible classes for each factor are listed in full in Table 3.7, along with the classification codes. Some factors are optional, depending on what the notation is being used for. For example, one can specify codes for the first 3 factors only, if one wants to define the partitioning only. Alternatively, if one wants to define which problems a transformation method can be reliably applied to, all seven factors should be included. When a factor is unspecified, or unnecessary, we suggest replacing it with a '.*'.

### 3.6.1 Demonstration

As an example, linear programs with a public objective function, row-partitioned constraints and RHS, pure inequality constraints, solely non-negative variables, a public optimal solution and a public objective value, would be referred to as P/R/R/\(\leq\)/N/P/P linear programs, using Table 3.7. Several explicit demonstrations of how to apply our notation system are given in Table 3.5. In particular, we apply our notation system to some of the secure linear programs that have been considered in existing papers, giving the program specification and corresponding classification.

We classify the linear programs covered by the existing transformation-based and cryptographic methods in Table 3.6. Additional early papers which are ambiguous are not included, e.g. [48]. A few words are in order for the cryptographic methods. The majority of these papers assume that all coefficients of \(c\), \(M\) and \(b\) are either secret-shared [25, 119] or additively split [79] \textit{a priori}. They don’t discuss how to deal with other types of partitioning, such as one party owning \(c\) and another owning \(M\) and \(b\). Hence we have used the ‘shared’ partitioning option for the majority of these methods, although we acknowledge that it may be possible to use them with other partitioning scenarios. In addition, these methods either show the original linear program as having all inequality constraints [25, 119, 122], or all equality constraints [79], but do not explicitly state whether or not other types of constraints are permitted. Similarly, the linear program is shown as having all non-negative variables, but no mention is made of whether free variables are allowed. Following our precautionary approach, we have only listed either ‘pure inequalities’ (\(\leq\)) or ‘pure equalities’ (\(=\)) and ‘non-negative variables’ (N) for all of the cryptographic methods.

### 3.6.2 Extensions

Further factors could be added to our notation system. For example, a factor could be added to specify the number of parties. The previous classifications, such as the type of partitioning, are still valid for the multiparty case.
Chapter 3. A Classification System for Privacy-Preserving LPs

Linear Program Description Notation

\[
\begin{align*}
\text{max } & \ f = c^T x \\
\text{s.t. } & \ Mx \leq b \\
& \ x \geq 0
\end{align*}
\]

\(c\) is owned by Alice; \(M\) and \(b\) are owned by Bob. All constraints are pure inequalities and all variables are non-negative. The optimal solution \(x^*\) is known by Bob only, while the optimal value \(f^*\) is public [11].

\[
\begin{align*}
\text{max } & \ f = c^T x \\
\text{s.t. } & \ \begin{bmatrix} M_1 \\ M_2 \end{bmatrix} x = \begin{bmatrix} b_1 \\ \vdots \\ b_2 \end{bmatrix} \\
& \ x \geq 0
\end{align*}
\]

\(c\) is public; \(M\) and \(b\) are row-partitioned; all constraints are pure equalities and all variables are non-negative. The optimal solution \(x^*\) is public, and so is \(f^*\) since \(c\) is public [87].

\[
\begin{align*}
\text{max } & \ f = [c_1; c_2]^T x \\
\text{s.t. } & \ \begin{bmatrix} M_1; M_2 \end{bmatrix} x \leq b
\end{align*}
\]

\(c\) and \(M\) are column-partitioned; \(b\) is public; all constraints are pure inequalities and all variables are free. The optimal solution \(x^*\) is row-partitioned, but each party can make their component public if desired. In either case \(f^*\) is public [86].

Table 3.5: Demonstration of our notation system

<table>
<thead>
<tr>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(A)/O(B)/O(B)/≤/N/O(B)/P</td>
<td>Linear Program</td>
</tr>
<tr>
<td>P/R/R/=/N/P/P</td>
<td>Linear Program</td>
</tr>
<tr>
<td>C/C/P/≤/F/(R or P)/P</td>
<td>Linear Program</td>
</tr>
</tbody>
</table>

Cryptographic Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Li &amp; Atallah [79]</td>
<td>S/S/S/=/N/P/Ø</td>
</tr>
<tr>
<td>Toft [119]</td>
<td>S/S/S/≤/N/S/S</td>
</tr>
<tr>
<td>Vaidya [122]</td>
<td>O(A)/O(B)/O(B)/≤/N/O(B)/Ø</td>
</tr>
</tbody>
</table>

Transformation Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bednarz et al. [11]</td>
<td>O(A)/O(B)/O(B)/≤/N/O(B)/P</td>
</tr>
<tr>
<td>Mangasarian [87]</td>
<td>P/R/R/=/N/P/P</td>
</tr>
<tr>
<td>Mangasarian [86]</td>
<td>C/C/P/≤/F/(R or P)/P</td>
</tr>
</tbody>
</table>

Table 3.6: Summary of the secure linear programs covered by the existing methods
3.7 Conclusion

We have presented a notation system to concisely classify a privacy-preserving linear program. We hope that, like Kendall’s queueing notation, our system will provide a common language for researchers in this field, and avoid ambiguities or vagaries in the specification of these programs. As well, we hope this notation can be extended as new types of secure optimisation problems, such as quadratic programming and non-linear programming, begin to be explored. We will use this notation system throughout the rest of this thesis.
### Objective Function Partitioning (c)

<table>
<thead>
<tr>
<th>P</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(X)</td>
<td>Owned by party ‘X’ e.g. O(A)</td>
</tr>
<tr>
<td>C</td>
<td>Column-partitioned</td>
</tr>
<tr>
<td>S</td>
<td>Shared (no known zeros)</td>
</tr>
</tbody>
</table>

### Constraint Partitioning (M)

<table>
<thead>
<tr>
<th>P</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(X)</td>
<td>Owned by party ‘X’ e.g. O(A)</td>
</tr>
<tr>
<td>R</td>
<td>Row-partitioned</td>
</tr>
<tr>
<td>C</td>
<td>Column-partitioned</td>
</tr>
<tr>
<td>S</td>
<td>Shared (no known zeros)</td>
</tr>
</tbody>
</table>

### RHS Vector Partitioning (b)

<table>
<thead>
<tr>
<th>P</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(X)</td>
<td>Owned by party ‘X’ e.g. O(A)</td>
</tr>
<tr>
<td>R</td>
<td>Row-partitioned</td>
</tr>
<tr>
<td>S</td>
<td>Shared (no known zeros)</td>
</tr>
</tbody>
</table>

### Constraint Type

- \( \leq \): Pure Inequality (only inequality constraints)
- \( = \): Pure Equality (only equality constraints)
- \( \leq,= \): Mixed (both inequality and equality constraints)

### Variable Type

- N: Non-negative variables only
- F: Free variables only
- N, F: Mixture of free and non-negative variables

### Optimal Solution Partitioning (x*)

<table>
<thead>
<tr>
<th>P</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(X)</td>
<td>Owned by party ‘X’ e.g. O(A)</td>
</tr>
<tr>
<td>R</td>
<td>Row-partitioned</td>
</tr>
<tr>
<td>S</td>
<td>Shared (no known zeros)</td>
</tr>
<tr>
<td>0</td>
<td>Null (No one is told x*)</td>
</tr>
</tbody>
</table>

### Objective Value Partitioning (f*)

<table>
<thead>
<tr>
<th>P</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(X)</td>
<td>Owned by party ‘X’ e.g. O(B)</td>
</tr>
<tr>
<td>S</td>
<td>Shared (no known zeros)</td>
</tr>
<tr>
<td>0</td>
<td>Null (No one is told f*)</td>
</tr>
</tbody>
</table>

| Table 3.7: Full list of classification codes |
Chapter 4

Transformation Methods: Hiccups and Solutions

An appealing group of approaches for privacy-preserving linear programming employ a ‘disguising’ transformation to allow one party to perform the joint optimisation without seeing the secret data of the other parties. These approaches are very appealing from the point of view of simplicity, efficiency, and the freedom to use any solver; but we show here that two of the existing transformations have a critical flaw.

We then show how to correct these transformations for the case when Alice owns the objective function and Bob owns the constraints. However, the fix introduces security concerns which must be carefully analysed. The core of this criticism and analysis was published in Bednarz et al. [11].

4.1 Background

As outlined in Chapter 2, solutions for privacy-preserving linear programming fall into one of two categories: ‘transformation approaches’, which disguise the LP using random matrices, or ‘cryptographic approaches’, which use secret sharing or homomorphic encryption to implement Simplex. Though the formal cryptographic approaches provide very good security, there are several appealing aspects of the transformation-based approaches. First, they are generally far more efficient than their cryptographic counterparts; they avoid much of the computational and communication complexity of the cryptographic techniques, because the information-hiding protocols are only needed once at the start of the algorithm, rather than at each step. Second, they do not tie the user to a specific optimisation algorithm — once the LP is disguised, users are free to solve it using Simplex, Interior Point, or any other algorithm of their choice.

However, transformation methods suffer from a major problem: their security is difficult to prove. Without a quantifiable security level, it is difficult to know how much faith
to place in their ‘heuristic’ approach to security. In this chapter we take the first step to remedying this, by presenting the first transformation method with a quantifiable security level. Our goal is to see if we can correct the problems in the existing transformation methods, and develop a secure, easy-to-use alternative to the cryptographic methods, whose security level can be quantified.

### 4.1.1 Transformation methods: the general idea

The idea behind the transformation methods is to convert the linear program into an *equivalent* system in which the constraints have been *disguised* by multiplying them by random matrices. One party solves the disguised problem, and then the solution is converted back to the original domain. For this to work, the methods rely on the fact that the transformed LP is equivalent to the original LP. In other words, the solution to the transformed LP must be a transformed version of the solution to the original LP.

Assume the original LP — which we will refer to as Program A — is as shown below:

#### Program A

\[
\begin{align*}
\text{max} & \quad f(x) = c^T x \\
\text{s.t.} & \quad Mx \leq b \\
& \quad x \geq 0,
\end{align*}
\]

where each of \(M\), \(b\) and \(c\) can be partitioned in any of the ways outlined in Chapter 3. We do not specify a particular partitioning yet, since it will cloud the presentation of the disguising procedure and does not affect the mathematical validity of the disguise.

Each component of Program A is ‘disguised’ via multiplication by random matrices. The disguised components are denoted by \(\hat{c}\), \(\hat{M}\) and \(\hat{b}\). The exact form of the disguise is dependent on the method, but, for example, \(M\) might become \(\hat{M} = PMQ\), where \(P\) and \(Q\) are random matrices. The important thing is that it must not be possible for Alice to determine \(M\) from knowledge of \(\hat{M}\). Bob generates the random matrices, and then Alice and Bob jointly engage in secure multiplication protocols such that only Alice receives the coefficients of the disguised program, Program B. In the new space, \(x\) is transformed to \(\hat{x}\).

#### Program B

\[
\begin{align*}
\text{max} & \quad \hat{f}(\hat{x}) = \hat{c}^T \hat{x} \\
\text{s.t.} & \quad \hat{M}\hat{x} \leq \hat{b} \\
& \quad \hat{x} \geq 0
\end{align*}
\]

Alice solves Program B locally, using whichever linear programming algorithm she desires. She finds \(\hat{x}^*\), the solution to the transformed problem, and sends it to Bob. Finally, Bob converts the solution back to the original coordinates using his knowledge of the random matrices.
It is important to point out that the transformation methods are not ‘cryptography-free’ — in the majority of methods, cryptographic protocols are needed in order to apply the disguise. Usually, secure matrix product protocols are required. Cryptographic protocols may also be needed to convert \( \hat{x} \) back to the original coordinates. The key is that the cryptographic protocols are only needed at the start and end points of the algorithm; thus the overhead is fixed, rather than a function of the number of steps in the solution, as is the case in the cryptographic methods.

Two of the first transformation-based methods were proposed by Du [48] and Vaidya [121]. However, we show there is a flaw in these techniques. The problem may mean that the answer returned by these methods may actually be infeasible.

Let’s examine in more detail the method of Du, and see why what appears to be a very simple and elegant technique falls down in practice.

### 4.2 Du’s transformation method

The first transformation method was proposed by Du [48]. Central to Du’s method, as with all transformation methods, is the assertion that the transformed LP is equivalent to the original LP. Here we consider the critical elements and see why this equivalence fails.

The original LP, Program A (4.1.1), is disguised via two random matrices: an \( m \times m \) matrix \( P \) and an \( n \times n \) matrix \( Q \). These matrices are populated with random elements, with the restrictions that the matrices are invertible, and the elements of \( P \) and \( Q^{-1} \) are all strictly positive. Bob generates the random matrices, and then Alice and Bob jointly engage in secure multiplication protocols such that only Alice receives the coefficients of the disguised program, Program B:

\[
\begin{align*}
\text{max} & \quad \hat{f}(\hat{x}) = \hat{c}^T \hat{x} \\
\text{s.t.} & \quad \hat{M} \hat{x} \leq \hat{b} \\
& \quad \hat{x} \geq 0,
\end{align*}
\]

(4.2.1)

where \( \hat{M} = PMQ \), \( \hat{b} = Pb \), \( \hat{c}^T = c^T Q \), and \( \hat{x} = Q^{-1} x \).

To make the disguising procedure more transparent for the reader, we also write Program B out in full so that its underlying structure is visible.
Chapter 4. Transformation Methods: Hiccups and Solutions

Program B (transparent version)

\[
\begin{align*}
\max \ & \ h'(Q^{-1}x) = c^TQQ^{-1}x \\
\text{s.t.} \ & \ PMQQ^{-1}x \leq Pb \\
& \ Q^{-1}x \geq 0.
\end{align*}
\] (4.2.2)

Alice must not be told \( P \) or \( Q \), and Bob is not permitted to know Program B. Otherwise, they could derive the other’s private data. Alice solves Program B and obtains the optimal solution \( \hat{x}^* \). She sends this to Bob, who transforms it back to the original variables by premultiplying by \( Q \) to obtain: \( x^* = Q\hat{x}^* \), which Du claims is the optimal solution to the original program, Program A.

On the surface, the transformation process from Program A to Program B appears to be valid — after all, \( QQ^{-1} \) is simply the identity matrix. Indeed, for all \( x \),

\[
f(x) = c^Tx = c^TQQ^{-1}x = \hat{c}^T\hat{x} = \hat{f}(\hat{x}).
\] (4.2.3)

Also, since \( P \) and \( Q^{-1} \) contain strictly positive elements, they should not disturb the inequality constraints or the non-negativity constraints (see [48, Theorem 4.3.1]). We now demonstrate why this equivalence relationship, which on the surface looks to be sound, does not hold.

### 4.2.1 Why Programs A & B are not equivalent

Du’s method hinges on the supposition that Program B is an equivalent formulation to Program A, and will therefore have the same solution. However, this supposition is false.

For Du’s method to hold, Programs A and B need only have the same feasible region (if we plot them both in terms of \( x \)) since we have shown in (4.2.3) that the objective functions are equivalent.

The problem with Du’s method is that by choosing \( P \) and \( Q^{-1} \) to be positive, we force \( P^{-1} \) and \( Q \) to contain some negative elements (see [70, Theorem 1.1]). These play havoc with inequality constraints: for a simple example consider what happens to the constraint \( x \geq 0 \) when we multiply by \(-1\); it becomes \( x \leq 0 \). The combination of negative matrix elements and the inequality constraints mean that the constraints forming the two feasible regions are not equivalent. More precisely, if the constraint \( \hat{x} \geq 0 \) is satisfied in Program B, this does not guarantee that the corresponding constraint, \( x \geq 0 \) will be satisfied in Program A, and likewise \( \hat{M}\hat{x} \leq \hat{b} \) does not guarantee that \( Mx \leq b \).

Let us consider a brief example to bring home the point. We consider the following optimisation problem (Program A):
max $f(x) = 5x_1 + x_2$
\[\begin{align*}
\text{s.t.} \quad & 2x_1 + 3x_2 \leq 12 \\
& 2x_1 + x_2 \leq 8 \\
& x_1, x_2 \geq 0,
\end{align*}\] (4.2.4)

and transform it using matrices $P = \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$ and $Q^{-1} = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$.

Obviously, these are not random, but the problem is generic and will occur with almost all choices of $P$ and $Q^{-1}$. The transformed problem (Program B) is then

max $\hat{f}(\hat{x}) = -\hat{x}_1 + 3\hat{x}_2$
\[\begin{align*}
\text{s.t.} \quad & \frac{4}{3}\hat{x}_1 + \frac{10}{3}\hat{x}_2 \leq 36 \\
& \frac{20}{3}\hat{x}_1 + \frac{2}{3}\hat{x}_2 \leq 68 \\
& \hat{x}_1, \hat{x}_2 \geq 0.
\end{align*}\] (4.2.5)

The optimal solution for the original LP, (4.2.4), is $x^* = (4, 0)$, with objective function $f_{\text{max}} = 20$. The optimal solution to the transformed problem (4.2.5) is $\hat{x}^* = (0, 10.8)$ with $\hat{f}_{\text{max}} = 32.4$. When we transform back into the original coordinates this solution corresponds to $x^* = (7.2, -3.6)$ which is infeasible since $x_2$ is negative.

How can the transformed problem result in an infeasible solution? Consider the associated feasible regions shown in Figure 4.1. The feasible region of Program A, (4.2.4), is shown in black, while the feasible region of Program B, (4.2.5), represented in the original coordinates, is shown in grey. Notice that it is larger than the original feasible region, and it even extends outside the non-negative quadrant (where feasible solutions cannot exist). Hence, it is possible to find a solution to (4.2.5) that is infeasible for (4.2.4).

By choosing $P$ and $Q^{-1}$ to be strictly positive, we ensure that any feasible solution to Program A also satisfies the constraints of Program B, and hence the feasible set of A is a subset of that of B. The problem with Du’s transformation is that the converse is not true, since the inverses of $P$ and $Q^{-1}$ must contain negative elements. Indeed, for the example just considered, $Q = \begin{bmatrix} -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} \end{bmatrix}$. Hence we cannot correct the problem by instead requiring $P^{-1}$ and $Q$ to be positive, as this will break the equivalence in the other direction.

### 4.3 Vaidya’s transformation method

Vaidya [121] recognised that Du’s method fails in some cases. Vaidya developed a new method for a specific scenario in which one party holds the objective function, and
the other party holds all the constraints. For this case, he proposed a transformation approach based on Du’s method, with the difference being that matrix $P$ is not used, so the disguise is imparted only by matrix $Q$. Vaidya avoids the troubles caused by $P$’s impact on the inequality constraints, but his approach does not resolve the problems with the non-negativity constraints. For instance, let’s reuse the example program (4.2.4), and this time apply only the associated $Q^{-1}$ to arrive at a new Program B:

$$\begin{align*}
\text{max} & \quad \hat{f}(\hat{x}) = -\frac{4}{3}\hat{x}_1 + 3\hat{x}_2 \\
\text{s.t.} & \quad \frac{4}{3}\hat{x}_1 + \frac{1}{3}\hat{x}_2 \leq 12 \\
& \quad \hat{x}_2 \leq 8 \\
& \quad \hat{x}_1, \hat{x}_2 \geq 0,
\end{align*}$$

(4.3.1)

where $\hat{x} = Q^{-1}x$, $\hat{M} = MQ$, $\hat{c}^T = c^TQ$.

The transformed problem (4.3.1) has solution $\hat{x}^* = (0, 8)$ with $\hat{f}_{\text{max}} = 24$. When we transform back into the original coordinates, we obtain $x^* = (5\frac{1}{3}, -2\frac{2}{3})$, which is once again infeasible since $x_2$ is negative.

Comparing the feasible regions of Program A and Program B in Figure 4.2, we see that the feasible region of Program B still extends outside the non-negative quadrant. However, within the non-negative quadrant the feasible regions now agree, as a result of removing $P$ from the transformation.
One may wonder if the disruption to the non-negativity constraints could be avoided simply by moving them into the main constraint matrix, $M$. In other words, why not rewrite program (4.2.4) as follows:

\[
\begin{align*}
\max & \quad f(x) = 5x_1 + x_2 \\
\text{s.t.} & \quad 2x_1 + 3x_2 \leq 12 \\
& \quad 2x_1 + x_2 \leq 8 \\
& \quad -x_1 \leq 0 \\
& \quad -x_2 \leq 0
\end{align*}
\] (4.3.2)

That way, the non-negativity constraints won’t be disrupted. However, we now have a security problem in that the last $n$ rows of $\hat{M} = MQ$ will expose $-Q$ (and hence $Q$) in full, as shown:

\[
\hat{M} = MQ = \begin{bmatrix}
2 & 3 \\
2 & 1 \\
-1 & 0 \\
0 & -1
\end{bmatrix} \begin{bmatrix}
-\frac{1}{3} & \frac{2}{3} \\
\frac{2}{3} & -\frac{1}{3}
\end{bmatrix} = \begin{bmatrix}
\frac{4}{3} & \frac{1}{3} \\
0 & 1 \\
\frac{1}{3} & -\frac{2}{3} \\
-\frac{2}{3} & \frac{1}{3}
\end{bmatrix}.
\]

Once $Q$ is known, the entire disguise is useless. Therefore the generalised permutation approach is indeed necessary.

### 4.4 Validity vs security

The oversight in both methods is that we cannot choose a random matrix $A$ such that both $A$ and $A^{-1}$ are strictly positive. If we relax strict positivity to allow non-negative elements (we allow zeros) then there are some matrices that satisfy this requirement.
These are generalised permutation matrices (products of non-singular diagonal and permutation matrices) [70, Theorem 1.1]. So Vaidya’s transformation will be valid if we require $Q$ to be a generalised permutation matrix. We can then deconstruct $Q$ into a positive diagonal scaling matrix $D$ and a permutation matrix $L$.

The disguise imparted by $Q$ consists of a scaling (from $D$) and a shuffle (from $L$). If we post-multiply a matrix by $Q$, the effect would be to scale and shuffle its rows. If we pre-multiply a matrix by $Q$, the effect would be to scale and shuffle its columns.

An alternative approach to correct the transformation would be to convert the inequality constraints into equality constraints, so that we can be free to multiply them by negative numbers without fear of disrupting them. This avenue is considered in detail in Chapter 5. For now, we note that one cannot easily remove the non-negativity constraints; if we attempt to convert them to equalities by adding slack variables to them, we produce more non-negativity constraints, since the slack variables are required to be non-negative. As well, adding slack variables inserts a known structure into the constraint matrix, which can present its own security problems. For now, we focus on the generalised-permutation matrix approach. However, resorting to a generalised permutation matrix means that our data is vulnerable to attack by Alice, as we illustrate in the next section.

### 4.4.1 Alice’s attack

The drawback is that generalised permutation matrices provide a much lower degree of disguise than purely random matrices, and can severely compromise security. In fact, if Alice knew the permutation $L$, then she could use her knowledge of $c$ and $\hat{c}$ to deduce $D$ precisely. Table 4.1 provides a quick summary of the parameters known to Alice and Bob during the execution of the protocol. As we can see, Bob knows $\{M, b, Q\}$ and Alice knows $\{c, \hat{c}, \hat{M}\}$ and also $b$, since it is undisguised. There are only finitely many possible permutations, and so Alice can consider each in turn. For instance, Alice might start by guessing the permutation to be $L_i$, and from this determine $D_i$ and hence $Q_i$ such that

$$\hat{c}^T = c^T Q_i.$$  \hspace{1cm} (4.4.1)

<table>
<thead>
<tr>
<th>Who knows what</th>
<th>$c$</th>
<th>$M$</th>
<th>$b$</th>
<th>$\hat{c}$</th>
<th>$\hat{M}$</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bob</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 4.1:** A table showing which parameters are known to whom in the execution of our transformation method. Note that although $b$ belongs to Bob, it is undisguised in Program B and therefore known to Alice in the clear.
In the current form of the problem Alice also learns both the transformed solution $\hat{x}^*$, and the untransformed solution $x^*$, and so can then test whether $x^* = Q_i \hat{x}^*$. If the equation doesn’t hold, then she can rule out that particular permutation. She will in fact be able to rule out most permutations in this fashion. The only permutations she cannot eliminate occur when $x^*$ contains repeated values: in these cases she cannot eliminate permutations that swap such repeated values.

Consider our example (4.2.4) again and choose $Q = DL$ to be the generalised permutation matrix

$$Q = \begin{bmatrix} 2 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ \frac{1}{2} & 0 \end{bmatrix}. $$

Alice knows $c^T = \begin{bmatrix} 5 & 1 \end{bmatrix}$, and from Bob she receives $\hat{c}^T = c^T Q = \begin{bmatrix} \frac{1}{2} & 10 \end{bmatrix}$. Since Alice knows that $Q$ only applies a generalised permutation, she considers the two possible cases:

**Case 1:** $L_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and hence $Q_1 = \begin{bmatrix} \frac{1}{10} & 0 \\ 0 & 10 \end{bmatrix}$

**Case 2:** $L_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ and hence $Q_2 = \begin{bmatrix} 0 & 2 \\ \frac{1}{2} & 0 \end{bmatrix}$

Alice knows the transformed solution $\hat{x}^* = (0, 2)$ and the untransformed solution $x^* = (4, 0)$, and just tests which $Q_i$ matrix is consistent.

i = 1: $Q_1 \hat{x}^* = \begin{bmatrix} \frac{1}{10} & 0 \\ 0 & 10 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 20 \end{bmatrix} \neq x^*$

i = 2: $Q_2 \hat{x}^* = \begin{bmatrix} 0 & 2 \\ \frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \end{bmatrix} = x^*$

Therefore Alice knows that $Q = Q_2$, and can immediately deduce Bob’s private data $M$, using the equation $M = \hat{M} Q^{-1}$. Thus this approach does not have satisfactory security.

Clearly for larger problems, Alice’s task will not be quite so simple. If there are $n$ variables, there will be $n!$ permutations for her to consider, each with an associated $Q$ matrix. For a large value of $n$, it may be infeasible to perform the above analysis. For example, if there were 100 variables, Alice would need to test $100! \approx 9.3 \times 10^{157}$ possible shuffles. However, there are ways we can prevent Alice from even theoretically being able to perform this attack, as discussed in the next section.

As a quick aside, one might be wondering if the security would improve with the addition of the $P$ matrix from Du’s method (see (4.2.1)). Why not let $P$ be a second generalised-permutation matrix, and let $\hat{M} = PMQ$? Perhaps this would provide an extra layer of disguise? Actually, $P$ will not be of help due to our choice of data-partitioning
scenario. Remember that the $P$ matrix does nothing more than shuffle and scale the rows. Since the ordering of the constraints is not important for most problems, shuffling the rows achieves nothing. Similarly, scaling them does not alter the core structure of the constraint. We take the view that knowing Bob’s constraint up to a scale factor is as good as knowing his constraint in its original form.

### 4.4.2 Restoring security

Consider Vaidya’s problem \cite{121}, where Alice holds the objective function $c$, and Bob holds the constraint matrix $M$, and now also impose the restriction that $Q$ is a generalised permutation matrix. The key observation is that the security breach discussed in Section 4.4.1 occurs because Alice knows both $x^*$ and $\hat{x}^*$. If we prevent Alice from learning $x^*$, then we argue below that she can never verify which of her $n!$ possible matrices $Q_i$ is correct.

First note that Alice has already exploited the information available in $\hat{c}$ and $c$ to deduce the set of matrices $Q_i$ according to (4.4.1), and $b$ meanwhile is untransformed. Suppose Alice has narrowed the $Q$ matrix down to a set of $n!$ candidates for $Q_i$, and she is now keen to see which of these she can eliminate. For each $Q_i$, Alice can form a linear program, Program $i$, defined as follows:

$$
\max f(w) = c^T w \\
\text{s.t.} \\
M_i w \leq b \\
w \geq 0,
$$

(4.4.2)

where $M_i = \hat{M}Q_i^{-1}$ and $w = Q_i\hat{x}$. (Since Alice is attempting to find the real Program A, and hence the real $M$, she has attempted to transform back to the original variables, but unless she has the correct $Q_i$ this will not be the case. We hence denote the variables in Program $i$ by $w$.)

One of these Program $i$’s is the real Program A (the original linear program), and thus contains Bob’s actual constraint data. Can Alice discover which Program $i$ is correct? We show that in fact every Program $i$ is equivalent to Program B, and thus she cannot distinguish the correct program from the set of $n!$ equivalent programs. To see this, consider what would happen if one of the Program $i$’s, say Program $j$, wasn’t equivalent to Program B. Alice could then identify this fact and hence $Q_j$ and Program $j$ could be eliminated from Alice’s candidate set.

Consider the components of Program $i$ — the constraints and the objective function. The constraints $M_i\hat{x} \leq b$ are equivalent to those in Program B, since

$$
M_i w = \hat{M}Q_i^{-1}Q_i\hat{x} \\
= \hat{M}\hat{x}.
$$
Similarly, the objective function in Program $i$ is also equivalent to the objective function in Program B, since

$$f(w) = c^T w$$
$$= c^T Q_i \hat{x},$$
$$= \hat{c}^T \hat{x} = \hat{f}(\hat{x}), \text{ due to (4.4.1).} \quad (4.4.3)$$

Therefore, for all $i$, Program $i$ is equivalent to Program B, so Alice cannot learn any more about which $Q_i$ she should select.

Having ruled out any other avenues of attack, we conclude that preventing Alice from learning both $x^*$ and $\hat{x}^*$ will be sufficient to restore security. There are three ways that we can prevent Alice from learning both $x^*$ and $\hat{x}^*$:

(i) instruct Bob not to broadcast $x^*$ to Alice,

(ii) ask a semi-trusted third party, Tim, to solve the disguised problem, or

(iii) provide the output $x^*$ in additively-split fashion — Alice receives $x^*_A$ and Bob receives $x^*_B$ such that $x^* = x^*_A + x^*_B$.

In scheme (i), Alice receives the transformed solution $\hat{x}^*$ only, while in scheme (ii), Alice receives the true solution $x^*$ only. In scheme (iii), Alice receives the transformed solution $\hat{x}^*$ and part of the true solution $x^*$. We now discuss each of these schemes in more detail.

4.4.2.1 Alice does not receive $x^*$

In scheme (i), instead of both parties receiving the optimal solution at the end, only Bob receives $x^*$, while Alice knows only $\hat{x}^*$. That is, the protocol is asymmetric in its output. Preventing Alice from receiving the optimal solution $x^*$ means that she can’t apply the technique demonstrated in Section 4.4.1 to identify $Q$, and, as we’ve just argued, the other information available to her is of no use. Her only option will be to try and guess the correct program, with probability $\frac{1}{n!}$. We will further investigate this claim in Chapter 5.

There is a set of problems where withholding the optimal solution from Alice would make sense. For example, suppose that Bob is trying to decide between several suppliers, as in Figure 4.3. Bob has a set of private constraints $M$ relating to his company, and wants to know which supplier to choose. The suppliers in turn each have a private objective function, $c_i$, representing their cost structure. Our method allows Bob to find $x^*$, while the supplier only knows $\hat{x}^*$. Bob also needs to know the optimal objective value, $f(x^*)$. He must do this without learning the supplier’s private objective coefficients $c$ and without the supplier learning $x^*$. The beauty is that the supplier can send
Bob would like to choose between 3 suppliers. Each supplier has a private objective function $c_i$. Bob has a set of private constraints $M$. Using our new transformation method, Bob can receive $x_i^*$ and $f_i^*$ from each supplier, while protecting the privacy of his constraints and also maintaining the privacy of the suppliers’ objective functions.

$\hat{f}(\hat{x}^*)$ to Bob, which does not reveal $c$, and yet is actually equal to $f(x^*)$, as shown in (4.2.3). Thus Bob receives his desired information, and no private information is revealed.

### 4.4.2.2 A semi-trusted third party solves the disguised LP

Alternatively, the security breach described can be avoided if we follow scheme (ii) and assume the existence of a semi-trusted third party, Tim. By ‘semi-trusted’, we mean that Tim must be trusted not to collude with either Alice or Bob. We also assume that Tim is ‘honest but curious’, meaning that he will follow the instructions of the protocol but will attempt to discover information about Alice’s and Bob’s data by performing calculations on the data given to him (which are guaranteed to be fruitless).

Tim solves the disguised problem in place of Alice, thus eliminating the attack arising from any party knowing both $\hat{c}$ and $c$. Of course, if Alice colludes with Tim, and discovers $\hat{c}$, we are back to square one. The advantage of the semi-trusted third party approach is that both Alice and Bob can receive $x^*$.

In more detail, the semi-trusted third party approach would work as follows: Bob can send $\hat{M} = MQ$ and $b$ directly to Tim. For the calculation of $c^TQ$, Alice and Bob engage in a secure multiplication protocol such that the answer is split into two shares $s_A$ and
Figure 4.4: Alice and Bob engage the services of Tim, a semi-trusted third party, who solves the disguised problem in place of Alice. This way, Alice can learn $x^*$, since she no longer has enough information to perform any attack. The only danger is collusion between Tim and Bob/Alice.

$s_B$, and then they each send their shares to Tim. This is illustrated in Figure 4.4. Since Tim doesn’t know $c$, he can’t use the knowledge of $c^TQ$ to construct a list of candidate Qs. That is, it is ‘safe’ for Tim to know $c^TQ$, as long as he doesn’t collude with Alice. Also, Tim cannot deduce $M$ from knowledge of $MQ$, since he doesn’t know $Q$. Under this scheme, Alice can safely learn $x^*$ and $f^*$. Although this output is deemed ‘safe’ in terms of the protocol, one must be careful that the output itself does not reveal too much, as discussed in Chapter 3.

The danger of this approach is of course the potential for collusion between Tim and either Bob or Alice. If Tim colludes with Bob, he can send Bob $\hat{c}^T$ which Bob can multiply by $Q^{-1}$ to reveal Alice’s data $c$. Similarly, if Tim colludes with Alice, he can tell her $\hat{c}$, $M$ and $\hat{x}^*$, which together with $x^*$ allow her to perform the attack described in Section 4.4.1.

4.4.2.3 Additively-split output

For scheme (iii), the transformation method is modified so that it returns shares of the output $x^*$. This has the further advantage of allowing the linear programming module to be used as part of a larger protocol (recall that when composing protocols, all intermediate output must be in shares). If shares of $f^*$, as well as $x^*$, are desired, we must engage a semi-trusted third party, who provides the shares. Otherwise, Alice and Bob
can calculate shares of \( x^* \) (but not \( f^* \)) amongst themselves.

Schemes (i) - (iii) are formally presented as algorithms in the next section.

4.5 The generalised-permutation algorithms

We formally present the algorithms for the three schemes discussed above — the scheme conducted solely between Alice and Bob is termed the ‘Base Protocol’, while the scheme using a semi-trusted third party is presented as ‘Variation 1’, and the scheme producing additively-split output is presented as ‘Variation 2’. Each version provides a different output option for the optimal solution \( x^* \), as summarised in Table 4.2.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Output option for ( x^* )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Protocol (two-party)</td>
<td>Told to Bob only (Alice is not told ( x^* ))</td>
</tr>
<tr>
<td>Variation 1 (semi-trusted third party)</td>
<td>Public (both parties are told ( x^* ))</td>
</tr>
<tr>
<td>Variation 2 (shared output)</td>
<td>Shared (( x^* = x^<em>_A + x^</em>_B ))</td>
</tr>
</tbody>
</table>

Table 4.2: Output options for each protocol.

Each algorithm includes an ‘LP Classification’ statement which defines the types of privacy-preserving linear programs the method applies to, following the notation introduced in Chapter 3.

4.5.1 Base protocol: two-party protocol

Protocol 4.1. (Base Protocol)

LP Classification: \( O(A)/O(B)/P/\leq/N/O(B)/P \)

Input: Alice has an \( n \)-dimensional vector of objective function coefficients \( c \); Bob has an \( m \times n \) matrix of constraint coefficients \( M \). They both know the vector of RHS values \( b \). They wish to solve the LP shown below.

\[
\begin{align*}
\max \quad & f(x) = c^T x \\
\text{s.t.} \quad & Mx \leq b \\
& x \geq 0.
\end{align*}
\] (4.5.1)

Output: Alice learns only \( \hat{x}^* \), the solution to the transformed problem, and \( \hat{f}^* = f^* \). Bob learns the optimal solution \( x^* \), and \( f^* = f^* \).
Chapter 4. Transformation Methods: Hiccups and Solutions

Protocol:

1. Bob generates a random positive $n \times 1$ vector $d$, which he sets as a diagonal to form the scaling matrix $D$, and a random $n \times n$ permutation matrix $L$, which he multiplies together to form $Q = DL$.

2. Bob sends to Alice $\hat{M} = MQ$.

3. Alice and Bob use Protocol 4.2 to compute $\hat{c}^T = c^T Q$ such that only Alice learns $\hat{c}^T$.

4. Alice now has all the components of the transformed problem, defined below:

$$
\begin{align*}
\max \quad & \hat{f}(\hat{x}) = \hat{c}^T \hat{x} \\
\text{s.t.} \quad & \hat{M} \hat{x} \leq b \\
& \hat{x} \geq 0,
\end{align*}
$$

where $\hat{M} = MQ$, $\hat{x} = Qx$ and $\hat{c}^T = c^T Q$. Alice solves the transformed problem and sends both the solution $\hat{x}^*$ and the corresponding objective value $\hat{f}^*$ to Bob.

5. Bob untransforms $\hat{x}^*$ to obtain $x^* = Q\hat{x}^*$, and as shown in (4.2.3), $\hat{f}^* = f^*$.

4.5.2 Sub-protocol: secure evaluation of $\hat{c}$

Protocol 4.2. (Secure evaluation of $\hat{c}$)

This protocol is adapted from Du and Atallah [50]. We have modified the protocol to improve its efficiency. In the Du-Atallah protocol, $p$ and $d$ are security parameters which are chosen so that performing $p^d$ additions is computationally infeasible. We instead require that $p^d$ be chosen such that Bob having a 1 in $p^d$ chance of guessing Alice’s vector $c$ is acceptable.

Input: Alice has an $n$-dimensional vector $c$; Bob has a random $n \times n$ matrix $Q$.

Output: Alice receives $\hat{c}^T = c^T Q$. Bob receives no output.

Protocol:

1. Alice and Bob agree on two random numbers $p$ and $d$, such that the 1 in $p^d$ chance of Bob guessing Alice’s data is acceptably low.

2. Alice generates $d - 1$ random vectors $x_1, \ldots, x_{d-1} \sim \mathcal{H}(n \times 1)$, where $\mathcal{H}$ is a distribution chosen by Alice, and sets

$$
x_d = c - \sum_{i=1}^{d-1} x_i.
$$

3. For each $j = 1, \ldots, d - 1$, Alice and Bob conduct the following sub-steps:
(a) Alice generates \( p - 1 \) random vectors \( \mathbf{g}^{(j)} \sim \mathcal{H}(n \times 1) \), and sends the following sequence to Bob:

\[
(\mathbf{h}^{(j)}_1, \ldots, \mathbf{h}^{(j)}_p),
\]

where \( \mathbf{h}^{(j)}_k = \mathbf{x}_j \), and \( k \in [1, p] \) is a secret random number known only by Alice, and the \( \mathbf{g}^{(j)} \)'s are assigned to the remaining \( (p - 1) \mathbf{h}^{(j)}_i, i \neq k \).

(b) Bob computes \( (\mathbf{h}^{(j)}_i)^T Q + \mathbf{r}^T_j \) for each \( i = 1, \ldots, p \), where \( \mathbf{r}_j \) is a random \( 1 \times n \) vector chosen by Bob.

(c) Using the 1-out-of-\( N \) Oblivious Transfer protocol, Alice gets back the result of

\[
(\mathbf{h}^{(j)}_k)^T Q + \mathbf{r}^T_j = \mathbf{x}_j^T Q + \mathbf{r}^T_j.
\]

(4) Bob sends \( \sum_{j=1}^d \mathbf{r}^T_j \) to Alice.

(5) Alice computes \( \hat{c} = \sum_{j=1}^d (\mathbf{x}_j^T Q + \mathbf{r}^T_j) - \sum_{j=1}^d \mathbf{r}^T_j = \mathbf{c}^T Q \).

### 4.5.3 Variation 1 — semi-trusted third party

**Protocol 4.3.** (Variation 1 — semi-trusted third party)

**LP Classification:** \( O(A)/O(B)/O(B)/\leq/N/P/P \)

**Input:** As per Base Protocol.

**Output:** Alice and Bob both receive \( \mathbf{x}^* \) and \( f^* \).

**Protocol:**

(1) Bob generates a random positive \( n \times 1 \) vector \( \mathbf{d} \), which he sets as a diagonal to form the scaling matrix \( \mathbf{D} \), and a random \( n \times n \) permutation matrix \( \mathbf{L} \), which he multiplies together to form \( \mathbf{Q} = \mathbf{DL} \).

(2) Bob sends \( \hat{M} = \mathbf{MQ} \) and \( \mathbf{b} \) to Tim.

(3) Alice and Bob compute \( \mathbf{c}^T \mathbf{Q} = \mathbf{s}_A + \mathbf{s}_B \) using a secure multiplication protocol (e.g., Protocol 1 from [52]), which returns the output in two shares, \( \mathbf{s}_A \) (known to Alice) and \( \mathbf{s}_B \) (known to Bob).

(4) Alice and Bob each send their share of \( \mathbf{c}^T \mathbf{Q} \) to Tim. Tim adds the shares together to form \( \hat{\mathbf{c}}^T \) in full.

(5) Tim now has all the components of the transformed problem. Tim solves the transformed problem and sends both the solution \( \hat{\mathbf{x}}^* \) and the corresponding objective value \( \hat{f}_{\text{max}} \) to Bob.

(6) Bob untransforms \( \hat{\mathbf{x}}^* \) to obtain \( \mathbf{x}^* = \mathbf{Q}\hat{\mathbf{x}}^* \), and as shown in (4.2.3), \( \hat{f}^* = f^* \).

(7) Bob sends \( \mathbf{x}^* \) and \( f^* \) to Alice.
4.5.4 **Variation 2 — additively-split output**

**Protocol 4.4.** (Variation 2 — additively-split output)

**LP Classification:** $O(A)/O(B)/P/\leq/N/S/S$

**Input:** As per Base Protocol.

**Output:** Alice receives $x_A^*$, and Bob receives $x_B^*$, such that $x_A^* + x_B^* = x^*$.

**Protocol:** Run the Base Protocol (Protocol 4.1) as usual, except Alice doesn’t send $\hat{x}^*$ to Bob in Step 4, only $\hat{f}^*$. Alice and Bob compute $x^* = Q\hat{x}^*$ via a secure multiplication (an adaptation of Protocol 4.2), such that the output consists of shares $x_A^*$ and $x_B^*$, held by Alice and Bob respectively, such that $x^* = x_A^* + x_B^*$.

### 4.6 Conclusion

Although we have ruled out the possibility that Alice can find Bob’s data with certainty, there is nothing to stop Alice in Protocols 4.1 and 4.4 from taking a guess from the available permutations. By comparing $c$ and $\hat{c}$, Alice can come up with a list of $n!$ candidate $Q$s, one for each possible shuffle. Therefore, Alice has a $\frac{1}{n!}$ chance of selecting the correct $Q$. This is by no means the end of the story, though, as we will see in Chapter 5.

For now, we note that the security level is satisfactory for large problems. For large values of $n$, the value of $\frac{1}{n!}$ is negligible. Even for a modest problem with 10 variables, Alice’s chances of guessing Bob’s data are 1 in 3.6 million! Typical optimisation problems can have $n > 100$, giving an even better security level.

We have shown that two existing disguise methods are invalid, due to the random matrices interfering with the inequality constraints of the linear program. Replacing the random disguising matrix by a random generalised-permutation matrix will restore validity, but weakens the security level. However, we have quantified the security level in terms of the probability of Alice correctly guessing Bob’s constraint matrix, and shown that for large problems (e.g., with 10 or more variables) this probability is negligible.

In Chapter 5, we analyse the security of our generalised-permutation method more deeply, considering a number of ‘side issues’ that arise when the algorithm is implemented in practice.
Chapter 5

Practical Security Analysis

In this chapter we provide an in-depth analysis of security for the transformation method developed in Chapter 4. In particular, we consider a number of ‘side issues’ that occur when one attempts to implement the algorithm in practice. These include: the potential for Alice to uncover partial private data; multiple runs of the algorithm with similar LPs; why Alice must not be told the distributions of Bob’s random numbers; and features that cannot be disguised using this method. Our analysis is guided by the acceptable security paradigm.

5.1 The acceptable security paradigm

Secure multiparty computation traditionally defines security with respect to an ideal model [62]. The ideal model for privacy-preserving linear programming is that Alice and Bob send their inputs to a trusted third party, who solves the joint-LP and returns the output $x^*$ to each of them. In the ideal model, the parties learn nothing about each other’s data aside from what can be inferred from their input and their output. A computation is said to be secure if the parties learn nothing more about each other’s data during the execution of the protocol than they would have in the ideal model [62].

Du and Zhan [53] state that ‘achieving this kind of security is not difficult, but achieving it efficiently is’. As an alternative, they introduce the notion of ‘acceptable security’ — a tradeoff between ideal security and an efficient method. The model allows partial information disclosure in favour of better performance. The authors point out that ‘if the ideal security is too expensive to achieve, people might prefer low-cost solutions that can achieve security at an “acceptable” level’, and that ‘sacrificing some security or disclosing some limited information about the private data is often acceptable in practice.’ [53, p.128].

79
Du and Zhan [53, p.128] provide the following informal definition of acceptable security:

‘A protocol achieves acceptable security if an adversary can only narrow down all the possible values of the secret data to a domain with the following properties:

(1) The number of values in this domain is infinite, or the number of values in this domain is so large that a brute-force attack is computationally infeasible.

(2) The range of the domain [difference between the upper and lower bounds] is acceptable for the application.’

There are problems with this definition. One problem is that infinite domains can still convey meaningful information, even if the range of possible values is large. For example, knowing a constraint up to a scale factor (which can be one of an uncountable number of values in an infinite range) is unacceptable in many applications, as scaled versions of the constraints may contain equivalent information.

We feel that it is more important for the security level to be explicitly specified. For example, this may be given in terms of the probability of an adversary learning a party’s raw data. We will put our efforts into calculating this probability.

5.2 Partial exposure of private data

In this section, our aim is to quantify the security-level of scheme (i) (i.e., Protocol 4.1) from Chapter 4 more precisely. We start by assessing the likelihood that Alice correctly uncovers part of Bob’s constraint matrix (as opposed to the complete matrix).

While it is true that the probability of Alice exactly obtaining Bob’s complete matrix is $\frac{1}{n!}$, her probability of obtaining some of Bob’s matrix is greater than this. The reason is that $Q$ only applies a scale and a shuffle to $M$; thus even if Alice selects an incorrect $Q$, it may still have part of the shuffle correct, and will therefore produce some correct columns of $M$. It may be the case that Alice has a negligible chance of getting the whole matrix correct, but a very high chance of getting most of it correct.

What is the probability that a randomly-chosen $Q$ will reveal one correct column of $M$? $k$ correct columns of $M$? We devise an answer shortly. We stress that Alice will not know which columns are correct — she will only know her chances of having $k$ of them correct.

One could argue that this partial exposure of $M$ is not a threat since Alice doesn’t know which columns are correct, but we would still like to be assured that the chance
of Alice revealing a high proportion of correct columns is very low. If it were the case that the majority of the incorrect Qs still gave 90% of the columns of M correct, we would abandon this algorithm.

Hence our first aim is to quantify the number of permutations with k positions correct (with respect to the true permutation chosen by Bob), for k = 0, 1, 2, ..., n. To do this, we need some results from the field of combinatorics; but first, let’s consider a small example and enumerate the probabilities ‘by hand’.

5.2.1 Preliminary example

Suppose \{2, 1, 3, 4\} is the permuted order of the columns of M under Bob’s chosen permutation. That is, column 2 has been moved to position 1, column 1 has been moved to position 2, and columns 3 and 4 are unchanged. How many permutations of the numbers \{1, 2, 3, 4\} will have 0 elements in the correct position (with respect to the true permutation)? 1 element in the correct position? 2 elements in the correct position? etc.

We can determine these counts by direct enumeration, shown in Table 5.1. Out of the 24 possible permutations, 9 have 0 columns of M correct, 8 have 1 column correct, 6 have 2 columns correct and 1 has all 4 columns correct. From this we see that even an incorrect permutation may reveal part of M. Hence, our claimed security level of \frac{1}{n!}
may need some adjustment. For a problem with 4 variables, we can say that Alice has:
- a \frac{1}{4!} = 4.2% chance of revealing M in full.
- a \frac{9}{4!} = 0.0% chance of revealing 3 correct columns of M.
- a \frac{9}{4!} = 25% chance of revealing 2 correct columns of M.
- a \frac{8}{4!} = 33.3% chance of revealing 1 correct column of M.
- a \frac{9}{4!} = 37.5% chance of having 0 columns of M correct.

It is important to keep in mind that Alice will not be able to tell which columns of M are correct (if any) — she will only know that her chances of having at least one correct column before her are \frac{15}{24} = \frac{15}{24} = 62.5%.

What will these probabilities be in general? For us to have any faith in our method, Alice’s chances of having a large proportion of M correct should be very low. In the next section we utilise some results from the field of combinatorics to derive general formulae for the probabilities we’ve calculated in this section, to see if this is indeed the case.
### Table 5.1: A list of all permutations of the numbers \{1,2,3,4\}; alongside each is a count showing the number of matches with respect to the actual permutation chosen by Bob of \{2,1,3,4\}.

<table>
<thead>
<tr>
<th>True Perm</th>
<th>Number of positions correct (with respect to True Perm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 3 4</td>
<td></td>
</tr>
<tr>
<td>Permutations</td>
<td></td>
</tr>
<tr>
<td>1 2 3 4</td>
<td>2</td>
</tr>
<tr>
<td>1 2 4 3</td>
<td>0</td>
</tr>
<tr>
<td>1 3 2 4</td>
<td>1</td>
</tr>
<tr>
<td>1 3 4 2</td>
<td>0</td>
</tr>
<tr>
<td>1 4 2 3</td>
<td>0</td>
</tr>
<tr>
<td>1 4 3 2</td>
<td>1</td>
</tr>
<tr>
<td>2 1 4 3</td>
<td>2</td>
</tr>
<tr>
<td>2 1 3 4</td>
<td>4</td>
</tr>
<tr>
<td>2 3 4 1</td>
<td>1</td>
</tr>
<tr>
<td>2 3 1 4</td>
<td>2</td>
</tr>
<tr>
<td>2 4 3 1</td>
<td>2</td>
</tr>
<tr>
<td>2 4 1 3</td>
<td>1</td>
</tr>
<tr>
<td>3 1 2 4</td>
<td>2</td>
</tr>
<tr>
<td>3 1 4 2</td>
<td>1</td>
</tr>
<tr>
<td>3 2 4 1</td>
<td>0</td>
</tr>
<tr>
<td>3 2 1 4</td>
<td>1</td>
</tr>
<tr>
<td>3 4 2 1</td>
<td>0</td>
</tr>
<tr>
<td>3 4 1 2</td>
<td>0</td>
</tr>
<tr>
<td>4 1 2 3</td>
<td>1</td>
</tr>
<tr>
<td>4 1 3 2</td>
<td>2</td>
</tr>
<tr>
<td>4 2 3 1</td>
<td>1</td>
</tr>
<tr>
<td>4 2 1 3</td>
<td>0</td>
</tr>
<tr>
<td>4 3 2 1</td>
<td>0</td>
</tr>
<tr>
<td>4 3 1 2</td>
<td>0</td>
</tr>
</tbody>
</table>

5.2.2 Alice’s probability of revealing \(k\) correct columns of \(M\)

The probability that Alice reveals \(k\) correct columns of \(M\) can be thought of as the probability that \(k\) elements of a randomly-chosen permutation will match the correct permutation.

The problem of counting the number of matches was originally posed in 1708 by the French mathematician Pierre de Montmort as a problem involving playing cards [41,42], but is more popularly phrased as the ‘Matching Hats Problem’ [110]:

\[ n \text{ men with hats have had a bit too much to drink at a party. As they leave the party, each man randomly takes a hat from the hatstand. What is the probability that at least one man takes his own hat?} \]

This is equivalent to the probability that at least one column of \(M\) is correct. This probability is much easier to calculate once we have an expression for the number of permutations with no columns correct (i.e. the number of derangements).
5.2.2.1 Derangements of \( n \) items

A permutation in which none of the elements remain in their original positions is known as a ‘derangement’. In our context, a derangement is a permutation (from Alice’s candidate set) in which none of the elements match Bob’s chosen permutation of the columns of \( M \). The number of derangements for a set of \( n \) elements is given by

\[
D_n = n! - \sum_{j=0}^{n} \frac{(-1)^j}{j!}.
\]

The formula for \( D_n \) is derived by first taking all permutations, and then subtracting the number of permutations with at least one match, i.e.,

\[
D_n = n! - \left(n! - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \ldots \pm \frac{1}{n!}\right),
\]

which simplifies to (5.2.1). The number of permutations with at least 1 match can be calculated using the inclusion-exclusion principle of combinatorics; the interested reader is referred to [34] for an in-depth explanation.

The probability of having zero matches is then given by:

\[
P(0 \text{ matches}) = 1 - \sum_{j=0}^{n} \frac{(-1)^j}{j!},
\]

which is simply the expression for \( D_n \) in (5.2.1) divided through by \( n! \), the total number of permutations.

We can approximate the expression in (5.2.3). Consider the Taylor series for \( e^x \):

\[
e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots
\]

Note that

\[
e^{-1} = 1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \ldots
\]

which is equal to the probability of having 0 matches (for reasonably large \( n \)).

Therefore, the probability of Alice having at least 1 correct column in her guess of the \( M \) matrix is approximately given by:

\[
P(\text{at least 1 match}) = 1 - P(0 \text{ matches}) \approx 1 - \frac{1}{e} = 0.6321\ldots
\]
We also want to find the probability of Alice having \( k \) correct columns of \( M \). Let \( b_n(k) \) be the number of permutations of \( n \) objects with exactly \( k \) matches. To calculate \( b_n(k) \), all one has to do is: (i) select the \( k \) elements that will match; (ii) multiply this by the number of permutations of the remaining \( n - k \) elements which have no matches.

Hence the number of permutations with exactly \( k \) matches is

\[
b_n(k) = C_k^n \cdot D_{n-k} = C_k^n \cdot (n - k)! \sum_{j=0}^{(n-k)} \frac{(-1)^j}{j!} \quad \text{, using the formula for } D_n \text{ in (5.2.1)}
\]

\[
= \frac{n!}{(n - k)!k!} (n - k)! \sum_{j=0}^{(n-k)} \frac{(-1)^j}{j!}
\]

\[
= \frac{n!}{k!} \sum_{j=0}^{(n-k)} \frac{(-1)^j}{j!}.
\]

Put another way, \( b_n(k) \) gives the number of permutations which will result in exactly \( k \) correct columns of \( M \). Therefore, the probability of a randomly selected permutation having \( k \) correct columns is

\[
P_n(k) = \frac{1}{k!} \sum_{j=0}^{(n-k)} \frac{(-1)^j}{j!}.
\]

See Figure 5.1 for an illustration of Alice’s choice of \( Q_i \). Even if Alice’s corresponding \( M_i \) matrix reveals \( k \) correct columns, they are still no good to her unless she knows which columns are correct. Thus this attack is of no threat unless there is a high chance of having close to all of the columns of \( M \) correct.

Table 5.2 lists some concrete examples displaying Alice’s probabilities of obtaining a certain number of correct columns for problems with 2, 3, \ldots, 10 variables. Here, a correct column has both correct values and is in its correct location in the matrix.

The final two rows of Table 5.2 show the limiting values of the probabilities. Since the term \( \sum_{j=0}^{(n-k)} \frac{(-1)^j}{j!} \to \frac{1}{e} \) as \( n - k \to \infty \), the general form of the limit for each value of \( k \) is:

\[
\frac{1}{k!} \cdot \frac{1}{e}.
\]

We see that convergence towards the limiting values happens very quickly as \( n \) grows. The sum when \( k = 0 \) converges to \( \frac{1}{e} \) at about \( n = 5 \); thus for \( n - k = 5 \) we should find ourselves getting close to the limiting factor. e.g. for \( n = 10, k = 5 \), i.e. \( n - k = 5 \), we are approximately equal to the limit.
Alice’s candidate $Q$ matrices

$$Q_1 \quad Q_2 \quad Q_3 \quad \ldots \quad Q_n$$

Alice selects one of the $Q$s at random.

![Alice](image)

Alice uses $Q_3$ to compute $M_3$.

$$P(M_3 = M) = \frac{1}{n!}$$

$$P(M_3 \text{ has } k \text{ columns correct}) = \frac{1}{k!} \sum_{j=0}^{k} \frac{(-1)^j}{j!}$$

$$\approx \frac{1}{k!} \frac{1}{e} \quad \text{for } (n-k) \geq 5$$

Figure 5.1: Alice chooses one of the candidate $Q_i$ matrices at random and uses it to find the corresponding $M$ matrix. The probability that it exactly equals the true $M$ matrix is $\frac{1}{n!}$. The probability that $k$ of its columns are correct is higher than this; however this probability becomes small for $k \geq 3$. Although Alice has a reasonable chance of getting one column correct, her chance of getting 3 or more correct is small.

We can also calculate the expected number of correct columns:

$$E(k) = \sum_{i=0}^{\infty} P(k = i) \cdot i$$

$$= \sum_{i=0}^{\infty} \frac{1}{e \cdot i!} \cdot i$$

$$= \frac{1}{e} \sum_{i=0}^{\infty} \frac{1}{(i - 1)!}$$

$$= \frac{1}{e} \sum_{i=1}^{\infty} \frac{1}{i!}$$

$$= 1, \text{ since } \sum_{i=1}^{\infty} \frac{1}{i!} = e.$$
Probability of revealing exactly $k$ correct columns of $M$

<table>
<thead>
<tr>
<th></th>
<th>$k = 0$</th>
<th>$k = 1$</th>
<th>$k = 2$</th>
<th>$k = 3$</th>
<th>$k = 4$</th>
<th>$k = 5$</th>
<th>$k = 6$</th>
<th>$k = 7$</th>
<th>$k = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.50</td>
<td>n/a</td>
<td>0.50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>0.33</td>
<td>0.50</td>
<td>n/a</td>
<td>0.17</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>0.38</td>
<td>0.33</td>
<td>0.25</td>
<td>n/a</td>
<td>0.042</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>0.37</td>
<td>0.38</td>
<td>0.17</td>
<td>0.083</td>
<td>n/a</td>
<td>8.3 $\times 10^{-3}$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>0.37</td>
<td>0.37</td>
<td>0.19</td>
<td>0.056</td>
<td>0.021</td>
<td>n/a</td>
<td>1.4 $\times 10^{-4}$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0.37</td>
<td>0.37</td>
<td>0.18</td>
<td>0.062</td>
<td>0.014</td>
<td>4.2 $\times 10^{-3}$</td>
<td>n/a</td>
<td>2.0 $\times 10^{-4}$</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>0.37</td>
<td>0.37</td>
<td>0.18</td>
<td>0.061</td>
<td>0.016</td>
<td>2.8 $\times 10^{-3}$</td>
<td>6.9 $\times 10^{-4}$</td>
<td>n/a</td>
<td>2.5 $\times 10^{-5}$</td>
</tr>
<tr>
<td>9</td>
<td>0.37</td>
<td>0.37</td>
<td>0.18</td>
<td>0.061</td>
<td>0.015</td>
<td>3.1 $\times 10^{-3}$</td>
<td>4.6 $\times 10^{-4}$</td>
<td>9.9 $\times 10^{-5}$</td>
<td>n/a</td>
</tr>
<tr>
<td>10</td>
<td>0.37</td>
<td>0.37</td>
<td>0.18</td>
<td>0.061</td>
<td>0.015</td>
<td>3.1 $\times 10^{-3}$</td>
<td>5.2 $\times 10^{-4}$</td>
<td>6.6 $\times 10^{-5}$</td>
<td>1.2 $\times 10^{-5}$</td>
</tr>
</tbody>
</table>

Table 5.2: The table shows Alice’s probability of revealing $k = 0, 1, \ldots, n$ correct columns of Bob’s matrix $M$, for $n = 2, 3, \ldots, 10$ variables. Note that if there are $n$ variables, it is impossible to have $n-1$ columns correct. These cases have been highlighted with ‘n/a’ in the table.

However, we can’t stop here. What happens if the same linear program is run again, with a different objective function? Alice will now have more information at her disposal, and further analysis is required to examine what impact this will have on the security level.

### 5.2.3 Repeated instances of similar LPs

It is important to consider what would happen if the parties repeat the problem with the same $M$ and $b$, but a different $c$ vector (and of course a different $Q$ matrix). This may be necessary if the objective function changes, say. We show here that such a scenario can be very dangerous. The reason is that Alice can compare all possible $M$ matrices she obtains on the first run with the new set of possible $M$ matrices she obtains on the second run — the correct matrix will be the only one which appears in both sets and will thus ‘give itself away’.

Let’s consider a small example where $M$ has 2 rows and 3 columns. Suppose

$$M = \begin{bmatrix} 4 & 3 & 7 \\ 2 & 5 & 1 \end{bmatrix}.$$
Also suppose that Bob chooses
\[
Q = \begin{bmatrix}
0 & \frac{1}{2} & 0 \\
3 & 0 & 0 \\
0 & 0 & 7
\end{bmatrix},
\]
which would apply a permutation of \(\{2,1,3\}\) to the columns of \(M\), as well as a scaling.

In this example, suppose that Alice’s objective function vector is given by:
\[
c^T = [5 \quad 2 \quad 6].
\]
From Bob she receives:
\[
\hat{c}^T = c^T Q = [6 \quad \frac{5}{2} \quad 42].
\]
Table 5.3 lists all permutations Alice would have to try in her attempt to deduce \(M\). The first column lists the permutation, the second column lists the permutation in matrix form, the third column shows the corresponding \(Q_i\) matrix, and the final column gives the matrix \(\hat{M}Q_i^{-1}\). Remember that \(\hat{M} = MQ\), so only when \(Q_i^{-1} = Q^{-1}\) will \(M\) be revealed in full.

When a column of \(M\) is correct, it will appear in its rightful spot. The exception is when \(c\) has repeated elements, in which case it is possible for the scaling to be correct but the permutation to be wrong.

Now suppose that Alice and Bob run the LP a second time, with the same constraint set, but a different objective function \(c\) and a different \(Q\). This time,
\[
c^T = [1 \quad 2 \quad 3],
\]
and Bob chooses
\[
Q = \begin{bmatrix}
7 & 0 & 0 \\
0 & 5 & 0 \\
0 & 0 & 2
\end{bmatrix}.
\]
From Bob, Alice receives:
\[
\hat{c}^T = c^T Q = [7 \quad 10 \quad 6].
\]
allowing her to come up with a list of candidate \(Q\) matrices, and from these generate a list of candidate \(M\) matrices. Alice’s candidate sets are shown in Table 5.4.

Alice can compare the output in Table 5.4 with that in Table 5.3 and find the matching \(M_i\), thereby revealing Bob’s data. We see that running an LP twice using the same constraint set is very dangerous — it will allow Alice to discover Bob’s data with certainty, since the only \(M_i\) matrix which appears in both candidate sets is the correct
Example 2: This LP had the same constraint set as Example 1, but a different vector and a different \( Q \) matrix. In this example, \( \mathbf{c} = [1, 2, 3] \) and \( \tilde{\mathbf{c}}^T = [7, 10, 6] \). The correct \( M \) matrix is highlighted in bold in row 5. Correct columns appearing in other rows are also highlighted in bold. Alice can compare the output here with that in Table 5.3 and find the matching \( M_i \), thereby revealing Bob’s data.

### Table 5.3: Example 1
For each candidate permutation, Alice finds the corresponding \( Q_1 \) and \( M_i \), using \( M_i = \tilde{MQ}_i^{-1} \). Line 4 corresponds to the true permutation chosen by Bob, with \( M \) highlighted in bold. Matching columns in other rows are shown in bold. Notice that other permutations still result in \( M_i \) having one column correct (eg. in Lines 2, 3 and 5).

<table>
<thead>
<tr>
<th>( i = )</th>
<th>Permutation</th>
<th>( L ) matrix</th>
<th>( Q ) matrix</th>
<th>( M_i = \tilde{MQ}_i^{-1} )</th>
</tr>
</thead>
</table>
| 1. | \{3, 2, 1\} | \[
\begin{bmatrix}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0
\end{bmatrix}
\] | \[
\begin{bmatrix}
0 & 0 & \frac{13}{2} \\
0 & \frac{5}{4} & 0 \\
1 & 0 & 0
\end{bmatrix}
\] | \[
\begin{bmatrix}
\frac{35}{8} & \frac{5}{8} & 9 \\
\frac{5}{8} & \frac{35}{8} & 15 \\
\frac{5}{8} & \frac{5}{8} & \frac{5}{8}
\end{bmatrix}
\] |
| 2. | \{3, 1, 2\} | \[
\begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
1 & 0 & 0
\end{bmatrix}
\] | \[
\begin{bmatrix}
\frac{1}{2} & \frac{1}{2} & 0 \\
0 & 0 & 21 \\
1 & 0 & 0
\end{bmatrix}
\] | \[
\begin{bmatrix}
4 & \frac{7}{3} & 9 \\
\frac{2}{3} & \frac{1}{3} & 15 \\
\frac{2}{3} & \frac{1}{3} & \frac{1}{3}
\end{bmatrix}
\] |
| 3. | \{2, 3, 1\} | \[
\begin{bmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\] | \[
\begin{bmatrix}
0 & 0 & \frac{13}{2} \\
3 & 0 & 0 \\
0 & \frac{5}{4} & 0
\end{bmatrix}
\] | \[
\begin{bmatrix}
\frac{35}{8} & \frac{5}{8} & 9 \\
\frac{5}{8} & \frac{35}{8} & 15 \\
\frac{5}{8} & \frac{5}{8} & \frac{5}{8}
\end{bmatrix}
\] |
| 4. | \{2, 1, 3\} | \[
\begin{bmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}
\] | \[
\begin{bmatrix}
\frac{1}{2} & \frac{1}{2} & 0 \\
0 & 0 & 7 \\
0 & \frac{5}{4} & 0
\end{bmatrix}
\] | \[
\begin{bmatrix}
4 & 3 & 7 \\
2 & 5 & 1
\end{bmatrix}
\] |
| 5. | \{1, 2, 3\} | \[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\] | \[
\begin{bmatrix}
\frac{1}{2} & \frac{1}{2} & 0 \\
\frac{1}{2} & \frac{1}{2} & 0 \\
0 & 0 & 7
\end{bmatrix}
\] | \[
\begin{bmatrix}
\frac{1}{2} & \frac{1}{2} & 0 \\
\frac{1}{2} & \frac{1}{2} & 0 \\
\frac{1}{2} & \frac{1}{2} & \frac{1}{2}
\end{bmatrix}
\] |
| 6. | \{1, 3, 2\} | \[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0
\end{bmatrix}
\] | \[
\begin{bmatrix}
\frac{1}{2} & \frac{1}{2} & 0 \\
\frac{1}{2} & \frac{1}{2} & 0 \\
0 & \frac{5}{4} & 0
\end{bmatrix}
\] | \[
\begin{bmatrix}
\frac{1}{2} & \frac{1}{2} & 0 \\
\frac{1}{2} & \frac{1}{2} & 0 \\
\frac{1}{2} & \frac{1}{2} & \frac{1}{2}
\end{bmatrix}
\] |

### Table 5.4: Example 2
This LP had the same constraint set as Example 1, but a different vector and a different \( Q \) matrix. In this example, \( \mathbf{c} = [1, 2, 3] \) and \( \tilde{\mathbf{c}}^T = [7, 10, 6] \). The correct \( M \) matrix is highlighted in bold in row 5. Correct columns appearing in other rows are also highlighted in bold. Alice can compare the output here with that in Table 5.3 and find the matching \( M_i \), thereby revealing Bob’s data.
one. Attacks such as this highlight the way the security of a protocol can change when it is run more than once. In this case, a protocol that was secure in a stand-alone sense does not maintain its security when it is run more than once, although the security breakdown only occurs when the same constraint matrix is used on both occasions.

All is not lost, however. The good news is that the security breakdown is only a problem for small linear programs. Once the value of $n$ becomes large, the amount of work needed to perform the attack renders it infeasible to execute. For $n = 150$, say, there are $150! \approx 5.71 \times 10^{262}$ possible $Q$s. Alice must perform $n$ divisions to find each candidate $Q_i$ matrix, where $i = 1, \ldots, n!$, as well as one matrix multiplication on matrices of size $(m \times n) \times (n \times n)$ to compute $M_i = MQ_i^{-1}$. She also needs to compute $Q_i^{-1} = (DL_i)^{-1} = L_i^{-1}D^{-1} = L_i^TD^{-1}$ — calculating $D^{-1}$ requires $n$ divisions. Altogether, to perform the attack, Alice must perform $2n(n!)$ divisions and $n!$ matrix multiplications on the first run of the LP; and then a maximum of $2n(n!)$ divisions and $n!$ matrix multiplications on the second run of the LP (with the new objective function), since she can stop as soon as a matching $M_i$ is found. Further work would be required to test whether each $M_i$ matches any of the preceding set.

We remark that if $c$ is changed but $Q$ is reused, Alice’s workload will be considerably reduced. Rather than comparing her two sets of candidate $M$ matrices for a match, she can compare her two sets of candidate $Q$ matrices for a match. This saves her having to compute the $M_i$s. Since she knows the same $Q$ was used on both runs, a match between her two sets of candidate $Q$s will be the correct $Q$. Hence the parties are always advised to choose a new $Q$ each time they execute the transformation algorithm.

### 5.3 Other side issues to consider to ensure the claimed security-level

In Chapter 4, we claimed that Alice can do no better than to guess the correct $Q$ matrix, with probability $\frac{1}{n!}$, subject to the discussion in Section 5.2. However, to be sure that this security level is correct, there are a number of side issues that need to be considered. We must check that there is no other information Alice can exploit in her quest to deduce Bob’s data. Similarly, we must consider whether there is any way for Bob to attack Alice’s data.

The issues to consider are:

(i) The distribution from which Bob chooses his random elements for the $Q$ matrix, and why this should not be made known to Alice. Since we cannot use modulo arithmetic for optimisation, if Alice knows the range over which Bob is selecting his numbers, she can infer something about the elements of $Q$. 
Preliminary constraint checks: Bob needs to ensure that the units of his constraints are not obvious. If they are, he can fix this by conducting preliminary re-scaling. For example, if one of his constraints is in dollars, he should scale it by a large or small number so that the units will not signal to Alice that this could relate to his budget.

Exactly what information cannot be concealed by the protocol. That is, knowledge of which features of a linear program cannot be kept private using our transformation-based method.

Any attacks that Bob can mount against Alice.

Although at first glance these may seem like minor issues, in fact they are quite important. In security systems, it is rarely the cryptographic algorithm itself that is broken — the weakest links tend to be the very things that seem the most benign and inconsequential, such as where the keys for a cryptosystem are stored, or how the random seed for a generator is chosen. A system is only as secure as its weakest link.

We now consider each of the issues identified in turn.

### 5.3.1 Distribution of the random elements

It is very important that Alice is not privy to the distribution from which Bob draws the random elements of the $Q$ matrix. Let’s consider a simple uniform distribution to illustrate this.

Assume Bob is drawing the elements of $Q$ uniformly from the interval $[1, 1000]$, and that this range is also known to Alice. Suppose $Q = \begin{bmatrix} 300 & 0 \\ 0 & 40 \end{bmatrix}$. Alice receives

$$\hat{c}^T = c^T Q = \begin{bmatrix} 5 & 1 \end{bmatrix} \begin{bmatrix} 300 & 0 \\ 0 & 40 \end{bmatrix} = \begin{bmatrix} 1500 & 40 \end{bmatrix}.$$ 

Alice learns $\hat{c}$, and by comparing it to $c = \begin{bmatrix} 5 & 1 \end{bmatrix}$ she knows that the first element of $Q$ must be 300. There is no way the element ‘1’ of $c$ could have been mapped to 1500, since Alice knows Bob’s range terminated at 1000. Therefore ‘5’ must have been mapped to ‘1500’, and ‘1’ to ‘40’. Hence the alternative permutation of $Q$ (where the elements of $c$ are shuffled) can immediately be ruled out.

Simply by knowing the range of Bob’s random elements, Alice has been able to uniquely determine that $Q = \begin{bmatrix} 300 & 0 \\ 0 & 40 \end{bmatrix}$. She will be able to perform this trick whenever the
product of $c_1$ and $q_1$ is greater than 1000. This would occur for any $q_1 > 200$. However, if $q_1$ was, say, 100, Alice would not be able to decide which of the two permutations was correct.

In this example there were only 2 possible permutations to consider (since there were 2 variables) and Alice was able to rule one of them out altogether to determine $Q$ uniquely. Although the security breach may not always be as extreme as this, the example demonstrates the danger of Alice knowing the range of Bob’s random-number distribution.

We must also consider other distributional properties besides range. Suppose the random numbers are no longer uniform but are drawn from some other distribution. Knowing the distribution of the random elements can allow Alice to conclude that some permutations are more likely than others to be correct (as opposed to downright impossible, as was the case in the previous example), and hence weaken the $\frac{1}{n}$ security level. Although not as clear cut as the previous example, this type of inference can still weaken security.

In general, Bob must choose a distribution from which to generate his random numbers but must not tell Alice the distribution he used.

5.3.2 Preliminary constraint rescalings

Alice is going to learn the $b$ vector, which contains all of the right-hand-side values of Bob’s constraints, thus Bob must take responsibility for ensuring his data is in as meaningless a form as possible. It is advisable for Bob to perform some preliminary checks and rescaling of his constraints before the protocol commences, in order to render his constraints as meaningless as possible. For instance, Bob needs to scale the constraints away from any obvious units (e.g. dollars). To do this, Bob is free to multiply each of his constraints by whatever constant he desires.

5.3.3 Features that cannot be disguised

There are some features that cannot be disguised using our method. For instance, if Bob has two parallel constraint lines in his set, they will remain parallel after the $Q$ transform. This is because $Q$ scales the columns, and thus the initial ratio between elements in the same column remains the same since they are both multiplied by the same scalar. For example, suppose $M$ contains the following 2 parallel constraints:

$$M = \begin{bmatrix} 2 & 4 \\ 1 & 2 \end{bmatrix}.$$
Suppose $Q$ scales column 1 by $a$ and column 2 by $b$; we will get:

$$
\hat{M} = \begin{bmatrix}
2a & 4b \\
1a & 2b
\end{bmatrix}.
$$

After the scaling, the ratios between the rows remain the same so the lines are still parallel. Similarly, coincident lines will remain coincident. (The coincident case is easily fixed though, since if two constraints are coincident, Bob can simply remove one of them before beginning the protocol.)

The inability to disguise parallel lines is a fundamental problem which is not unique to this example. The disguise has to preserve certain properties of the LP in order for the raw and disguised problems to be equivalent. No disguise can avoid exposing parallel lines because to do so it would have to violate the equivalence between the raw and disguised problems.

It is important that these non-disguisable features be made explicit to the users, so that they are aware of them before they enter into a transaction involving this method.

### 5.3.4 Attacks Bob can mount against Alice

Repeated runs of LPs with the same objective function can sometimes give Bob enough information to discover $c$. If $c$ has $n$ elements, then $n$ runs of an LP with the same objective function can enable Bob to find $c$. On each run, Bob receives $x_i^*$ and $f_i^*$. If Alice and Bob solve two LPs in succession, each with the same $c$ vector but different $M$ matrices, Bob will obtain a second equation for $c$. That is, he will know

$$
c^T x_1^* = f_1^*
$$

$$
c^T x_2^* = f_2^*.
$$

If $c$ has two elements (i.e., $c = \begin{bmatrix} c_1 & c_2 \end{bmatrix}$), then Bob has two equations in two unknowns, which, provided they are linearly independent, he can solve uniquely for $c_1$ and $c_2$. Of course, this attack is only possible if the same objective function is used $n$ times with different constraint matrices, and provided that each run gives a different linearly independent optimal solution $x_i^*$.

This thesis is focusing on problems where $n$ is large; thus this attack is unlikely to be a problem. For example, if the LP has 20 variables, Alice and Bob would need to run it at least 20 times before Bob could have any chance of deducing Alice’s objective function.

A final avenue of attack is for Bob to manipulate his constraint set prior to the computation in order to gain information about Alice’s objective function. A detailed presentation of this attack is given in Chapter 9.
5.4 Conclusions

We have discussed the security level in more depth, including the impact of side issues and attacks Bob and Alice can mount against each other. The main points are:

- running the same LP multiple times with different objective functions but the same constraint matrix is only dangerous for small problems (with fewer than 10 variables). The attack by Alice requires $O(n!)$ work; thus once $n!$ is large, the attack is infeasible to execute.

- $n$ runs of an LP with the same objective function $c$ and linearly independent optimal solutions will enable Bob to find $c$ (Alice’s private data), where $n$ is the number of variables in the LP. This is unlikely to be a problem in practice, except possibly for very small $n$.

- Alice should never be told the distribution from which Bob draws the random elements of $Q$.

- Bob should rescale his constraints and right-hand-side constraint vector $b$ away from any obvious units. This is to protect the information in $b$ from Alice, as $b$ is not disguised by the transformation method.

In this chapter we have only considered one partitioning scenario, namely that the objective is held by one party, and the constraint set is held by the other party. There are many other variations we could consider. In the next chapter, we build on the ideas in Chapters 4 and 5 to develop a new transformation method for row-partitioned problems.
Chapter 6

Transformation Method for Row-Partitioned LPs

In Chapter 4, we developed a transformation method for the case when one party owns the objective, and the other owns the constraint matrix; but this is a very specific case. In this chapter we consider a more general case. We present a transformation method for S/R/R/\(\leq/N\) linear programs. The primary advantage of our method is that it allows both inequality constraints and non-negative variables in conjunction with row-partitioned constraints, a common combination in practice which was not solvable with any of the existing transformation methods. Further, we quantify the worst-case security in terms of the workload required to deduce Bob’s data. This chapter introduces the algorithm, and Chapter 7 analyses its security.

6.1 Background

As discussed in Chapters 2 and 3, the transformation methods are heavily dependent on the way the data is partitioned between the participants, the types of constraints and variables, and the output partitioning; the same disguise will not work for any arbitrary combination of these factors.

Recently, Mangasarian [86,87] proposed transformation methods for (i) column-partitioned data in the absence of non-negativity constraints [86]; and (ii) row-partitioned data in the absence of inequality constraints [87]. The inability to simultaneously handle inequality constraints and non-negativity constraints limits the classes of problems to which these methods can be applied, especially since real-world problems often require this combination of constraints. Our previous method in Chapter 4 could handle this combination, but only for \(O(A)/O(B)/O(B)\) LPs.

The existing transformation methods are summarised in Table 6.1, using the notation from Chapter 3. We can see that none of the existing transformation methods can
handle row-partitioned constraints (R) in conjunction with inequality constraints (≤) and non-negativity constraints (N). This case covers a large class of practical problems.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Type of LP (see Table 3.7, Ch. 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bednarz et al.</td>
<td>O(A)/O(B)/O(B)/≤/N/O(B)/P</td>
</tr>
<tr>
<td>Mangasarian [87]</td>
<td>P/R/R/=/N/P/P</td>
</tr>
<tr>
<td>Mangasarian [86]</td>
<td>C/C/P/≤/F/(R or P)/P</td>
</tr>
</tbody>
</table>

Table 6.1: Summary of existing transformation methods

6.2 Problem definition

Let Alice have \( m_1 \) constraints \( \overline{M}_1 \mathbf{x} \leq \overline{b}_1 \) and Bob have \( m_2 \) constraints \( \overline{M}_2 \mathbf{x} \leq \overline{b}_2 \), defined on the same set of \( n \) variables. We introduce the bar notation to distinguish Alice’s and Bob’s actual constraints from their total constraint matrices, \( M_1 \) and \( M_2 \), which are both of size \( m \times n \), where \( m = m_1 + m_2 \) is the total number of constraints. The implicit assumption is that each party contributes at least one constraint (i.e., \( m_1, m_2 \geq 1 \)), otherwise the problem would no longer be row-partitioned, and would fall under a different scenario.

We propose a new transformation approach for securely solving Program A, below.

\[
\text{Program A}

\begin{align*}
\max & \quad f(\mathbf{x}) = (c_1 + c_2)^T \mathbf{x} \\
\text{s.t.} & \quad (M_1 + M_2) \mathbf{x} \leq \mathbf{b}_1 + \mathbf{b}_2 \\
& \quad \mathbf{x} \geq \mathbf{0},
\end{align*}
\]

(6.2.1)

where Alice owns

\[
M_1 = \begin{bmatrix} \overline{M}_1 \\ \cdots \\ \overline{0} \end{bmatrix}, \quad \mathbf{b}_1 = \begin{bmatrix} \overline{b}_1 \\ \cdots \\ \overline{0} \end{bmatrix} \quad \text{and} \quad \mathbf{c}_1;
\]

and Bob owns

\[
M_2 = \begin{bmatrix} \overline{0} \\ \cdots \\ \overline{M}_2 \end{bmatrix}, \quad \mathbf{b}_2 = \begin{bmatrix} \overline{0} \\ \cdots \\ \overline{b}_2 \end{bmatrix} \quad \text{and} \quad \mathbf{c}_2.
\]

The objective vector \( \mathbf{c}_1 + \mathbf{c}_2 \) is shared between Alice and Bob. It is imperative that the objective function be genuinely shared, not merely column-partitioned. That is, we must have \( \mathbf{c}_1^T = (c_{11}, c_{12}, \ldots, c_{1n}) \) and \( \mathbf{c}_2^T = (c_{21}, c_{22}, \ldots, c_{2n}) \), where there must be no known structure behind any zeros in the objective function components. It is also possible to have the objective function owned entirely by Bob (\( \mathbf{c}_1 = \mathbf{0} \)) or Alice (\( \mathbf{c}_2 = \mathbf{0} \)). These are discussed as special cases in Section 6.3.2.
6.3 Solution approach

To solve Program A, we apply some techniques from papers by Du and Atallah [50] and Yang et al. [131]. These papers consider how to solve a system of linear equations securely, using a transformation-based approach. Bob chooses two invertible random matrices, $K (m \times m)$ and $Q^{-1} (n \times n)$. The following disguise is applied:

$$(M_1 + M_2)x = b_1 + b_2 \iff K(M_1 + M_2)QQ^{-1}x = Kb$$

where we can see that the two equations are equivalent, since inserting $QQ^{-1}$ has no impact, and multiplying both sides by $K$ will not change the solution. A change of variables is performed by setting $\hat{x} = Q^{-1}x$, and if we let $\hat{M} = K(M_1 + M_2)Q$ and $\hat{b} = Kb$, we can simplify the disguised system to $\hat{M}\hat{x} = \hat{b}$. Alice solves this for $\hat{x}$, and Bob transforms the solution back to the original space, using his knowledge of $Q^{-1}$.

When we move from solving a system of linear equations to solving a linear program, a number of changes must be made. We must first convert Program A into equality form by adding $m$ slack variables $s$ to give:

Program A – Equality Form

$$\begin{align*}
\max f(x) &= \left[(c_1 + c_2)^T, 0^T\right]\begin{bmatrix} x \\ s \end{bmatrix} \\
\text{s.t.} &\begin{bmatrix} M_1 + M_2 & I \end{bmatrix}\begin{bmatrix} x \\ s \end{bmatrix} = b_1 + b_2 \\
&\begin{bmatrix} x \\ s \end{bmatrix} \geq 0.
\end{align*}$$

We introduce new notation for the equality form. Let

$$(d_1 + d_2)^T = \left[(c_1 + c_2)^T, 0^T\right],$$

where $d_1^T = \left[c_1^T, 0^T\right]$ and $d_2^T = \left[c_2^T, 0^T\right]$. Similarly, let

$$N_1 + N_2 = \begin{bmatrix} M_1 + M_2 & I \end{bmatrix},$$

where $N_1 = \begin{bmatrix} M_1 & I_{m_1} & 0 \\ 0 & 0 & 0 \end{bmatrix}$ and $N_2 = \begin{bmatrix} 0 & 0 & 0 \\ M_2 & 0 & I_{m_2} \end{bmatrix}$. From this point forward, for simplicity, $[x \ s]^T$ will simply be denoted by $x$, which is now of size $(n + m) \times 1$.

As mentioned, naively inserting slack variables can cause problems. We will discuss these problems via an example, and fix them later on by strengthening the disguise.
With the system in equality form, we are now free to multiply both sides of the constraints by a random matrix without worrying about disrupting any inequalities. However, we still face a complication due to the non-negativity constraints. We will have to multiply the non-negativity constraints by $Q^{-1}$ to be consistent with the change of variables ($\hat{x} = Q^{-1}x$). However, as discussed in [11] and Chapter 4 of this thesis, the constraint $Q^{-1}x \geq 0$ does not guarantee that $QQ^{-1}x \geq 0$ (i.e. that $x$ itself is $\geq 0$) since $Q^{-1}$ or $Q$ may contain negative elements which could reverse some of the inequalities.

Therefore we must place a restriction on matrix $Q$ to ensure the non-negativity constraints are not disrupted. We need both $Q$ and $Q^{-1}$ to contain strictly non-negative elements. To achieve this, we require $Q$ to be a generalised permutation matrix, as discussed in Chapter 4. Such a matrix is the product of a diagonal scaling matrix $D$ and a permutation matrix $L$. These matrices are the only matrices that have both $Q^{-1}$ and $Q$ strictly non-negative [70]. This is the only class of matrices that ensures that $\hat{x} = Q^{-1}x \geq 0$ is equivalent to $x \geq 0$.

The disguise imparted by $Q = DL$ consists of a scaling (from $D$) and a shuffle (from $L$). If we post-multiply a matrix by $Q$, the effect would be to scale and shuffle its rows. If we pre-multiply a matrix by $Q$, the effect would be to scale and shuffle its columns.

The drawback is that generalised permutation matrices only provide a ‘low-grade’ disguise — they merely scale and shuffle the columns/rows of a matrix. If we force $Q$ to be a generalised permutation matrix, we see that it will not obscure the data as much as desired. Let $P = (N_1 + N_2)Q$. Therefore, $P$ is merely $N_1 + N_2$ with its columns scaled and shuffled. This level of disguise is not sufficient, as illustrated by the following scenario.

Suppose $N_1 + N_2 = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 3 & 4 & 0 & 1 \end{bmatrix}$, where Alice has $N_1 = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$, and Bob has $N_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 3 & 4 & 0 & 1 \end{bmatrix}$. Suppose Bob chooses

$$Q = DL = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 \end{bmatrix},$$

which gives

$$Q^{-1} = \begin{bmatrix} \frac{1}{5} & 0 & 0 & 0 \\ 0 & \frac{1}{6} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{2} & 0 \end{bmatrix}.$$
Alice will receive $P$, where

\[ P = (N_1 + N_2)Q \]

\[ = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 3 & 4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 \end{bmatrix} \]

\[ = \begin{bmatrix} 5 & 12 & 0 & 2 \\ 15 & 24 & 3 & 0 \end{bmatrix}. \]

We see from this result that $Q$ has scaled the columns of $(N_1 + N_2)$ and interchanged the last two columns. The weak point of this approach is that it does not remove the structure in the slack variables. By observing how the slack variables have deviated from their original (known) identity-matrix layout, Alice can deduce part of the permutation applied by $Q$, and the corresponding scale factors. In this case she can deduce that columns 3 and 4 have been interchanged, and thus the last two columns of $Q$ must be:

\[
\begin{bmatrix}
\cdots & 0 & 0 \\
\cdots & 0 & 2 \\
\cdots & 3 & 0
\end{bmatrix}.
\]

The non-slack columns are not as straightforward: the best Alice can do is consider every possible permutation of the first two columns, and then match up the columns of $P$ with the columns of her matrix $N_1$ to deduce the corresponding scale factors of $Q$. This will leave her with a set of candidate $Q$s. There are $n$ non-slack variables; hence Alice must consider $n!$ permutations.

In our example there are 2 non-slack variables and thus 2 permutations to consider. Alice knows that the first two columns of $L$ are either:

a) \[ L = \begin{bmatrix}
1 & 0 & \cdots \\
0 & 1 & \cdots \\
0 & 0 & \cdots \\
0 & 0 & \cdots
\end{bmatrix}, \]

or b) \[ L = \begin{bmatrix}
0 & 1 & \cdots \\
1 & 0 & \cdots \\
0 & 0 & \cdots \\
0 & 0 & \cdots
\end{bmatrix}. \]

If Alice starts by assuming permutation a), and compares the first column of $P$ with the first column of $N_1$, then the first column must have been scaled by 5, and the second
column by 6. In total, this gives her a hypothetical $Q$ matrix, $Q_a$, and its inverse:

$$
Q_a = \begin{bmatrix}
5 & 0 & 0 & 0 \\
0 & 6 & 0 & 0 \\
0 & 0 & 0 & 2 \\
0 & 0 & 3 & 0
\end{bmatrix}
$$

and

$$
Q_a^{-1} = \begin{bmatrix}
\frac{1}{5} & 0 & 0 & 0 \\
0 & \frac{1}{6} & 0 & 0 \\
0 & 0 & 0 & \frac{1}{3} \\
0 & 0 & \frac{1}{2} & 0
\end{bmatrix}.
$$

If Alice assumes permutation b), she compares the first column of $P$ with the second column of $N_1$ (since the order of columns has been switched), and hence reasons that the second column has been scaled by $\frac{5}{2}$. She then compares the second column of $P$ with the first column of $N_1$, and reasons that the first column has been scaled by 12. In total, this gives her:

$$
Q_b = \begin{bmatrix}
0 & 12 & 0 & 0 \\
\frac{5}{2} & 0 & 0 & 0 \\
0 & 0 & 0 & 2 \\
0 & 0 & 3 & 0
\end{bmatrix}
$$

and

$$
Q_b^{-1} = \begin{bmatrix}
0 & \frac{5}{2} & 0 & 0 \\
0 & \frac{12}{5} & 0 & 0 \\
0 & 0 & 0 & \frac{1}{3} \\
0 & 0 & \frac{1}{2} & 0
\end{bmatrix}.
$$

From the candidate $Q$s, Alice creates a set of candidate $Q^{-1}$s, and then candidate $N$s (where $N = N_1 + N_2$) by post-multiplying $P$ by the $Q^{-1}$. Her candidates are:

$$
N_a = P(Q_a^{-1}) = \begin{bmatrix}
1 & 2 & 1 & 0 \\
3 & 4 & 0 & 1
\end{bmatrix},
$$

$$
N_b = P(Q_b^{-1}) = \begin{bmatrix}
1 & 2 & 1 & 0 \\
2 & 6 & 0 & 1
\end{bmatrix}.
$$

Alice cannot tell which of these is the true $N$, but she could take a guess. This means that Bob’s data is protected up to a level of 1 out of $n!$. However, this can be improved.

### 6.3.1 A second layer of disguise: matrix $K$

If Alice receives $P$ she can find $n!$ possibilities for Bob’s matrix $N_2$. To combat this problem, a second layer of disguise is desirable. If Alice instead receives $KP$, where $K$ is a random $m \times m$ matrix chosen by Bob, she can no longer perform the attack just described. The multiplication by $K$ has the further advantage of protecting the $b_1 + b_2$ values. The disguised program, Program B, is as follows.

**Program B**

$$\begin{align*}
\text{max} & \quad f(x) = \tilde{d}^T \tilde{x} \\
\text{s.t.} & \quad \tilde{N} \tilde{x} = \tilde{b} \\
& \quad \tilde{x} \geq 0,
\end{align*}$$

where $\tilde{N} = K(N_1 + N_2)Q$, $\tilde{b} = K(b_1 + b_2)$, $\tilde{d}^T = d^T Q$, and $\tilde{x} = Q^{-1} x$.

The disguise applied to the main constraint matrix is very similar to Du and Atallah’s [50] disguise for linear equations. However, we have shown why the two layers of dis-
guise ($K$ and $Q$) are necessary for the row-partitioned case, and we have also derived the conditions needed for the matrix $Q$ to maintain equivalence between Program A and Program B in the presence of non-negativity constraints.

Let’s return to our example and introduce $K$, a random $m \times m$ invertible matrix chosen by Bob, where $K$ can contain negative elements. Suppose Bob chooses:

$$K = \begin{bmatrix} -5 & 4 \\ 8 & -9 \end{bmatrix}.$$

Also, $Q$ is as before, namely,

$$Q = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}.$$

Alice receives

$$K(N_1 + N_2)Q = \begin{bmatrix} 35 & 36 & 12 & -10 \\ -95 & -120 & -27 & 16 \end{bmatrix}.$$

We can see that the locations of the slack variables have been hidden. Alice can no longer find a set of candidate $Q$s from this. Alice also receives $Kb$, where:

$$Kb = \begin{bmatrix} -5 & 4 \\ 8 & -9 \end{bmatrix} \begin{bmatrix} 6 \\ 12 \end{bmatrix} = \begin{bmatrix} 18 \\ -60 \end{bmatrix}.$$

The two layers of disguise are applied simultaneously, using a privacy-preserving matrix product protocol similar to the one in [50]. The calculation of $Kb$ and $d^TQ$ are done similarly. We have modified the protocol from [50] to make it more efficient, but as the main focus is on the disguise process, we defer the details of the sub-protocols until Section 6.5. We now describe the complete algorithm for the transformation method.

### 6.3.2 The transformation method

Protocol 6.1 gives the core method of this chapter, designed for linear programs of type ‘{$S$ or $O(B)$}/R/R/≤/N’ (see Chapter 3 for notation). The protocol requires three sub-protocols, given in Section 6.5. In addition, the protocol offers three output options for the optimal solution $x^*$.

**Protocol 6.1.** (Transformation method for row-partitioned LPs)

**Input:** Alice has an $n \times 1$ vector of objective function coefficients $c_1$, an $m_1 \times n$ matrix of constraint coefficients $M_1$, and corresponding RHS vector $b_1$. Bob has an $n \times 1$ vector of objective function coefficients $c_2$, an $m_2 \times n$ matrix of constraint coefficients $M_2$, and corresponding RHS vector $b_2$. Let $m_1 + m_2 = m$, which is known to both parties, and $m_1, m_2 \geq 1$. They wish to solve Program A. If $c_1 = 0$ or $c_2 = 0$, refer to ‘Special Case’ at the end of the protocol.
Output: Alice and Bob both receive $f^*$, the optimal value of Program A. There are three options for $x^*$, the optimal solution of Program A:

- (Option 1) Only Bob receives $x^*$.
- (Option 2) Alice receives $x_A^*$ and Bob receives $x_B^*$, such that $x_A^* + x_B^* = x^*$.
- (Option 3) Both parties receive $x^*$.

Protocol:

1. Alice and Bob convert the system to equality form by adding slack variables to their constraint matrices, to give

$$N_1 + N_2 = \begin{bmatrix} M_1 & I_{m_1} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ M_2 & I_{m_2} \end{bmatrix}.$$  

They also add $m$ zeros to $c_1$ and $c_2$ for the slack variable coefficients, to give:

$$(d_1 + d_2)^T = \begin{bmatrix} c_1^T; 0^T \\ c_2^T; 0^T \end{bmatrix}.$$  

2. Bob generates a random positive $(m+n) \times 1$ vector $r$, which he sets as a diagonal to form the scaling matrix $D$, and a random $(m+n) \times (m+n)$ permutation matrix $L$, which he multiplies together to form $Q = DL$. Note that the random numbers in $r$ can be taken from any distribution, and are not restricted to integers only.

3. Bob generates a random $m \times m$ invertible matrix $K$.

4. Using Protocol 6.2 (Section 6.5.1), Alice and Bob compute $\hat{N} = K(N_1 + N_2)Q$ so that only Alice learns $\hat{N}$.

5. Using Protocol 6.3 (Section 6.5.2), Alice and Bob compute $\hat{b} = K(b_1 + b_2)$ so that only Alice learns $\hat{b}$.

6. Using Protocol 6.4 (Section 6.5.3), Alice and Bob compute $\hat{d}^T = (d_1 + d_2)^T Q$ such that only Alice learns $\hat{d}^T$.

7. Alice now has all the components of the transformed problem, Program B. Alice solves the transformed problem to find $\hat{x}^*$ and the corresponding objective value $\hat{f}^*$. Note that $\hat{f}^* = f^*$, since $\hat{d}^T \hat{x}^* = d^T Q Q^{-1} x^* = d^T x^*$.

8. (Option 1) Alice sends both $\hat{x}^*$ and $\hat{f}^*$ to Bob. Bob untransforms $\hat{x}^*$ to obtain $x^* = Q \hat{x}^*$.

(Option 2) Alice sends only $\hat{f}^*$ to Bob and withholds $\hat{x}^*$. Alice and Bob conduct a secure multiplication to compute $x^* = Q \hat{x}^* := x_A^* + x_B^*$, such that Alice owns $x_A^*$ and Bob owns $x_B^*$. This can be done using the matrix multiplication protocol.
given in Han et al. [66], or any other available algorithm that provides the output in shares.

(Option 3) As per Option 1, except Bob sends $x^*$ to Alice at the end.

Special Case:

i) Entire objective owned by Bob (the party who owns the objective must play the role of Bob). Run the protocol as normal, except no secure multiplication is needed in step 6). Bob computes $d^TQ$ locally and sends it to Alice.

6.3.2.1 Small example

To demonstrate the complete algorithm, consider the following optimisation problem, where $n = 2$ and $m = 2$:

$$
\begin{align*}
\text{max} \quad & f(x) = 5x_1 + x_2 \\
\text{s.t.} \quad & x_1 + 2x_2 \leq 6 \quad \text{(Alice’s)} \\
& 3x_1 + 4x_2 \leq 12 \quad \text{(Bob’s)} \\
& x_1, x_2 \geq 0,
\end{align*}
$$

(6.3.1)

To this we add slack variables $x_3$ and $x_4$ and re-write the LP in matrix form.

$$
\begin{align*}
\text{max} \quad & d^T x = \begin{bmatrix} 5 & 1 & 0 & 0 \end{bmatrix} x \\
\text{s.t.} \quad & \begin{bmatrix} 1 & 2 & 1 & 0 \\ 3 & 4 & 0 & 1 \end{bmatrix} x = \begin{bmatrix} 6 \\ 12 \end{bmatrix} \\
& x \geq 0,
\end{align*}
$$

(6.3.2)

where Alice owns

$$
N_1 = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, b_1 = \begin{bmatrix} 6 \\ 0 \end{bmatrix}, \text{ and } d_1^T = \begin{bmatrix} 3 & \frac{1}{2} & 0 & 0 \end{bmatrix};
$$

and Bob owns

$$
N_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 3 & 4 & 0 & 1 \end{bmatrix}, b_2 = \begin{bmatrix} 0 \\ 12 \end{bmatrix}, \text{ and } d_2^T = \begin{bmatrix} 2 & \frac{1}{2} & 0 & 0 \end{bmatrix}.
$$

The optimal solution for Program A is $x^* = (4, 0, 2, 0)$, with objective function $f^* = 20$.

Bob chooses a non-negative generalised permutation matrix of size $(n + m) \times (n + m)$ for $Q$ and a random $m \times m$ matrix for $K$. Let $Q$ and $K$ be as in Section 6.3.1.

Alice and Bob compute $\hat{d}^T = d^T Q = \begin{bmatrix} 25 & 6 & 0 & 0 \end{bmatrix}$ such that only Alice receives the result. Alice also receives $K(N_1 + N_2)Q$ and $Kb$ (as given in Section 6.3.1) and then
Chapter 6. Transformation Method for Row-Partitioned LPs

solves Program B:

$$\begin{align*}
\max & \quad \mathbf{d}^T \mathbf{x} = [25 \ 6 \ 0 \ 0] \mathbf{x} \\
\text{s.t.} & \quad \begin{bmatrix} 35 & 36 & 12 & -10 \\ -95 & -120 & -27 & 16 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 18 \\ -60 \end{bmatrix} \\
& \quad \mathbf{x} \geq 0,
\end{align*}$$

(6.3.3)

which has solution $\mathbf{x}^* = (\frac{8}{10}, 0, 0, 1)^T$ and $f^* = 20$.

We emphasise that Alice is free to solve Program B using whichever algorithm best suits the nature of the problem — she is free to choose Simplex, Interior Point, or any other linear programming method. This is not possible in the current cryptographic methods, which tie the user to Simplex.

Alice sends $\mathbf{x}^*$ and $f^*$ to Bob, who applies $Q$ to reveal $x^*$, where

$$x^* = Q \mathbf{x}^* = (4, 0, 2, 0)^T.$$

(6.3.4)

The security level differs depending on whether or not Bob tells Alice $x^*$. This is discussed in detail in Chapter 7.

6.4 Complexity

Here we analyse the overhead of our method. The complexity is obviously dependent on the algorithm chosen by the user to solve the disguised LP (Simplex, Interior Point, etc.). The user is free to choose the most efficient method for their particular disguised problem, so we focus here on the cost of applying the transformation.

In contrast to the cryptographic methods of [25, 79, 119], the transformation protocols are only run once, at the start of the algorithm, rather than on every iteration. This dramatically reduces the computation and communication cost.

The transformation stage requires three sub-protocols. In each sub-protocol, $p$ and $d$ are security parameters such that Bob has a $1$ in $p^d$ chance of guessing Alice’s data; $p$ and $d$ are chosen to make this chance acceptably low. Each protocol has a linear algebra overhead, and then a cryptographic component, based on the 1-out-of-$N$ oblivious transfer protocol ($N$ is equal to $p$ for our protocols). The bit length, $k$, of all values is a constant and is thus ignored. Naor and Pinkas [92] give a 1-out-of-$N$ oblivious transfer protocol that requires $\log N$ exponentiations, with a communication overhead of $O(N)$.

Each sub-protocol requires $d$ rounds; on each round a 1-out-of-$p$ oblivious transfer is conducted. We state the complexity of each sub-protocol in turn.
Protocol 6.2 securely evaluates $K(N_1 + N_2)Q$. In total, $pd[m^2(n+m) + m(n+m)^2]$ multiplications are needed. Each round of $d$ needs a 1-out-of-$p$ oblivious transfer on each element of a matrix of size $m \times (n+m)$. Protocol 6.3 securely evaluates $K(b_1 + b_2)$. The algorithm requires $pdm^2$ multiplications. One 1-out-of-$p$ oblivious transfer is required on each of $d$ rounds on vectors of size $m$. Protocol 6.4 securely evaluates $(d_1 + d_2)^TQ$. The algorithm requires $pd(n+m)^2$ multiplications. Once again we need $d$ 1-out-of-$p$ oblivious transfers, this time on vectors of size $(n+m)$. Note that choosing the shared output option (Option 2) adds one further secure matrix multiplication — the protocol from [66] requires $(n+m)$ invocations of a secure scalar product protocol (e.g. [61]).

By contrast, Toft’s secure Simplex method [119] requires $O(m(m+n))$ secure multiplications and $O(m)$ secure comparisons to be executed on every pivot. In the worst case, the Simplex method takes an exponential number of pivots [76]; thus Toft’s algorithm can be impractical.

### 6.5 Detailed protocols

The following protocols have been adapted from [50,131], with an important modification to improve their efficiency. In the original protocols, $p$ and $d$ are security parameters and are chosen so that the cost of breaking the security is computationally infeasible. This could make the algorithm prohibitively expensive. We instead require that $p$ and $d$ be chosen so that the chance of Bob guessing Alice’s data is acceptably low. In addition, we modify the protocol so that even if Bob evaluates all $pd$ possibilities for Alice’s matrix, they will all be equally likely from his point of view. Under the protocols from [50,131], if Alice’s matrix had a known structure of zeros, the $pd$ possibilities would not be equally likely — the correct matrix would be obvious.

#### 6.5.1 Secure evaluation of $\hat{N}$

This protocol is adapted from Yang et al. [131] and Du and Atallah [50]. Since we are dealing with row-partitioned matrices, which have a specific structure, we modify the protocol to ensure that the structure cannot be easily exploited by Bob. The main change is to truncate the zeros and the identity structure from $N_1$ before we begin. Alice transmits shares of the truncated $N_1$, and we have Bob ‘fill in’ the known structures himself before performing the calculations.

**Protocol 6.2.** (Compute $\hat{N} = K(N_1 + N_2)Q$) [adapted from [50,131]]

**Input:** Alice has a private $m_1 \times n$ matrix $\overline{M_1}$, stored in $N_1$; Bob has a private $m_2 \times n$ matrix $\overline{M_2}$, stored in $N_2$, a random $m \times m$ matrix $K$, and a $(n+m) \times (n+m)$ generalised
permutation matrix $Q$. Matrices $N_1$ and $N_2$ have the following structure:

$$N_1 = \begin{bmatrix} M_1 & I_{m_1} & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad N_2 = \begin{bmatrix} 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ M_2 & I_{m_2} & 0 \end{bmatrix}$$

Shares of $M_1$ will be denoted $X_i$, $i = 1, \ldots, d$.

**Output:** Alice receives an $m \times (n + m)$ matrix $\hat{N} = K(N_1 + N_2)Q$. Bob receives no output.

**Protocol:**

i) Alice and Bob agree on two integers $p$ and $d$, such that the $1$ in $p^d$ chance of Bob guessing Alice’s data is acceptably low.

ii) Alice generates random matrices $\bar{X}_1, \ldots, \bar{X}_{d-1} \in \mathcal{H}(m_1 \times n)$, where $\mathcal{H}$ is a distribution chosen by Alice, and sets

$$\bar{X}_d = M_1 - \sum_{i=1}^{d-1} \bar{X}_i.$$  

iii) Bob generates random matrices $Y_1, \ldots, Y_{d-1} \in \mathcal{H}(m_2 \times n)$, and sets

$$Y_d = N_2 - \sum_{i=1}^{d-1} Y_i.$$  

iv) For each $j = 1, \ldots, d$, Alice and Bob perform the following sub-steps:

(a) Alice generates $p - 1$ random matrices $G^{(j)}_i \sim \mathcal{H}(m_1 \times n)$, and sends the following sequence to Bob:

$$\langle \overline{H}_1^{(j)}, \ldots, \overline{H}_p^{(j)} \rangle$$

where $\overline{H}_k^{(j)} = \overline{X}_j$, and $k \in [1, p]$ is a secret random number known only by Alice, and the $G^{(j)}_i$’s are assigned to the remaining $(p - 1)$ $\overline{H}_i^{(j)}$’s, $i \neq k$.

(b) Bob “fills out” $\overline{H}_i^{(j)}$ to create $H_i^{(j)}$.

He does this by appending the fraction $\frac{1}{d}$ of an $m_1 \times m_1$ identity matrix and an $m_1 \times m_2$ zero matrix to $\overline{H}_i^{(j)}$, and inserting a $m_2 \times (m + n)$ zero matrix below $\overline{H}_i^{(j)}$, i.e.

$$H_i^{(j)} = \begin{bmatrix} \overline{H}_i^{(j)} & I_{m_1/d} & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$  

In each case, Bob then computes

$$Z_i^{(j)} = K(H_i^{(j)} + Y_j)Q + R_j$$

for $i = 1, \ldots, p$, where $R_j \in \mathcal{H}(m \times (n + m))$ is a random matrix.
(c) Using the 1-out-of-$N$ Oblivious Transfer protocol, Alice gets back the result of

$$Z_k^{(j)} = K(H_k^{(j)} + Y_j)Q + R_j$$
$$= K(X_j + Y_j)Q + R_j,$$

where $H_k^{(j)}$ is denoted by $X_j$.

v) Bob sends $\sum_{j=1}^{d} R_j$ to Alice.

vi) Alice computes

$$\hat{N} = \sum_{j=1}^{d} Z_k^{(j)} - \sum_{j=1}^{d} R_j$$
$$= \sum_{j=1}^{d} (K(X_j + Y_j)Q + R_j) - \sum_{j=1}^{d} R_j$$
$$= K(N_1 + N_2)Q.$$

6.5.2 Secure evaluation of $\hat{b}$

Protocol 6.3. (To compute $\hat{b} = K(b_1 + b_2)$)

**Input:** Alice has an $m_1$-dimensional vector $b_1$; Bob has an $m_2$-dimensional vector $b_2$, where

$$b_1 + b_2 = \begin{bmatrix} b_1 \\ \vdots \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \vdots \\ b_2 \end{bmatrix}.$$

and Bob also has a random $m \times m$ matrix $K$.

**Output:** Alice receives $\hat{b} = K(b_1 + b_2)$. Bob receives no output.

**Protocol:** This protocol is a special case of Protocol 6.2, with $Q = I$. Also, the ‘filling out’ procedure is simpler since we don’t need to augment the vector with an identity component. Bob only needs to add $m_2$ zeros to the shares of $b_1$.

6.5.3 Secure evaluation of $\hat{d}$

Protocol 6.4. (To compute $\hat{d}^T = (d_1 + d_2)^T Q$)

**Input:** Alice has an $(n + m)$-dimensional vector $d_1$; Bob has an $(n + m)$-dimensional vector $d_2$, and a random $(n + m) \times (n + m)$ matrix $Q$.

**Output:** Alice receives $\hat{d}^T = (d_1 + d_2)^T Q$. Bob receives no output.

**Protocol:** This protocol is a special case of Protocol 6.2, with $K = I$. Let $c_1$ be $d_1$ with the $m$ trailing zeros corresponding to the slack variables truncated. The ‘filling out’ procedure is simpler, since Bob only needs to add $m$ zeros to the shares of $c_1$. 
6.6 Conclusion

We have presented a transformation method which can handle both inequality constraints and non-negative variables in conjunction with row-partitioned constraints. Although we have detailed the protocols and the method’s overall complexity, we have yet to analyse its security in detail. In the next chapter, we consider several attacks that Alice can perform in an attempt to deduce Bob’s data, and show that the associated workload is infeasible.
Chapter 7

Security Analysis: Attacks and Workloads

This chapter presents the security analysis for the transformation method developed in Chapter 6. Despite the simplicity of the transformation, the security analysis is quite involved. Altogether, we isolate three separate attacks that Alice can perform in an attempt to deduce Bob’s data. In particular, we consider Alice’s potential to ‘invert’ the disguise via a system of non-linear equations. We show that whenever Alice owns more than one constraint she can deduce Bob’s constraint matrix with certainty.

However, the good news is that Alice’s workload increases factorially with the size of the problem. For even a modest-sized problem with 30 variables and 6 constraints, the attacks would take of the order of 10 million years to execute! Our transformation method thus relies on a ‘workload defence’. This is the same type of defence protecting public key cryptography.

7.1 Introduction

As we did in Chapter 5, we will follow Du and Zhan’s acceptable security paradigm [53], and explicitly specify the security level in terms of the probability of an adversary learning a party’s raw data. This time we will also consider the associated workload to do so. All of the analysis in this chapter is with reference to Protocol 6.1 from Chapter 6.

First, we consider the case when Alice does not receive the optimal solution $x^*$. Alice’s goal is to ‘invert’ the disguise to uncover Bob’s private matrix $M_2$. We analyse the information available to Alice in the transformed problem, and show how she can derive a set of possible permutations for the disguising matrix $Q$, and, for each of these, set up systems of non-linear equations to obtain candidates for Bob’s matrix. This will be termed the ‘non-linear-algebra attack’.
Further, whenever Alice owns more than one constraint, we show that she can perform a ‘consistency-check’ attack in conjunction with the non-linear-algebra attack. This will allow her to deduce Bob’s constraints exactly, by enabling her to pinpoint which system of equations is correct. This may appear to spell the end of our transformation method. However, this is only half the story: we also need to consider Alice’s workload in performing these attacks. As we show in Section 7.7, her workload rapidly becomes prohibitively large as the number of variables, \( n \), and the number of constraints, \( m \), grow. Thus our method offers ‘workload’ protection — Alice can find Bob’s data, but she will have to do a prohibitively large amount of work (i.e., solve many systems) to do so.

Second, we consider the case when Alice does receive the optimal solution \( x^* \). In addition to the non-linear-algebra and consistency-check attacks, Alice can compare \( x^* \) with the transformed solution \( \hat{x}^* \) to reduce the set of candidate permutations she needs to consider. We will refer to this as the ‘permutation-reduction’ attack. Thus her workload will be slightly reduced under this scenario, but still becomes impractical very quickly.

The security afforded by our approach is satisfactory for large problems. More importantly, the workload is quantifiable — the user is able to calculate the number of sets of equations Alice potentially has to solve, and judge whether or not this risk is acceptable to them.

In summary, Alice’s attack on the protocol consists of a number of stages:

i) Derive a list of candidate permutations.

ii) For each candidate permutation, perform the following attacks:

(a) Non-Linear-Algebra Attack

(b) Consistency-Check Attack (if Alice owns more than one constraint)

(c) Permutation-Reduction Attack (if Alice receives \( x^* \)).

These steps are discussed in more detail in the following sections. We divide our analysis into two cases, according to whether or not Alice receives the optimal solution \( x^* \).

### 7.2 Case 1: Alice does not receive \( x^* \)

Recall the linear program defined in Program A (6.2.1), and the corresponding transformation algorithm given in Protocol 6.1. We begin by analysing security for output options 1 and 2, which are when Alice either does not receive \( x^* \) at all, or receives a share \( x^*_A \) of \( x^* \). These options are equivalent, because knowing a share \( x^*_A \) (where \( x^* = x^*_A + x^*_B \)) is equivalent to Alice knowing none of \( x^* \), in terms of the attacks she can perform. This will become clear as we progress.
7.2.1 Information deducible from the transformation

Consider the transformed LP, Program B, from Alice’s point-of-view:

\[
\begin{align*}
\text{Program B} \\
\text{max } f(x) &= \hat{d}^T \hat{x} \\
\text{s.t. } \hat{N} \hat{x} &= \hat{b} \\
\hat{x} &\geq 0,
\end{align*}
\]

where \( \hat{N} = K(N_1 + N_2)Q \), \( \hat{b} = K(b_1 + b_2) \), \( \hat{d}^T = (d_1 + d_2)^T Q \), and \( \hat{x} = Q^{-1} x \).

Under our transformation scheme (Protocol 6.1), zeros in a vector or matrix are preserved, since the transformation matrix \( Q = DL \) only scales and shuffles the columns. (Recall that \( D \) is an \((m + n) \times (m + n)\) positive diagonal matrix, and \( L \) is an \((m + n) \times (m + n)\) permutation matrix, where \( m \) is the number of constraints.) Therefore, knowledge of the transformed objective \( \hat{d}^T = (d_1 + d_2)^T Q \) allows Alice to discover something about the permutation applied by \( Q \). This is because the slack variables have a zero coefficient in the objective function and thus ‘give themselves away’. If there are \( n \) variables and \( m \) slack variables, then comparing the locations of the zeros in \( d \) and \( \hat{d} \) enables Alice to narrow down the possible permutations to \( n!m! \) possibilities, rather than \((n + m)!\) had there been no zero coefficients present in \( d \). However, she cannot deduce the corresponding scale factors, since she either only knows a share of \( d \) (if she knows \( d_1 \)) or none of \( d \) (if Bob owns the entire objective).

For instance, in the example problem from Section 6.3.2.1, Alice knows that

\[
d^T = \begin{bmatrix} * & * & 0 & 0 \end{bmatrix} \quad \text{and} \quad \hat{d}^T = \begin{bmatrix} * & * & 0 & 0 \end{bmatrix},
\]

and reasons that either the first two columns were switched, or remained the same, and likewise the last two columns were either switched, or remained the same. This gives her four possible permutations for \( L \):

\[
\begin{align*}
[1 \ 2 \ 3 \ 4], & \quad [1 \ 2 \ 4 \ 3], & \quad [2 \ 1 \ 4 \ 3], & \quad [2 \ 1 \ 3 \ 4],
\end{align*}
\]

where the notation \([1 \ 2 \ 4 \ 3]\) is interpreted as ‘element 1 remains in position 1, element 2 remains in position 2; element 4 moves to position 3; element 3 moves to position 4’. That is, the notation we use for permutations lists the permuted order of the elements, with respect to their original ordering of \( 1, \ldots, (n + m) \). For example,

- the permutation \([1 \ 2 \ 3 \ 4]\) is equivalent to \( L = I \);

- the permutation \([2 \ 1 \ 4 \ 3]\) is equivalent to \( L = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \).
Once Alice has narrowed down the set of permutations, she can perform a non-linear-algebra attack for each candidate $L$.

### 7.3 Non-linear-algebra attack

For each of the $n!m!$ possible permutations, Alice creates a system of equations which, although non-linear, can be carefully solved to find a candidate for Bob’s constraint matrix. That is, for each permutation $L$, Alice can create a candidate $Q$ matrix, and solve the following system of equations:

$$K N Q = \hat{N} \quad \text{and} \quad K b = \hat{b}$$

for the unknown row(s) of $N$ and $b$, bearing in mind that $K$ is also unknown. Each system will be internally consistent, and will give Alice a candidate for Bob’s constraint(s). We don’t give the full solution method yet, but instead give some intuition as to how to solve it below.

Referring to the example from Section 6.3.2.1, the known and unknown values from Alice’s point of view are:

$$K = \begin{bmatrix} a & b \\ c & d \end{bmatrix}; \quad M_1 = \begin{bmatrix} 1 & 2 \end{bmatrix}; \quad M_2 = [B_1 \quad B_2]; \quad b = \begin{bmatrix} 6 \\ B_3 \end{bmatrix},$$

where $\{a, b, c, d\}$ and $\{B_1, B_2, B_3\}$ are unknown scalars. Also consider the disguising matrix $Q = DL$ from Alice’s point of view. Let

$$D = \begin{bmatrix} r_1 & 0 & 0 & 0 \\ 0 & r_2 & 0 & 0 \\ 0 & 0 & r_3 & 0 \\ 0 & 0 & 0 & r_4 \end{bmatrix},$$

where $\{r_1, \ldots, r_4\}$ are unknown scalars, and we shall consider the case where Alice tries the permutation $[1 \quad 2 \quad 4 \quad 3]$, giving $L_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$. Hence let $Q_1 = DL_1$.

Alice can create the following two systems of equations:
\[ KNQ_1 = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 & 0 \\ B_1 & B_2 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & 0 & 0 & 0 \\ 0 & r_2 & 0 & 0 \\ 0 & 0 & 0 & r_3 \\ 0 & 0 & r_4 & 0 \end{bmatrix} = \begin{bmatrix} r_1(1a + B_1b) & r_2(2a + B_2b) & r_4b & r_3a \\ r_1(1c + B_1d) & r_2(2c + B_2d) & r_4d & r_3c \end{bmatrix} \]

\[ = \begin{bmatrix} 35 & 36 & 12 & -10 \\ -95 & -120 & -27 & 16 \end{bmatrix} = \hat{N}, \]

where all the elements of \( \hat{N} \) are known to Alice; and

\[ K\hat{b} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 6 \\ B_3 \end{bmatrix} = \begin{bmatrix} 6a + B_3b \\ 6c + B_3d \end{bmatrix} = \begin{bmatrix} 18 \\ -60 \end{bmatrix} = \hat{b}, \]

where all the elements of \( \hat{b} \) are known to Alice.

Equating the last two columns of \( KNQ_1 \) with the last two columns of \( \hat{N} \) will allow Alice to reduce the number of unknowns, since she will be able to write \( c \) in terms of \( a \), and \( d \) in terms of \( b \).

Now consider the \( K\hat{b} \) equations. Using the relationships deduced above, Alice can now write \( K\hat{b} \) entirely in terms of \( a \) and \( B_3b \). If we equate it with \( \hat{b} \) we can see we have a system of 2 equations in 2 unknowns. This system has to be consistent by construction. Alice can solve uniquely for \( a \) and \( B_3b \). Thus she knows Bob’s coefficient \( B_3 \) up to a scale factor of \( b \).

Applying the same logic to columns 1 and 2 of \( KNQ_1 \), Alice can solve uniquely for \( r_1a \) and \( r_1B_1b \). Then, using her knowledge of \( a \), she can find \( r_1 \), and hence get Bob’s coefficient \( B_1 \) up to a scale factor \( b \). In this example, Alice can find \( B_1 = \frac{12}{b} \), \( B_2 = \frac{16}{b} \) and \( B_3 = \frac{48}{b} \). Thus one candidate for Bob’s constraint is:

\[ 12x_1 + 16x_2 \leq 48. \]

Alice will obtain \( n!m! \) such candidates by repeating this procedure for every candidate \( L \). One of these will be correct (up to a scale factor), but Alice won’t be able to tell...
which one. Hence, the best she can do is take a guess, with a \( \frac{1}{n!m!} \) chance of being correct. However, we will see later in Section 7.4 that once Alice owns more than one constraint she will be able to detect which candidate is correct.

### 7.3.1 Detailed calculations

For completeness, we describe one execution of the non-linear-algebra attack in detail, by giving the full calculations for the above example. To see all of the ‘true’ values for the above example, refer to the LP given in (6.3.1) in Chapter 6, which is the original LP upon which this example is based.

#### Step 1: Equate the slack variable columns

Alice begins by equating the columns of \( KNQ_1 \) relating to the slack variables with the corresponding columns of \( \hat{N} \):

\[
\begin{bmatrix}
r_4b & r_3a \\
r_4d & r_3c
\end{bmatrix} =
\begin{bmatrix}
12 & -10 \\
-27 & 16
\end{bmatrix}.
\]

This allows Alice to derive the following relationships between the elements of \( K \):

\[
\begin{align*}
r_3a &= -10 & r_4b &= 12 \\
r_3c &= 16 & r_4d &= -27
\end{align*}
\]

\[
\Rightarrow -\frac{10}{a} = \frac{16}{c} \quad \Rightarrow \frac{12}{b} = -\frac{27}{d}
\]

\[
\therefore c = -\frac{8}{5}a. \quad \therefore d = -\frac{9}{4}b.
\]

Alice uses the above relationships to solve the \( Kb \) equations.

#### Step 2: Solve the \( Kb \) equations

The \( Kb \) equations are:

\[
Kb = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 6 \\ B_3 \end{bmatrix} = \begin{bmatrix} 6a + B_3b \\ 6c + B_3d \end{bmatrix} = \hat{b} = \begin{bmatrix} 18 \\ -60 \end{bmatrix}.
\]

Substituting \( c = -\frac{8}{5}a \) and \( d = -\frac{9}{4}b \) gives:

\[
\begin{align*}
6a + B_3b &= 18 \\
6(-\frac{8}{5}a) + B_3(-\frac{9}{4}b) &= -60,
\end{align*}
\]
which has solution:

\[
\begin{align*}
a &= -5, \\
B_3b &= 48 \\
\therefore B_3 &= \frac{48}{b}.
\end{align*}
\]

**Step 3: Equate the non-slack variable columns**

Alice now equates the columns of \( KNQ_1 \) relating to the non-slack variables with the corresponding columns of \( \hat{N} \). i.e.,

\[
\begin{bmatrix}
r_1(1a + B_1b) & r_2(2a + B_2b) \\
r_1(1c + B_1d) & r_2(2c + B_2d)
\end{bmatrix} = \begin{bmatrix} 35 \\
-95 \end{bmatrix}.
\]

We consider each column separately.

**Column 1:** Alice equates column 1 of \( KNQ_1 \) with column 1 of \( \hat{N} \), to obtain the following system of equations:

\[
\begin{align*}
ar_1 + B_1br_1 &= 35 \\
cr_1 + B_1dr_1 &= -95.
\end{align*}
\]

Substituting \( a = -5, c = \frac{-8}{5}a = 8 \) and \( d = \frac{-9}{4}b \) gives:

\[
\begin{align*}
-5r_1 + B_1br_1 &= 35 \\
8r_1 + B_1\left(\frac{-9}{4}b\right)r_1 &= -95.
\end{align*}
\]

Letting \( B = B_1br_1 \) gives:

\[
\begin{align*}
-5r_1 + B &= 35 \\
8r_1 + \left(\frac{-9}{4}\right)B &= -95,
\end{align*}
\]

which gives:

\[
\begin{align*}
r_1 &= 5 \\
B &= 60 \\
\implies B_1br_1 &= 60 \\
\implies B_1b &= \frac{60}{5} \implies B_1 = \frac{12}{b}.
\end{align*}
\]

**Column 2:** Alice equates column 2 of \( KNQ_1 \) with column 2 of \( \hat{N} \) to obtain:

\[
\begin{align*}
2ar_2 + B_2br_2 &= 36 \\
2cr_2 + B_2dr_2 &= -120.
\end{align*}
\]
Substituting \( a = -5, c = 8 \) and \( d = -\frac{9}{4}b \) gives:

\[
-10r_2 + B_2br_2 = 36 \\
16r_2 + B_2(-\frac{9}{4}b)r_2 = -120.
\]

Letting \( B = B_2br_2 \) gives:

\[
-10r_2 + B = 36 \\
16r_2 + (-\frac{9}{4})B = -120,
\]

which has solution

\[
r_2 = 6 \\
B = 96 \\
\implies B_2br_2 = 96 \\
\implies B_2b = 16 \implies B = \frac{16}{b}
\]

Thus a candidate for Bob’s constraint is:

\[
\frac{12}{b}x_1 + \frac{16}{b}x_2 \leq \frac{48}{b},
\]

which is indeed a scaled version of Bob’s true constraint \( 3x_1 + 4x_2 \leq 12 \) (because we used the correct permutation).

Alice can now revisit Step 1 and evaluate \( r_3 \) as \( a \) has now been determined. Recall that

\[
r_3a = -10 \implies -5r_3 = -10 \implies r_3 = 2.
\]

The only scale factor Alice can’t find is \( r_4 \), since it is tied up with \( b \); and \( b \) in turn is tied up with Bob’s coefficients \( B_1 \) and \( B_2 \), so she can never find \( b \) on its own. However, Alice doesn’t need to know \( r_3 \) or \( r_4 \) to find Bob’s constraint. If there are \( n \) variables, Alice only needs to know the first \( n \) scale factors of \( \mathbf{r} = [r_1 \; r_2 \; \ldots \; r_n \; r_{n+1} \; \ldots \; r_{n+m}] \) to form a candidate for Bob’s constraint matrix, as the remaining scale factors are associated with the slack variables.

### 7.3.2 Summary of the non-linear-algebra attack

The non-linear-algebra attack consists of the following steps. (Assume that Alice stores all of her candidate permutations in a matrix called candidate_perms.)

1. for \( i = 1 \) to size(candidate_perms) do;
   
   cur_perm = candidate_perms(i);
(a) Set-up Phase. Set $L_i = \text{cur_perm}$. Let $D = \begin{bmatrix}
 r_1 & 0 & 0 & 0 \\
 0 & r_2 & 0 & 0 \\
 0 & 0 & \ddots & 0 \\
 0 & 0 & 0 & r_{n+m} 
\end{bmatrix}$.

Set $Q_i = DL_i$. Replace the elements of $K$ and the unknown elements of $N$ and $b$ with variables, and calculate $KNQ_i$ and $Kb$.

(b) Slack-variable columns. Equate the columns of $KNQ_i$ corresponding to the $m$ slack variables with the corresponding columns of $\hat{N}$.

(c) Solve the $Kb$ equations. Solve the $Kb$ equations to find the first $m_1$ elements of the first row of $K$. The $Kb$ equations are an $m \times m$ system of equations.

(d) Non-slack variable columns. Find scale factors $r_1, r_2, \ldots, r_n$. Do this by equating the columns of $KNQ_i$ corresponding to the $n$ non-slack variables with the corresponding columns of $\hat{N}$, and then substituting the known values of $K$. Each time you equate a pair of columns, you need to solve an $m \times m$ system of equations.

In the next section, we consider an extension of the non-linear-algebra attack for the case when Alice owns more than one constraint, and show how it will allow her to deduce Bob’s constraint matrix.

7.4 Consistency-check attack

The consistency-check attack is a powerful extension of the non-linear-algebra attack which can be performed whenever Alice owns more than one constraint. (In the above calculations, Alice only had one constraint.)

The key behind the attack is that each constraint Alice owns gives her an additional independent means of calculating the scale factors $r_1, \ldots, r_n$. This is because each additional constraint gives Alice an additional column of $K$. For example, if Alice owns two constraints, she can obtain two columns of a candidate $K$ for each candidate permutation, which means that she has two independent ways of calculating each scale factor $r_i$, $i = 1, \ldots, n$. That is, Alice can use the first or the second column of $K$ to calculate each scale factor, and, if the results are consistent either way, Alice will know that she has the correct permutation. Similarly, if the two independent ways give inconsistent values for $r_i$, Alice knows the permutation she’s used must be incorrect.

In general, if Alice owns $m_1$ constraints, she will obtain $m_1$ columns of $K$ for each permutation tested, giving her $m_1$ independent ways of checking each $r_i$ value. As soon as
one of the $r_i$ values is not consistent for all columns of $K$, Alice knows that the current permutation is incorrect. Thus the consistency-check attack allows Alice to rule out permutations as she goes.

We now demonstrate the attack using the above systems of equations again, but this time suppose Alice has two constraints. The sizes of $K$ and $Q$ will need to be increased accordingly. The attack requires very little additional work on top of the non-linear-algebra attack.

### 7.4.1 Demonstration

Suppose Alice owns 2 constraints, and Bob owns 1 constraint, and Program A is given as follows:

\[
\begin{align*}
\text{max } f(x) &= 5x_1 + x_2 \quad \text{(Additively Shared)} \\
\text{s.t.} &\quad x_1 + 2x_2 \leq 6 \quad \text{(Alice’s)} \\
&\quad 3x_1 + 4x_2 \leq 12 \quad \text{(Alice’s)} \\
&\quad 5x_1 + 6x_2 \leq 30 \quad \text{(Bob’s)} \\
&\quad x_1, x_2 \geq 0,
\end{align*}
\]

Therefore the constraint matrix $N$ (once we add slack variables) and RHS vector $b$ are given by:

\[
N = \begin{bmatrix}
1 & 2 & 1 & 0 & 0 \\
3 & 4 & 0 & 1 & 0 \\
5 & 6 & 0 & 0 & 1
\end{bmatrix} ; \quad b = \begin{bmatrix}
6 \\
12 \\
30
\end{bmatrix}.
\]

Suppose Bob chooses $K$ and $D$ as follows:

\[
K = \begin{bmatrix}
-5 & 4 & 2 \\
8 & -9 & 1 \\
6 & 3 & 10
\end{bmatrix} ; \quad D = \begin{bmatrix}
5 & 0 & 0 & 0 & 0 \\
0 & 6 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 4 & 0 \\
0 & 0 & 0 & 0 & 3
\end{bmatrix},
\]

and $L$ to be simply the identity matrix. Thus, the true $Q$ is:

\[
Q = DL = DI_5 = \begin{bmatrix}
5 & 0 & 0 & 0 & 0 \\
0 & 6 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 4 & 0 \\
0 & 0 & 0 & 0 & 3
\end{bmatrix}.
\]

After Alice and Bob conduct the secure scalar products, Alice receives the following
components of the transformed problem:

\[
\hat{N} = \begin{bmatrix}
85 & 108 & -10 & 16 & 6 \\
-70 & -84 & 16 & -36 & 3 \\
50 & 108 & 12 & 12 & -3
\end{bmatrix};
\hat{b} = \begin{bmatrix}
78 \\
-30 \\
42
\end{bmatrix};
\hat{d}^T = [25 \ 6 \ 0 \ 0 \ 0].
\]

Consider the disguising matrices and components of the original problem from Alice’s point of view. The disguising matrix \(K\) is completely unknown to her and given by:

\[
K = \begin{bmatrix}
a & b & c \\
d & e & f \\
g & h & i
\end{bmatrix},
\]

where \(a, b, \ldots, i\) are unknown constants. The constraint matrix \(N\) and RHS vector \(b\) (from Alice’s point of view) are given by:

\[
N = \begin{bmatrix}
1 & 2 & 1 & 0 & 0 \\
3 & 4 & 0 & 1 & 0 \\
B_1 & B_2 & 0 & 0 & 1
\end{bmatrix};
\quad b = \begin{bmatrix}
6 \\
12 \\
B_3
\end{bmatrix},
\]

where \(B_1\) and \(B_2\) are scalars representing Bob’s constraint coefficients, and \(B_3\) represents Bob’s RHS value. The diagonal matrix \(D\) is given by:

\[
D = \begin{bmatrix}
r_1 & 0 & 0 & 0 & 0 \\
0 & r_2 & 0 & 0 & 0 \\
0 & 0 & r_3 & 0 & 0 \\
0 & 0 & 0 & r_4 & 0 \\
0 & 0 & 0 & 0 & r_5
\end{bmatrix},
\]

where \(r_1, \ldots, r_5\) are unknown scalars.

Recall that the true objective function \(d\) is:

\[
d^T = [5 \ 1 \ 0 \ 0 \ 0],
\]

but because \(d\) is shared, to Alice’s eyes it looks like:

\[
d^T = [* \ * \ 0 \ 0 \ 0].
\]

That is, Alice can distinguish zeros from non-zeros, but nothing more. Meanwhile, the disguised objective is given by \(\hat{d}^T = [25 \ 6 \ 0 \ 0 \ 0]\). Comparing \(d\) and \(\hat{d}\) using the logic described in Section 7.2.1, Alice has to consider \(2!3! = 12\) permutations. We will consider two of them — the true permutation, and one of the incorrect permutations — to demonstrate the consistency-check attack.
As mentioned, the consistency-check attack is an extension of the non-linear-algebra attack. Consider performing the non-linear-algebra attack on the following two permutations:

\[
\begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 2 & 5 & 4 & 3 \\
\end{bmatrix}.
\]

**Permutation 1:** Assume the permutation is \( [1 \ 2 \ 3 \ 4 \ 5] \)

Alice begins by assuming the shuffle is \( L_1 = I \); thus \( Q_1 \) is simply \( D \). Alice writes out the product \( K N Q_1 \) and sets it equal to \( \hat{N} \):

\[
K N Q_1 = \begin{bmatrix}
(1a + 3b + B_1c) r_1 & (2a + 4b + B_2c) r_2 & ar_3 & br_4 & cr_5 \\
(1d + 3e + B_1f) r_1 & (2d + 4e + B_2f) r_2 & dr_3 & er_4 & fr_5 \\
(1g + 3h + B_1i) r_1 & (2g + 4h + B_2i) r_2 & gr_3 & hr_4 & ir_5 \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
85 & 108 & -10 & 16 & 6 \\
-70 & -84 & 16 & -36 & 3 \\
50 & 108 & 12 & 12 & -3 \\
\end{bmatrix}
= \hat{N}.
\]

**Equate the slack-variable columns:**

Using the same logic as she did in the non-linear-algebra attack, Alice deduces the following relationships between the variables:

\[
\begin{bmatrix}
ar_3 \\
dr_3 \\
gr_3 \\
\end{bmatrix} = \begin{bmatrix}
-10 \\
16 \\
12 \\
\end{bmatrix} \implies \begin{bmatrix}
d = -1.6a \\
g = -1.2a \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
br_4 \\
er_4 \\
hr_4 \\
\end{bmatrix} = \begin{bmatrix}
16 \\
-36 \\
12 \\
\end{bmatrix} \implies \begin{bmatrix}
e = -2.25b \\
h = 0.75b \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
cr_5 \\
f r_5 \\
ir_5 \\
\end{bmatrix} = \begin{bmatrix}
6 \\
3 \\
-3 \\
\end{bmatrix} \implies \begin{bmatrix}
f = 0.5c \\
i = -0.5c \\
\end{bmatrix}.
\]

Alice uses these relationships to solve the \( Kb \) equations.

**Solve the \( Kb \) equations:**

\[
Kb = \begin{bmatrix}
6a + 12b + B_3c \\
6d + 12e + B_3f \\
6g + 12h + B_3i \\
\end{bmatrix} = \begin{bmatrix}
78 \\
78 \\
42 \\
\end{bmatrix}
\]
\[
6a + 12b + B_3c = 78 \\
6(-1.6a) + 12(-2.25b) + B_3(0.5c) = -30 \\
6(-1.2a) + 12(0.75b) + B_3(-0.5c) = 42.
\]

Solving this set of equations for \(a\), \(b\) and \(B_3c\) gives:

\[a = -5, \quad b = 4, \quad B_3c = 60.\]

\[\therefore d = -1.6(-5) = 8 \quad \therefore e = -2.25(4) = -9\]

\[\therefore g = -1.2(-5) = 6. \quad \therefore h = 0.75(4) = 3.\]

Note that Alice owns 2 constraints, therefore she obtains 2 elements \((a\) and \(b\)) out of the first row of \(K\). Hence she can deduce the first 2 columns of \(K\). (Although the fact that she can find the whole column is irrelevant, because the values in the remaining rows are just derived from the values in the first row.) That is, Alice knows that a candidate for \(K\) corresponding to the permutation under consideration is:

\[
K = \begin{bmatrix}
-5 & 4 & \cdot \\
8 & -9 & \cdot \\
6 & 3 & \cdot 
\end{bmatrix}.
\]

**Equate the non-slack variable columns:**

**Column 1:**

Alice now equates column 1 of \(KNQ_1\) with the corresponding column of \(\hat{N}\):

\[
1ar_1 + 3br_1 + B_1cr_1 = 85 \\
1(-1.6a)r_1 + 3(-2.25b)r_1 + B_1(0.5c)r_1 = -70 \\
1(-1.2a)r_1 + 3(0.75b)r_1 + B_1(-0.5c)r_1 = 50,
\]

to obtain the solution:

\[ar_1 = -25, \quad br_1 = 20, \quad B_1cr_1 = 50.\]

Note that Alice has a choice of two independent equations for \(r_1\): she can either find \(r_1\) using column one of \(K\) (by substituting in the value of \(a\)) or using column two of \(K\) (by substituting in the value of \(b\)). Substituting \(a = 5\) and \(b = 4\) into the above equations gives:

\[
ar_1 = -25 \quad br_1 = 20 \quad B_1cr_1 = 50
\]

\[
\implies r_1 = \frac{-25}{-5} = 5 \quad \implies r_1 = \frac{20}{4} = 5 \quad \implies B_1c = \frac{50}{r_1} = \frac{50}{5} = 10.
\]

Note that the value of \(r_1\) found using \(b\) is consistent with the value of \(r_1\) found using \(a\).
Chapter 7. Security Analysis: Attacks and Workloads

Alice has two checks for $r_1$, since she knows two columns of $K$. The number of checks is equal to $m_1$, the number of constraints owned by Alice. Note that Alice can never find the remaining $m_2$ columns of $K$, because they are tied up with Bob’s constraints.

**Column 2:**

Alice now equates column 2 of $KNQ_1$ with the corresponding column of $\hat{N}$:

\[
2ar_2 + 4br_2 + B_2cr_2 = 108 \\
2(-1.6a)r_2 + 4(-2.25b)r_2 + B_2(0.5c)r_2 = -84 \\
2(-1.2a)r_2 + 4(0.75b)r_2 + B_2(-0.5c)r_2 = 108,
\]

\[
\Rightarrow ar_2 = -30 \\
\Rightarrow br_2 = 24 \\
\Rightarrow B_2cr_2 = 72
\]

\[
\Rightarrow r_2 = -30 \\
\Rightarrow r_2 = 24 \\
\Rightarrow B_2c = \frac{72}{6} = 12
\]

Note that the value of $r_2$ found using $b$ is consistent with the value of $r_2$ found using $a$. Since the values found for $r_1$ were also consistent with each other, we say that this permutation ‘passes’ the consistency-check attack. In general, if all $m_1$ ways of finding $r_1, \ldots, r_n$ give consistent results, the permutation passes the consistency-check attack.

**Form a candidate for Bob’s constraint:**

The final step is to compute the candidate for Bob’s constraint. From column 1, column 2 and the $Kb$ equations, Alice has that:

\[
B_1c = 10 \\
B_2c = 12 \\
B_3c = 60
\]

\[
B_1 = \frac{10}{c} \\
B_2 = \frac{12}{c} \\
B_3 = \frac{60}{c}.
\]

Therefore the candidate for Bob’s constraint resulting from this permutation is:

\[
B_1x_1 + B_2x_2 \leq B_3 \Rightarrow \left(\frac{10}{c}\right)x_1 + \left(\frac{12}{c}\right)x_2 \leq \left(\frac{60}{c}\right) \Rightarrow 10x_1 + 12x_2 \leq 60.
\]

This is indeed a scaled version of Bob’s true constraint, $5x_1 + 6x_2 \leq 30$ (since we used the correct permutation).
Permutation 2: Assume the permutation is [1 2 5 4 3]

Alice now assumes the permutation is [1 2 5 4 3], and executes the non-linear-algebra attack a second time. Note that this permutation is incorrect with respect to the one chosen by Bob. This time, \( L = L_2 \), where

\[
L_2 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0
\end{bmatrix}.
\]

Thus \( KNQ_2 \) is as given below, and Alice sets it equal to \( \hat{N} \):

\[
KNQ_2 = \begin{bmatrix}
(1a + 3b + B_1c)r_1 & (2a + 4b + B_2c)r_2 & cr_5 & br_4 & ar_3 \\
(1d + 3e + B_1f)r_1 & (2d + 4e + B_2f)r_2 & fr_5 & cr_4 & dr_3 \\
(1g + 3h + B_1i)r_1 & (2g + 4h + B_2i)r_2 & ir_5 & hr_4 & gr_3
\end{bmatrix}
\]

\[
= \begin{bmatrix}
85 & 108 & -10 & 16 & 6 \\
-70 & -84 & 16 & -36 & 3 \\
50 & 108 & 12 & 12 & -3
\end{bmatrix}
\]

Thus \( KNQ_2 \) is as given below, and Alice sets it equal to \( \hat{N} \):

\[
KNQ_2 = \begin{bmatrix}
65 & 108 & -10 & 16 & 6 \\
-70 & -84 & 16 & -36 & 3 \\
50 & 108 & 12 & 12 & -3
\end{bmatrix}
\]

Equate slack-variable columns (incorrect permutation):

By equating the slack-variable columns in \( KNQ_2 \) and \( \hat{N} \), Alice deduces the following relationships between the slack variables:

\[
\begin{bmatrix}
ar_3 \\
dr_3 \\
gr_3
\end{bmatrix} = \begin{bmatrix}
6 \\
3 \\
-3
\end{bmatrix} \implies \begin{bmatrix}
d = 0.5a \\
g = -0.5a
\end{bmatrix}
\]

\[
\begin{bmatrix}
br_4 \\
cr_4 \\
hr_4
\end{bmatrix} = \begin{bmatrix}
16 \\
-36 \\
12
\end{bmatrix} \implies \begin{bmatrix}
e = -2.25b \\
h = 0.75b
\end{bmatrix}
\]

\[
\begin{bmatrix}
cr_5 \\
fr_5 \\
ir_5
\end{bmatrix} = \begin{bmatrix}
-10 \\
16 \\
12
\end{bmatrix} \implies \begin{bmatrix}
f = -1.6c \\
i = -1.2c
\end{bmatrix}
\]

\( Kb \) equations (incorrect permutation):

Alice solves the \( Kb \) equations:

\[
Kb = \begin{bmatrix}
6a + 12b + B_3c \\
6d + 12e + B_3f \\
6g + 12h + B_3i
\end{bmatrix} = \begin{bmatrix}
78 \\
-30 \\
42
\end{bmatrix}.
\]
Solving this set of equations for $a$, $b$, and $B_3c$ gives:

\[
\begin{align*}
\begin{cases}
a = 10, \\
b = 4, \\
B_3c = -30.
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\therefore d &= 0.5(10) = 5 \\
\therefore e &= -2.25(4) = -9 \\
\therefore g &= -0.5(10) = -5 \\
\therefore h &= 0.75(4) = 3
\end{align*}
\]

Note that Alice owns 2 constraints, hence she can get the first 2 columns of $K$. Thus, Alice knows that the $K$ corresponding to this candidate permutation is:

\[
K = \begin{bmatrix}
10 & 4 \\
5 & -9 \\
-5 & 3
\end{bmatrix}.
\]

**Column 1 (incorrect permutation):**

Alice equates column 1 of $KNQ_2$ with the corresponding column of $\hat{N}$:

\[
\begin{align*}
1ar_1 + 3br_1 + B_1c &= 85 \\
1(0.5a)r_1 + 3(-2.25b)r_1 + B_1(-1.6c)r_1 &= -70 \\
1(-0.5a)r_1 + 3(0.75b)r_1 + B_1(-1.2c)r_1 &= 50
\end{align*}
\]

\[
\begin{align*}
\Rightarrow ar_1 &= 50 \\
\Rightarrow br_1 &= 20 \\
\Rightarrow B_1c &= -25
\end{align*}
\]

\[
\begin{align*}
\Rightarrow r_1 &= \frac{50}{10} \\
\Rightarrow r_1 &= \frac{20}{4} \\
\Rightarrow B_1c &= \frac{-25}{r_1} = \frac{-25}{5}
\end{align*}
\]

\[
\begin{align*}
\Rightarrow r_1 &= 5 \\
\Rightarrow r_1 &= 5 \\
\Rightarrow B_1c &= -5
\end{align*}
\]

Note that the values for $r_1$ are consistent, so this permutation still remains a candidate. Alice thus moves on to column 2.

**Column 2 (incorrect permutation):**

Alice equates column 2 of $KNQ_2$ with the corresponding column of $\hat{N}$:

\[
\begin{align*}
2ar_2 + 4br_2 + B_2c &= 108 \\
2(0.5a)r_2 + 4(-2.25b)r_2 + B_2(-1.6c)r_2 &= -84 \\
2(-0.5a)r_2 + 4(0.75b)r_2 + B_2(-1.2c)r_2 &= 108
\end{align*}
\]
Here we can see we have an inconsistent $r_2$ estimate! (Substituting the value of $a$ gives $r_2 = 3.6$, while substituting the value of $b$ gives $r_2 = 6$.) Therefore Alice knows this permutation is incorrect. She terminates the attack here and moves on to the next candidate permutation.

A full demonstration of the consistency-check attack for this example is given in Figure 7.1. It can be seen that only the true permutation results in consistent scale factors down columns 1 and 2 of the scale factors. Columns 3, 4 and 5 correspond to the slack variable scale factors $r_3, r_4$ and $r_5$, and are consistent by construction (see step one of the non-linear-algebra attack). However, note that Alice can’t find $r_5$, since it is tied up with Bob’s constraint; she can only find $r_3$ and $r_4$.

The big question is: will there always only be one permutation that passes the consistency-check attack? Well, yes and no. As the number of constraints increases, experimental trials in MATLAB\(^1\) show that in fact there will be $m_2!$ permutations that pass the consistency-check attack, where $m_2$ is the number of constraints owned by Bob (in Figure 7.1, Bob only owned 1 constraint, therefore only one constraint passed).

Consider a 5-constraint, 2-variable problem where Alice owns 2 constraints and Bob owns 3 constraints, and the true permutation is $[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7]$. In this case, the following 6 permutations will all pass the consistency-check attack:

\[
\begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 2 & 3 & 4 & 5 & 7 & 6 \\
1 & 2 & 3 & 4 & 6 & 5 & 7 \\
1 & 2 & 3 & 4 & 6 & 7 & 5 \\
1 & 2 & 3 & 4 & 7 & 6 & 5 \\
1 & 2 & 3 & 4 & 7 & 5 & 6
\end{bmatrix}
\]

Notice that these permutations differ only in the order of the final $m_2$ (3) columns (i.e., columns 5, 6 and 7). This pattern holds for all values of $m_2$, i.e., altering the positions of the last $m_2$ variables does not affect the consistency-check attack. (We will explain why in the next section.) Note that these variables do not have to occupy the last $m_2$ columns, as shown here; no matter which columns the last $m_2$ variables occupy, their ordering will not affect the consistency-check attack.

\(^1\)Code available on request.
<table>
<thead>
<tr>
<th>Permutations</th>
<th>Candidate matrices</th>
<th>Candidate scale factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 5 4 3</td>
<td>-8.33 -10 -30</td>
<td>7.2 2.5 0.6 4</td>
</tr>
<tr>
<td>2 1 5 3 4</td>
<td>-5 -6.67 -30</td>
<td>12 3.75 2 1.2</td>
</tr>
<tr>
<td>2 1 4 5 3</td>
<td>13.33 24 48</td>
<td>7.2 2.5 0.6 4</td>
</tr>
<tr>
<td>2 1 4 3 5</td>
<td>6 13.33 60</td>
<td>12 3.75 2 4</td>
</tr>
<tr>
<td>2 1 3 4 5</td>
<td>6 20 60</td>
<td>12 2.5 2 4</td>
</tr>
<tr>
<td>2 1 3 5 4</td>
<td>8 24 48</td>
<td>12 2.5 2 1.2</td>
</tr>
<tr>
<td>1 2 5 4 3</td>
<td>-5 -16.67 -30</td>
<td>5 3.6 0.6 4</td>
</tr>
<tr>
<td>1 2 5 3 4</td>
<td>-3.33 -10 -30</td>
<td>7.5 6 2 1.2</td>
</tr>
<tr>
<td>1 2 4 5 3</td>
<td>12 26.67 48</td>
<td>5 3.6 0.6 4</td>
</tr>
<tr>
<td>1 2 4 3 5</td>
<td>6.67 12 60</td>
<td>7.5 6 2 4</td>
</tr>
</tbody>
</table>

**Figure 7.1:** A demonstration of the consistency-check attack, showing Alice’s candidate permutations, the corresponding candidates for Bob’s constraint matrix, and the associated scale factors. Only one permutation, the true permutation, gives a set of scale factors (highlighted) that is consistent across both rows. Note that the slack variables (in the last three columns, although Alice can only find two out of the three) are always consistent by construction, regardless of the permutation.
Unfortunately, though, deeper inspection reveals that all of these permutations in fact produce the same candidate for Bob’s constraint matrix! This can be seen in Table 7.1, where the resulting candidate constraint matrix is listed alongside each permutation. More precisely, the candidate constraint matrices produced by these permutations are identical except for the order of the rows, but the row-order is of no consequence. Each row represents a constraint, but the order the constraints are arranged in is irrelevant. Therefore, Alice will still only obtain one possibility for Bob’s constraint matrix — is this the end for our method?

Recall that Alice cannot calculate the final $m_2$ scale factors, since these are tied up with Bob’s constraints; these scale factors are represented by dashes in Table 7.1, and in all examples to follow.

<table>
<thead>
<tr>
<th>Permutations that passed the consistency-check attack</th>
<th>Candidates for Bob’s matrix</th>
<th>Candidate scale factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7</td>
<td>1234567</td>
<td>5 6 2 4 – – –</td>
</tr>
<tr>
<td></td>
<td>1234576</td>
<td>5 6 2 4 – – –</td>
</tr>
<tr>
<td></td>
<td>1234657</td>
<td>5 6 2 4 – – –</td>
</tr>
<tr>
<td></td>
<td>1234675</td>
<td>5 6 2 4 – – –</td>
</tr>
<tr>
<td></td>
<td>1234765</td>
<td>5 6 2 4 – – –</td>
</tr>
<tr>
<td></td>
<td>1234756</td>
<td>5 6 2 4 – – –</td>
</tr>
</tbody>
</table>

Table 7.1: Six permutations pass the consistency-check attack, but they all produce the same candidate for Bob’s constraint matrix (only the order of the rows is different each time).
All is not lost, however; we have not considered Alice’s workload in performing the non-linear-algebra and consistency-check attacks. The total number of permutations she may need to consider is $n!m!$, and this soon becomes an infeasible number for her to check. e.g. for a problem with $n = 100$ variables and $m = 50$ constraints, $n!m! = 2.8 \times 10^{222}$. Modern computers can perform about $10^{12}$ operations per second, so even if each check required only one operation, it would still take Alice around $10^{204}$ years to get through them all!

Thus we say that our method has a ‘workload’ defence, in that Alice has a sure-fire way of finding Bob’s data, but it requires an infeasible amount of work. Of course, there is always the chance that Alice could strike it lucky and find a match on her first go. However, as the number of candidate permutations increases, the chance of this happening becomes increasingly unlikely.

We now explain why there will always be $m_2!$ permutations that result in the same candidate constraint matrix, and show how Alice can use this knowledge to reduce her workload.

### 7.5 Observation that reduces Alice’s workload

As mentioned, $m_2!$ permutations out of $n!m!$ permutations will pass the consistency-check attack. However, these permutations are all ‘equivalent’, in the sense that they lead to the same candidate for Bob’s matrix, only with the order of the rows permuted.

Given that $m_2!$ permutations will pass the consistency-check attack, and that these only differ in the ordering of the final $m_2$ variables, Alice may as well only consider $\frac{n!m!}{m_2!}$ permutations to begin with. She can get away with only testing one of each group of $m_2!$ permutations that differ only in the order of the final $m_2$ variables.

Of course, whenever Alice owns more than one constraint, all of the incorrect permutations will be eliminated via the consistency-check anyway; this observation simply allows Alice to eliminate all of the ‘equivalent’ permutations to start with, to reduce the number of times she needs to perform the non-linear-algebra attack.

Only the true permutation would pass all $m$ checks of all the scale factors, but since Alice doesn’t have all the scale factors, she accepts all permutations that pass the first $m_1$ checks of all the scale factors. The reason that the ordering of the final $m_2$ columns is irrelevant is that Alice can never find the associated scale factors associated with these columns, so if these columns are wrong, it won’t ‘mess up’ the consistency-check attack.
7.5.1 The row-switching phenomenon

We showed that permutations that differ only in the positions of the last \( m_2 \) variables, where \( m_2 \) is the number of constraints owned by Bob, will all pass the consistency-check attack. Furthermore, they will all lead to the same candidate constraint matrix \textit{except} that the ordering of the rows will be different each time. We refer to this effect as the ‘row-switching phenomenon’. The row-switching phenomenon occurs whenever Bob has more than one constraint.

Why does the row-switching occur?

The row-switching is due to the structure of the matrix inverses when solving the systems of equations. Swapping the order of the last \( m_2 \) columns of slack variables swaps the order of the corresponding \( m_2 \) rows of the resulting matrix inverse used to solve the system of equations, which in turn swaps the order of the rows in the candidate constraint matrix.

\textbf{Theorem 7.1.} Consider an \( n \times n \) matrix \( A \), and its inverse \( A^{-1} \). If matrix \( B \) is matrix \( A \) with the columns permuted, then \( B^{-1} \) is \( A^{-1} \) with its rows permuted. i.e. If you permute the columns of a matrix, you permute the rows of its inverse.

\textit{Proof.} Let \( B = AP \), where \( P \) is an \( n \times n \) permutation matrix, and \( A \) and \( B \) are \( n \times n \) matrices. Then

\[
B^{-1} = (AP)^{-1} \\
= P^{-1}A^{-1} \\
= P^T A^{-1} \\
\text{(since } P^{-1} = P^T \text{)}
\]

That is, \( B^{-1} \) is \( A^{-1} \) with its rows permuted according to \( P^T \). \( \square \)

In Figures 7.3 and 7.4 we present detailed output from one of our MATLAB trials to demonstrate why the row-switching phenomenon occurs. The example output is from a 4-constraint x 2-variable problem where Alice owns 1 constraint, and Bob owns 3 constraints. The program inputs are given in Figure 7.2. The output from two selected permutations is given in Figures 7.3 and 7.4. The differing column ordering produced by these two permutations is highlighted in yellow, and all other points of difference throughout the calculations are also highlighted in yellow. We can see that switching the ordering of the final three slack variables will result in candidate constraint matrices that are identical except for the ordering of the rows. Thus Alice need not consider both of these permutations.

Even when Alice only owns one constraint, she still only needs to check one of each of these \( m_2! \) permutations. She will thus end up with \( \frac{m! m!}{m_2!} \) effective candidates for Bob’s constraint matrix. This is illustrated in Figure 7.5.
PROBLEM SET-UP FOR THE $4 \times 2$ EXAMPLE:

The components of the original problem are:

$$M_1 = \begin{bmatrix} 1 & 2 \end{bmatrix}; \quad M_2 = \begin{bmatrix} 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}; \quad M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}; \quad b = \begin{bmatrix} 6 \\ 12 \\ 30 \\ 40 \end{bmatrix}; \quad d^T = [5 \ 1 \ 0 \ 0 \ 0].$$

Once we add slack variables to $M$, we get:

$$N = \begin{bmatrix} 1 & 2 & 1 & 0 & 0 & 0 \\ 3 & 4 & 0 & 1 & 0 & 0 \\ 5 & 6 & 0 & 0 & 1 & 0 \\ 7 & 8 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Bob chooses:

$$K = \begin{bmatrix} -5 & 4 & 2 & 6 \\ 8 & -9 & 1 & 11 \\ 6 & 3 & 10 & 1 \\ 4 & 11 & 3 & -13 \end{bmatrix}; \quad Q = \begin{bmatrix} 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 19 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 \end{bmatrix}.$$

The components of the transformed problem are:

$$\hat{N} = \begin{bmatrix} 295 & 726 & -95 & 32 & 40 & 42 \\ 315 & 814 & 152 & -72 & 20 & 77 \\ 360 & 1012 & 114 & 24 & 200 & 7 \\ -195 & -374 & 76 & 88 & 60 & -91 \end{bmatrix}; \quad \hat{b} = \begin{bmatrix} 318 \\ 410 \\ 412 \\ -274 \end{bmatrix}; \quad \hat{d}^T = [25 \ 11 \ 0 \ 0 \ 0 \ 0].$$

Figure 7.2: Inputs for the $4 \times 2$ MATLAB Example
### Row-switching phenomenon

**4 x 2 Example: Alice owns 1 constraint, Bob owns 3 constraints.**

<table>
<thead>
<tr>
<th>cur_perm = [1 2 3 4 5 6]</th>
<th>cur_perm = [1 2 3 4 5 6]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S = \begin{bmatrix} -10 &amp; 16 &amp; 6 &amp; 42 \ 16 &amp; 36 &amp; 3 &amp; 77 \ 12 &amp; 30 &amp; 7 &amp; 8 \ 8 &amp; 44 &amp; 9 &amp; -91 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( S = \begin{bmatrix} -10 &amp; 42 &amp; 6 &amp; 16 \ 16 &amp; 77 &amp; 3 &amp; -36 \ 12 &amp; 30 &amp; 12 &amp; 8 \ 8 &amp; -91 &amp; 9 &amp; 44 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( R = \begin{bmatrix} 295 &amp; 396 \ 315 &amp; 444 \ 360 &amp; 552 \ -195 &amp; -204 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( R = \begin{bmatrix} 295 &amp; 396 \ 315 &amp; 444 \ 360 &amp; 552 \ -195 &amp; -204 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{ratio_matrix} = \begin{bmatrix} 1.0000 &amp; 1.0000 &amp; 1.0000 &amp; 1.0000 \ -1.6000 &amp; 1.8333 &amp; 0.5000 &amp; -2.2500 \ -1.2000 &amp; 0.1667 &amp; 5.0000 &amp; 0.7500 \ -0.8000 &amp; -2.1667 &amp; 1.5000 &amp; 2.7500 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{diag_b} = \begin{bmatrix} 6 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{diag_b} = \begin{bmatrix} 6 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{coeffs}_{Kb} = \begin{bmatrix} 6.0000 &amp; 1.0000 &amp; 1.0000 &amp; 1.0000 \ -9.6000 &amp; 1.8333 &amp; 0.5000 &amp; -2.2500 \ -7.2000 &amp; 0.1667 &amp; 5.0000 &amp; 0.7500 \ -4.8000 &amp; -2.1667 &amp; 1.5000 &amp; 2.7500 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{coeffs}_{Kb} = \begin{bmatrix} 6.0000 &amp; 1.0000 &amp; 1.0000 &amp; 1.0000 \ -9.6000 &amp; 1.8333 &amp; 0.5000 &amp; -2.2500 \ -7.2000 &amp; 0.1667 &amp; 5.0000 &amp; 0.7500 \ -4.8000 &amp; -2.1667 &amp; 1.5000 &amp; 2.7500 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{inv_coeffs}_{Kb} = \begin{bmatrix} -0.0128 &amp; -0.0979 &amp; 0.0381 &amp; -0.0858 \ 0.6462 &amp; 0.4758 &amp; -0.2430 &amp; 0.5000 \ -0.1231 &amp; -0.2041 &amp; 0.3068 &amp; -0.2059 \ 0.5538 &amp; 0.3154 &amp; -0.2923 &amp; 0.5000 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{inv_coeffs}_{Kb} = \begin{bmatrix} -0.0128 &amp; -0.0979 &amp; 0.0381 &amp; -0.0858 \ 0.6462 &amp; 0.4758 &amp; -0.2430 &amp; 0.5000 \ -0.1231 &amp; -0.2041 &amp; 0.3068 &amp; -0.2059 \ 0.5538 &amp; 0.3154 &amp; -0.2923 &amp; 0.5000 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{soln}_{Kb} = \begin{bmatrix} -5.0000 \ 48.0000 \ 60.0000 \ 240.0000 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{soln}_{Kb} = \begin{bmatrix} -5.0000 \ 240.0000 \ 60.0000 \ 48.0000 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{Bob_RHS} = \begin{bmatrix} 48.0000 \ 60.0000 \ 240.0000 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{Bob_RHS} = \begin{bmatrix} 240.0000 \ 60.0000 \ 48.0000 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{K_Alice} = \begin{bmatrix} -5.0000 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{K_Alice} = \begin{bmatrix} -5.0000 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{diag}<em>{M</em>{col1}} = \begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{diag}<em>{M</em>{col1}} = \begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{coeffs}_{col1} = \begin{bmatrix} 1.0000 &amp; 1.0000 &amp; 1.0000 \ -1.6000 &amp; 1.8333 &amp; 0.5000 \ -1.2000 &amp; 0.1667 &amp; 5.0000 \ -0.8000 &amp; -2.1667 &amp; 1.5000 \end{bmatrix} )</td>
<td></td>
</tr>
<tr>
<td>( \text{coeffs}_{col1} = \begin{bmatrix} 1.0000 &amp; 1.0000 &amp; 1.0000 \ -1.6000 &amp; 1.8333 &amp; 0.5000 \ -1.2000 &amp; 0.1667 &amp; 5.0000 \ -0.8000 &amp; -2.1667 &amp; 1.5000 \end{bmatrix} )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.3: MATLAB output illustrating why the row-switching phenomenon occurs.
Figure 7.4: Continuation of MATLAB output illustrating why the row-switching phenomenon occurs.
**Figure 7.5:** A demonstration of the ‘row-switching’ phenomenon. The first column lists Alice’s candidate permutations, and alongside each, the resultant candidate constraint matrix and set of scale factors found by Alice. Since Alice only owns one constraint, she can’t apply the consistency-check attack. The pairs of permutations that lead to the same candidate constraint matrices (except that the order of the rows is switched) are highlighted. Notice that the permutations in each pair have the positions of variables 1, 2 and 3 fixed, and only differ in the positions of variables 4 and 5. The effect of switching variables 4 and 5 is to switch the order of the rows in the resultant candidate constraint matrix.
7.6 Case 2: Alice does receive \( x^* \)

Although likely to be the more useful option in practice, revealing \( x^* \) to Alice does impact the security level by reducing Alice’s workload. Essentially, giving Alice \( x^* \) allows her to reduce the number of permutations in her candidate set. We will refer to this attack as the ‘permutation-reduction attack’, described below. Since we are relying on a workload defence, any reduction in the workload will lessen security. We quantify the minimum number of permutations Alice would need to check and show that, fortunately, this still increases factorially in the size of the problem.

7.6.1 Permutation-reduction attack

When Alice knows \( x^* \), she can conduct the permutation-reduction attack. The only advantage is that it reduces Alice’s workload; the consistency-check attack means that she’ll still only have one possibility for Bob’s matrix.

Alice can compare \( x^* \) and \( \hat{x}^* \), in addition to \( d \) and \( \hat{d} \), to narrow down the possible permutations still further. The key is that \( x^* \) provides a (possibly) different partitioning of the variables into a set of zero values and a set of non-zero values than was provided by \( d^T \). To see this, consider the example from Section 6.3.2.1. Non-zero numbers are represented by ‘\( * \)’. Recall that from \( d^T \) and \( \hat{d}^T \) (see Section 7.2.1), the four possible permutations were:

\[
\begin{bmatrix} 1 & 2 & 3 & 4 \\ \end{bmatrix}, \quad \begin{bmatrix} 1 & 2 & 4 & 3 \\ \end{bmatrix}, \quad \begin{bmatrix} 2 & 1 & 4 & 3 \\ \end{bmatrix}, \quad \begin{bmatrix} 2 & 1 & 3 & 4 \\ \end{bmatrix}
\]

Also suppose:

\[
(x^*)^T = \begin{bmatrix} * & 0 & * & 0 \\ \end{bmatrix} \quad \text{and} \quad (\hat{x}^*)^T = \begin{bmatrix} * & 0 & 0 & * \\ \end{bmatrix} \tag{7.6.1}
\]

From \( (x^*)^T \), Alice knows \( x_1 \) and \( x_3 \) are non-zero while \( x_2 \) and \( x_4 \) are zero. Therefore, looking at (7.6.1), the four possible permutations are:

\[
\begin{bmatrix} 1 & 2 & 4 & 3 \\ \end{bmatrix}, \quad \begin{bmatrix} 1 & 4 & 2 & 3 \\ \end{bmatrix}, \quad \begin{bmatrix} 3 & 2 & 4 & 1 \\ \end{bmatrix}, \quad \begin{bmatrix} 3 & 4 & 2 & 1 \\ \end{bmatrix}
\]

Comparing the permutations from \( \hat{d}^T \) and \( (\hat{x}^*)^T \) gives \( \begin{bmatrix} 1 & 2 & 4 & 3 \\ \end{bmatrix} \) as the only common permutation. Hence Alice knows it is the correct permutation.

How many common permutations will there be in general? Although not obvious from this example, the final number of permutations boils down to the overlap between the sets of zero and non-zero variables in \( d \), and the sets of zero and non-zero variables in \( x^* \). We have four possible overlaps:

- non-zeros in \( d \) and non-zeros in \( x^* \),
- non-zeros in \( d \) and zeros in \( x^* \),
If we write \((x^*)^T\) beneath \(d^T\) for the example just considered, we see that there was one member in each of these four groups:

\[
\begin{align*}
d^T &= \begin{bmatrix} * & * & 0 & 0 \\ (x^*)^T &= \begin{bmatrix} * & 0 & * & 0 
\end{align*}
\]

Assume \(d\) is fixed. The worst case for security is when all four overlaps are present and membership in the four resulting groups is approximately equal. To create the worst-case security scenario, we do the following: (i) Populate \(x^*\) with non-zeros and zeros so that we partition the \(n\) non-zero variables in \(d\) (or the \(m\) zero variables, whichever is the larger set) into two groups of sizes as equal to each other as possible. (ii) Partition the other set of variables into two groups of sizes as equal to each other as possible. Note that sometimes the worst case can occur when some of the basic variables have value zero, and so there are more zeros in \(x^*\) than \(n\).

Let the number of permutations in the worst case be \(R_{n,m}\). It is tedious but straightforward to show (see Section 7.9 for details) that \(R_{n,m}\) has the following formulae:

\[
R_{n,m} = \begin{cases} 
\left\lceil \frac{n}{2} \right\rceil! \left\lceil \frac{m}{2} \right\rceil! \left\lfloor \frac{m}{2} \right\rfloor! \left\lfloor \frac{m}{2} \right\rfloor! & n \leq m \\
\left\lceil \frac{m}{2} \right\rceil! \left\lfloor \frac{n}{2} \right\rfloor! \left( m - \left\lfloor \frac{n}{2} \right\rfloor \right)! \left\lfloor \frac{n}{2} \right\rfloor! & n > m, \text{ but } m \geq \frac{n}{2} \\
m! (n-m)! m! & n > m, \text{ but } m < \frac{n}{2} \end{cases} \tag{7.6.2}
\]

We need to divide \(R_{n,m}\) through by \(m_2!\) to give the actual number of permutations Alice needs to consider, since for every permutation, there are \(m_2!\) equivalent permutations. Example values of \(\frac{R_{n,m}}{m_2!}\) will be given in the next section.

The only difference when Alice is told the output \(x^*\) is that \(R_{n,m}\), is less than \(n!m!\), thereby reducing Alice’s workload. However, for all but very small problems, such a reduction soon becomes inconsequential (i.e., Alice’s workload is still too large to be feasible, even with the reduction).

### 7.7 Security level

In summary, the security level of Protocol 6.1 under each output option is:

1. **Option 1: Alice is not told \(x^*\).**

   In this case Alice needs to consider \(\frac{n!m!}{m_2!}\) candidate permutations. For each of these, she can solve a system of non-linear equations to find a candidate for Bob’s matrix \(\tilde{M}_2\). Moreover, if Alice owns more than one constraint, she can deduce
Table 7.2: A summary of the number of candidates for Bob’s constraint matrix and Alice’s corresponding workload under different circumstances.

<table>
<thead>
<tr>
<th>Circumstance</th>
<th>Security Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice is...</td>
<td>Alice has...</td>
</tr>
<tr>
<td>not told x*</td>
<td>1 constraint</td>
</tr>
<tr>
<td>&gt; 1 constraint</td>
<td>m2 constraints</td>
</tr>
<tr>
<td>told x*</td>
<td>1 constraint</td>
</tr>
<tr>
<td>&gt; 1 constraint</td>
<td>m2 constraints</td>
</tr>
</tbody>
</table>

which candidate matrix is correct. If Alice has only one constraint, she has a 1 in $\frac{n!m!}{m_2!}$ chance of guessing Bob’s matrix.

2. **Option 2: x* is additively shared.**

Alice knowing only a share of x* is equivalent to Alice knowing none of x*; thus the security level is the same as for Option 1.

3. **Option 3: Alice is told x*.**

Alice now has two independent sources of information about the permutation. This allows her to narrow down the number of candidate permutations to $\frac{R_{n,m}}{m_2!}$, where $R_{n,m}$ is given in (7.6.2).

The number of candidates for Bob’s matrix, and the associated workload to find them, is summarised in Table 7.2 for the cases when Alice does and does not receive x*. When Alice has more than one constraint, she can find Bob’s matrix with certainty, however the associated workload soon becomes unmanageable. When Alice has one constraint, the best she can do is guess from among either $\frac{n!m!}{m_2!}$ or $\frac{R_{n,m}}{m_2!}$ equally-likely candidates for Bob’s constraint matrix.

A summary of the attacks Alice can perform under the different output options and numbers of constraints is given in Table 7.3. Note that Alice can perform all three attacks whenever she has more than one constraint and is told x*.
Table 7.3: A summary of the attacks Alice can perform under different circumstances.

<table>
<thead>
<tr>
<th>CIRCUMSTANCE</th>
<th>ATTACKS POSSIBLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice is...</td>
<td>Non-Linear</td>
</tr>
<tr>
<td></td>
<td>Algebra</td>
</tr>
<tr>
<td>Alice has...</td>
<td>Consistency Check</td>
</tr>
<tr>
<td></td>
<td>Permutation</td>
</tr>
<tr>
<td></td>
<td>Reduction</td>
</tr>
<tr>
<td>not told $x^*$</td>
<td>1 constraint ✓</td>
</tr>
<tr>
<td>told $x^*$</td>
<td>1 constraint ✓</td>
</tr>
</tbody>
</table>

Table 7.4: Values of $\frac{n!m!}{m^2!}$ (Alice’s workload for Options 1 and 2) and $R_{n,m}$ (Alice’s workload for Option 3) for several values of $n$ & $m$.

Our transformation method offers the greatest security when applied to large LPs. In any case, the user is able to calculate the number of sets of equations Alice potentially has to solve, and judge whether or not this risk is acceptable to them. Table 7.4 gives the values of $\frac{R_{n,m}}{m^2!}$ and $\frac{n!m!}{m^2!}$ for several values of $n$ and $m$.

These days, typical optimisation problems often have $n, m > 100$, or even $n, m > 1000$, so for practical problems, $\frac{R_{n,m}}{m^2!}$ would be much, much larger than the values shown in Table 7.4. For example, for a problem with $m = 50$ and $n = 150$, the number of candidate permutations is of the order of $10^{224}$. Modern computers can perform about $10^{12}$ operations a second. Supposing that each permutation took only one operation to check, performing $10^{224}$ operations would take approximately $10^{204}$ years, but remember that the number of operations required for a check also increases with $n$ and $m$. 
A problem with 3000 constraints and 14000 variables is considered to be of medium size by today’s standards [125, p.377]. Supply chain optimisation problems can even have up to millions of variables (e.g. Braun [20] mentions a problem with 4 million variables). For these cases, the security level would be acceptable with any currently conceivable computational resources.

7.8 Conclusion and future work

We have provided a simple, efficient transformation method which can handle a more general class of linear programs than the current transformation algorithms. Most importantly, we have quantified the worst-case security of our method in terms of the workload required to deduce Bob’s data. Natural extensions of this work could include:

- extending the method to handle multiple parties.
- considering the case of column-partitioned data in the presence of non-negativity constraints. In other words, an extension of Mangasarian’s method [86] to handle non-negativity constraints.
- considering an arbitrary partitioning scenario; such a scenario is likely to be easier than the current scenario, as there should be less structure in the constraint matrices for an adversary to exploit.
- strengthening the transformation further so that we do not have to rely on a workload defence; this way, the method could be used for small LPs as well.
- numerical stability of the transformed problem. In all of our experimental trials, a solution to the transformed problem was always able to be found — will there be times when this is not the case?

Although it would be highly desirable, we believe that it is not possible to devise a generic transformation method that can handle any arbitrary data-partitioning scenario. This is one of the downfalls of this approach — the price paid for efficiency and simplicity is a lack of generality. On the upside, however, the transformation methods do lend themselves nicely to other types of optimisation problems, such as quadratic programming and convex programming. This avenue has yet to be explored.

A final important note is that the sub-protocols needed for our method can be revised if a more efficient approach becomes available. For example, if an improved protocol for secure multiplication is developed, the complexity of our method instantly improves.
7.9 Additional calculations: deriving the formula for $R_{n,m}$

In this section we explain how to derive the formula for $R_{n,m}$, the reduced number of permutations when Alice receives $x^*$. We start by assuming $d$ is fixed, but we do not know the optimal solution $x^*$, so $x^*$ is thought of as being able to be chosen.

The number of candidate permutations boils down to the overlap between the groups of zero and non-zero variables in $d$ and $x^*$. An example showing why this is the case is given in Figure 7.6.

The problem of counting the number of permutations essentially becomes one of counting the number of permutations of a set partitioned into four sets. The set has $(n + m)$ elements, where $n$ is the original number of variables, and $m$ is the number of constraints, and hence the number of slack variables added.

The optimal solution of a linear program consists of $m$ basic variables, some of which may be equal to zero, and $n$ non-basic variables, which are zero by definition. Thus, the vector $x^*$ partitions the set of variables into two subsets, by its zero elements. And, by linear programming theory, there will be at least $n$ zero valued variables in the solution, but there may be more (if some of the basic variables are also zero).

Similarly, the vector $d$, representing the objective function, will again partition the variables by its zero elements which correspond, at least, to the $m$ slack variables. The key thing is that $x^*$ and $d$ provide potentially different partitionings of the set of variables, thereby reducing Alice’s set of candidate permutations.

Recall that the worst case for security is when all four overlaps (of zero and non-zero values) described in Section 7.6.1 are present and membership in the four resulting groups is approximately equal. When $n$ and $m$ are equal, the number of zero and non-zero variables is balanced, so it is easy to create four similar-sized groups, as we saw in the example in Section 7.6.1. We have two groups of size $\frac{n}{2}$ and two groups of size $\frac{m}{2}$. If $m$ and $n$ are odd, we can’t split a variable down the middle so we introduce the floor and ceiling functions. The notation $\lfloor x \rfloor$ denotes the nearest integer $\leq x$ (the “floor” of $x$), while $\lceil x \rceil$ denotes the nearest integer $\geq x$ (the “ceiling” of $x$). It is always the case that $\lfloor x \rfloor + \lceil x \rceil = x$.

To create the worst-case security scenario, we do the following: (i) Populate $x^*$ with non-zeros and zeros so that we partition the $n$ non-zero variables in $d$ (or the $m$ zero variables, whichever is the larger set) into two groups of sizes as equal to each other as possible. From this point onwards, we refer to such groups as ‘equal-sized’ groups, to avoid unnecessary lengthy descriptions. (ii) Partition the other set of variables into
Chapter 7. Security Analysis: Attacks and Workloads

Small Example

Suppose $d, \hat{d}, \hat{x}^*$ and $x^*$ from Alice’s point-of-view are as follows:

\[
d = \begin{bmatrix} * & * & 0 & 0 & 0 \end{bmatrix}, \\
\hat{d}^T = \begin{bmatrix} 25 & 6 & 0 & 0 & 0 \end{bmatrix}, \\
(\hat{x}^*)^T = \begin{bmatrix} 0.8 \ 0 \ 1 \ 0 \ 3.5 \end{bmatrix}, \\
x^* = \begin{bmatrix} 4 & 0 & 2 & 0 & 10 \end{bmatrix}.
\]

Beneath the values in $\hat{d}$, Alice writes which variables they could correspond to. From $d$, she knows that variables $x_1$ and $x_2$ are non-zero, while $x_3, x_4$ and $x_5$ are zero; therefore non-zero values in $\hat{d}$ must correspond to either $x_1$ or $x_2$, while zero values must correspond to ($x_3, x_4, x_5$). Alice does the same beneath the variables in $\hat{x}^*$. From $x^*$, she knows that $x_1, x_3$ and $x_5$ are non-zero, while $x_2$ and $x_4$ are zero. Altogether, Alice writes:

\[
\hat{d}^T = \begin{bmatrix} * & * & 0 & 0 & 0 \end{bmatrix}, \\
\begin{array}{rrrrr}
\text{possible variables} & 1 & 2 & 3 & 4 & 5 \\
1.2 & 1.2 & 3.4.5 & 3.4.5 & 3.4.5 \\
\end{array}, \\
(\hat{x}^*)^T = \begin{bmatrix} * & 0 & * & 0 & * \end{bmatrix}, \\
\begin{array}{rrrrr}
\text{possible variables} & 1 & 3.5 & 2.4 & 1.3.5 & 2.4 & 1.3.5 \\
1.3.5 & 2.4 & 1.3.5 & 2.4 & 1.3.5 \\
\end{array}
\]

where the common variables in each position of the permutation have been underlined. Thus, Alice concludes that the only possible permutations that are compatible with both partitionings must have $x_1$ in position 1, $x_2$ in position 2, $x_3$ or $x_5$ in position 3, $x_4$ in position 4, and $x_3$ or $x_5$ in position 5. Thus, only two permutations are possible candidates:

\[
\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\
1 & 2 & 5 & 4 & 3 \\
\end{bmatrix}.
\]

Notice that the number of permutations boiled down to the overlap between the partitions of zero and non-zero variables in $d$ and $x^*$:

\[
\begin{array}{cc}
\text{zeros in } d & (3, 4, 5) \\
\text{zeros in } x^* & (2, 4) \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{non-zeros in } d & (1, 2) & \text{non-zeros in } d & (1, 2) \\
\text{non-zeros in } x^* & (1, 3.5) & \text{zeros in } x^* & (2, 4) \\
\text{zeros in } d & (3, 4, 5) & \text{non-zeros in } x^* & (1, 3.5) \\
\end{array}
\]

Figure 7.6: Example showing how the number of candidate permutations boils down to the overlap between the partitions of zero and non-zero variables in $d$ and $x^*$.
two equal-sized groups. Note that sometimes the worst case can occur when some of the basic variables have value zero and so there are more zeros in $x^*$ than $n$.

When $n$ and $m$ are equal, the four groups, from left to right, are of size $\lceil \frac{n}{2} \rceil$, $\lfloor \frac{n}{2} \rfloor$, $\lceil \frac{m}{2} \rceil$, and $\lfloor \frac{m}{2} \rfloor$. Thus the number of possible shuffles in the worst case is:

$$\lceil \frac{n}{2} \rceil! \lfloor \frac{n}{2} \rfloor! \lceil \frac{m}{2} \rceil! \lfloor \frac{m}{2} \rfloor!$$

If $m \neq n$, the number of zero and non-zero variables will be unbalanced and the formula will differ slightly. We consider two cases separately.

- **Case 1:** $n < m$ For example, $n = 2$, $m = 4$:

  $d^T = \begin{bmatrix} \ast \ast 0 0 0 \end{bmatrix}$

  $$(x^*)^T = \begin{bmatrix} 0 0 \ast \ast \ast \end{bmatrix}$$

  This case is actually equivalent to $n = m$. Although the number of zeros and non-zeros is unbalanced across the two rows, we can ‘rebalance’ them by setting $(m - n)$ of the $m$ basic variables to 0, since we are assuming the worst case. Hence the number of possible shuffles in the worst case is the same as for $n = m$.

- **Case 2:** $n > m$ We may not have enough non-zero values in $x^*$ to split $n$ into two equal-sized classes, since there are only $m$ non-zero values, and $m$ is less than $n$. Thus we consider case 2(a): $m \geq \frac{n}{2}$, when we can partition $n$ in half, and 2(b): $m \leq \frac{n}{2}$, when we can’t.

  - **Case 2a:** $n > m$, but $m \geq \frac{n}{2}$

    There are $m$ non-zero values in $x^*$, and since $m$ is more than half of $n$, we will be able to partition the $n$ non-zero variables in $d$ into two equal-sized classes. We place $\lceil \frac{n}{2} \rceil$ non-zeros and $\lceil \frac{n}{2} \rceil$ zeros beneath the $n$ non-zeros in $d$. The leftover non-zero values and zeros will be placed beneath the zeros in $d$. There will be $m - \lceil \frac{n}{2} \rceil$ non-zeros left, and $n - \lceil \frac{m}{2} \rceil = \lfloor \frac{m}{2} \rfloor$ zeros left. Hence the number of possible shuffles in the worst case is:

    $$\lceil \frac{n}{2} \rceil! \lfloor \frac{n}{2} \rfloor! (m - \lceil \frac{n}{2} \rceil)! \lfloor \frac{m}{2} \rfloor!$$

  - **Case 2b:** $n > m$, but $m \leq \frac{n}{2}$

    There are $m$ non-zero values in $x^*$, and since $m$ is less than half of $n$, we won’t be able to partition the $n$ non-zero variables in $d$ into two equal-sized classes. The best we can do is use up all the non-zeros to partition $n$ as best we can, and then fill the remainder of the array with zeros. The first two classes will be of size $m$ and $n - m$, while the final class will not be partitioned at all, and will have $m$ elements. The number of possible shuffles in the worst-case is: $m!(n - m)!m!$.

These cases give us the formulae for $R_{n,m}$ defined previously in (7.6.2).
Chapter 8

A Privacy-Preserving Interior Point Method

This chapter considers a new type of cryptographic method for privacy-preserving linear programming. Existing cryptographic methods have all been based on the Simplex method. Since interior point methods are known to offer superior performance over Simplex for certain types of problems, it is important that users are not restricted to using Simplex when solving privacy-preserving linear programs. Here we present the first interior-point privacy-preserving LP solution. We implement primal affine-scaling in a secure setting, for the case when one party owns the objective function, and the other owns the constraint set.

8.1 Motivation

Solution methods for privacy-preserving linear programming have followed two major directions: cryptographic approaches, which implement secure versions of the Simplex algorithm [25, 79, 119, 122], and matrix transformation approaches, which ‘disguise’ the linear program via random matrices [11, 48, 86, 87, 121]. The matrix transformation approaches have the advantage of being able to be used in conjunction with any linear programming algorithm (i.e., not just the Simplex algorithm), but their security properties are generally not as robust as the cryptographic approaches [79].

All of the existing cryptographic approaches are based on the Simplex algorithm. Although it generally performs very well in practice, the Simplex algorithm has exponential worst-case performance [76], and there are certain classes of linear programs for which interior point methods are considered superior. Interior point methods offer better computational complexity than Simplex (polynomial vs exponential) and are good for large-scale problems [4]. For example, Vanderbei [125, p.377] has an example where an interior point method is 900 times faster than the Simplex method on a standard test problem. Also, unlike in Simplex, degeneracy is not a problem in interior point methods
(this will be explained further in the next section). It is therefore advantageous for users to have the option of selecting either algorithm, depending on which one best suits their purpose. Hence a secure implementation of an interior point method would be desirable.

Interior point methods are a fundamental departure from the Simplex algorithm. Whereas the Simplex algorithm traverses the vertices of the feasible region, interior point methods move through the interior of the feasible region, and approach the optimal vertex from inside the feasible region, rather than along an external edge. Various interior point methods have been developed, including Karmarkar’s method [72], and primal-dual interior point methods [16].

Here we focus on one of the simplest interior point methods, the Primal Affine-Scaling method (PAS), and adapt it to be privacy-preserving. This method was selected primarily because it does not require any special form for the linear program beyond standard equality form. This is an advantage, since attempting to convert a jointly-owned linear program into Karmarkar’s [72] standard form in a secure setting is a challenging task. In addition, PAS works with the primal form of the linear program only, which reduces the amount of data sharing needed. Other interior point methods which use both the dual and the primal are potentially faster optimisers, but in a privacy-preserving setting they require additional secure data sharing (and hence have a higher communication overhead).

Hence, although PAS is not the most efficient interior point method available, it is conceptually simple, performs reasonably well in practice [4], and has less privacy-related overhead than alternatives. As well, most interior point methods are variations on a common theme, so being able to implement one of them in a privacy-preserving fashion is a step towards being able to implement all of them.

The PAS method has two phases. Phase I finds a feasible interior starting point, and Phase II finds the optimal solution. Since we are assuming that Bob holds the entire constraint set, Bob is able to execute Phase I locally; thus we focus on securing Phase II.

Our approach is to secure each step of Phase II with a secure ‘building block’. These building blocks are implementations of common operations, such as multiplication and division of private values, that are more efficient than using Yao’s generic circuit approach [133]. In addition, we have modified some of the steps of the regular (non-privacy-preserving) affine-scaling algorithm so as to make them more easily computable in a secure environment. The key to protecting Alice’s and Bob’s data is to ensure that, throughout our protocol, all intermediate output remains split between participants, and hence is not revealed to either. Only the final solution $x^*$ is made public, if desired, or it can remain split if the linear program is part of a larger calculation.
Chapter 8. A Privacy-Preserving Interior Point Method

We emphasise that our algorithm is intended to serve as a framework. Users are free to insert their preferred implementation of each building block at each step. Our contribution is to show how to adapt the traditional PAS algorithm to match the available secure building blocks, and how to put these blocks together efficiently and effectively.

8.2 Background and related work

8.2.1 Simplex method vs interior point methods

Karmarkar’s [72] publication of the first polynomial-time algorithm for linear programming in 1984 generated a flood of research into interior point methods. The interior point method we consider here, the affine-scaling method, was in fact developed in Russia by Dikin [46], but did not receive global exposure, and consequently was independently developed in the West as a modification of Karmarkar’s algorithm [126]. As their name suggests, interior point methods move through the interior of the feasible region towards the optimal solution. This is in contrast to the Simplex method, which moves along the boundary of the feasible region, jumping from vertex to vertex, as illustrated in Figure 8.1.

Figure 8.1: The Simplex method moves from vertex to vertex along the boundary of the feasible region, whereas interior point methods move through the interior of the feasible region towards the optimal solution.

The Simplex method performs well for small and medium-scale problems [68]. Although its theoretical worst-case performance is exponential, it performs well on many practical problems [96]. One of the main problems with Simplex is degeneracy [68]. A basic solution is said to be ‘degenerate’ if there is a basic variable which has a value of zero. A consequence of this is that the objective value remains the same from one iteration to the next, even though the sets of basic and non-basic variables are changing. Degeneracy can cause Simplex to ‘stall’ in practice [68]. It can ‘plateau’ at a certain objective
value and remain there for a finite (but possibly very long) sequence of iterations before the solution improves [57]. Sometimes, cycling can occur, which is when the same sequence of vertices are visited again and again in an infinite loop [57].

Interior point methods avoid degeneracy by moving through the centre of the feasible region, rather than moving around the vertices on the edges of the feasible region. Moving along a path through the interior means that cycling does not occur.

Even though interior point methods have a better worst-case complexity, in practice there is no clear winner between the Simplex algorithm and interior point methods [68]. However, the consensus is that interior point methods perform better for very large, sparse problems [4]. According to Illes and Terlaky:

‘When solving huge problems, possibly involving millions of variables, and solving highly degenerate problems [interior point methods] outperform Simplex method based codes.’ [68, p.183]

In practice, depending on the problem to be solved, it will sometimes be preferable to use Simplex, and other times an interior point method. Thus it is important that users are not restricted to using Simplex-based methods to solve privacy-preserving linear programs.

### 8.3 Traditional primal affine-scaling method

The privacy-preserving algorithm requires a clear understanding of the optimisation algorithm, so before we give our algorithm, we explain the particular strategy used by the primal affine-scaling (PAS) method. The PAS method requires that the original linear program be expressed in standard equality form (8.3.1).

#### 8.3.1 How interior point methods work

Consider problem (8.3.1), below.

\[
\begin{align*}
\min & \quad f = c^T x \\
\text{s.t.} & \quad Ax = b \\
& \quad x \geq 0.
\end{align*}
\] \quad (8.3.1)

Suppose we have a feasible point \(x^{(k)}\) which is strictly interior, i.e., \(x^{(k)} > 0\). We wish to find a new point \(x^{(k+1)}\) by searching in a direction \(d\) that decreases the objective function. This interior movement is expressed as:

\[
x^{(k+1)} = x^{(k)} + \alpha d,
\] \quad (8.3.2)
where $\alpha > 0$ is the step-length, and $d$ is the direction of movement. Two key questions are (i) How do we find a “good” direction $d$? and (ii) How far should we move? We address each question in turn.

A “good” search direction is one which (i) reduces the value of the objective value and (ii) maintains feasibility. To satisfy (i), the following condition must hold: $c^T x^{(k+1)} \leq c^T x^{(k)}$, so $c^T d \leq 0$. Therefore a candidate for $d$ is $-c$, the negative gradient of the objective function, which is the direction of steepest descent. However, we must ensure we remain in the feasible region.

To maintain feasibility we require (i) $Ax^{(k+1)} = b$ and (ii) $x^{(k+1)} > 0$ (we want the new point to be strictly interior). When choosing the descent direction, we concentrate only on requirement (i); we will deal with requirement (ii) when we choose the step size. By definition, $Ax^{(k)} = b$. We need $Ax^{(k+1)} = b$, so we need $Ad = 0$. Hence we require $d$ to be in the nullspace of matrix $A$, denoted $\mathcal{N}(A) = \{x : Ax = 0, x \in \mathbb{R}^n\}$.

To choose a direction $d$ that lies in the nullspace of $A$ but is still ‘close’ to $-c$, we orthogonally project $-c$ onto the nullspace of $A$. This is achieved by multiplying $c$ by the following matrix $P$, called the orthogonal projector:

$$P = I - A^T (AA^T)^{-1} A,$$

where $I$ is an $n \times n$ identity matrix. Thus, the best descent direction $d$ is given by $d = -Pc$.

Among all feasibility-preserving directions, this gives the largest rate of decrease of $c^T x$ per unit step length [125]. This is a desirable property so long as the current point $x^{(k)}$ is not too close to the boundaries of the feasible region. If the current point is close to a ‘wall’, the overall decrease in one step will be small, since we will not be able to move far before hitting the boundary. Hence, we require a strategy that keeps points away from the walls [125]. Karmarkar’s method re-centres each point using a projective transformation, while PAS achieves a similar effect using a simpler transformation, an affine transformation.

### 8.3.2 Scaling transformation

In the PAS method, the strategy is to rescale the starting point so that it is a unit distance away from the non-negative axes. This is done via an affine transformation. Essentially, $x^{(k)}$ is scaled to the vector $1$, the vector consisting of all ones. We compute the step direction in the scaled problem, and then translate this direction back into the original system. By defining an $n \times n$ transformation matrix $D^{(k)}$ whose diagonal
elements are equal to the current point, this scaling transformation is written as follows:

\[ y^{(k)} = (D^{(k)})^{-1}x^{(k)} = \begin{bmatrix} 1 & 1 & \ldots & 1 \end{bmatrix}^T, \quad (8.3.4) \]

where \( D^{(k)} = \text{diag}(x^{(k)}) \), that is,

\[
D^{(k)} = \begin{bmatrix}
 x_1^{(k)} & 0 & 0 & 0 \\
 0 & x_2^{(k)} & 0 & 0 \\
 \vdots & \vdots & \ddots & \vdots \\
 0 & 0 & 0 & x_n^{(k)}
\end{bmatrix}
\quad \text{and} \quad (D^{(k)})^{-1} = \begin{bmatrix}
 \frac{1}{x_1^{(k)}} & 0 & 0 & 0 \\
 0 & \frac{1}{x_2^{(k)}} & 0 & 0 \\
 \vdots & \vdots & \ddots & \vdots \\
 0 & 0 & 0 & \frac{1}{x_n^{(k)}}
\end{bmatrix}.
\]

The original primal linear program from (8.3.1) is rearranged using \( x = D^{(k)}y \), yielding the transformed affine linear program [88]:

**Transformed Linear Program**

\[
\begin{align*}
\text{Minimise} & \quad (c^{(k)})^T y \\
\text{subject to} & \quad A^{(k)}y = b \\
& \quad y \geq 0,
\end{align*}
\]

where \( A^{(k)} = AD^{(k)} \) and \( c^{(k)} = D^{(k)}c \).

In this way, the current iterate is transformed to a point which is equidistant from the boundaries of the positive orthant \( \{x : x \geq 0\} \) and closer to the centre of the feasible set [30]. The effect is illustrated for a sample problem in Figure 8.2.

We repeat the logic described previously to find \( d \), except now we perform all steps in the transformed space. In the transformed space, the formula for the search direction is (see Bhatti [16] for details):

\[
d^{(k)} = -D^{(k)} \left[ c - A^T w^{(k)} \right] = -D^{(k)}z^{(k)}, \quad (8.3.5)
\]

where we define \( z^{(k)} = c - A^T w^{(k)} \), and where \( w^{(k)} \) is defined as

\[
w^{(k)} = [A(D^{(k)})^2 A^T]^{-1} A(D^{(k)})^2 c. \quad (8.3.6)
\]

Note that \( w^{(k)} \) provides an estimate of the solution to the dual problem, and the vector \( z^{(k)} \) provides an estimate of the reduced-cost vector [4].

The final point to address is step length. In the transformed space, we know that \( y^{(k)} = 1 \). The condition \( y^{(k+1)} > 0 \) must be satisfied in order for \( y^{(k+1)} \) to be strictly interior. That is,

\[
y^{(k+1)} = y^{(k)} + \alpha d^{(k)} = 1 + \alpha d^{(k)} > 0. \quad (8.3.7)
\]
Figure 8.2: Affine scaling transforms the current point in \( x \)-space to a vector of 1s in \( y \)-space to prevent points getting too close to the boundary.

If all elements of \( d^{(k)} \) are positive, then we can step as far as we like; such a situation means that the problem is unbounded. Otherwise, the only elements of \( d^{(k)} \) which could make \( y^{(k+1)} \) fall below zero are those which are negative. From (8.3.7) we obtain the condition 
\[
\alpha d^{(k)} > -1,
\]
therefore
\[
\alpha^{\text{max}} = \min \left\{ -\frac{1}{d_i^{(k)}}, \quad d_i^{(k)} < 0, \quad i = 1, \ldots, n \right\}. \tag{8.3.8}
\]

If we set the step length to be \( \alpha^{\text{max}} \), at least one element of \( y^{(k+1)} \) will be zero and therefore on the boundary. Thus we take \( \alpha = \rho \alpha^{\text{max}}, \quad 0 < \rho < 1, \) where \( \rho \) is typically 0.99 in order to go as far as possible without actually touching the boundary of a constraint.

Using the scaling transformation, we can express the next point in terms of the actual problem variables as follows:
\[
x^{(k+1)} = D^{(k)}(y^{(k+1)}) = D^{(k)}(y^{(k)} + \alpha d^{(k)}) = x^{(k)} + \alpha D^{(k)}d^{(k)}. \tag{8.3.9}
\]

Stopping criteria will be discussed later, in Section 8.5.6. The full algorithm for regular (non-privacy-preserving) primal affine-scaling is given in Table 8.1.

### 8.4 Security model and secure building blocks

Loosely speaking, a protocol is ‘secure’ if the parties learn no more information than they would have if the computation had been performed by a trusted third party. In
other words, the parties learn nothing about each other’s data other than what can be inferred from their own input and the output.

We make the standard assumption [62] that the parties are ‘semi-honest’ (also called ‘honest-but-curious’), which means that they will follow the protocol correctly, but will attempt to find out extra information ‘on the side’ from any intermediate and final output.

### 8.4.1 Storing intermediate output

All intermediate output will be additively split between Alice and Bob. For example, if the output is a vector $w$, then Alice and Bob receive vectors $w_A$ and $w_B$, respectively, where $w = w_A + w_B$. The shares $w_A$ and $w_B$ are chosen so that one cannot be inferred from the other. *NB:* Throughout the paper, items subscripted with $A$ are known to Alice but not to Bob, while those subscripted with $B$ are known to Bob but not to Alice.

### 8.4.2 Composition of privacy-preserving protocols

Our interior point protocol is composed of a number of secure building blocks. The security of our protocol is derived primarily from the security of these blocks, but in
order for the complete protocol to be secure, we need some results about the composition of privacy-preserving protocols in series. By Canetti’s [23] composition theorem (see Section 2.1.6), to prove that a protocol that sequentially invokes functions $f_1, \ldots, f_n$ is secure in the semi-honest model, all that needs to be shown is that: (i) the individual functions are secure and (ii) the intermediate output does not reveal information about the private inputs [79]. Since all of our secure protocols produce additively-split outputs, and we choose secure blocks, we satisfy these conditions.

### 8.4.3 Notation

Secure protocols will be represented using the following notation, introduced by Du, Han and Chen [52]. Assume Alice has a private matrix $M_A$, and Bob has $M_B$, which they wish to multiply together to produce additively-split output. Such a computation is represented as follows.

$$[M_A : M_B] \rightarrow [P_A : P_B | P_A + P_B = M_A M_B]. \quad (8.4.1)$$

The data on the left of the arrow represents the input to the secure protocol, and the data on the right of the arrow represents the output. The notation shows that the input of the computation is $M_A$ from Alice, and $M_B$ from Bob, while the output of the computation is $P_A$ for Alice, and $P_B$ for Bob, where the sum of $P_A$ and $P_B$ equals $M_A M_B$. This notation will often be shortened to $M_A M_B = P_A + P_B$, with an arbitrary letter (in this case $P$) chosen to denote the output matrix.

### 8.4.4 Building blocks

Our protocol uses a number of secure building blocks. The most frequently used building block is secure matrix multiplication. There are multiple methods for performing secure scalar product [61] and secure matrix multiplication [52, 66]; any method can be used, so long as it produces split output.

The other secure building blocks used in our protocol are listed below, where the input and output of each building block is given using the notation described in Section 8.4.3:

1. **secure matrix inverse** [52, 65]
   
   $$[M_A : M_B] \rightarrow [P_A : P_B | P_A + P_B = (M_A + M_B)^{-1}]$$

2. **privacy-preserving linear system of equations protocol** [50, 131]
   
   $$[M_A, n_A : M_B, n_B] \rightarrow [x_A : x_B | (M_A + M_B)(x_A + x_B) = (n_A + n_B)]$$

3. **secure minimum-finding protocol** [9]
   
   $$[m_A : m_B] \rightarrow [n_A : n_B | n_A + n_B = \min(m_A + m_B)]$$
(iv) secure division [1]
\[ [m_A, n_A : m_B, n_B] \rightarrow \begin{bmatrix} p_A : p_B \mid p_A + p_B = \frac{(m_A + m_B)}{(n_A + n_B)} \end{bmatrix} \]

(v) secure comparison [2, 56, 132]
\[ [m_A : m_B] \rightarrow [1 \text{ (if } m_A > m_B \text{) or } 0 \text{ otherwise : 1 or 0}] \]

These protocols are discussed in more detail where needed. Note that users are free to use their preferred implementation of each building block, as long as it returns the output in shares (or can be safely adapted to do so). Graphical representations of some of these building blocks can be found in Appendix A.

8.5 Privacy-preserving primal affine-scaling

We need to compute all steps in the original algorithm, with the restriction that only Alice knows \( c \), and only Bob knows \( A \) and \( b \). More precisely, we consider solving a linear program of the type ‘O(A)/O(B)/O(B)/*/N/*\{S or P\}/\{S or P\}' (see Chapter 3 for notation). The general idea is to implement each step with the help of one or more secure building blocks, such that the output of the step (e.g., the search direction) is additively split. As well, several changes must be made to the original algorithm, which will be discussed as they arise. We provide a graphical overview of our method in Figure 8.3; each step is explained in more detail in the subsequent sections. As mentioned, Bob can run Phase I and find an initial feasible interior point \( x^{(0)} \) on his own. (Note that Bob learns nothing new from \( x^{(0)} \), because he already knows \( A \) and \( b \).) On the first round of Phase II, since Bob owns all of \( x^{(0)} \), we set Alice’s share to be \( x^{(0)}_A = 0 \), while Bob’s share is \( x^{(0)}_B = x^{(0)} \).

8.5.1 Step 1: Preliminary computations

In this step, the parties compute some initial sharings in preparation for finding the dual variable vector \( w^{(k)} \), given by (8.3.6). The goal of the preliminary computations is to find shares of \( (A(D^{(k)})^2A^T) \) and \( (A(D^{(k)})^2c) \). We build up to these goals by first computing shares of two smaller multiplications. In total, four additive sharings are computed, each requiring one secure multiplication, in the following order:

(i) Find shares \( (F_A, F_B) \) of \( (D^{(k)})^2 \), so that \( F_A + F_B = (D^{(k)})^2 \).

(ii) Find shares \( (H_A, H_B) \), so that \( H_A + H_B = A(D^{(k)})^2 \).

(iii) Find shares \( (K_A, K_B) \), so that \( K_A + K_B = A(D^{(k)})^2A^T \).

(iv) Find shares \( (f_A, f_B) \), so that \( f_A + f_B = A(D^{(k)})^2c \).

We now explain how to compute each case in detail.
Chapter 8. A Privacy-Preserving Interior Point Method

**Figure 8.3:** A summary of our privacy-preserving primal affine-scaling method. The arrows show the inputs and outputs to each of the blocks from Alice and Bob (denoted by subscripts \( A \) and \( B \)). Note that the output of each step is additively split between Alice and Bob.
8.5.1.1 Find shares of \((D^{(k)})^2\)

Let \(D^{(k)}_A = \text{diag}(x^{(k)}_A)\), and \(D^{(k)}_B = \text{diag}(x^{(k)}_A)\). Then, \((D^{(k)})^2\) is given by

\[
(D^{(k)})^2 = (D^{(k)}_A + D^{(k)}_B)^2 = (D^{(k)}_A)^2 + 2D^{(k)}_A D^{(k)}_B + (D^{(k)}_B)^2.
\]  

(8.5.1)

The first and last terms of (8.5.1) can be computed locally by Alice and Bob respectively, while the middle term requires a secure multiplication. Our aim is to determine \(E_A\) and \(E_B\) such that \(E_A + E_B = D^{(k)}_A D^{(k)}_B\), with \(E_A\) held by Alice, and \(E_B\) held by Bob.

Since \(D^{(k)}_A\) and \(D^{(k)}_B\) are both \(n \times n\) diagonal matrices, multiplying them together requires only \(n\) secure scalar multiplications. The resulting shares, \(E_A\) and \(E_B\), are then most easily kept as diagonal matrices. Substituting \(E_A + E_B = D^{(k)}_A D^{(k)}_B\) into (8.5.1) gives

\[
(D^{(k)})^2 = (D^{(k)}_A)^2 + 2E_A + (D^{(k)}_B)^2 = (D^{(k)}_A)^2 + 2E_A + [2E_B + (D^{(k)}_B)^2].
\]

where \(F_A = (D^{(k)}_A)^2 + 2E_A\) is given to Alice, and \(F_B = 2E_B + (D^{(k)}_B)^2\) is given to Bob. Since \(D^{(k)}_A, D^{(k)}_B, E_A\) and \(E_B\) are all diagonal matrices, \(F_A\) and \(F_B\) will also be diagonal.

8.5.1.2 Find shares of \(A(D^{(k)})^2\)

The next step is to use the sharing of \((D^{(k)})^2\) to find a sharing of \(A(D^{(k)})^2\).

\[
A(D^{(k)})^2 = A(F_A + F_B) = AF_A + AF_B.
\]

A secure multiplication is needed to find \(AF_A\), since \(F_A\) is held by Alice and \(A\) by Bob, while \(AF_B\) can be computed locally by Bob. Let \(AF_A = G_A + G_B\). Thus

\[
A(D^{(k)})^2 = (G_A + G_B) + AF_B = [G_A] + [G_B + AF_B] = H_A + H_B,
\]

where \(H_A = G_A\) and \(H_B = G_B + AF_B\).

We will use this approach throughout this chapter — multiplications involving shared values will be decomposed into terms that can be computed locally, and ‘cross-terms’ that involve data from both parties and hence require secure multiplication. In the above calculation, \(AF_A\) was such a ‘cross-term’.
8.5.1.3 Find shares of $A(D^{(k)})^2 A^T$

The sharing of $A(D^{(k)})^2$ is post-multiplied by $A^T$ to obtain shares of $A(D^{(k)})^2 A^T$.

\[
A(D^{(k)})^2 A^T = (H_A + H_B) A^T = H_A A^T + H_B A^T = (J_A + J_B) + H_B A^T = [J_A] + [J_B + H_B A^T] = K_A + K_B,
\]

where $K_A = J_A$ and $K_B = J_B + H_B A^T$, and the cross-term $H_A A^T = J_A + J_B$ is found via secure matrix multiplication, since $H_A$ is held by Alice and $A$ by Bob. Note that $H_B A^T$ can be computed locally by Bob.

8.5.1.4 Find shares of $A(D^{(k)})^2 c$

This time we post-multiply the sharing of $A(D^{(k)})^2$ by $c$ to find a sharing of $A(D^{(k)})^2 c$. The steps are essentially identical to those above (with $c$ in place of $A^T$), except the secure matrix multiplication is replaced by a secure matrix-vector multiplication. The result is

\[
A(D^{(k)})^2 c = f_A + f_B,
\]

where $f_A$ is held by Alice, and $f_B$ is held by Bob. At this point, the preparation for finding $w^{(k)}$ is complete.

8.5.2 Step 2: Find shares of the dual variable vector

We need to compute shares of $w^{(k)}$, given by (8.3.6). The calculation of $w^{(k)}$ is the most demanding step in the traditional affine-scaling interior point algorithm [4], and, rather than using matrix inversion, it is more efficient to compute the vector $w^{(k)}$ as the solution of the following linear system of equations:

\[
(A(D^{(k)})^2 A^T) w^{(k)} = (A(D^{(k)})^2) c. \tag{8.5.2}
\]

There are two approaches available for calculating $w^{(k)}$ in a privacy-preserving environment. We can either solve (8.5.2) using a privacy-preserving linear system of equations protocol, or we can evaluate (8.3.6) using a secure matrix inverse protocol. We consider each option in turn.

8.5.2.1 Secure system of linear equations approach

One way to find $w^{(k)}$ is to solve a privacy-preserving system of linear equations. Using the sharings $K_A + K_B = (A(D^{(k)})^2 A^T)$ and $f_A + f_B = A(D^{(k)})^2 c$ from the pre-
computations, the system in (8.5.2) can be written as
\[(K_A + K_B)w^{(k)} = (f_A + f_B),\]  
(8.5.3)

This approach relies on having a good implementation of a privacy-preserving linear equations solver.

Currently, there are two classes of methods for solving a system of linear equations securely. One class are essentially disguising methods, e.g., Du and Atallah [50] and Yang et al. [131]. The idea is to multiply both sides of the system by a random matrix $B$ known only to Bob, and then have Alice solve the disguised system. Bob then recovers the true solution using his knowledge of $B$. The problem with the Du-Atallah protocol is that it does not provide the output in shares. This can be easily fixed as follows. Let the solution to the disguised system be $\hat{w}$, and let the solution to the original system be $w$. Normally, Alice sends $\hat{w}$ to Bob, who then computes $w = B^{-1}\hat{w}$. The result is that Bob has the answer in full. To rectify this, we simply replace the last step with a secure multiplication, where Alice holds $\hat{w}$, Bob holds $B^{-1}$ and the output is $w = w_A + w_B$, in shares as desired.

Although it is possible to use these methods, they are inefficient due to their heavy reliance on oblivious transfer. Since a new system of equations must be solved on every iteration, this approach may be impractical.

A different method, proposed by Kang and Hong [71], employs a semi-trusted third-party. Essentially, the parties locally disguise their individual components using the same random matrices, and then a third party solves the disguised problem. The advantage is that it does not require oblivious transfer or any other computationally-intensive step, and is thus highly efficient. However, the use of a semi-trusted third-party or ‘commodity-server’ is sometimes frowned upon and considered to only satisfy a ‘weak model’ of security. Yet Amirbekyan and Estivill-Castro [1] argue that researchers should not be too quick to dismiss methods which use a commodity-server (such as the secure scalar product protocol of [53]), since such protocols can still be secure in the spirit of the semi-honest model. However, the third party in the Kang-Hong protocol plays a more active role than the commodity-server in the Du-Zhan protocol [53], and thus there may be unidentified security concerns.

### 8.5.2.2 Secure matrix inverse approach

An alternative is to find $w^{(k)}$ directly using (8.3.6), since the form of the solution to the system of linear equations is known explicitly. This approach involves a matrix inverse computation, which becomes an expensive operation (in the traditional non-privacy-preserving algorithm) once the matrices get large [4]. However in the privacy-preserving context we can see that the alternative is also expensive, so inversion may be competi-
Several privacy-preserving algorithms have been proposed for finding the inverse of a shared matrix. There is an efficient method by Han and Ng [65] for calculating a matrix inverse securely, which has the added bonus of providing the output in shares.

Applying the Han-Ng protocol, Alice and Bob input their shares \((K_A, K_B)\) of \((A(D^{(k)})^2A^T)\), and obtain shares \((L_A, L_B)\) of its inverse. That is, the Han-Ng protocol achieves the following:

\[
[K_A : K_B] \rightarrow [L_A : L_B \mid L_A + L_B = (K_A + K_B)^{-1}].
\]

Substituting \(L_A+L_B\) for \((A(D^{(k)})^2A^T)^{-1}\) in (8.3.6), and using the sharing \((A(D^{(k)})^2)c = f_A + f_B\) from the preliminary computations, gives:

\[
w = (L_A + L_B)(f_A + f_B). \tag{8.5.4}
\]

Two secure multiplications will be required for the cross-terms. Let \(L_Af_B = g_A + g_B\) and \(L_Bf_A = h_A + h_B\). Expanding the RHS of (8.5.4):

\[
w = L_Af_A + L_Af_B + L_Bf_A + L_Bf_B
\]
\[
= [L_Af_A + g_A + h_A] + [g_B + h_B + L_Bf_B]
\]
\[
= w_A + w_B,
\]

where \(w_A = L_Af_A + g_A + h_A\) and \(w_B = g_B + h_B + L_Bf_B\).

### 8.5.3 Step 3: Compute shares of the descent direction

Although most PAS algorithms compute \(d^{(k)}\) in the original space, computing \(d^{(k)}\) in the transformed space allows us to use a simpler formula (in a privacy-preserving sense) for \(\alpha_{\text{max}}\), the maximum step-size. In the transformed space, the formula for \(\alpha_{\text{max}}\) in (8.3.8) is not dependent on \(x^{(k)}\). If instead we compute \(d^{(k)}\) in the original space, the formula for \(\alpha_{\text{max}}\) becomes dependent on \(x^{(k)}\) (see Kipp-Martin [88]), and since \(x^{(k)}\) is shared, this would create unnecessary work.

For clarity, we drop the superscript \(k\) to make the sharing operations more visible, and consider a simplified representation:

\[
d = -Dz = -D(c - A^T w).
\]

We begin by using the sharing of \(w\) to finding a sharing of \(z\), as follows.

\[
z = c - A^T w
\]
\[
= c - A^T(w_A + w_B) \quad \text{(using our sharing of } w) 
\]
\[
= c - A^T w_A - A^T w_B.
\]
The cross-term $A^T w_A$ must be calculated via secure multiplication. Let $A^T w_A = i_A + i_B$, which has dimension $n \times 1$. Therefore

$$z = c - (i_A + i_B) - A^T w_B$$

$$= [c - i_A] + [-i_B - A^T w_B]$$

$$= z_A + z_B,$$

where $z_A = [c - i_A]$ and $z_B = [-i_B - A^T w_B]$. The search direction vector in the transformed space is then

$$d = -D(z_A + z_B)$$

$$= -(D_A + D_B)(z_A + z_B)$$

$$= -[D_A z_A + D_A z_B + D_B z_A + D_B z_B].$$

The terms $D_A z_A$ and $D_B z_B$ can be computed locally by Alice and Bob, respectively, while the cross-terms require secure multiplication. Let $D_A z_B = j_A + j_B$ and $D_B z_A = k_A + k_B$. Resuming our computation of $d$ with these sharings yields

$$d = -[D_A z_A + (j_A + j_B) + (k_A + k_B) + D_B z_B]$$

$$= [-D_A z_A - j_A - k_A] + [-D_B z_B - j_B - k_B]$$

$$= d_A + d_B,$$

where $d_A = [-D_A z_A - j_A - k_A]$, and $d_B = [-D_B z_B - j_B - k_B]$. In total, three secure matrix-vector multiplications were needed to compute the shares of $d$.

### 8.5.4 Step 4: Find the step length

In order to find the maximum step length, $\alpha^{\text{max}}$, Alice and Bob must find the minimum element of $d = d_A + d_B$. We make the following observation which will simplify the calculation. Suppose $d = (-2, -5, 4)^T$. In this case,

$$\alpha^{\text{max}} = \min \left\{ -\frac{1}{d_i} \mid d_i < 0, i = 1, 2, 3 \right\} = \min \left\{ \frac{-1}{-2}, \frac{-1}{-5} \right\} = \frac{1}{5}. \quad (8.5.5)$$

An equivalent calculation is to find the minimum value in $d$, which will give the largest (in terms of absolute value) negative number, if any negative numbers exist; then take the negative reciprocal of it to obtain $\alpha^{\text{max}}$. i.e.,

$$\alpha^{\text{max}} = -\frac{1}{d^{\text{min}}}, \text{ where } d^{\text{min}} = \min \{d_i, i = 1, \ldots, n\}. \quad (8.5.6)$$

In our example, the minimum element in $d$ is $-5$, and taking the negative reciprocal gives $\alpha^{\text{max}} = \frac{1}{5}$, as before. This approach for calculating $\alpha^{\text{max}}$ is preferable from a privacy-preserving viewpoint. The formula for $\alpha^{\text{max}}$ in (8.5.5) involves first searching
for negative elements in a shared array, then taking negative reciprocals, and finally finding the minimum of the negative reciprocals. Such a calculation is much less efficient from a security point of view, as all intermediate output of these steps would need to be shared. By constrast, the approach in (8.5.6) simply involves finding a minimum over a shared array and then conducting a secure division, so that we obtain two shares of $\alpha^{\text{max}}$.

In theory, the minimum of a shared array can be found through repeated application of a secure comparison protocol, as discussed in [9], but the problem is how to secure the intermediate results. There is one protocol available which manages to do this, the ‘minimum finding protocol for split data’ by [9], which also provides the output in shares, as desired. Upon completion of the protocol, Alice has $d_A^{\text{min}}$ and Bob has $d_B^{\text{min}}$, such that $d_A^{\text{min}} + d_B^{\text{min}} = d^{\text{min}}$. The drawback is that it uses homomorphic encryption, which is only applicable to integer input, so the real input would have to undergo a discretisation, resulting in some loss of precision. However, as this only occurs in computing the step size, the loss of precision does not cascade in the way it does in some implementations of secure Simplex.

Alternatively, a recent protocol for secure comparison [2] is suitable for use with real numbers, and could potentially be adapted into a secure minimum-finding protocol. We leave this as an avenue for future research. The drawback is that this protocol uses a semi-trusted third party.

Once Alice and Bob hold shares of $d^{\text{min}}$, they conduct a secure division to find $\alpha^{\text{max}}$. Applying the secure division protocol from Amribekyan and Estivill-Castro [1], Alice and Bob can compute

$$\alpha^{\text{max}} = -\frac{1}{d_A^{\text{min}} + d_B^{\text{min}}},$$

such that Alice receives $\alpha_A^{\text{max}}$ and Bob receives $\alpha_B^{\text{max}}$, and $\alpha^{\text{max}} = \alpha_A^{\text{max}} + \alpha_B^{\text{max}}$.

Alice and Bob must also test $d^{\text{min}}$ for non-negativity, since if $d^{\text{min}} > 0$, it means that the linear program is unbounded. Note that testing $d^{\text{min}} > 0$ is equivalent to testing $d^{(k)} > 0$, but is much simpler to perform in a secure setting. Writing $d^{\text{min}}$ in shared form gives this condition as:

$$d_A^{\text{min}} + d_B^{\text{min}} > 0 \iff d_A^{\text{min}} > -d_B^{\text{min}},$$

which can be checked via secure comparison. If (8.5.8) is true, the parties terminate the algorithm. Otherwise, the parties move on to find shares of the next interior point.
8.5.5 Step 5: Find shares of the next interior point

The formula for the next iterate $x^{(k+1)}$ is given by

$$x^{(k+1)} = x^{(k)} + \alpha D^{(k)} d^{(k)},$$  \hspace{1cm} (8.5.9)

where $\alpha = \rho \alpha^{\max}$. The factor $\rho$ is public, while $\alpha^{\max}$ is shared, so $\alpha$ is shared also. Alice holds $\alpha_A = \rho \alpha^{\max}_A$ and Bob holds $\alpha_B = \rho \alpha^{\max}_B$.

To compute shares of $x^{(k+1)}$, we must replace all terms on the RHS of (8.5.9) with their shared values and carry out the multiplications; this will be done in stages. First consider the product $D^{(k)} d^{(k)}$:

$$D^{(k)} d^{(k)} = (D^{(k)} A + D^{(k)} B)(d^{(k)} A + d^{(k)} B).$$  \hspace{1cm} (8.5.10)

When expanded, the RHS of (8.5.10) will consist of two locally-computable terms and two cross-terms. Each cross-term requires a secure multiplication. Let $D^{(k)} A = l_A + m_A$ and $D^{(k)} B = m_A + m_B$. Thus we can write (8.5.10) as

$$D^{(k)} d^{(k)} = n_A + n_B,$

where $n_A = [D^{(k)} A d^{(k)} A + l_A + m_A]$ and $n_B = [D^{(k)} B d^{(k)} B + l_B + m_B]$. So far,

$$x^{(k+1)} = (x^{(k)} A + x^{(k)} B) + \alpha(n_A + n_B).$$  \hspace{1cm} (8.5.11)

Replacing $\alpha$ by $(\alpha_A + \alpha_B)$ gives:

$$x^{(k+1)} = \{x^{(k)} A + (\alpha_A + \alpha_B)(n_A)\} + \{x^{(k)} B + (\alpha_A + \alpha_B)(n_B)\}
= x^{(k)} A + \alpha_A n_A + \alpha_B n_A + x^{(k)} B + \alpha_A n_B + \alpha_B n_B.$$

Two secure multiplications are needed for the cross-terms: Let $\alpha_B n_A = o_A + o_B$ and $\alpha_A n_B = p_A + p_B$. In total, Alice computes $x^{(k+1)} A$ and Bob computes $x^{(k+1)} B$, where

$$x^{(k+1)} A = x^{(k)} A + \alpha_A n_A + o_A + p_A,$$
and
$$x^{(k+1)} B = x^{(k)} B + \alpha_B n_B + o_B + p_B.$$

A total of two secure matrix-vector multiplications and two vector-scalar multiplications were needed to compute the next iterate.

8.5.6 Step 6: Check stopping conditions

Miller [90] notes that a number of stopping rules can be used — these are listed in Table 8.2. Options 3 and 4 are the simplest to implement in a privacy-preserving environment, as they involve the least amount of shared computation. In the following, we drop the superscript $k$ for clarity.
1. Relative change in objective value is small. 
\[ \frac{|c^T x^{(k+1)} - c^T x^{(k)}|}{|c^T x^{(k)}|} < \epsilon. \]
2. Relative change in solution vector is small. 
\[ \frac{||x^{(k+1)} - x^{(k)}||}{||x^{(k)}||} < \epsilon. \]
3. Direction vector close to zero. 
\[ ||d^{(k)}|| < \epsilon. \]
4. Duality gap close to zero. 
\[ |c^T x^{(k)} - b^T w^{(k)}| < \epsilon. \]

**Table 8.2:** Choice of stopping conditions ($\epsilon$ typically set to $10^{-6}$ or $10^{-8}$).

We begin with option 3, $||d|| < \epsilon$. Any vector norm could be used to implement this; here we use the standard Euclidean norm. Thus we require $\sqrt{d^T d} < \epsilon$ or $d^T d < \epsilon^2$.

Expanding $d^T d$ gives:

\[
d^T d = (d_A + d_B)^T (d_A + d_B) \\
= d_A^T d_A + 2d_A^T d_B + d_B^T d_B \\
= d_A^T d_A + 2(q_A + q_B) + d_B^T d_B \\
= [d_A^T d_A + 2q_A] + [2q_B + d_B^T d_B] \\
= s_A + s_B,
\]

where $s_A$ and $s_B$ are scalars. One secure vector dot-product was required for the cross term $d_A^T d_B$. Finally, we check whether $s_A + s_B < \epsilon^2$, which is equivalent to checking $s_A < \epsilon^2 - s_B$. This can be checked via a secure comparison.

For option 4, we require the duality gap to be close to zero. It can be shown that option 4 is equivalent to

\[ |x^T z| < \epsilon, \]  \hspace{1cm} (8.5.12)

where $z = c - A^T w$. This follows from substitution of $b = Ax$.

since

\[
c^T x - b^T w = c^T x - (Ax)^T w \\
= x^T c - x^T A^T w \\
= x^T [c - A^T w] \\
= x^T z. \]  \hspace{1cm} (8.5.13)

Substituting $z = z_A + z_B$ and $x = x_A + x_B$ into (8.5.13) we have

\[ x^T z = (x_A + x_B)^T (z_A + z_B), \]

which when expanded involves two locally-computable terms and two cross-terms. Two secure vector dot-products are needed for the cross-terms. Let $(x_A)^T z_B = t_A + t_B$ and
let \((x_B)^Tz_A = u_A + u_B\), where \(t_A, t_B, u_A\) and \(u_B\) are scalars. Altogether, we have

\[
x^Tz = (x_A)^Tz_A + (x_A)^Tz_B + (x_B)^Tz_A + (x_B)^Tz_B
\]

\[
= (x_A)^Tz_A + t_A + t_B + u_A + u_B + (x_B)^Tz_B
\]

\[
= v_A + v_B,
\]

where \(v_A = (x_A)^Tz_A + t_A + u_A\) and \(v_B = (x_B)^Tz_B + t_B + u_B\), are scalars.

If \(|v_A + v_B| < \epsilon\), the stopping condition is satisfied. Therefore we must check: 

\[-\epsilon < v_A + v_B \quad \text{and} \quad v_A + v_B < \epsilon.
\]

This is equivalent to checking: (i) \(-v_A - \epsilon < v_B\) and (ii) \(v_A < \epsilon - v_B\). Since the LHS of each inequality is owned by Alice and the RHS is owned by Bob, these tests can be done via two secure comparisons. A secure ‘and’ protocol can be used to test whether both relations hold, for example using Yao’s circuit protocol [133].

### 8.5.6.1 Obtaining the output

There are several different output options. Once the stopping conditions are satisfied, Alice and Bob each hold a share of the optimal solution. If shared output is desired, the algorithm ends here. Shares of the corresponding optimal objective function \(f^*\) can be calculated via a secure multiplication: 

\[
f^* = c^T(x_A^* + x_B^*) = c^Tx_A^* + c^Tx_B^*
\]

so that each party receives a share of \(f^*\). Otherwise, Alice and Bob can make their shares public, and both can learn \(x^*\). Once \(x^*\) is made public, Alice can compute \(f^*\) and send it to Bob.

We have ignored the purification phase, which is when the interior solution is converted into a vertex solution, but it would be relatively easy to add. However, omitting the purification phase offers protection against the constraint-insertion attack described in Chapter 9 of this thesis. The fact that interior point methods return a point which is not a vertex is an advantage in terms of security. Further details of the attack can be found in Chapter 9.

### 8.6 Complexity analysis

The complexity of our algorithm is determined by the complexity of the building blocks. The number of building blocks used per iteration is given in Table 8.3. The complexity varies depending on which instantiation of each building block is used. Ideally, a much more detailed analysis of complexity should be considered, which breaks the building blocks down further, but this depends on the implementation of these blocks. Also, if \(w^{(k)}\) is calculated using the matrix inverse approach, two additional secure matrix-vector multiplications are required per iteration, as shown in the second column of Table 8.3.
Building Block | Number of instances per iteration $w^{(k)}$ via (8.5.2) | $w^{(k)}$ via (8.3.6)
--- | --- | ---
secure multiplication protocol | 2 | 2
- matrix-matrix | 6 | 8
- matrix-vector | 3 | 3
- vector-vector | 2 | 2
- scalar-vector | $n$ | $n$
- scalar-scalar | | |
secure division protocol | 1 | 1
secure linear equation protocol | 1 | 0
secure matrix inverse protocol | 0 | 1
secure comparison protocol | 4 | 4
secure minimum-finding protocol | 1 | 1

Table 8.3: Summary of the number of secure building blocks used per iteration.

### 8.7 Conclusion

We have presented the first interior-point-based method for privacy-preserving linear programming; all previous secure-protocol-based methods were based on the Simplex algorithm. Our method offers users an alternative, since interior point methods are superior to Simplex for certain problems. Our method applies to linear programs of type ‘$O(A)/O(B)/O(B)/\ast/N\{S\ or\ P\}/\{S\ or\ P\}$’. Further investigation into the complexity and security of this method is required, but this chapter serves as a ‘proof of concept’, that such an implementation is possible. Future work could look at extending this method to implement other types of interior point methods so that they are privacy-preserving, as well as considering other data-partitioning scenarios. In addition, future work could look at developing more efficient building blocks, and considering the most efficient combination of implementations of the building blocks to use.
Chapter 9

An Attack on Privacy-Preserving Linear Programming

Throughout this thesis, we have seen the potential for privacy-preserving linear programming as an optimisation tool. But can this technology be attacked? The technology guarantees the privacy of the algorithm, but does the output reveal so much information that the technology is useless? Here we describe an attack where the malicious party owns all of the constraints, and a second party owns the objective function. The attack is largely independent of the solution algorithm and involves strategically adding extra constraints to the feasible region in order to gain the maximum information possible about the other party’s private objective function. The question of how best to allocate the extra constraints gives rise to a new optimisation problem for which we devise a greedy heuristic and prove that it is optimal.

9.1 Introduction

In linear programming, if one knows the constraint set and the optimal solution, one can derive information about the slope of the objective function. For instance, knowing that the optimal solution occurs at a given vertex means that the slope of the objective must be bounded by the slopes of the constraints forming that vertex. This is usually not a problem, since the person solving the linear program normally holds all of the information, but it becomes important when we consider the case of privacy-preserving linear programming. Here, the privacy of the algorithm is circumvented because the solution reveals more information than we wish.

For example, consider a two-party privacy-preserving linear programming problem where one party, Alice, owns the objective function (which must be kept secret from Bob) and the other party, Bob, owns the complete constraint set (which must be kept secret from Alice). Knowledge of the optimal solution of this problem immediately allows Bob to place bounds on Alice’s objective function, based on the observation just
Chapter 9. An Attack on Privacy-Preserving Linear Programming

described. This information leakage is independent of the privacy-preserving algorithm used. The secure computation field takes the view that an algorithm is secure if it leaks nothing about any party’s private data aside from what can be derived from the output [84]. However, few papers have considered the potential for the output itself to reveal information, and possibly undermine the work of the algorithm.

The problem is exacerbated by the fact that it creates an attack Bob can carry out to learn more precise information about Alice’s objective function. Specifically, Bob can add extra constraints to obtain tighter bounds on the slope of Alice’s objective than he would have on the original feasible region. This will be explained more precisely in the next section. Although it won’t significantly affect the optimal solution, manipulating the constraint set will greatly increase Bob’s ability to learn Alice’s data. We refer to this attack as the ‘constraint-insertion attack’.

To the best of our knowledge, none of the existing papers in the privacy-preserving literature have considered this particular attack. Moreover, the attack is not unique to a particular privacy-preserving LP algorithm — any such algorithm (whether it be cryptographic-based or transformation-based) is vulnerable.

The constraint-insertion attack is unusual in that (i) it will not significantly affect the accuracy of the computation (i.e., the optimal point will only be very slightly removed from the true optimum) and (ii) it is not easily detectable. In order to ensure (ii), Bob only wants to add a modest number of additional constraints. Call this number of constraints $N$. Bob wants to allocate these constraints to his vertices in such a way that his information gain about Alice’s objective function upon learning the optimal solution $\mathbf{x}^*$ will be maximised. This leads us to an optimisation problem, discussed in more detail in Section 9.3.2.

Although the constraint-insertion attack is possible in any number of dimensions, it becomes more difficult to describe as the dimensionality of the feasible region increases. Here we analyse the 2-D case, where the linear program has only two variables, $x_1$ and $x_2$. We describe the attack in more detail in Section 9.2, and in Section 9.3 we formulate the execution of the attack as an optimisation problem. In Sections 9.4 and 9.5 we describe a simple greedy heuristic for solving this problem, and prove that it will always provide the optimal way of implementing the attack.
9.2 Description of the attack

We consider the following linear program:

\[
\begin{align*}
\text{max} & \quad f(x) = c^T x \\
\text{s.t.} & \quad Ax \leq b \\
& \quad x \geq 0,
\end{align*}
\]

(9.2.1)

where the cost vector \( c \in \mathbb{R}^\ell \) is owned by Alice, and the constraint matrix \( A \in \mathbb{R}^{m \times \ell} \) and vector \( b \in \mathbb{R}^m \) are owned by Bob. More specifically, using the notation from Chapter 3, we have an \( O(A)/O(B)/O(B)/\leq /N/P/P \) linear program. Note that, for this chapter, we use the notation \( \ell \) for the number of variables in place of the usual \( n \), because \( n \) is used for another purpose, and we will take \( \ell = 2 \) in much of the discussion anyway.

The aim is to find the optimal solution \( x^* \) without revealing any privately-held data. There are now a number of techniques which do so without any leakage of information from intermediate steps (e.g. [11,119,122]), as have been discussed in previous chapters.

In a two-dimensional linear programming problem, i.e. \( \ell = 2 \), the feasible region is a convex polygon, and the optimal point occurs at one of the vertices. Before formulating the attack mathematically, we review some standard results about polygons.

A \( V \)-sided polygon has \( V \) vertices and \( V \) interior angles [78]. At each vertex an exterior angle may be drawn by extending one of the sides. This is illustrated in Figure 9.1(i), where the exterior angle at vertex \( C \) is labelled as \( \alpha_i \), and the interior angle is labelled as \( \gamma_i \). At each vertex, the sum of the interior angle and the exterior angle is 180°. Below we give two standard theorems for the sum of the angles of a polygon — one for the sum of the interior angles, and another for the sum of the exterior angles.

**Theorem 9.1.** (Interior angle sum) The sum of the measures of the interior angles of a polygon having \( V \) sides is \( 180(V - 2)° \) [78].

**Theorem 9.2.** (Exterior angle sum) The sum of the measures of the exterior angles of any polygon, regardless of the number of sides, is 360° [78].

Knowledge of the optimal vertex \( x^* \) discloses some information about Alice’s private objective to Bob. Specifically, \( x^* \) enables Bob to place bounds on the slope of Alice’s objective function. Figure 9.1(i) illustrates the situation. Recall that Bob owns the entire constraint set, and thus can plot the feasible region. Suppose Alice’s objective function is optimal at vertex \( C \) (i.e. \( x^* = C \) so the level curve \( c^T x = c^T x^* \) touches the constraint set at the vertex \( C \)); thus Bob knows that the slope of Alice’s objective is somewhere between the slope of \( \overline{BC} \) and the slope of \( \overline{CD} \). This range is denoted by \( \alpha_i \), the exterior angle at vertex \( i \).
Figure 9.1: Demonstration of the constraint-insertion attack. Figure (i) shows the situation before Bob carries out the attack. If he leaves his feasible region in its unaltered state, we see that Alice’s objective will touch at vertex C, and Bob will be able to determine that the slope of her objective function falls somewhere in the range labelled $\alpha_i$, the original exterior angle at that vertex. Figure (ii) shows Bob’s improved range for Alice’s objective function following the addition of an extra constraint line near vertex C. The range has narrowed to $\theta_i^{(2)}$. The more lines Bob adds to the vertex, the narrower his range becomes for Alice’s objective function. In reality, Bob must add lines to every vertex, since he will usually not know in advance precisely where Alice’s objective will touch.
However, a little craftiness on the part of Bob and the range for Alice’s objective function can be made much slimmer. Specifically, Bob can insert extra constraint lines at each vertex to refine his knowledge of Alice’s objective function. In Figure 9.1(ii), Bob has added one extra constraint at vertex C. Below we define more precisely what ‘adding a constraint’ means.

**Definition 9.1.** (Adding a constraint at a vertex) *If the original linear program’s feasible region has a vertex C, adding a constraint to vertex C means adding another constraint to the linear program such that it makes vertex C infeasible, but no other vertex infeasible.*

Note that adding one constraint to a vertex results in two vertices (where there was previously only one). In general, adding $w_i$ constraints to a vertex produces $w_i + 1$ vertices in its place, and hence $w_i + 1$ exterior angles at that vertex. Let $\theta_i = (\theta_i^{(1)}, \ldots, \theta_i^{(w_i+1)})$ be the newly-created exterior angles at vertex $i$ following the insertion of $w_i$ constraint lines. Adding one line will decrease Bob’s range for Alice’s slope to $\theta_i^{(2)}$, say, as shown in Figure 9.1. Adding further lines will decrease the range still further. Note that $\sum_j \theta_i^{(j)} = \theta_i^{(1)} + \theta_i^{(2)} = \alpha_i$, since the total exterior angle sum must remain constant.

The attack does come at some cost to accuracy. As can be seen in Figure 9.1(i), the optimal vertex of the original feasible region is vertex C, whereas the optimal vertex of the modified feasible region in Figure 9.1(ii) is vertex $C_2$, which is some distance from C. However, this distance can be made arbitrarily small, simply by shifting the new constraint line so that it is as close as possible to the boundary of the original feasible region. We therefore regard any loss of accuracy as negligible.

In Figure 9.1, Bob only added a constraint to the optimal vertex, but in reality he would need to perform the attack on all vertices as the optimal vertex is not known *a priori*. This is illustrated in Figure 9.2, which shows the constraint insertion attack applied to an entire feasible region. Figure 9.2(i) shows the original feasible region, while Figure 9.2(ii) shows the feasible region after three additional constraints have been added. The original region had a maximum exterior angle of $120^\circ$, which is an upper bound on how accurately Bob can determine Alice’s objective function, while the new feasible region has a maximum exterior angle of only $60^\circ$. Thus Bob’s range of Alice’s objective function has been considerably reduced as a result of the attack.

### 9.3 Optimal attack

#### 9.3.1 Optimal attack at a single vertex

In this section, we consider the optimal way for Bob to implement the attack. If Bob adds too many additional constraints, it may unnecessarily increase the complexity of the problem, or be too obvious that he has carried out an attack, therefore he only
Figure 9.2: An example of the constraint-insertion attack applied to the entire feasible region. Figure (i) shows the original feasible region, while Figure (ii) shows the feasible region after one additional constraint has been added to each of vertices 1, 3 and 4, for a total of three extra constraint lines. The original region had a maximum exterior angle of 120° (equivalent to Bob’s range for Alice’s objective function), while the new feasible region has a maximum exterior angle of only 60°.

wants to add a limited number. But what do we mean by “optimal”? The optimal allocation, subject to a fixed number of additional constraints, $N$, is the one in which Bob’s information about the slope of Alice’s objective function will be maximised. Since Bob will not know in advance which vertex will be the optimal vertex, he needs to minimise the exterior angle at all of the vertices.

There are various ways of expressing this objective. One option is to have Bob minimise the worst case, which equates to minimising the maximum exterior angle. Another option is to have Bob minimise the average case, which equates to minimising the expected exterior angle. Here we focus on the first option.

We first show that under the objective of minimising the maximum exterior angle, the optimal strategy at a single vertex is to make all of the resulting angles equal. Using this result, we can then simplify the multi-vertex problem.

Let $\alpha_i$ be the initial exterior angle at vertex $i$ and let $\theta_i = (\theta_i^{(1)}, \ldots, \theta_i^{(w_i+1)})$ be the newly-created exterior angles at vertex $i$ following the insertion of $w_i$ constraint lines (which creates $w_i + 1$ vertices in place of the original vertex).
Claim: The strategy that minimises the maximum exterior angle $\theta_i^{(j)}$, i.e. $\min \max \theta_i^{(j)}$, when inserting $w_i$ constraints at a vertex $i$ is to ensure that $\theta_i^{(1)} = \theta_i^{(2)} = \ldots = \theta_i^{(w_i+1)}$.

Proof. Suppose setting $\theta_i^{(1)} = \theta_i^{(2)} = \ldots = \theta_i^{(w_i+1)}$ is not the optimal set of angles. Therefore there exists a preferable set of exterior angles $\beta_i = (\beta_i^{(1)}, \beta_i^{(2)}, \ldots, \beta_i^{(w_i+1)})$.

For $\beta$ to be the preferred angle vector, we must have $\beta_i^{(j)} < \theta_i^{(j)}$ for some $j \in \{1, \ldots, w_i+1\}$. By construction, since the sum of the exterior angles of a polygon is constant, $\sum_{k=1}^{w_i+1} \beta_i^{(k)} = \alpha_i = \sum_{k=1}^{w_i+1} \theta_i^{(k)}$. If $\beta_i^{(j)} < \theta_i^{(j)}$ then $\exists \epsilon$ such that $\beta_i^{(\epsilon)} > \theta_i^{(\epsilon)}$; hence $\beta$ is a worse allocation, since it has a larger maximum angle.

Thus we conclude that $\theta_i^{(j)} = \frac{\alpha_i}{w_i+1}$, for all $j = 1, 2, \ldots, (w_i + 1)$. In general, if we add $w_i$ lines to vertex $i$ such that the exterior angles $\theta_i^{(j)}$ at the resultant vertices will all be equal, we have:

$$\theta_i^{(j)} = \frac{\alpha_i}{w_i+1}. \quad (9.3.1)$$

We therefore drop the superscript $j$ and write $\theta_i$, since all exterior angles at vertex $i$ are equal.

9.3.2 Optimal assignment of constraints to vertices

Suppose Bob wishes to add $N$ extra constraints to his feasible region, in such a way that he minimises the maximum exterior angle of the polygon forming the feasible region. The question is, how should he best distribute the $N$ extra constraints among the existing vertices?

Ideally, Bob would like all exterior angles at each vertex in the entire feasible region to be equal, but this is often not possible due to the integer restriction on $w_i$ (e.g. we can’t add 1.5 lines, say, to a vertex).

For a given initial angle vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_V)$ and an allocation of $N$ lines given by $w = (w_1, w_2, \ldots, w_V)$ where $\sum_{i=1}^{V} w_i = N$, the resulting angle at each vertex is stored in $\theta = (\theta_1, \theta_2, \ldots, \theta_V)$, where $\theta_i$ is given by (9.3.1). Note that although multiple angles are created at a vertex, they are all equal, as shown in Section 9.3.1; therefore we only store a single copy of the angle at each original vertex in $\theta$.

Recall that Bob wants to minimise the exterior angle at each vertex. Again, we can formulate the optimisation problem as a min-max problem, where the goal is to minimise the maximum exterior angle, using the available number of lines. Thus our optimisation
Chapter 9. An Attack on Privacy-Preserving Linear Programming

problem can be formulated as:

\[
\min \max_{i=1,...,V} \theta_i = \left( \frac{\alpha_i}{w_i + 1} \right)
\]

\[ \text{s.t.} \]

\[ \sum_{i=1}^{V} w_i = N, \]

\[ w_i \in \mathbb{Z}^+. \] (9.3.2)

In the following sections we show that despite this problem being an integer linear program, it is easily solvable.

9.4 A greedy heuristic for solving the minimisation problem

We outline here a simple greedy heuristic that can be applied to the problem in (9.3.2). Briefly, the idea is to add a line to the vertex with the largest exterior angle, adding one line per round until all \(N\) lines have been allocated.

The important thing to note is that although the total number of vertices in the polygon grows as the method proceeds, the vertex array is not expanded to accommodate the new vertices — we always add lines to the original \(V\) vertices, not the newly-created vertices. Since we produce equal angles every time we add a line to a vertex, we only need to store one copy of the new exterior angle.

More formally, the greedy heuristic proceeds as follows. We denote the final assignment vector of \(N\) lines to \(V\) vertices by \(w = [w_1, w_2, \ldots, w_V]\).

Protocol 9.1. Greedy Heuristic

1. Initialization: Let \(\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_V]\). Set \(\theta = \alpha\). Set \(w = [0, 0, \ldots, 0]\).

2. while \((\sum_{i=1}^{V} w_i < N)\)

   (a) Find the index of the maximum angle in \(\theta\). If there is a tie, choose the smallest index. Call this index \(r\).

   (b) Increment \(w_r\) by 1.

   (c) Recalculate the exterior angles at vertex \(r\) using the formula \(\theta_r = \frac{\alpha_r}{w_r + 1}\), and update \(\theta\) with the new value.

Each round of the greedy heuristic involves one sort operation to find the maximum angle in the array, and the remaining two steps are constant-time. The number of
rounds is independent of the size of the input array. Thus the complexity of the greedy heuristic is $O(n^2)$ (or better, depending on the choice of sorting algorithm), where $n$ represents the size of the input array $\theta$.

The greedy heuristic is trivially easy to perform, but as we show in the next section, it will always produce the optimal assignment of the $N$ additional constraints. That is, it will always give the optimal solution to the min-max problem in (9.3.2).

9.5 Proof that the greedy heuristic is optimal

Despite its simplicity, the greedy heuristic will always produce the optimal assignment of the $N$ additional constraints. We now present a formal proof. This proof is from the paper by Bednarz, Bowden, Bean and Roughan (2011, submitted), and we wish to acknowledge the contribution of Rhys Bowden in formulating this proof.

Let $C : \mathbb{Z}^+ \rightarrow \mathbb{R}$, be defined by $C(w) = \max_i \left\{ \frac{\alpha_i w_i}{w_i+1} \right\}$ where $w = (w_1, w_2, \ldots, w_V)$. That is, $C(w)$ returns the maximum exterior angle in the feasible region after applying allocation $w$.

Let $Z : \mathbb{Z}^+ \rightarrow \mathbb{R}$ be defined by $Z(n) = \min_{w \in W_n} C(w)$, where $W_n = \{ w | \sum_{i=1}^V w_i = n \}$. That is, $Z(n)$ is the minimax exterior angle over all possible allocations $w$ of $n$ lines.

Following Protocol 9.1, we say a sequence $(w(n))_{n=0}^N$ is a greedy sequence for $\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_V]$ if:

(i) $w(0) = \mathbf{0}$, and

(ii) $w(n+1) = w(n) + e_j$, where $j \in \operatorname{argmax}_i \left\{ \frac{\alpha_i}{w_i(n)+1} \right\}$. If there is a tie for $j$, choose the smallest index.

That is, at each step we add the line to the largest angle. Consequently,

$$C(w(n)) = \max_i \frac{\alpha_i}{w_i(n)+1} = \frac{\alpha_j}{w_j(n)+1}.$$ 

Note $\sum_{i=1}^V w_i(n) = n$. Also note that $C(w(n))$ is a non-increasing sequence since adding $e_j$ (the $j^{th}$ standard unit vector in $\mathbb{R}^V$) to $w$ can never increase $C(w)$, as the denominator of (9.3.1) will increase for $\theta_j$ and remain the same for all other $\theta_i$, $i \neq j$, $i = 1, 2, \ldots, V$.

**Proposition 9.2.1.** Any greedy sequence $(w(n))_{n=0}^N$ is optimal for each $n$, i.e., $C(w(n)) = Z(n)$, $\forall n = 0, 1, \ldots, N$. 
Chapter 9. An Attack on Privacy-Preserving Linear Programming

Proof. Let \((w(n))_{n=0}^{N}\) be a greedy sequence for \(\alpha\). Assume that \(w(k)\) is not the optimal assignment; that is, \(C(w(k)) \neq Z(k)\). Then there exists an alternative assignment \(h^*(k)\) with a smaller maximum exterior angle, i.e.,

\[
C(h^*(k)) < C(w(k)), \tag{9.5.1}
\]

and \(\sum_{i=1}^{V} h^*_i(k) = k\).

Since \(h^*(k) \neq w(k)\) yet \(\sum_{i=1}^{V} h^*_i(k) = \sum_{i=1}^{V} w_i(k) = k\), the two assignments must differ in at least two positions. In particular, there exists an index \(p\) such that:

\[
h^*_p(k) < w_p(k). \tag{9.5.2}
\]

Now consider the element \(w_p(n)\) as a function of \(n = 0, 1, \ldots, k\).

Since \(w(0) = [0, 0, \ldots, 0]\), we have that \(w_p(0) = 0 \leq h^*_p(k)\), and at the final round \(k\) we have that \(w_p(k) > h^*_p(k)\), by (9.5.2). Also, \(w_p(n) - w_p(n-1) = 0\) or \(1\), by definition of a greedy sequence (in each round, an index either remains unchanged or is increased by 1 unit).

Therefore, there exists some \(n < k\) at which \(w_p(n) = h^*_p(n)\). Let \(n = L\) be the greatest such \(n\). That is, round \(L\) is the last time that allocations \(w\) and \(h^*\) had the same value at index \(p\). Then \(w(L+1) = w(L) + e_p\) by the definition of \(L\) and by (9.5.2).

The maximum angle in round \(L\) (under allocation \(w(L)\)) must therefore have been at index \(p\). Thus

\[
C(w(L)) = \frac{\alpha_p}{w_p(L) + 1} = \frac{\alpha_p}{h^*_p(L) + 1} \leq C(h^*(k)), \tag{9.5.3}
\]

since \(C(h^*(k)) = \max_i \left\{ \frac{\alpha_i}{h^*_i(k) + 1} \right\} \). Also, since \(C(w(n))\) is non-increasing, and \(k > L\),

\[
C(w(k)) \leq C(w(L)). \tag{9.5.4}
\]

Therefore, combining (9.5.3) and (9.5.4), we have

\[
C(w(k)) \leq C(w(L)) \leq C(h^*(k)) \Rightarrow C(h^*(k)) \geq C(w(k)),
\]

thus contradicting our assumption that \(C(h^*(k)) < C(w(k))\). Therefore \(C(w(k)) = Z(k)\) for all \(k = 0, 1, \ldots, N\).

\(\square\)

Remark: Note that the greedy heuristic described is more broadly applicable than the simple problem considered here. It is optimal for any min-max problem with \(C(w)\) satisfying \(C(w) = \max_i f(w_i)\) with \(f\) non-increasing.
9.6 Discussion

Many of the existing papers on privacy-preserving linear programming have either made no explicit assumptions about adversarial behaviour [86, 87], or have assumed that the participants are ‘honest-but-curious’ [11, 79, 121, 122]; that is, they will follow the protocol correctly, but will attempt to derive extra information from any intermediate output, and the final output.

In particular, few of the existing papers have considered malicious behaviour. This is presumably because malicious behaviour (such as providing false input) normally results in a non-optimal solution, and since it is assumed that both parties are interested in the result, such malicious behaviour is discounted. As Li and Atallah [79] point out, ‘There is no incentive for a dishonest participant to deviate from the protocol in order to obtain a non-optimal solution.’ However, as we show in this paper, malicious behaviour and a correct optimal solution need not be mutually exclusive; Bob’s attack can be designed to have negligible effect on the result of the optimisation.

Li and Atallah [79] point out that a party may decide to deviate from the protocol if their malicious actions won’t be detected by the other party. However the attack they describe is specific to their protocol, and involves a participant deliberately manipulating their shares of additively split data values in order to cause the calculation to be incorrect. Such an attack is unlikely to be appealing to an adversary, unless the adversary genuinely doesn’t care about the optimality of the solution. To prevent this form of dishonest behaviour, Li and Atallah introduce cryptographic commitment schemes and a post-solution checking phase in which each participant proves to the other party that the solution to the linear program satisfies all the constraints. As Toft [119] notes, although this type of ad-hoc checking may constrain malicious behaviour, it cannot rule it out altogether. Indeed, we note that this type of verification would not work against our attack, since Bob could simply commit to his additional constraints as well.

Toft’s [119] protocol also works in the presence of malicious adversaries. However, Toft’s method assumes that all parameters (constraints, objective function) are split element-wise between the participants; thus his protocol does not apply to our scenario.

It is important to point out that the constraint-insertion attack relies on the malicious adversary receiving the output of the linear program. If the output $x^*$ is revealed only to Alice, and not to Bob, the attack is no longer a threat. Alice could safely tell Bob $f(x^*)$, but nothing more (note that Bob doesn’t know $f$, so he can’t test every vertex in turn to see which $x$ gives him $f(x^*)$).

Other algorithms which do not reveal the final LP solution to both parties may, or may not, be immune to this attack. For example, Vaidya’s revised Simplex method [122]
only reveals the optimal solution to one party, but it happens to be the party in possession of the constraints, and therefore the attack is still possible. Toft’s method [119], on the other hand, provides shares of the final answer to each party, thereby preventing this type of attack. Indeed, sharing the output in additively-split fashion, if permitted, is the simplest strategy to prevent our attack. This means that Alice receives $x^*_A$ and Bob receives $x^*_B$ such that $x^* = x^*_A + x^*_B$. This attack provides yet another reason why care must be taken when deciding whether or not it is safe to reveal the output of a secure computation to the parties.

As an aside, one may wonder whether Alice or Bob could apply duality theory to learn more information than the attack described here; for example, the marginal costs. However, without knowledge of the objective function coefficients, Bob is unable to formulate the dual problem. Meanwhile, Alice can potentially formulate the dual problem for the transformed linear program (if the parties are using a transformation method), but this would only reveal the marginal prices of the transformed constraints, and thus would give her no meaningful information about Bob’s true constraints.

### 9.7 Conclusion

We have presented an attack on privacy-preserving linear programming, which is undetectable and largely independent of the solution algorithm. Furthermore, the attack will not cause the output to significantly deviate from the correct solution (a common deterrent against other attacks, such as inputting false data, since we assume both parties would like to know the correct result). The attack applies to the two-party case where the malicious party owns the constraint set, and the other party owns the objective function. It may be possible to perform a variation of this attack under other data-partitioning environments, but we leave this as an avenue for future research.

In addition, we have shown how to optimally execute the attack, and proven that a greedy heuristic for achieving this is guaranteed to always give the optimal strategy. Given that the optimisation problem is an integer program, the fact that it can be solved optimally using a greedy heuristic makes the attack easy to execute. Our aim in describing this attack is to show that no matter how ‘secure’ a method is, there may still be ways for a malicious adversary to extract more information from the output than expected. The only way to avoid this attack is to prevent the party holding the constraint set from learning the optimal solution, or to provide the output in shares.

The door is open to consider other attacks that may be possible in this domain. In particular, this attack highlights the need to always be aware of what the output of a secure computation may reveal, and whether it is safe to reveal it to all parties.
Chapter 10

Conclusion and Outlook

Privacy-preserving linear programming is a relatively new application of secure multiparty computation. The research contained in this thesis has added to the body of work relating to the privacy-preserving linear programming methodology by:

- Providing a notation system for privacy-preserving linear programs, inspired by Kendall’s queueing notation. This will provide a common language for these problems, and address the extra features not able to be captured by regular linear programming notation.

- Demonstrating a flaw in two of the existing transformation methods that could lead to infeasible results, and explaining how to correct these methods for a certain partitioning scenario (O(A)/O(B)/O(B)/≤/N/O(B)/P problems).

- Providing a thorough analysis of security for a transformation-based method; these methods had previously been criticised for their ‘heuristic’ security guarantees. Our approach focused on determining the probability that private data is uncovered (as a function of the size of the problem), so that users can quantify the risk of performing the optimisation and assess whether it is acceptable to them.

- Providing a new transformation method for row-partitioned linear programs with non-negativity constraints, a common case in practice which was not able to be addressed with any of the existing transformation methods.

- Developing an interior-point-based cryptographic method; all previous cryptographic methods had been based on the Simplex method.

- Demonstrating an attack on privacy-preserving linear programming that is largely independent of the solution algorithm, and, more importantly, is undetectable and will not cause the optimal solution to deviate significantly from the true optimal.

These findings open the door for further developments in this field. Future research could examine:
• Other partitioning scenarios

There are partitioning scenarios for which there are currently no transformation-based methods available. e.g. column-partitioned constraints with non-negative variables. As well, there may be some partitioning scenarios for which it is impossible to develop a secure transformation method; it would be useful to identify these.

• Ways to strengthen the security of the transformation-based methods

There may be ways to improve the security level of the transformation-based method developed in Chapter 6 of this work so that it doesn’t rely on a workload defence. This would allow the method to be applied to small problems.

• Other optimisation problems

The methods for privacy-preserving linear programming could be extended to solve similar problems in quadratic programming, non-linear programming and integer programming. In addition, one could look at privacy-preserving versions of some NP-Hard problems, such as:

– privacy-preserving travelling salesman problem (some work has already been done by [109], as discussed in Chapter 2).

– privacy-preserving bin-packing problem — some objects are owned by one party, and some by another; the objective is to pack them all in the minimum number of shipping containers.

It is our belief that, with increased awareness of the technique in the mathematics and computer science community, and beyond, privacy-preserving linear programming may one day become a commercially-viable technique that can assist people in making decisions involving private information, with minimal risk to the security of their data.
Appendix A

Library of Methods

This appendix provides the algorithms for some common building blocks — the 1-out-of-2 oblivious transfer algorithm, along with four others.

A.1 1-out-of-2 oblivious transfer algorithm

The steps of the 1-out-of-2 oblivious transfer algorithm are illustrated in Figure A.1. Alice has an array of two messages, $M_1$ and $M_2$, and Bob would like to access message $M_2$ (i.e. his desired index is 2). The aim is for Alice to send precisely $M_2$ to Bob, without knowing which message she sent; and for Bob to learn $M_2$ and $M_2$ only.

The method can be instantiated using public key cryptography. Briefly, public key cryptography is based on the idea of each user having two keys: a ‘public’ key, known to everyone in the system, and a corresponding ‘private’ key, known to them only. The public and private key form a pair, in that any message encrypted with the public key can be decrypted with the corresponding private key. Given someone’s public key, it is very difficult to find the corresponding private key; it is equivalent to factorising a large prime number. Thus it is safe to ‘publish’ your public keys. If anyone wishes to write a message for your eyes only, they must encrypt it using your public key. This ensures that you alone will be able to decrypt it, since only you know the private decryption key.

In 1-out-of-2 oblivious transfer, Alice generates one public/private key pair for each of her messages. The public keys are denoted $K_1$ and $K_2$, and the corresponding private keys are denoted by $K_1^{-1}$ and $K_2^{-1}$ (the inverse notation is used to indicate that the private and public key ‘undo’ each other). Bob generates one public/private key pair $C$ and $C^{-1}$. The remaining steps are described in Figure A.1.

Note that there are several ways of implementing oblivious transfer. The version we present here was adapted from Mu et al. [91] and Fischer [55]. Other protocols for oblivious transfer can be found in Naor and Pinkas [94].
Oblivious Transfer Protocol

**Set-up Stage**

<table>
<thead>
<tr>
<th>Alice’s Data</th>
<th>Bob’s Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Messages</strong></td>
<td>Desired index 2</td>
</tr>
<tr>
<td><strong>Public keys</strong></td>
<td>Public key C</td>
</tr>
<tr>
<td><strong>Private keys</strong></td>
<td>Private key C⁻¹</td>
</tr>
</tbody>
</table>

**Notation**

Use public key to encrypt a message $M_1$: $K_1(M_1)$

Use private key $K_i^{-1}$ to decrypt a message: $K_i^{-1}[K_i(M_i)] = M_i$

**Protocol**

1. Alice sends her public keys to Bob.

2. Bob encrypts his public key $C$ with either $K_1$ or $K_2$, depending on whether he wants to read $M_1$ or $M_2$. In this case he wants to read $M_2$, so he chooses $K_2$. He sends his encrypted key to Alice.

3. Alice doesn’t know whether Bob has used $K_1$ or $K_2$, so she has no choice but to try both of them. She tries to decrypt Bob’s message using $K_1^{-1}$ and $K_2^{-1}$ in turn. Alice will obtain a “junk” key in cell 1, and Bob’s true key in cell 2. Importantly, these keys will look the same to Alice — she will not be able to tell which one is Bob’s key.

4. Alice uses the first key, $J$, to encrypt $M_1$ and the second key, $C$, to encrypt $M_2$. In this way, we ensure that Bob’s desired message will be encrypted with his key. Alice sends the encrypted messages to Bob.

5. Bob uses $C^{-1}$ to decrypt cell 2 and obtain $M_2$. If he attempts to decrypt cell 1, he will get rubbish, since cell 1 was encrypted with a different key. In this way, Alice’s other data is protected.

**Figure A.1:** The 1-out-of-2 Oblivious Transfer Protocol
A.2 Selected secure building blocks

In this section we present some of the common secure building blocks from existing papers in a step-by-step illustrated format. The four building blocks given, in order, are:

- Secure multiplication protocol using a commodity server [52], illustrated in Figure A.2;
- Secure system of equations protocol using a semi-trusted third party [71], illustrated in Figure A.3;
- Secure inverse of matrix sum protocol [65], illustrated in Figure A.4; and
- Secure division protocol [1], illustrated in Figure A.5.
Matrix Product with Semi-Trusted Third-Party
(Du, Han and Chen 2004)

Goal: $AB = V_A + V_B$

random: $R_A (n \times N), R_B (N \times m), r_A (n \times m)$
Then sets $r_B = R_A R_B - r_A$

$A = A + R_A$
$B = B + R_B$

Let $A = A + R_A$
$B = B + R_B$

$T = A \cdot B + (R_A - V_B)$

$V_A = T + r_A - (R_A \cdot B)$

CHECK: $V_A + V_B = AB$

LHS = $T + r_A - (R_A \cdot B) + AB + r_B - T$

$= r_A - R_A (B + R_B) + (A + R_A) B + r_B$

$= r_A - R_A B - R_A R_B + AB + R_A B + r_B$

$= AB + r_A - R_A R_B$

$= AB$

$= RHS$

Figure A.2: The secure multiplication protocol using a commodity server, from Du, Han and Chen [52]
Privacy-Preserving Linear System of Equations Solver
(with semi-trusted third party)
(adapted from Kang and Hong 2007)

Goal:
each party receives $x$, where

\[(A_1 + A_2)x = b_1 + b_2\]

random: $T (m \times m)$, invertible

\[S\]

$E_1 = TB_1S$
$g_1 = Tc_1$

$E_2 = TB_2S$
$g_2 = Tc_2$

Tim solves for $y$:

\[(E_1 + E_2)y = g_1 + g_2\]
\[i.e. \ (T(B_1+B_2)SS^{-1} x = T(c_1 + c_2))\]

$y = Sy$

$x = Sy$

Figure A.3: The secure linear system of equations solver, adapted from Kang and Hong [71]. The secure split protocol step, where $A_1 + A_2 = b_1 + b_2$ is changed into $B_1 + B_2 = c_1 + c_2$, is omitted since it appears to afford no additional security. If Tim colludes with Bob, say, and sends Bob $TB_1S$, then Bob can uncover $B_1$. With the step omitted, $B_1 = A_1$, and Bob knows Alice’s data. With the step included, it’s no different, since Bob knows $B_2$. Thus Bob can find $B_1 + B_2$, and he’ll know that it’s equal to $A_1 + A_2$. He can subtract $A_2$ to find $A_1$. 
Secure Inverse of Matrix Sum
(Han and Ng 2008)

Goal:
\[(A+B)^{-1} = M_a + M_b\]

Secure Matrix Multiplication
\[AP = S_a + S_b\]

\[T = S_b + BP\]

\[V = S_a + T\]
\[= S_a + S_b + BP\]
\[= AP + BP\]
\[= (A + B)P\]

\[V^{-1} = P^{-1}(A + B)^{-1}\]

Secure Matrix Multiplication
\[PP^{-1}(A+B)^{-1} = M_a + M_b\]

Figure A.4: The secure inverse protocol, from Han and Ng [65].
Secure Division

(Amirbekyan and Estivill-Castro 2007)

Goal:

\( \frac{(a_1 + b_1)}{(a_2 + b_2)} = M_a + M_b \)

random number \( r_1 \)

random number \( r_2 \)

Scalar product protocol

(only Alice learns the result)

\( (a_2, 1) \cdot \begin{bmatrix} r_2 \\ r_2b_2 \end{bmatrix} = r_2(a_2 + b_2) \)

Scalar product protocol

(only Bob learns the result)

\( (b_2, 1) \cdot \begin{bmatrix} r_1 \\ r_1a_2 \end{bmatrix} = r_1(a_2 + b_2) \)

Scalar product protocol

\( \begin{bmatrix} r_1a_1 \\ 1 \\ r_1(a_2 + b_2) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ r_2(a_2 + b_2) \\ r_2b_2 \end{bmatrix} = \frac{r_1a_1}{r_1(a_2 + b_2)} + \frac{r_2b_1}{r_1(a_2 + b_2)} = \frac{r_1a_1 + r_2b_1}{r_1(a_2 + b_2)} = \frac{(a_1 + b_1)}{(a_2 + b_2)} \)

CHECK:

\( \begin{bmatrix} r_1a_1 \\ 1 \\ r_1(a_2 + b_2) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ r_2(a_2 + b_2) \\ r_2b_2 \end{bmatrix} = \frac{r_1a_1}{r_1(a_2 + b_2)} + \frac{r_2b_1}{r_1(a_2 + b_2)} = \frac{r_1a_1 + r_2b_1}{r_1(a_2 + b_2)} = \frac{(a_1 + b_1)}{(a_2 + b_2)} \)

Figure A.5: The secure division protocol, from Amirbekyan and Estivill-Castro [1].
Appendix B

Regular LPs in Canonical Form

This appendix explains the standard conversion processes for regular LPs.

B.1 Standard conversion procedures for regular LPs

In regular linear programming all problem forms are all equivalent, and it is a simple matter to convert a linear program from one form to another. Two common forms of linear programs are ‘standard form’ and ‘canonical form’.

Standard Form
\[
\begin{align*}
\text{max} & \quad f(x) = c^T x \\
\text{s.t.} & \quad Mx = b \\
& \quad x \geq 0, \\
\end{align*}
\]  
(B.1.1)

Canonical Form
\[
\begin{align*}
\text{max} & \quad f(x) = c^T x \\
\text{s.t.} & \quad Mx \leq b \\
& \quad x \geq 0, \\
\end{align*}
\]  
(B.1.2)

Any linear program can be converted into these forms using the procedures explained below.

B.1.1 Constraint conversions

To convert a greater-than-or-equal-to constraint to a less-than-or-equal-to constraint, simply multiply through by $-1$. To change an inequality constraint to an equality constraint, we introduce new variables, called slack or surplus variables, to ‘balance out’ the inequalities, much as we would balance a set of scales. Slack variables are added to $\leq$ constraints to ‘weigh out’ the LHS so that it reaches the RHS. Surplus variables are subtracted from $\geq$ constraints to reduce (‘lighten’) the LHS so that it equals the RHS.

\footnote{There is no consensus regarding the definitions of standard and canonical form \cite{102}, so these formulations may differ from those in other texts.}
For example, the constraint $3x_1 + 5x_2 \leq 7$ can be converted into the equivalent form

$$3x_1 + 5x_2 + s_1 = 7$$

by using the slack variable $s_1 \geq 0$, where $s_1 = 7 - 3x_1 + 5x_2$ (i.e. the difference between the RHS and the LHS of the constraint). A feasible solution $(x_1, x_2)$ for the inequality constraint generates an equivalent feasible solution $(x_1, x_2, s_1)$ for the equality version [102].

Similarly, the constraint $2x_1 + x_2 \geq 5$ can be converted into the equivalent form

$$2x_1 + x_2 - s_2 = 5,$$

by using the surplus variable $s_2 \geq 0$, where $s_2 = 2x_1 + x_2 - 5$.

To convert an equality constraint to inequality form, we can write it as two inequality constraints. For example, the equality constraint $2x_1 + 5x_2 = 3$ can be equivalently expressed as the following pair of inequalities: $2x_1 + 5x_2 \leq 3$ and $2x_1 + 5x_2 \geq 3$.

### B.1.2 Objective function conversions

To convert a minimisation problem to a maximisation problem (or vice versa), multiply the objective by $-1$. For example, the objective

$$\text{Min } f = 5x_1 + x_2$$

is equivalent to

$$\text{Max } = -5x_1 - x_2,$$

in the sense that both objectives will return the same optimal solution $x^*$. However, the optimal objective value of the old problem is $-1$ times the optimal objective value of the new problem.

### B.1.3 Variable conversions

To change a non-positive variable to a non-negative variable, multiply the constraint through by $-1$ and define a new variable. E.g., the constraint $x_2 \leq 0$ becomes $-x_2 \geq 0$, and we define a new variable $x_2' = -x_2$ and re-write the constraint as $x_2' \geq 0$. To change unrestricted (i.e. free) variables to non-negative variables, rewrite them as the difference of two non-negative variables. For example, if $x_2$ is free, set $x_2$ to be the difference of two non-negative variables. That is, set $x_2 = x_2' - x_2''$, where $x_2', x_2'' \geq 0$. This formulation still allows $x_2$ to take any possible value.
Bibliography


