Université Catholique de Louvain
Ecole Polytechnique de Louvain

Center for Operations Research and Econometrics
CORE
1348 Louvain-la-Neuve – Belgium

Network Flow Problems with Secure Multiparty Computation

Abdelrahaman Aly

Doctoral Thesis

Jury:
Philippe Chevalier (UCL, Belgium)
Olivier Pereira (UCL, Belgium)
Frits Spieksma (KULeuven, Belgium)
François-Xavier Standaert (UCL, Belgium)
Laurence Wolsey (UCL, Belgium)

Supervisor:
Mathieu Van Vyve (UCL, Belgium)

President:
Philippe Chevalier (UCL, Belgium)

PhD Organization

Abdelrahaman Aly
Université catholique de Louvain
Ecole Politechnique de Louvain
Center of Operations Research and Econometrics

Thesis Supervisor

Mathieu Van Vyve
Professor, Université catholique de Louvain
Louvain School of Management
Center of Operations Research and Econometrics

Supervisory Committee

Phillippe Chevalier
Professor, Université catholique de Louvain
Louvain School of Management
Center of Operations Research and Econometrics

Olivier Pereira
Professor, Université catholique de Louvain
Ecole Polytechnique de Louvain
Crypto Group

We are not simply in the universe, we are part of it. We are born from it. One might even say we have been empowered by the universe to figure itself out.

by Neil deGrasse Tyson

Acknowledgments

I have to confess, my life as a PhD Student has been quite an adventure, with many ups and some downs. But, most of all, full of many people that selflessly offered me a hand, without expecting anything in return. Finally, after the struggles and the happy moments, the smiles and the tough moments, I get the opportunity to thank everybody that has helped me.

I feel I should start from the beginning. If there is one person I have to thank, more than anybody else, it would be my advisor Mathieu Van Vyve. There are so many things I could tell you about the experience of working beside him. I could tell you, for instance, about how many times his brilliant insights opened many research paths that seemed to be closed. Or the many times that with honesty and in the most straight-forward fashion he trimmed many of my constant flaws to try to make me a better researcher, and a better writer for that matter. Or the infinite patience and wisdom with which he observed my first steps. But I would honestly prefer to tell you that during all these years he was, more than anything, my friend, someone I could count on. I prefer to tell you about the smiles we shared so many times in his office, talking about anything you can think of. The times when early in the morning he tried to cheer me up showing me the latest poll results, or when he sympathetically "convinced" me to run half marathons. Thanks so much Mathieu, for this opportunity that changed my life, but more importantly for your support and kindness.

To Philippe Chevalier from whom, after 4 years and more at CORE, I cannot remember a single instance when he did not have a smile to offer me. In any situation, either about this PhD dissertation, student visits or their grades, no matter how big the problem was (and we had some!) he always tried to make me feel comfortable and happy, even allowing me, from time to time, to exchange with him some words in Spanish. Thanks for entrusting me, together with Mathieu, your students and your confidence for the last 4 years.

To Olivier Pereira, who's valuable insights helped us from the very beginning of this quest. His love for what he does is self evident and inspiring. I learned so much from him, and this is because he never had his doors closed for me, even coming to my office to chat about data-obliviousness and Oblivious RAM. Thanks so much Olivier, for the chance to learn with you and from you, not only about academic matters, but how cryptology can impact the world, and how passion for what you do feels like.

To the PhD Jury, composed by Prof. Spieksma, Prof. Standaert and Prof. Wolsey, for your nice comments, improvements, suggestions and help. For you to have accepted to be part of my Jury or just the fact that you have taken the time to read my work, even with my so frequent typos, is for me, an undeserved honor. Thanks so much for the interest you showed, and your valuable time spent on improving this dissertation.

To all the people who I had the opportunity to share my work with and, in return always got an open mind, comments, suggestions and improvements. To everyone who helped me to open seemingly closed doors, and had patience to share the knowledge I did not have. To Sophie Mawet, Edouard Cuvelier, François Koeune, Thomas Peters, Tomas Toft, Claudio Orlandi, Marcel Keller, Mehdi Madani and of course to CORE's chilean community: Alejandro Lamas and her wife Tanja Mlinar, Claudio Telha, Ignacio Aravena and Gustavo Angulo. To Angela Ocampo for the patience, time and love you dedicated to proofreading this thesis. For all other people like Sonia Trabelsi and Sebastian Martin who did something to make of this a better dissertation. Thanks so much for your support through the years. Research in Operational Research and Cryptology have an ample and exciting future with researchers like you among their ranks.

To so many people that made me feel at home during my time at CORE, like Adel Hatami, Fahimeh Shamsaei, Bartosz Filipecky, Francisco Santana Ferra, but specially to Catherine Germain or, as many people calls her, our mom at CORE. From the smallest detail to the biggest issue, she was always there, worried for us, sharing stories to help us abstract ourselves from the bureaucratic issues that at times can be stressful. Dear Catherine, I can say with confidence, after this 4 years, that CORE would not be the same without you. Thanks so much.

To all the non-expected friends that even without knowing me, did everything they could to make me feel at home. I will always be thankful for the opportunity to share this experience with you guys. From Steph's taste for amercain, to Mikel's love for football. From the gentle invitation to Claudia's Birthday, to Margherita's farewell party. From Paolo trying to teach me a bit about Paretto efficiency to Corinna always trying to speak to me in Spanish to make me feel comfortable. From Rafael teaching how to solve some non-linear problems in the laundry at 10 pm to Gilles talking about how he would love to stop smoking and Sophie sleeping on the floor after a long day at work. I will treasure every single moment.

To all the wonderful people I got to meet at Louvain-la-neuve, this small almost mystical corner of Belgium, the friends and the not so friends, thanks for sharing this adventure along side me. I would like to thank my family, friends and professors at home. To Danilo Martinez and Rolando Reyes at the ESPE, my university, for always being there pushing me forward.

Finally, thanks so much to everybody that was part of my life these past 4 years and, for letting me share a little of myself with you. When I started my PhD, I wanted to keep something from every difficult day, so I could appreciate better the road traveled. I end up keeping all the RedBull cans I drunk every long day during my PhD. Thanks to this and together with other PhD Students (Dr. Stéphane Bouché and Mr. Andrea Pirrone) we have formalized the following model: Consider r_i to be the daily RedBull input, c_i the daily cost in terms of effort and h is the health of the subject. In our model t is the time for the completion of the PhD (a bit more than 4 years in my case). $min \quad \sum_{i=1}^{t} c_i$ such that $c_i = S_i - \sqrt{r_i - \alpha \cdot r_{i-1}} \quad \forall i \in \{1, ..., t\}$, $h = \nu - \sum_{i=1}^{t} ln(r_i + \alpha r_{i-1})$ and $0 < h \leq \nu$. We derive that the only feasible solution possible to the system is 178 RedBull cans. As an empirical proof we offer as evidence the 178 cans above one of the drawers in the office D-360 at CORE building. The values of the constants are left as an exercise for the reader. Although I'm not sure whether RedBull gives you wings, I'm sure you did, and I will always be thankful for that.

Abstract

In this PhD dissertation, we investigate how to solve some classical combinatorial optimization problems and applications using secure multiparty computation. Our study highlights the differences between traditional and secure adaptations of some algorithms to later test its implementation. It also explores various trade-offs between performance and security. We provide protocols that can be used as building blocks to solve more complex problems. Additionally, we report on practical applications, more specifically, we study the problem of securely building auction mechanisms with transmission constraints. We focus on improving performance for real life applications. We report on the design of a specific Object Oriented C++ implementation of the necessary secure multiparty computation protocols used for the experimentation on practical applications. Areas of interest for our work can be found in: auction markets, communication networks, routing data from rival company hubs, distribution problems, amongst others.

Contents

I Introduction 1

1 Generalities 3
1.1 Motivation . . . 3
1.1.1 Theoretical Perspective . . . 4
1.1.2 Practical Perspective . . . 5
1.2 Thesis Structure . . . 5

2 Foundations 7
2.1 Secure Multiparty Computation: Security Model . . . 7
2.1.1 Communicational Model . . . 8
2.1.2 Adversarial Behavior . . . 9
2.1.3 Achievable Security under MPC . . . 10
2.2 Secret Sharing . . . 11
2.2.1 Finite Fields . . . 12
2.2.2 Shamir's Secret Sharing . . . 12
2.2.3 Other Sharing Mechanisms . . . 16
2.3 Secure Multiparty Computation: Applications and Primitives . 17
2.3.1 Secure Multiplication . . . 17
2.3.2 Secure Comparison . . . 18
2.3.3 Software Frameworks . . . 20
2.4 Combinatorial Optimization . . . 22
2.4.1 Graph Theory Preliminaries . . . 23
2.4.2 Some Classical Network Flow Problems . . . 24
2.4.3 Matroids . . . 25
2.4.4 Submodular Functions . . . 25

II Secure Classical Network Flow Problems 27

3 Securely Solving The Shortest Path Problem 29
3.1 Introduction . . . 29

3.1.1 Our Contribution 30
3.1.2 Related Works 32
3.1.3 Overview 34
3.2 Preliminaries 34
3.2.1 Security 34
3.2.2 Cryptographic Assumptions 34
3.2.3 The Arithmetic Black-Box 35
3.2.4 Notation 35
3.3 Shortest Path Problem 36
3.4 Bellman-Ford's Algorithm 37
3.5 Dijkstra's Algorithm 39
3.5.1 Partial Data-Oblivious Adaptation 41
3.6 Computational Experimentation 41
3.6.1 Bellman Ford 43
3.6.2 Data-Oblivious Dijkstra 44

4 Securely Solving the Max Flow, Minimum Cost Flow and Other Related Problems 47
4.1 Introduction 47
4.1.1 Our Contributions 48
4.1.2 Related Work 50
4.1.3 Overview 51
4.2 Preliminaries 51
4.2.1 Cryptographic Foundations 51
4.2.2 Notation 52
4.2.3 On Network Flows and Matrix Representation 53
4.3 Maximum Flow Problem 53
4.4 Privacy-Preserving FIFO Push/Relabel Algorithm 55
4.5 Minimum Mean Cycle Problem 56
4.6 Privacy-Preserving Minimum Mean Cycle 57
4.7 Minimum Cost Flow Problem 59
4.8 Privacy-Preserving Minimum-Cost Flow 61
4.9 Computational Experiments 63
4.9.1 Maximum Flow Problem 64
4.9.2 Minimum Cost Flow Problem 66

5 Conclusions 68

III Towards Practically Efficient Applications 69

6 A Modular and Compact MPC Toolkit 71
6.1 Introduction 71

6.1.1 Related Work 72
6.1.2 Application Features 74
6.1.3 Overview 76
6.2 Architecture and Design 76
6.2.1 Architecture and Task Division 77
6.2.2 Information Movement and Operation Counter 79
6.2.3 Communications Support 81
6.2.4 Parametrization 81
6.2.5 MPC Functionality 82
6.3 Prototype Capabilities and Technical Characteristics 82
6.4 Engine Functionality 87
6.4.1 Arithmetic Operations 88
6.4.2 Bitwise Operations 89
6.4.3 Logical Operations 90
6.5 Computational Experimentation and Network Benchmarking . 91
6.6 Road Map and Future Work 96

7 Practically Efficient Secure Auctions with Transmission Constraints 98
7.1 Introduction 98
7.1.1 Our Contribution 99
7.1.2 Related Work 100
7.1.3 Overview 102
7.2 Problem Overview 102
7.2.1 Auction Mechanism 102
7.2.2 Problem Definition 103
7.2.3 Problem Formulation 104
7.3 Network Flow Formulation 104
7.3.1 Greedy Algorithm 105
7.3.2 Correctness 106
7.4 Cryptographic Preliminaries 107
7.4.1 Security Model 107
7.4.2 Basic Building Blocks 108
7.4.3 Complex building Blocks 109
7.5 Secure Auction Mechanism 112
7.5.1 Notation 112
7.5.2 Secure Auction with Transmission Constraints 113
7.5.3 Security and Correctness 116
7.6 Computational Experimentation 116
7.6.1 Prototype Capabilities and Technical Characteristics . . 117
7.6.2 Numerical Results 118

8 Conclusions 122

List of Figures

2.2.1 Example: Secret Polynomials 15
2.2.2 Example: Secure Addition 15

3.6.1 Bellman-Ford CPU Times and Ratio Analysis 44
3.6.2 Dijkstra CPU Times and Ratio Analysis 45

4.9.1 Secure Max Flow CPU Times and Ratio Analysis 65
4.9.2 Secure MMCC CPU Times and Ratio Analysis 66

6.2.1 `SmcEngines:ShamirSharesEngine` Invocation Diagram 77
6.2.2 MPC Toolkit Component Architecture 78
6.2.3 MPC Toolkit Logical Architecture 78
6.5.1 MPC Toolkit Multiplication and Comparison Life Cycle 92
6.5.2 MPC Toolkit Bellman-Ford Test CPU Times 95

7.6.1 Secure Auction Protocol Life Cycle 120

List of Tables

2.1 Sharing Example 13
2.2 Addition Example: sharing phase 14
2.3 Addition Example: parties privately add shares 15

3.1 Worst-case bounds of Original and Privacy-Preserving algorithmic versions 32
3.2 CPU Time from Secure Bellman-Ford protocol 43
3.3 CPU Time from Secure Dijkstra protocol 44

4.1 Worst-case bounds of Original and Privacy-Preserving algorithms 50
4.2 Execution times per phase Secure Max Flow Algorithm for a complete graph. 64
4.3 Execution times per phase MMCC Algorithm for a complete graph. 66

6.1 Basic MPC functionality provided by the ToolKit 82
6.2 Secure Arithmetic Operations on MPC ToolKit 88
6.3 Secure Bitwise Operations on MPC ToolKit 89
6.4 Secure Logic Operations on MPC ToolKit 90
6.5 Secure Catrina and Hoogh Comparison Formulations on MPC ToolKit 90
6.6 CPU Time in Seconds from Multiplications over LAN Ethernet 93
6.7 CPU Time in Seconds from Comparisons over LAN Ethernet . 94
6.8 Secure Logic Operations on MPC ToolKit 95

7.1 List of Primitives used by the Secure Auction protocol implementation 117
7.2 Overall Results 119

It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to.

by J.R.R. Tolkien

Part I

Introduction

Chapter 1

Generalities

1.1 Motivation

Imagine a setting where computations that involve private data from several sources cannot be done without fear of disclosing this information to your direct competition. Now, consider that such computations are done over some kind of finite set for a problem that is combinatorial in nature, and that problem has to be solved to optimality. Moreover, the data from calculations is provided by competing agents that could use the information coming from other participants in the process for their own advantage.

It is easy to imagine such scenario in various environments, from Information Technology (IT) platforms where competing agents have to configure their appliances (e.g. routers, switches or satellites) to improve traffic speeds and avoid any type of collisions, to day-ahead electricity markets, where demand and supply bids from competing parties with different interests have to be accepted and rejected.

A solution approach would require for these agents to trust a third party such that they can transfer to this party all private information, and in exchange, it would return to all the parties the result of the computation. This ideal approach would create the need of finding such agent, and elements like security and fairness (parties learn about the results at the same time) to be its responsibility. Note that this is not always possible in practice. In many cases such third party could be coerced to modify the calculations in favor of any other party or disclose sensible information to other agents. Detection of such corruption is not always obvious.

Secure Multiparty Computation (MPC) is the theoretical answer to the absence of this ideal third party. Roughly speaking, MPC allows the parties involved in the computation to simulate the third party functionality, such that the computations can be done between the competing agents alone. MPC can guarantee different levels of security and fairness in various models. The basic concept of the problem was introduced by Yao's [1] paper in 1982 with the millionaires problem. Basically, two millionaires want to find who is richer. The caveat is that none of them wants to disclose inadvertently any information to the other about its wealth. The question Yao's raises refers to how such conversation should be carried out (a solution can be achieved using 1-2 oblivious transfers, which is a way to obtain an entry from a data collection such that the receiver does not learn anything from the other entries and neither the sender about the element that was queried). Nowadays, this question has muted towards a field of itself, embodied by two-party computation. MPC, tries to answer similar questions regardless the number of players.

In this dissertation we discuss how to solve a series of combinatorial problems to optimality using MPC protocols. We introduce some trade-offs between efficiency, and security and provide some computational experimentation. Our conclusions on the more theoretical aspects of the thesis, inspired us to investigate a more practical approach. We not only use available software tools, but also build customizable implementations of the MPC functionality needed by our protocols, not only to accelerate performance, but to provide modularity, adaptation capabilities and scalability. They are later used to solve real life combinatorial problems in auctions of the type encountered day-ahead electricity markets.

It is our belief that, since the early 80's, the study of MPC has evolved from a purely theoretical endeavor towards real and practical applications. This is the reason why we focus our attention not only in providing theoretical results but practical implementations as well. This dissertation gives emphasis to both, a theoretical outlook where polynomial time algorithms are adapted to provide security and correctness, and a practical approach, with experimentation and prototyping.

1.1.1 Theoretical Perspective

The selected combinatorial problems studied, can be solved to optimality by theoretically efficient algorithms (i.e polynomial time) that guarantee correctness. In many of these algorithms, the flow of the algorithm depends on the data (non data-oblivious). Thorough adaptation is needed for these algorithms, such that they can correctly calculate their outputs using MPC protocols and methods (primitives). We introduce secure protocols that are adaptations of such algorithms and provide polynomial bounds with some level of security, (typically perfect security). We provide a proof of their correctness, complexity estimate, and their security.

1.1.2 Practical Perspective

The performance of theoretically efficient algorithms is affected by several factors in real life scenarios. Topics of implementation and the design of the applications and protocols, as well as available resources, including CPU power, have great influence on their behavior. State of the art MPC protocols have an associated computational cost, and they pose challenges on design and workload distribution. For instance, an MPC addition would be less expensive in terms of performance than a multiplication, and in the same way, a multiplication would be less expensive than a comparison. We have adapted our secure protocols to this reality. Moreover, we have built prototypes of our secure protocols such that we can measure and evaluate their performance. In addition, we have extended our research to realistic life scenarios, and provided custom compact and modular MPC tools to our prototypes.

Finally, note that our work centers on the multiparty case, where our secure protocols provide security to any number of parties to securely solve some combinatorial problems to optimality. Moreover, our secure protocols could be easily adapted to be used in the two-party case.

1.2 Thesis Structure

This dissertation is organized as follows: Chapter 2 introduces some basic concepts on security and combinatorial optimization that are universal to the thesis. Problem specifics, like mathematical formulations, are described in detail in further chapters.

Part II introduces theoretically efficient protocols to some known Network Flow Problems. We provide complexity bounds and perform some computational experimentation over the Virtual Ideal Functionality Framework (VIFF) [2]. Part II is divided as follows: we first address the shortest path problem in Chapter 3. We begin by introducing different algorithmic solutions for the problem as well as some variations of Dijkstra's algorithm to provide security with different information access levels and asymptotic bounds. We later present the numerical results of our experimentation. Secondly, in Chapter 4 we discuss the Maximum Flow Problem, Minimum Mean Cost Cycle Problem and the Minimum Cost Flow Problem. We present algorithms to solve these three problems and report on computational experimentation. Finally we include some conclusions in Chapter 5.

Inspired by the results of the experimentation of Part II, we introduce in Part III secure algorithms to real-life problems using MPC protocols. We include novel secure algorithms for such problems, emphasizing practicality, performance analysis and experimentation, with a novel set of custom tools specifically created for this purpose. We have also structured this Part in three chapters as follows: Chapter 6 introduces a secure implementation of the primitives that will be needed by results introduced by following chapters. Previous experimentation raised the need to introduce an architecture based on software composition, modular, adaptable, scalable and relatively efficient (when benchmarked with VIFF) Toolkit, implemented in C++ using an Object Oriented paradigm approach. The toolkit can be used as a third party library in custom-software applications, and its modular design allows easy expansion and code accessibility. Chapter 7 introduces a secure protocol for auctions with transmission constraints inspired by day-ahead electricity markets. We model it as a combinatorial optimization problem. We report on primitive adaptations as well as trade-offs to obtain better performance. We provide computational experimentation with our MPC Toolkit and realistic data scenarios. Finally, Chapter 8 presents some conclusions.

Chapter 2

Foundations

2.1 Secure Multiparty Computation: Security Model

We use the terms "securely" and "privacy preserving" indistinctly. We can succinctly formalize their notion as follows:

Definition 1. *Parties $P_1, ..., P_n$ want to jointly and correctly compute the function $y = f(x_1, ..., x_n)$ where x_i is P_i's secret input. The security constraints are as follows: only y is allowed to be revealed to all parties. In other words, the security constraint is such that each player P_i learns y and what can be inferred from y, but no more. In particular, any information given during the computation process should not allow him to infer information about other secret inputs.*

Consider the case where there is trust and confidence between parties, or external parties in charge of such computation. Definition 1 can be satisfied by the direct exchange of messages between parties. When such confidence and trust is missing, however, such process might be simulated by a subset of untrusted parties. Several models describe the scenarios where similar results to the previously described naive case can be achieved. In other words, the conditions on which distrustful parties can obtain similar results in terms of definition 1.
As mentioned earlier, Yao [1] introduced the millionaires problem, and with it the concept of securely solving a function amongst mutually distrustful parties. This, combined with the work of many early researchers e.g. [3, 4, 5, 6], cemented the basis of secure multiparty computation. To mention an example, we could imagine competing agents forced to exchange information in a computer network, they still want to know what is the best way to send information

between 2 points, without disclosing sensible information about their network configuration. In this case the parties would have to build routing algorithms that are capable to use MPC protocols to calculate the shortest path. This kind of problems arise naturally in several contexts, from distribution networks to different market configurations.
A simplified security notion could be as follows: secure protocols should only disclose the same information to adversary $\mathbb{A}$ than what it might learn when a trusted ideal (no corruptible) third party computes the functionality. For the latter "ideal" setting, it suffices to execute a trivial (non-secure) version of such protocol. We say a protocol is secure if what the adversaries can learn from both settings is the same. In other words, the secure setting can emulate the trusted third party. This conceptualization is true for a variety of adversarial and communicational models.

2.1.1 Communicational Model

The model defines environmental capabilities for the adversary like whether all channels are or not tamper-free. It also defines the behavior of the network as a such. We explore some scenarios based on the categorization made by [7]. A more detailed treatment can be found in [7, 8].
In this context we can talk about two basic models of communication:

Private-Channel. When adversaries are not allowed to tap the communications between parties, this is called the private-model channel [4, 9], also referred to as the Information Theoretic Model [1]. This restricted model can describe some abstract scenarios and is useful for some applications that might not need encrypted communications. In many security configurations, it can provide a clean model without cryptographic questions on the side of the communicational channels, and deliver possible solution paths for the unrestricted model. It also assumes that all parties are connected directly to each other, and the adversary is not allowed to capture or modify incoming messages from parties that have not been corrupted. An example could be a set of competing agents that need to exchange information and securely compute some functions e.g. (Shortest Path Problem) using a Wide Area Network (WAN) where they extend point to point connections using optic fiber.

Standard Cryptographic Model. The adversaries are allowed to tap the communications from the parties involved in the computations (computational parties), this is a standard assumption in cryptography. Note that, in many scenarios and configurations, a solution on the private-channel model can be emulated by this scenario. This model has to consider the security of the channel and not only the protocol to guarantee security.

Furthermore, in MPC, the algorithm designer can introduce some additional characteristics to the communicational model:

Broadcast Channel. In some contexts, the existence of a broadcast channel for the parties might be suggested as well [10]. The results from Ben-Or, Goldwasser and Wigderson [4] (BGW) postulate its use against malicious adversaries.

Network Behavior. This is whether or not the transmission of the data is synchronous or asynchronous. Simply explained, parties have immediate access to the communicational channels and data do not have to be temporally stored to be later processed. This allows them to have immediate message exchange, this model is typically used when designing MPC protocols [7]. Asynchronous communications consider the opposite and assume that, for some reason, online calculations are not implementable. Temporal secrets have to be stored and mechanisms for offline processing have to be implemented e.g. [11].

2.1.2 Adversarial Behavior

The security model of any problem depends on the behavior of the computational parties involved in the calculations. Corrupted parties may do whatever is in their power to learn additional information. Their behavior and capabilities can be classified as follows:

Computational Limitations

In MPC, adversaries could either be computationally-bounded by a polynomial function with some probability, usually referred to as probabilistic polynomial time adversaries (PPT), or they can also be computationally unbounded. The latter is possible under the private-channel model. Moreover, any protocol that is secure against computational unbounded adversaries is trivially secure against computational bounded ones. Our protocols consider unbounded adversaries, unless we explicitly estate the opposite.

Adaptive and Non-Adaptive

A more general characteristic classifies adversaries on its ability to corrupt other parties during the protocol execution. It selects corruptible parties based on partial information that it has collected from the process. This behavior is usually called Adaptive. Moreover, in some settings it suffices to fix in advance the set of corrupted parties from the start of the process, this more restrictive model is called Non-Adaptive. Note that the set of dishonest players is not known by the honest computational parties. Our protocols consider the adaptive case.

Passive

This restricted model is reserved for corrupted players that do not deviate from the secure protocol. Although passive adversaries are allowed to collude amongst themselves, they respect all the algorithmic steps. Instead, they try to learn any additional information about other player's secret from the exchanged messages, the environment and the outputs. This model is usually called the semi-honest model or honest-but-curious. Finally, this restricted model is often used given that it provides methodological focus to the secure protocols.

Active

This model refers to the corrupted party that might take active steps to go against the protocol's behavior. In the case at hand, this implies sending messages that were not calculated by the secure protocol. His interest might be not only to learn extra information but to falsify or interrupt the computations. Corrupted parties are commonly referred to as malicious. If a protocol is secure against active adversaries (unrestricted model), it is also secure against passive adversaries.

2.1.3 Achievable Security under MPC

Different security properties can be achieved under these models by conventional MPC primitives. Secure protocols can be categorized by how difficult it would be for an adversary to reconstruct the information from honest parties. The following is a classical security classification based on [12]:

Perfect Security: In terms of security, the results of the secure protocol exactly reflect those of the ideal functionality (third party). The adversary does not learn any additional information that it would not learn under the ideal setting. This security level considers computational-unbounded adversaries under the information theoretic model.

Statistical Security: Similarly to perfect security, the adversary does not learn more than it would in the ideal setting, but this time only with a statistical probability. Typically the scenario where the adversary can effectively learn additional information is described by some negligible function. Once again this model considers computational - unbounded adversaries and the private channel model.

Computational Security This setting considers a probabilistic polynomial time adversary (PPT) instead of unlimited computational power. Roughly speaking To achieve security under this setting, secure schemes rely on some not deterministic polynomial-time problem such that a polynomial bounded adversary cannot break it e.g. factor decomposition.

Finally, it has been shown by Ben-Or et al. [4] and Chaum et al. [9] that any functionality can be computed securely against active and passive adversaries under the private-channel model. The security level (perfect or statistical) depends solely on the functionality. Moreover, the results of BGW [4], made use of different sharing mechanisms (Shamir Sharing [13] and VSS [14]) to provide security against passive and active adversaries. It has to be noticed that different MPC protocols of similar characteristics have been proposed since the advent of BGW e.g. [15, 16, 17]. Most notably, work by [18] included an improvement of the multiplication protocol from BGW, that is still used by many state of the art MPC implementations e.g [19]. This multiplication mechanism is later used in this dissertation.

2.2 Secret Sharing

Secret sharing mechanisms are one of the basic tools used by the MPC protocols. Although secret sharing has its own research path and development outside multiparty computation, its application and use in the field is vast. Our work pays special attention to Shamir's secret sharing scheme [13] which uses modulo arithmetic over a field $\mathbb{Z}$ greater than the number of players. For a detailed analysis on secret sharing mechanisms we refer the reader to [20]. The secret sharing mechanism is simple in nature. Imagine that you split your values into a set of shares, typically as many as your computational parties. These shares can be seen as randomized representations of the value with no meaning by themselves. Later, when a computation is at hand, you distribute these shares amongst the other computational players, usually a share (different) per player. It can be seen why such a scheme fits so well in a multiparty computation scheme. In the following pages, we focus our attention on the multiparty case, where the number of players n, is greater than 2 ($n > 2$). It has to be noticed that for the two-party computation case, homomorphic encryption is typically preferred over sharing mechanisms where, similar to CPU outsourcing, one player provides data and the other executes the calculation. Some sharing mechanisms require all inputs to reconstruct the problem, meanwhile some others let a subset of players do so, these mechanisms are usually referred to as threshold sharing schemes. In the literature, t is the number of inputs needed to reconstruct the problem (the threshold), and n, the number of players who interact with each other. We present the following definition of threshold schemes provided by Asharov and Lindell [8]:

Definition 2. *A (t,n) secret sharing scheme takes a secret input s and output n shares, such that any subset of t shares is capable to reconstruct s, but any* *subset of less than t elements does not learn anything about s.*

It has to be noticed that s has to belong to a finite field $\mathbb{Z}_p$ where p is a prime number bigger than n. Applications typically make use of considerably big prime numbers (much bigger than the inputs) to define the field size to avoid overflows. Moreover s_i correspond to the share of $i \quad \forall i \in \{1, .., n\}$.

2.2.1 Finite Fields

A field $\mathbb{Z}$ can be of finite or infinite nature. This algebraic object allows universal multiplication and addition (and their respective inverses) of its elements. Examples of infinite fields are for instance the rational numbers, or the real numbers. In this thesis, we make use of finite fields also known as Galois Fields, where their size is also referred to as the order of the field. A Galois field is usually defined as $GF(p^m)$ where p is any prime integer and n is any integer bigger or equal than 1 and $p^m = q$. Our interest however centers specially in the case where $m = 1$. Roughly speaking, that way a field is equal to all integers $mod \quad p$, then it suffices to calculate the modulo p after any addition or multiplication. This is specially useful given that it suffices to use the Standard Arithmetic and Logic Unit (ALU) of the CPU to perform basic sharing operations. It's worth noticing that secret sharing schemes like Shamir's Secret Sharing can work with any finite field.
Fields are a suitable tool to provide perfect security in many cryptographic mechanisms including secret sharing. This is because whenever a party receives a share from its counterparts that was correctly randomized from an integer secret, the probability is uniformly distributed over all elements of $\mathbb{Z}_p$.

2.2.2 Shamir's Secret Sharing

Adi Shamir introduced a secret sharing scheme [13] in 1979. A simple method to share a secret $[s] \in \mathbb{Z}_p$ between n players by giving each party P_i a share $s_i \in \mathbb{Z}_p \quad \forall i \in \{1, .., n\}$. In Shamir's secret sharing scheme, the bit length of the shares is the same as the longest secret $[s]$ to be shared. As mentioned, this scheme was later used by Ben-Or et al. work [4], in conjunction with the arithmetic circuit paradigm to securely solve any functionality. Remember that in this case the objective is for any party to secretly share its input amongst n parties.

Shares Process

The basic idea of Shamir's sharing mechanism is to calculate shares s_i of secret s over some polynomial $f(x)$ of degree of at least $t-1$. It can be seen that in case of a threshold scheme (t, n), such polynomial can only be reconstructed with at least t shares. The following definition provided by Resitad [21] introduces such principle:

Definition 3. *Let s be the secret over $\mathbb{Z}_p$ and coefficients $r_i \quad \forall i \in 1, ..., t-1$ random values on $\mathbb{Z}_p$. Shamir's secret sharing is a $(t, n)-$threshold scheme over $\mathbb{Z}_p$ where each s_i corresponds to $f(i) \quad \forall i \in 1, ..., n$ and $f(x)$ is a polynomial of degree $t-1$ such that:*

$$f(x) = s + r_1 \cdot x + ... + r_{t-1} \cdot x^{t-1} \quad mod \quad p \tag{2.1}$$

Notice that $f(0) = s$. The sharing process consists on each individual party calculating shares and then distributing them amongst the other parties. The degree $t-1$ of the polynomial requires at least t parties to reconstruct the secret. To secure that no minority coalition can learn s, the threshold has to be at least $\frac{n}{2}$. In other words, the adversary would need to corrupt at least $t < \frac{n}{2}$ parties to learn the secret s. We would like to illustrate the sharing procedure with the following example:

Example: A player holds a secret value $s = 6$ and wants to share it amongst three parties such that no minority coalition can reconstruct the secret $(t, n) = (2, 3)$. All parties have previously agreed to work over $\mathbb{Z}_{11}$ and use Shamir's Secret Sharing. It proceeds to randomly select $r_1 = 2$ and then construct the polynomial $f(x) = 6 + 2 \cdot x \quad mod \quad 11$. Then, following definition 3, shares have to be calculated. Table 2.1 shows that process:

$\boldsymbol{P_i}$	$\boldsymbol{f(x) = 6 + 2 \cdot x \quad mod \quad 11}$	$\boldsymbol{s_i}$
P_1	$f(1) = 6 + 2 \cdot 1 \quad mod \quad 11$	$8 \quad mod \quad 11 = 8$
P_2	$f(2) = 6 + 2 \cdot 2 \quad mod \quad 11$	$10 \quad mod \quad 11 = 10$
P_3	$f(3) = 6 + 2 \cdot 3 \quad mod \quad 11$	$12 \quad mod \quad 11 = 1$

Table 2.1: Sharing Example

To reveal the secret, given that all the r_i are unknown to the parties, it is necessary to interpolate the polynomial. First, all the shares related to the value are sent to the party or parties in charge of reconstructing the value. Then, such party could use an interpolation method such as the Lagrangian's method. Polynomial $f(x)$ can be redefined as follows:

$$f(x) = L(x) = \sum_{i=1}^{t} s_i \cdot l_i(x) \quad mod \quad p \tag{2.2}$$

Where polynomial $l_i(x)$ is defined as:

$$l_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{t} \frac{x - x_j}{x_i - x_j} \tag{2.3}$$

Note that it suffices to evaluate $f(0)$ to obtain the secret, hence further simplification on l_i is possible. Other simple interpolation processes like the use of the Vandermonde Matrix can be used as well.

Continuing with our example, to reconstruct the secret $s = 6$ we replace all the s_i in 2.2:

$$\begin{aligned} f(0) &= 8 \cdot l_1(0) + 10 \cdot l_2(0) + 1 \cdot l_3(0) \quad mod \quad 11 \\ &= 8 \cdot \frac{-2}{-1} \cdot \frac{-3}{-2} + 10 \cdot \frac{-1}{1} \cdot \frac{-3}{-1} + 1 \cdot \frac{-1}{2} \cdot \frac{-2}{1} \quad mod \quad 11 \\ &= 8 \cdot 3 - 10 \cdot 3 + 1 \cdot 1 \quad mod \quad 11 \\ &= -5 \quad mod \quad 11 \\ &= 6 \end{aligned}$$

Addition

A natural extension of this process is the addition of secret values e.g. Some subset of parties would like to add the values that they have previously shared. This linear operation can be calculated locally by simply having each player adding locally the corresponding shares. Later, these individual results can be interpolated to reconstruct the result of the addition following equation (2.2).

Consider secrets s_a and s_b, both secretly shared using polynomials $f_a(x)$ and $f_b(x)$. When each player adds locally $s_{i,a} + s_{i,b} \quad \forall i \in \{1, ..., n\}$, in our scenario, this is equivalent to say $f_a + f_b$. Such operation does not increase the degree of the polynomial and the reconstruction of the resulting share can be performed without further information exchange. This can be easily extended to any number of secret values. We illustrate such a process with the following example:

Example: Consider the following three secret values $s_a = 4$, $s_b = 2$, $s_c = 9$ over $\mathbb{Z}_{31}$. We would like to secretly share these values using Shamir's scheme and calculate $s_a + s_b + s_c$, such that no minority coalition could reconstruct the secret. Parties $P_i \quad \forall i \in \{1, ..., n\}$ are in charge of the secret sharing, addition process and reconstruction.

First, we have to randomly construct the polynomials as previously explained, and then calculate the corresponding shares. In this case linear polynomials would suffice. Once this process has been completed they have to be "transmitted" to the corresponding parties. Table 2.2 illustrates such process:

$\boldsymbol{P_i}$	$\boldsymbol{f_a(x) = 4 + 1 \cdot x}$	$\boldsymbol{f_b(x) = 2 + 3 \cdot x}$	$\boldsymbol{f_c(x) = 9 + 0 \cdot x}$
P_1	$f_a(1) = 4 + 1 \cdot 1 \quad mod \quad 31 = 5$	$f_b(1) = 2 + 3 \cdot 1 \quad mod \quad 31 = 5$	$f_c(1) = 9 + 0 \cdot 1 \quad mod \quad 31 = 9$
P_2	$f_a(2) = 4 + 1 \cdot 2 \quad mod \quad 31 = 6$	$f_b(2) = 2 + 3 \cdot 2 \quad mod \quad 31 = 8$	$f_c(1) = 9 + 0 \cdot 1 \quad mod \quad 31 = 9$
P_3	$f_a(3) = 4 + 1 \cdot 3 \quad mod \quad 31 = 7$	$f_b(3) = 2 + 3 \cdot 3 \quad mod \quad 31 = 11$	$f_c(3) = 9 + 0 \cdot 3 \quad mod \quad 31 = 9$

Table 2.2: Addition Example: sharing phase

The randomly selected polynomials are illustrated by figure 2.2.1:

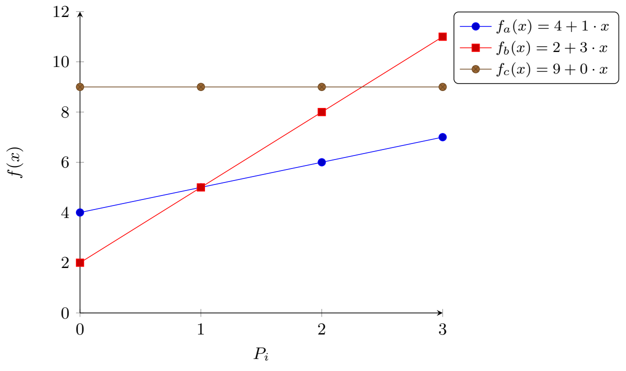


Figure 2.2.1: Example: Secret Polynomials

Each player then proceeds to add the shares it received from this process as follows:

P_i	**Addition**
P_1	$5 + 5 + 9 = 19 \quad mod \quad 31$
P_2	$6 + 8 + 9 = 23 \quad mod \quad 31$
P_3	$7 + 11 + 9 = 27 \quad mod \quad 31$

Table 2.3: Addition Example: parties privately add shares

Notice that the results of the addition can be interpolated to obtain the secret. Figure 2.2.2 shows the newly calculated shares and the result of their interpolation where coordinates $(x, f(x)) = (0, 15)$ are the result of the secure addition: $s_a + s_b + s_c$.

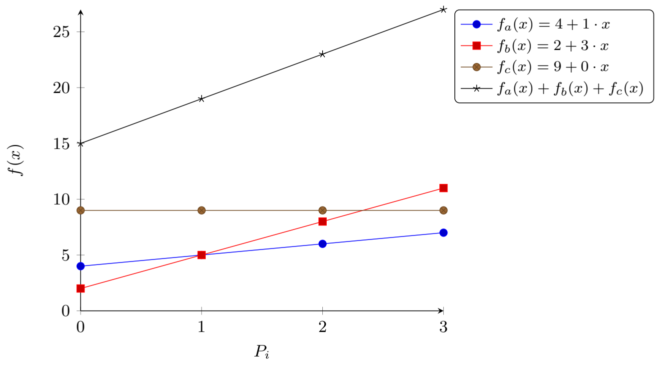


Figure 2.2.2: Example: Secure Addition

Other Linear Operations

Operations that involve secret shares and *public scalars* over $\mathbb{Z}_p$, namely addition and multiplication are linear operations. It suffices for all parties to add up or multiply their secret share by such number. The operation can be seen as a linear displacement of the secret functions that encode the secret.

2.2.3 Other Sharing Mechanisms

Different methodologies for secretly share information can be used with various forms of MPC (others than Shamir's Secret Sharing). We examine two commonly mentioned flavors, although they are not the subject of study in this thesis.

Additive Secret Sharing Scheme. A simple mechanism, that usually illustrates the basic principles of secret sharing: the shares have to be as big as the secret, they have to be generated randomly and parties need all shares to reconstruct the secret. The scheme works well on any finite field i.e. $\mathbb{Z}_q$ providing statistical security, and perfect security against passive adversaries. The intuition of the scheme is as follows: A party that wants to secretly share its input would select $n-1$ random numbers on $\mathbb{Z}_q$ and would compute the last share as $s_n = s - \sum_{i=1}^{n-1} s_i$. In other words, the secret is expressed as the addition of all the individual shares. To reconstruct the secret the only thing that is needed is to add all the shares. It can be observed why this is acceptable in case the adversary does not deviate from the protocol (does not alter the value of the shares), but not in case of malicious adversaries where the effect of changing the content of the shares could be predicted.

Paillier Cryptosystem The scheme was initially introduced by Paillier [22], a later generalization was introduced by Damgård et al. [23]. The Paillier scheme is considered to be partially homomorphic, this is because it can add cyphertexts and multiply by plaintexts. However it can be used for secure multiparty computation [23]. This is especially true for the two party case. Paillier's cryptosystem is commonly used in multiparty literature, and provides easy formulation for addition and multiplication in MPC. Its security is based on the Decisional Composite Residuosity Assumption [22]. Moreover, fully homomorphic solutions can satisfy the basic security definition we have described, and are suitable candidates for MPC, especially for two-party computation. However, for the multiparty case, literature usually prefers sharing mechanisms specially because of their performance and security properties. A detailed treatment of the cryptosystem in MPC can be found in [23].

2.3 Secure Multiparty Computation: Applications and Primitives

The results by Ben-Or et al. (BGW) [4] and Chaum et al. [9] bolstered the MPC perspectives on applications, proving that any functionality can be calculated with perfect security against passive and active adversaries with MPC. More specifically, BGW proposed the use of Shamir's Secret Sharing and a Verifiable Secret Sharing Scheme (VSS) [14] to achieve perfect security. Moreover, they were the first to propose a VSS scheme with perfect security. Their results allow to calculate a function f as an arithmetic circuit where inputs are connected to addition and multiplication (shares with shares or publicly available scalars with shares) gates that are subsequently the inputs of other gates.

2.3.1 Secure Multiplication

The BGW contribution includes a secure multiplication protocol that was later improved by Gennaro et al. [18]. Moreover, multiplication is considered as a basic arithmetic gate, and it is used to build complex applications such as comparisons. The process is somewhat similar to the secret addition protocol, only this time, information exchange between the players is needed. Each time parties exchange information amongst themselves is called a communicational round. We consider operations that do not require communicational rounds to be computationally negligible. This additional processing is needed because of the following: given that we work with secrets shared by polynomials of degree $t-1$, the polynomial resulting from the multiplication will be of degree $2t-2$. Thus, the number of shares that such polynomial needs to be reconstructed is $2 \cdot t - 1$. Following the arithmetic circuit paradigm, the degree of such polynomial will only increase by each multiplication, and with it the number of parties (shares) needed to reconstruct it. The information exchange is used to reduce the polynomial from its $2t-2$ form to the original $t-1$ degree, avoiding this issue. The process can be described as follows:

Prerequisites. Parties (P) secretly share inputs using, for instance, Shamir's secret sharing as showed in previous sections: `Share(`a`)`, `Share(`b`)`. This place the inputs on the following polynomials:

$$f_a = a + r_{1,a}x + ... + r_{t-1,a}x^{t-1} \tag{2.4}$$
$$f_b = b + r_{1,b}x + ... + r_{t-1,b}x^{t-1} \tag{2.5}$$

.

1. Each party (P_i) locally computes the product by multiplying the shares they hold of inputs a and b i.e. $f_a(P_i) \cdot f_b(P_i)$. Note that, if we were to interpolate the resulting values of the multiplications obtained by the parties, the resulting polynomial would be the following:

$$f_{a \cdot b} = a \cdot b + a \cdot r_{1,b} + ... + r_{t-1,a} \cdot r_{t-1,b}x^{2t-2} \tag{2.6}$$

2. Parties (P) secretly share the result of their last computation: `Share(`$f_a(P_i) \cdot f_b(P_i)$`)`.

3. Next, the polynomial $f_{a \cdot b}$ of degree $2t-2$ has to be reduced to a $t-1$ polynomial. Note that, at this point, parties hold shares of the polynomial $f_{a \cdot b}$ (the results of all $f_a(P_i) \cdot f_b(P_i)$ multiplications in shared form), it suffices for each party to apply the first line of the inverted Vandermonde Matrix to the shares of $f_{a \cdot b}$ they hold to obtain such reduction. This requires linear operations only i.e. secure share addition and scalar share multiplication.

4. (Optional) Open the resulting share of the reduction, interpolating the resulting $t-1$ polynomial.

Algorithm: 1: Secure Multiplication

We refer the reader to [24] for a detailed revision of the process.

2.3.2 Secure Comparison

As any other functionality, secure comparisons can be built from an arithmetic circuit based on "costless" addition and multiplication gates. Work by Damgård et al. [25] introduced comparisons mechanisms with perfect security that include bitwise decomposition of shares amongst other useful tools. Since then many methods have been introduced, building upon their results, several include some level of bit decomposition, and the use of other protocols for random bit generation, amongst others. Work by Limpaa and Toft [26] introduces comparison methods with sub-linear complexity on the online phase (where randomization is previously calculated by an offline phase). Cantrina and Hoogh's method [27] offers constant complexity using a novel secure modulo algorithm. these methods can offer a wide range of security properties from statistical to perfect, against active and passive adversaries. Unlike addition and multiplication on MPC, the performance of comparisons in real life scenarios (because of its common dependence on boolean arithmetics and bit representation of some of its temporal results) its tied to the bit-size of the shares, and the parallelization capabilities of the implementation. Given that the equality test ($a == b$) can be reduced to a zero test ($a - b = 0$) many of these protocols show how to solve the zero test instead. The same applies to the inequality test. There exist several comparison protocols, the selection of one by the algorithm designer depends not only on the asymptotic complexity of the algorithm but in properties like parallelization dependence and message size.

Latter chapters of this thesis make use of Catrina and Hoogh's method [27] to provide the comparison functionality. We have included the following algorithm, which is a high level description of the method inner works.

```
Input: The size of the field $p$, a secret shared value $[a]$ and the bit size of the input $m$.
Output: Shared bit $[d]$ (if $a < 0$ then $[d] == [1]$, otherwise $[d] == [0]$).
1. $([r''], [[r'], [r'_{m-1}], ..., [r'_0]) \leftarrow$ PRandM($m$);
2. $c \leftarrow [a] + 2^m[r''] + [r']$;
3. $c \leftarrow$open($c$);
4. $c' \leftarrow c \quad mod \quad 2^m$;
5. $[u] \leftarrow$BitLT($c', ([r'_{m-1}], ..., r'_0)$);
6. $[a'] \leftarrow c' - [r'] + 2^m[u]$;
7. $[d] \leftarrow ([a] - [a'])(2^{-m} \quad mod \quad \mathbb{Z}_p)$;
8. return: $-[d]$;
```

Algorithm: 2: Catrina and Hoogh's Secure Comparison (values between square brackets e.g. $[d]$ are considered secret)

The algorithm itself makes use of known results to generate random bits and random numbers. In our description, function `PRandM` encapsulates these functionality. Additionally, the algorithm needs of any bit level comparison protocol, capable to compare integer numbers against secretly shared numbers in bit form. This functionality is represented in our case by the function `BitLT`, and can be achieved, for instance, by using simple bitwise arithmetic i.e. bitwise subtraction. In other words for for bitwise values a and b of size m, if $c = a - b < 0$, then $1 - c_{m+1} == 1$ and 0 otherwise. Algorithm 2.3.2 shows exactly how to achieve this. Note that this is not the only method suggested in the literature. Details and implementation of these protocols, (random number generations and bit level comparison) can be found in [25, 27].

```
Input: A public integer $a$, a secret shared value $[b]$ and the bit size of the input $m$.

Output: Shared bit $[c]$ (if $a < [b]$ then $[c] == [1]$, otherwise $[c] == [0]$).

1. $(a_m, .., a_1) \leftarrow$ Bits$(a, m)$;
2. for each $i \in 1, ..., m$ do $[b'_i] \leftarrow 1 - [b_i]$;
4. $[c] \leftarrow [1]$;
5. for each $i \in 1, ..., m$ do
6.     $[c] \leftarrow a_i + [c] - 2 \cdot (a_i \cdot [c])$;
7.     $[c] \leftarrow [b'_i] + [c] - 2 \cdot ([b'_i] \cdot [c])$;
8. End For
9. return: $1 - [c]$;
```

Algorithm: 3: Complement 2 `BitLT` Algorithm (values between square brackets e.g. $[b]$ are considered secret)

2.3.3 Software Frameworks

Some frameworks have been designed and implemented to provide basic MPC functionality to algorithm designers such that they can build complex applications. The approach and scope varies with each different software solution. Different flavors can be found for security level, accessibility, software composition, usability, scalability and performance. For multiparty computation, the BGW [4] results opened the door to an array of functionality waiting to be implemented and these frameworks provide the basic tangible building blocks for that endeavor. We provide a more detailed comparison with our own implementation of MPC protocols further in Chapter 6. However, we now present a short overview of their basic characteristics.

Fairplay - FairplayMP First introduced in 2004, Fairplay [28] and its later multiparty version FairplayMP [29] provides an interesting approach to the problem despite being the first to provide, to the best of our knowledge, open access to a general purpose multiparty framework. Initially the functionality was designed to solve the two party case, making use of garbled circuits. Later multiparty extensions kept working with garbled circuits instead of the use of sharing mechanisms, which had great impact on its general performance. On usability, Fairplay provides a meta-language on which the algorithm designer writes secure protocols.

Sharemind This product from the University of Tartu (Estonia) and AS Cybernetica, is an optimized 3-party computation tool that can be used online. Their development has been centered on optimizations for the 3-party case, and some sections of its source code are programmed in Assembly [19]. However, the implementation of such optimizations for the multiparty case are not obvious. Moreover, student access is limited and commercial applications need licenses. The source code for the Sharemind Server is not available. These factors limit greatly scalability and malleability. Sharemind itself is an application server, which means that applications run over Sharemind, which somewhat limits its integration with standalone applications. A detailed treatment on Sharemind functionality can be found in [30].

VIFF The Virtual Ideal Functionality Framework by Martin Geisler [2] is a compact set of MPC tools programmed on Python, using the Twistter Framework to manage communications and GMPY (GNU Multiple Precision arithmetic library Python extension) for the precision arithmetic. VIFF uses an asynchronous approach that takes into account possible delays on share deliveries such that operations can be processed upon arrival of the shares they depend on. This results in a extensive use of parallelization in the form of deferred method callbacks. Basically, each new operation is assigned a new deferred that can be considered a thread by itself. Note that this is done in a platform with limited thread support such as Python. Although logically this would mean the existence of as many threads as active operations, this is not the same as saying they might be executed by different physical CPUs. Under the deferred approach, threads in Python do not make use of other CPUs. As a consequence, the thread scheduling scheme becomes a burden for the process itself in terms of performance. Nevertheless, even though VIFF is not the fastest of all the frameworks hereby explained, it grants total access to the algorithm designer to edit the source code, and provides a variety of security configurations that are useful in various settings e.g. [31]. The readability with Python as interface language is also an important point to consider. Part II of this thesis reports on experimentation with VIFF. However, our results point out the necessity of an accessible compact toolkit without the issues posed by VIFF design poses, not only in terms of performance, but also memory management (relatively fast monotonic increase of memory consumption prompted by its asynchronous model).

Other MPC Solutions There are several other constructions and MPC compilers that are dedicated to provide access to MPC primitives. Some of these center their attention on bringing access to Oblivious Random Access Memory data-structures or fast parallelization such as [19, 32]. Recent work has been done on specifications for MPC frameworks such as SPDZ [33] but currently, there is no implementation that is publicly available. Given that aspects such as performance analysis, memory consumption and functionality are tied to the implementation itself rather than to the specification, it is difficult for us to comment on such aspects.

2.4 Combinatorial Optimization

Combinatorial optimization studies the problem of efficiently arranging discrete sets of objects through the use of various algorithms. Briefly speaking an algorithm can be defined as follows:

Definition 4. *It is any well defined computational procedure or sequence of steps, that takes a value, or set of values as an input, and produces some value, or set of values as an output.*

A comparative characteristic of an algorithm is its complexity time. Korte and Vygen [34] define it as follows:

Definition 5. *Let A be an algorithm which accepts inputs from a setX, and let $f : X \longrightarrow \mathbb{R}_+$. If there exists a constant $\alpha > 0$ such that A terminates its computation after at most $\alpha \cdot f(x)$ elementary steps for each input $x \in X$, then we say that A runs in $\mathcal{O}(f)$. In other words the time complexity of A is $\mathcal{O}(f)$.*

It is in the interest of the field to find feasible bounds to complexity times e.g. polynomial times. This can be defined as follows:

Definition 6. *An algorithm with rational input is said to run in polynomial time if there is an integer k such that it runs in $\mathcal{O}(n^k)$ time, where n is the input size, and all intermediate computations can be stored with $\mathcal{O}(n^k)$ bits.*

In this section we include some basic useful concepts for combinatorial optimization, as well as succinctly review some classical problems. We refer the reader to [35, 34, 36, 37, 38] for a detailed treatment.

2.4.1 Graph Theory Preliminaries

We briefly describe some general notions on graph theory taken from [38, 36] that are often used during the following chapters.
A **Directed Graph** $G = (V, E)$ consists of a set of vertices V and a set of edges E whose elements are an ordered pair of distinct vertices i.e. $E \subseteq V \times V$.
A **Directed Network** is a directed graph, where the vertices and/or edges have a numerical value associated to them e.g. costs, capacities, supply, demands or any combination of these values.
An **Undirected Graph** $G = (V, E)$, consist of a set of vertices V and a set of edges E whose elements are unordered pairs of distinct edges. In other words we can refer to edge (v, w) as (w, v) indistinctly.
Furthermore, a **Undirected Network** is a undirected graph with values associated to its elements.
A vertex v has associated 3 concepts of **degree**:

- The **indegree** is the number of incoming edges towards the vertex.
- The **outdegree** is the number of outgoing edges from the vertex.
- The **degree** is the addition of the *indegree* and the *outdegree.*

A **Network Flow** is an assignment of flows: $E \longrightarrow \mathbb{R}_+$ such that flow conservation (difference between incoming and outgoing flow towards vertex v is 0) at each vertex is satisfied. It is called **capacitated** if the numerical value associated to each edge i.e. capacity, is respected.
An **Adjacency List** $E(v)$ of vertex v is the set of edges emanating from v, that is $E(v) = \{(v, w) \in E : w \in V\}$. From the definition it follows that $|E(v)|$ is equal to the *outdegree* of vertex v.
An **Adjacency Matrix** A of size $|V|x|V|$, stores the graph $G = (V, E)$ where each existing entry a_{ij} is equal to 0 in case $(i, j) \notin E$ and 1 otherwise. It is **weighted** if the matrix stores the edge associated value.
Strong Connectivity is a property of a directed graph $G = (V, E)$ where there is at least one path from every vertex to any other vertex. Moreover, a **Simple Graph** is a graph having no loops.
The **Residual Capacity** of a network is the difference between the capacity of an edge $(v, w) \in E$ and the flow circulating over it.
A **Residual Network** is the associated network defined by the edges with positive residual capacities.
A **Walk** is a list $v_1, (v_1, v_2), v_2, (v_2, v_3), v_3, ..., v_k$ of vertices and edges such that , for $1 \leq i \leq k$, the edge $(v_{i-1}), v_i$ has endpoints v_{i-1} and v_i.
Moreover, a **Path** can be defined as a simple graph whose vertices can be ordered such that 2 vertices are adjacent if and only if they are consecutive. Thus, it can be said that all paths are walks but not all walks are paths.
A **Cycle** is a graph with an equal number of vertices and edges whose vertices

can be placed around a circle so that 2 vertices are adjacent if and only if they appear consecutively along the circle.

2.4.2 Some Classical Network Flow Problems

We briefly describe the problems studied by this thesis, namely the Shortest Path Problem and Max-Flow and Minimum Cost Flow Problem. In this section, we present a quick overview of the problems based on the definitions provided by [38]. Each of these problems are reviewed more in detail in the following chapters:

Shortest Path Problem

The shortest path problem is often used as sub-routine, a sub-protocol, part of algorithms to solve more complex problems. In this case we want to find the shortest path in a network e.g. (minimum cost or length) from a source vertex s towards a sink vertex t using for that purpose an associated numerical value of the edges. Simple applications of the problem include the determination of the path of minimal length between any pair of vertices of the network or the shortest time to traverse the network. Other more complex applications can be found in communication systems e.g. message routing, cash flow management, project scheduling, supply chain and many others. A more detailed treatment can be found in Chapter 3.

Maximum Flow Problem

Is the problem of finding a capacitated directed flow that maximizes the amount of flow from a source vertex s towards sink vertex t. This problem can be seen as a complementary model for the shortest path problem. Indeed, the shortest path models the case where the flow is constrained by some kind of associated cost, but is not restricted by any capacity. In contrast the maximum flow problem studies the case where the flow is not constrained by costs, but the flow is restricted by capacity bounds. Applications can be found in petroleum or gas pipeline networks, routing messages in communication networks, road networks, electricity networks, amongst others. We refer the reader to Chapter 4 for a detailed treatment.

Minimum Cost Flow Problem: In this case the network does not only have a capacity associated to the edges but also a cost (a price for allocating a unit of flow). Moreover, the vertices in V can allocate or consume units of such flow. When a vertex injects flow onto the network is called a source, and sink vertices if it does the opposite. They can be seen as supply and demand entities. Vertices with 0 input on the network are called neutral vertices. The problem is that of finding the minimum cost of moving flow under this new set of constraints. In this way, the problem includes aspects from the shortest path and maximum flow problems. It restricts the flow to a certain capacity and minimizes the associated costs of the flows. See Chapter 4.

2.4.3 Matroids

Since its initial introduction and development by Hassler Whitney [39, 40], the use of matroid theory has expanded to several different fields. Applications can be found in graph theory, lattice theory and the topic at hand. Moreover, many combinatorial problems can be described as matroids. This is an important result given that greedy algorithms solve optimization problems over matroids.

To define a matroid we first need to illustrate what is an independence system. Korte and Vygen [34] provide the following definitions:

Definition 7. *Consider a set system $(E, \mathcal{F})$ i.e. a finite set E and some $\mathcal{F} \subseteq 2^E$ is an **independence system** if:*

$$\emptyset \in \mathbb{F}; \tag{2.7}$$

$$if \quad X \subseteq Y \in \mathbb{F} \quad then \quad X \in \mathcal{F}. \tag{2.8}$$

*The elements of $\mathbb{F}$ are called **independent**, the elements of $2^E \setminus \mathbb{F}$ **dependent**.*

Then, a matroid then can be defined as follows:

Definition 8. *An independence system is a **matroid** if $X, Y \in \mathcal{F}$ and $|X| > |Y|$, then there is an $x \in X \setminus Y$ with $Y \bigcup \{x\} \in \mathcal{F}$.*

The same authors provide some examples of independence systems that are matroids:

- Let us consider matrix A where E is a set of columns of A over some field and $F := \{F \subseteq E :\}$ such that all columns in F are linearly independent over the same field.

- Consider some undirected graph G where E is a set of edges of G, then $F := \{F \subseteq E : (V(G), F) \quad is \quad a \quad forest\}$

2.4.4 Submodular Functions

Also known as submodular set functions, we briefly make use of them in Chapter 7. Alexander Schrijver [41] provides the following definition, note that this is not the only valid description of a submodular function, rather is the one that suffices the purposes of this thesis:

Definition 9. *Let f be a set function on a set S, that is a function defined on the collection $P(S)$ of all subsets of S. The function f is called submodular if:*
$f(T) + f(U) \quad \geq \quad f(T \cap U) + f(T \cup U)$ *for all subsets* T, U *of* S.

Complete lists, definitions and proofs of both, matroids and submodular functions can be found in [35, 34, 41].

Part II

Secure Classical Network Flow Problems

Chapter 3

Securely Solving The Shortest Path Problem

3.1 Introduction

The computation of the single source shortest path is a well studied problem and a frequently used subroutine in various applications and different problems in combinatorial optimization. It is the problem of finding the shortest given path from one vertex to another in a network and its use is not limited to theoretical aspects but to real life applications. Interest areas of study include communications and networking, as well as transportation routes, amongst many others. Many real life problems use day to day applications that may need, at some point, to solve a shortest path problem. In many cases, data related to the computation, like the topology of the network or the associated length, is distributed amongst competing and distrustful parties. We can imagine for instance, a collaboration between distribution networks from competing companies, each one holding information about the warehouses they cover and rout costs. In this setting none of the parties is willing to disclose its network composition nor its costs, but still want to compute the cheapest way to go between 2 different warehouses. In such environments, adversarial parties would gain competitive advantages from any information that is disclosed. In this cases mechanisms have to be put in place to provide factors like correctness and fairness into the calculations.

Secure Multiparty Computation (MPC), introduces tools to facilitate interaction between distrustful agents providing security. Although initially regarded as a theoretical result, since Yao's introduction of the millionaire problem [1], the field has evolved to provide answers to complex problems and real life applications with the years [31, 33]. Nowadays it is possible to find MPC protocols that are secure against various different adversarial models using secret sharing techniques or homomorphic encryption, in two-party or multiparty settings.

We use MPC to solve the shortest path problem providing security in a distributed environment. Furthermore, our study shows the differences between traditional and secure implementations of some algorithms used to solve the problem, to later test its implementation. We also report on various trade-offs between performance and security. Additionally, we provide protocols that can be used as building blocks to solve more complex problems. Applications of our work can be found in competitive environments like: communication technologies, and satellite routing; retailer/supplier selection in multi-level supply chains that want to share routes without disclosing sensible information; amongst others.

The contents introduced on this Chapter come from the following papers:

2013 Securely Solving Simple Combinatorial Graph Problems (Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, Mathieu Van Vyve), *In Financial Cryptography*, pp. 239-257, 2013.

2014 Securely Solving Classical Network Flow Problems (Extended Treatment) (Abdelrahaman Aly, Mathieu Van Vyve), *In CORE Discussion series Vol:2014/57*, 2014.

2014 Securely Solving Classical Network Flow Problems (Abdelrahaman Aly, Mathieu Van Vyve), LNCS 8949 ICISC, 2014.

3.1.1 Our Contribution

We introduce a series of data-oblivious protocols designed to work in conjunction with MPC protocols such that they are able to solve the single source shortest path problem correctly and in a secure manner. Moreover, we selected and adapted the Bellman-Ford and Dijkstra's algorithms and introduce correctness and security analyses. Further on, we provide complexity bounds considering black box operations and introduce trade-offs to augment efficiency. Additionally, we report on prototyping with the Virtual Ideal Functionality Framework (VIFF) [2] and the results of computational experimentation. We also introduce a novel exchange technique to hide the vertex selected at each iteration of Dijkstra's algorithm, avoiding the overhead caused by the use of special data structures e.g. oblivious data structures. This is particularly relevant on dense graphs. We refer the reader to Section 3.1.2 for further analysis. The use of secure primitives, in our case, limits the use of some typical data structures such as Fibonacci heaps [42], which are not trivial to replicate. This is specially relevant in the case of Dijkstra.

On the input data, we consider the following information related to the graph to be secret: the graph topology and edges weights/length. We assume that the number of vertices or at least an upper bound to be public. We also consider the case when the topology of the graph is publicly known, this is specially relevant for our Bellman-Ford protocol where its complexity bound can be improved. The distribution of secret information is settled beforehand, according to the parties' preferences. The result of the computation is the length of the path and/or the path composition at the discretion of the parties. The players have the faculty to determine whether or not the path is disclosed as well as part of the solution. Moreover, all parties involved in the computation learn the result at the same time (fairness).

Setting examples include several scenarios, for instance: given a graph, each party involved in the computation is represented by vertices and owns all edges with an associated length (different than zero) exiting such vertex where this information is consider to be private; given a a graph, each edge with an associated length is owned by some party involved in the computation, and the calculation needs to be performed without revealing these lengths. As mentioned, there is no restriction on how information is distributed amongst the parties. This is true for all the algorithms presented in this Part, hence we will not come back to it.

We make use of a slightly extended version of the arithmetic black box introduced by Damgård and Nielsen [23], to abstract the algorithm designer from the primitive selection and focus on the protocol itself. It also makes our algorithms dependent on the security properties of the underlying primitives that implement the black box functionality. This means in practice that our algorithms will be as secure as the primitives they rely on e.g. information theoretic security with Shamir's secret Sharing [13], Ben-Or et al. [4] and [25]. Note that an homomorphic encryption method such as [22, 43] would only bring computational security.

Complexity on MPC applications is typically measured by communicational rounds (a coordinated exchange of messages between parties). Additions can be performed without any message exchange and are assumed to be costless, multiplications and sharing operations require a single round [4]. Although results on comparisons with constant round time have been introduced e.g. [27], it has to be noticed that in practice a comparison has higher constant time than a multiplication. As an example, a comparison in VIFF is ≈ 160 times slower than a multiplication. Our secure protocols reduces the use of comparisons such that better practical results can be achieved.

Table 3.1 showcases the complexity bounds of the main results of this work:

	Advance Impl.	Simple Impl.	Complete Graphs	Privacy Preserving	Secure Comparisons
Bellman-Ford	$\|E\| \cdot \|V\|$	$\|E\| \cdot \|V\|$	$\|V\|^3$	$\|V\|^3$	$\|V\|^3$
Dijkstra	$\|E\| + \|V\| \cdot log(\|V\|)$	$\|V\|^2$	$\|V\|^2$	$\|V\|^3$	$\|V\|^2$

Table 3.1: Worst-case bounds of Original and Privacy-Preserving algorithmic versions

Finally, our experimentation assesses the influence and effects of the secure primitives on performance, as well as the adaptations of our secure protocols. Moreover, VIFF was the tool selected for experimentation given its status of open source tool, and its easy readability in Python. VIFF provides a range of functionalities with security against passive adversaries, and methods like the Orlandi protocol [44] for the active setting. Our prototyping includes vanilla implementations of Bellman-Ford and Dijkstra algorithms in Python, similar to what a trusted third party would use to compute the shortest path, for benchmarking.

3.1.2 Related Works

Extensive effort has been made on comparisons and equality tests under MPC e.g. [45, 46, 47, 27]. The most efficient, to the best of our knowledge, are the comparison methods proposed by Limpa and Toft [26], with a sub-linear complexity bound on the online phase. Nonetheless, work has also been proposed for other kinds of applications using various MPC techniques.
Branching Programs. Branching mechanisms study the case where the algorithm flow is determined by certain parametrization and the nature and value of the input. Several authors have considered a two-party setting where the user does not want to reveal its input and the server wants to privately compute the branching e.g. [48, 49, 50]. While security is maintained, leakage of information e.g. branching length, can compromise the security of the algorithms.
Graph Theory Problems. Different alternatives to solve some graph theory problems have been studied by Aly et al. [51], namely the shortest path and maximum flow problems. They provide bounds on the Dijkstra algorithm using a different strategy than the results introduced by this thesis. Moreover, we slightly improve its worst-case bound. Indeed, [51] uses a searching array technique, similar to the indexation technique proposed by Launchbury et al. [52], to keep track of a secret shared index. Our proposed Dijkstra implementation does not require the use of this technique, eliminating its overhead. For the maximum flow problem: Edmonds-Karp and push-relabel bounds are also provided. Their implementation is secure in the *information-theoretic* model relying on the same arithmetic black-box. Moreover, Brickell and Shmatikov [53] have addressed the shortest path problem on the two-party case, limited to the honest but curious model. Their protocol reveals, at each iteration, the

newly selected edge of the shortest path. Our approach attacks the problem in a different fashion by eliminating this requirement. Moreover, our algorithms assume the capabilities the arithmetic black-box used, and are not limited to the two-party case. Furthermore, our algorithms offer data-oblivious alternatives that can be used not only in the context of MPC but asynchronous applications that need to schedule operations before data is made available. Finally, Blanton et al. [54] have proposed data-oblivious alternative for the Breath-First-Search (BFS) algorithm, which is later used to solve the special case of the shortest path problem where distance is measured in the number of edges of the path. We consider instead the use of a general version of Dijkstra's algorithm to solve the conventional shortest path problem where distance is measured by the added length of the path. We achieve this by conveniently exchanging the information contained in the data containers. Additionally, they use their BFS algorithm to provide bounds for the Max-Flow problem, where weighted edges with a positive residual capacity are mapped as 1 and its counterparts as 0, extending the definition of an existing edge.

Oblivious data structures over ORAM (Oblivious Random Access Memory). Data structures are used to speed-up Dijkstra's algorithm and achieve its optimal complexity. ORAM has been viewed as a suitable mechanism to build oblivious distributed data structures with some overhead and specific configuration e.g. The work of Wang et al. [55] designed to work on a client(s)-server configuration. Moreover, secure two-party computation protocols have been developed to take advantage of the recent advances on ORAM e.g. [56, 57]. The two-party tool and algorithmic implementations of Liu et al. [32] securely address the shortest path and other combinatorial problems by using these kinds of data structures. More recently, Keller and Scholl [58] show how to use oblivious data structures on a multiparty setting, where none of the players have to fulfill the role of the server. Furthermore, they use their data structures to implement Dijkstra's algorithm. Their experimentation shows how some MPC protocols in the absence of ORAM can perform better for certain kinds of graphs than their proposed counterparts i.e samples of smaller-to-medium sizes and complete graphs of any size. This is easily explained by the fact that the overhead coming from the ORAM exceeds the asymptotic advantage of the algorithms. Indeed, we address the problem differently, our Dijkstra algorithm is designed to work on plain vectors and matrices and does not require any secure data structure construction, slightly improving the bounds proposed by Aly et al. [51], who's work is later used in Keller and Scholl's analysis. This allows us to avoid any overhead caused by the use of ORAM or static secret sharing arrays. We refer to [58] for details.
Other Alternatives. The shortest path problem can be formulated as a linear program and solved using the simplex algorithm. MPC implementations of Simplex have been introduced by the following authors [59, 60, 61].

3.1.3 Overview

Section 3.2 presents some of the cryptographic principles as well as basic subroutines constantly used across this Chapter. We introduce the shortest path problem in Section 3.3. In Sections 3.4 and 3.5 we illustrate our privacy preserving Bellman-Ford and Dijkstra's protocols. Finally, in Section 3.6 we report on the results of our experimentation.

3.2 Preliminaries

3.2.1 Security

Succinctly speaking, we use the formalized notion of security from definition 1. Moreover, all our protocols are designed to work under the *information-theoretic* model in the presence of passive or active adversaries over an arithmetic black box $\mathcal{F}_{ABB}$ (see section 3.2.3). This implies that as long as the corrupted parties do not have access to other private data but their own, unbounded computing power would not allow them to obtain any additional information. In practice, this means that they will be as secure as the underlying MPC functionality and crypto-primitives they rely on.

3.2.2 Cryptographic Assumptions

Modulo arithmetic for some M or ring arithmetic allows to simulate secure integer arithmetic. Indeed, several multiparty computation protocols have been designed to work on modulo arithmetic for an appropriate M e.g. a sufficiently big prime number (transforming the ring in a finite field over some M, $\mathbb{Z}_M$), such that no overflow occurs. This is true for secret sharing schemes the likes of Shamir [13] sharing or additive sharing, as well for homomorphic threshold public key encryption.

Primitives like addition between secret shared inputs, as well as additions and multiplications of shares by public values, are linear operations and do not require any information transmission between players. When data is communicated between players it is called a communication round or just round. For complexity analysis purposes, we require constant-round protocols for multiplications. Sharing and reconstruction are done in one round as well. There are still local operations involved with all the primitives, but the performance cost is mainly determined by the communication processes, as explained by Maurer [62]. We assume that the execution flavor i.e. sequential and parallel, does not compromise the security of the private data.

3.2.3 The Arithmetic Black-Box

Multiparty computation, by secretly sharing inputs, can be performed in different ways, (using homomorphic encryption techniques e.g. [22, 43, 17], or secret sharing formulations, like the one introduced by Shamir [13]). The concept of the arithmetic black-box $\mathcal{F}_{ABB}$ [15] embeds this behavior and makes the process transparent for the algorithm designer. It creates an abstraction layer between the protocol construction and functionality specificities, and at the same time it provides the security guarantees desired. Following [15, 47, 51], we assume the following functionalities are available: storage and retrieval of ring $\mathbb{Z}_M$ elements, additions, multiplications, equality and inequality tests.

Several MPC schemes for all of these operations have been proposed in the last 30 years. Multiplications have been addressed by [3, 9, 63] amongst others. As mentioned, comparisons with statistical security and sub-linear complexity on the on-line phase have been explored in [47, 26], and perfect security in constant rounds by [25, 46, 27, 26].

3.2.4 Notation

We use the traditional square brackets e.g. $[x]$, to denote secretly shared or encrypted values contained in the $\mathcal{F}_{ABB}$. This notation is commonly used by secure applications [25, 27, 51]. The value $[\top]$ is used to designate a sufficiently large constant smaller than M (the size of the field) but much bigger than the values of the inputs. Its value depends on the application on which the protocol is going to be used as well. It has to be noted that some comparison protocols require a security parameter on the size of M that has to be taken into account when defining its size. Moreover, secure operations are described using the infix operation e.g. $[z] \leftarrow [x] + [y]$ for secure addition into the $\mathcal{F}_{ABB}$ and $[z] \leftarrow [x] \cdot [y]$ for secure multiplication. The secret result of any secure operation primitive is stored in $[z]$ and onto the $\mathcal{F}_{ABB}$. This notation covers all operations performed with secret values, including those performed with public scalars and secret values.

We use only weighted adjacency matrices with our Dijkstra protocol. That is because adjacency lists, where each vertex has a list of its neighbors and the corresponding length of the connecting edges, is another traditional way to represent graphs. Although It would allow us to save memory but it would also leak the degree of each vertex i.e. the size of the corresponding list. In an iterative process where the order of the vertices is shuffled for security reasons, it serves as an identifiable label compromising the security of the protocol.

We define two subroutines that are repeatedly used, to improve readability and simplify expressions. They only use the primitives available in the $\mathcal{F}_{ABB}$

and work under the same general assumptions.

conditional assignment : Overloaded functionality of the assignment operator represented by $[z] \leftarrow_{[c]} [x] : [y]$. Much like in [32, 60, 51], the behavior of the assignment is tied to a secretly shared binary condition $[c]$. If $[c]$ is one, $[x]$ is assigned to $[z]$ and $[y]$ otherwise. Our version employs a single multiplication for that purpose. Protocol 1 shows the implementation of the conditional assignment, using only supported $\mathcal{F}_{ABB}$ operations. The subroutine can be extended for other mathematical structures i.e. vectors, matrices.

Protocol 1: Implementation of secure conditional assignment.

```
Input: Binary expression $[c]$, assignment values $[x]$ and $[y]$.
Output: Value $[z]$ containing $[x]$ if 1 and $[y]$ if 0, evaluated according
        the binary expression $[c]$
1 $[a] \leftarrow [c] \cdot ([x] - [y])$;
2 $[z] \leftarrow [y] + [a]$;
```

conditional exchange : We define the operator $condexch([c], i, j, [v])$. It exchanges the values held in position i and j of secretly shared vector $[v]$ if a secretly shared binary condition $[c]$ is 1 and leaves the vector unchanged otherwise. We describe the algorithm as protocol 2. We also extend this operator to work with matrices. In that case both i^{th} and j^{th} rows and columns are swapped.

Protocol 2: condexch: Exchanges the values of 2 different vector positions

```
Input: Binary condition $[c]$. Any vector $[v]$. Indexes $i, j$
Output: The vector $[v]$ with values $i$,$j$ swapped if $[c]$ true.
1 $[a] \leftarrow [c] \cdot ([v]_j - [v]_i)$;
2 $[v]_i \leftarrow [v]_i + a$;
3 $[v]_j \leftarrow [v]_j - a$;
```

3.3 Shortest Path Problem

Consider the connected graph $G = (V, E)$ where V is the set of vertices and E the set of edges. Furthermore, $c_{v,w}$ is the associated length of edge $(v, w) \quad \forall (v, w) \in E$. Let s be the designated source vertex. The single source shortest path problem (SSSP) can be defined as the problem of finding the directed path of shortest length form s towards $v \quad \forall v \in V - \{s\}$.

In case a negative cycle is found in G, the solution becomes unbounded given that an infinite amount of flow can be allocated in the cycle. Indeed, the problem of finding the shortest path in a graph that contains negative cycles

is an NP-complete problem.
Note that the problem of finding the shortest path is not limited to the formulation hereby presented. This is in fact a generalization of the common single source shortest path problem. There are several polynomial time algorithms for this problem, amongst them, the two adaptations we report on, Bellman-Ford and Dijkstra's algorithms. With Dijkstra it is necessary for the graph G to have non-negative edges, meanwhile for Bellman-Ford it suffices that more classical assumption of non-negative cycles be present.
For a detailed treatment of the problem we refer the reader to [38, 37].

3.4 Bellman-Ford's Algorithm

The algorithm of Bellman-Ford is particularly simple, making it a natural target for building a secure version. This algorithm proceeds by repeatedly scanning all edges, in search of adding edges that decrease the ongoing distance from the source to the various vertices. If a pass over the edges did not improve the current solution, or if the edges were scanned $|V|$ times, the algorithm halts. An interesting feature of this algorithm is that its flow of operations only depends on the structure of the graph but not on the length of the edges. Its drawback is its time-complexity: its classical implementation runs in $\mathcal{O}(|V||E|)$ time.

The original algorithm is presented as Algorithm 3.

Algorithm 3: Classical Bellman-Ford algorithm

```
Input: A graph $G = (V, E)$ where $V$ is the list of vertices and $E$ the list
       of edges, a list of weights-lengths $w_e$ for each $e \in E$, and a
       source vertex $s \in V$.
Output: The list of immediate predecessors $p$ and/or total distances $d$.
1  $d \leftarrow \top$, $d_s \leftarrow 0$
2  for $i \leftarrow 1$ to $|V|$ do
3      for $e \leftarrow 1$ to $|E|$ do
4          if $d_{t(e)} + w_e < d_{h(e)}$ then
5              $d_{h(e)} \leftarrow d_{t(e)} + w_e$
6              $p_{h(e)} \leftarrow t(e)$
7          end
8      end
9      If no update during the last pass, terminate early , solution is
       optimal.
10 end
11 If there was an update during the very last pass, solution is unbounded
   ($\exists$ negative cycle).
```

Protocol 4 presents our secure shortest path protocol based on Bellman-Ford. Note that $h(e)$ and $t(e)$ represent the head and tail vertex of an edge

e respectively. Finally, note that in case the source vertex is also secretly shared, parties can make use of any of the secret index assignment techniques documented by the literature e.g. [47, 52, 58]. The protocol differs from the original algorithm only in a limited number of aspects: *a*) the branching corresponding to the discovery of a shorter path is handled on Lines 8–10 through arithmetic as in Protocol 2, *b*) the early termination condition of the Bellman-Ford algorithm, which is triggered if the inner loop happens to have no effect during one pass, is removed as it could leak information.

Protocol 4: protocol based on Bellman-Ford's algorithm

```
Input: A graph $G = (V, E)$ where $V$ is the list of vertices and $E$ the list
       of edges, a set of shared weights $[w]_e$ for each $e \in E$, and a share
       of the source vertex $[s] \in V$.
Output: The list of immediate predecessors $[p]$ and/or total distances
        $[d]$.
1  for $i \leftarrow 1$ to $|V|$ do
2      $p_i \leftarrow [0]$; $d_i \leftarrow [\top]$;
3  end
4  $[d]_{[s]} = 0$
5  for $i \leftarrow 1$ to $|V|$ do
6      for $e \leftarrow 1$ to $|E|$ do
7          $[y] \leftarrow [d]_{t(e)} - [d]_{h(e)} + [w]_e$;
8          $[x] \leftarrow [y] < 0$;
9          $[d]_{h(e)} \leftarrow [d]_{h(e)} + [x] \cdot [y]$;
10         $[p]_{h(e)} \leftarrow_{[x]} t(e) : [p]_{h(e)}$;
11     end
12 end
13 If there was an update during the very last pass, solution is unbounded
   ($\exists$ negative cycle). Open required output.
```

The structure of this algorithm makes it easy to implement with either of the two graph representations discussed above (list or matrix), making it possible to fully exploit the sparsity of graphs when it is public (we use the matrix representation if it has to be kept secret).

It can be seen that our implementation requires $|V||E|$ secure comparisons, dominating the time required to perform $2|V||E|$ secure multiplications and $5|V||E|$ additions. These complexities grow to $\mathcal{O}(|V|^3)$ when the graph structure is secret, as the graph is then treated as complete (i.e., augmented with edges of infinite weight). Very interestingly, this algorithm is the only one amongst those we analyzed in this dissertation in which our privacy-preserving algorithm does not cause any asymptotic overhead on its complexity bounds (when the structure is public).

Security. No leakage originates from the protocol process. Moreover the $\mathcal{F}_{ABB}$ provides secure functionality for the $+$, $\cdot$ and $! =$ operations. In other words, security follows from the call of such operators in a given order, oblivious to the data, where their quantity depends exclusively from public available information. In this case the size of sets $|V|$ and $|E|$ if the structure of the graph is publicly known or only $|V|$ in case the structure is secret. This can be seen as an execution list of operations predefined in advance and agreed by all parties. This is also true for all algorithms introduced in this dissertation.

3.5 Dijkstra's Algorithm

The algorithm provides a greedy way to find the shortest path from a source vertex s in a directed connected graph with non-negative lengths. Basically, it selects the vertex with the smallest accumulated distance and then propagates the path forward until all vertices have been explored. This ensures to get the shortest path from a source vertex to all other vertices in the graph. To find the shortest path to a single vertex is also possible. Our secure implementation can be adapted to detect at each iteration whether the target vertex has been reached to stop the algorithm.
Adapting Dijkstra to MPC. The input data in our case is a weighted adjacency matrix $[U]$ where non existing edges are represented by $[\top]$. Dijkstra's algorithm treats the vertices of the graph in an order that depends on the length of the edges. The main challenge is to hide this order. Earlier work [51] has proposed to hide the position of the vertex accessed by using a secretly shared unary vector $[0, 0, ..., 0, 1, 0, ..., 0]$. We introduce a different technique. The basic idea is to exploit the symmetry in the data structure. More precisely, the numbering of the vertices or equivalently, the position of a vertex in the data structure is indifferent for the algorithm. We exploit this by positioning at iteration i, the vertex with the lowest distance in position i. That way we align the vertex exploration of our protocol with the secret data stored in all the structures. This enables us to gain in the number of operations performed because we can avoid considering edges pointing to vertices already explored. The algorithm is detailed as Protocol 5.
Correctness Because the algorithm constantly reshuffles the positions of the vertices in all matrices and vectors used, we need to (secretly) track the position of the vertices. This is the role of the vector π. Throughout the algorithm π_j holds the node number that is currently in position j.

The loop on lines 5-8 determines the untreated vertex with current minimum distance. This vertex is brought to position i in all data structures. Loop on lines 9-14 scans all edges leaving node in position i to all other untreated vertices (positioned after i). If the edge improves the current best path (Line 11), the current best distances and predecessors are updated (Lines 12-13). The

Protocol 5: Shortest Path Protocol based on Dijkstra's algorithm

Input: A matrix of shared weights $[U]_{i,j}$ for $i, j \in \{1, ..., |V|\}$ and a unit vector $[S]$ encoding the source vertex.

Output: The vector of predecessors $[P]$ and/or the vector of distances $[d]_i$.

```
1  for $i \leftarrow 1$ to $|V|$ do
2      $[\pi]_i \leftarrow i$; $[d]_i \leftarrow_{[S_i]} [0] : [\top]$; $[P]_i \leftarrow i[S]_i$;
3  end
4  for $i \leftarrow 1$ to $|V|$ do
5      for $j \leftarrow |V|$ to $i + 1$ do
6          $[c] \leftarrow [d]_j < [d]_{j-1}$;
7          $([\pi], [P], [\mathbf{d}], [U]) \leftarrow condexch([c], j, j - 1, [\pi], [P], [d], [U])$;
8      end
9      for $j \leftarrow i + 1$ to $|V|$ do
10         $[a] \leftarrow [d]_i + [U]_{i,j}$;
11         $[c] \leftarrow [a] < [d]_j$;
12         $[d]_j \leftarrow_{[c]} [a] : [d]_j$;
13         $[P]_j \leftarrow_{[c]} [\pi]_i : [P]_j$;
14     end
15 end
```

predecessor of node i is recorded as P_j. If the path needs to be kept secret and subsequently used in a parent protocol, then it would be more suitable to record this information in a matrix with $P_{i,j} = 1$ indicating that the predecessor of i is j (and 0 otherwise). It is easy to adapt the algorithm for this case.

Security. Following the correctness analysis, the protocol does not need to leak intermediate values on any inner process. Moreover, Operations are provided by the $\mathcal{F}_{ABB}$. Additionally, The number of operations depends solely on the upper bound on the number of vertices (we assumed this to be public), therefore the same follows for the execution CPU time and memory usage. These adheres to our definition of security, no player learns anything but the output.

Complexity. The algorithm performs $|V|^2 + \mathcal{O}(|V|)$ comparisons (at Lines 6 and 11) and $\frac{4 \cdot |V|^3}{3} + \mathcal{O}(|V|^2)$ multiplications, dominated by Line 7 (the 4/3 factor is 4 times the sum of the square of the integers 1 to $|V|$). This distinction is important for small graph instances where the comparison complexity dominates over round complexity.

The performance of our privacy preserving version of Dijkstra has an extra factor of $|V|$ when compared with a vanilla implementation. Brickell [53] eliminates this overhead by revealing at each iteration the current shortest path. Our approach does not leak any information but the final shortest path. Moreover, it can also be extended to obtain the shortest path between any pair of vertices $(v, w) \in V$. It can also be seen that no special data-structure is needed, giving the $\mathcal{F}_{ABB}$ autonomy on the MPC-primitive selection.

3.5.1 Partial Data-Oblivious Adaptation

Data oblivious algorithms can be used in several contexts, including non secure applications. For example, imagine an asynchronous scenario, where operations have to be scheduled in advance, before the information arrives to the computational station. The private preserving protocol introduced in the previous section, can be used in such a context as well. However, it is possible, under the correct circumstances, to adapt the behavior of the protocol such that some decisions can be made in function of the data, improving the asymptotic complexity of the protocol with a simple adaptation.

When the information to be disclosed to the public is not only the length of the path but the path composition itself, the protocol complexity can be reduced to $\mathcal{O}(|V|^2)$. We propose to follow algorithm 5 and identify the minimum not treated vertex, but to do this without exchanging data with `condExch`. The vertex is part of the shortest path and can be opened without the risk of leaking extra information. Then, it suffices then to swap the information as before, but without the necessity of the `condExch` method, and more importantly, outside the loop of lines 5-8. This will achieve the reduction of the asymptotic complexity.

However, sometimes the algorithm designer would rather prefer to disclose the complete shortest path at the end of the computations or not disclose it at all. In this case, it suffices to securely and randomly permute the vertex associated data before the start of the computation e.g. the rows of the weighted adjacency matrix that describes the graph instance. A pseudo-label is used to identify the permuted information. In this way, the original vertex identifiers can be disclosed at the end of the computation, while the pseudo-labels can be used during the process.

Note that, this changes would limit the applicability of the algorithm on other non-oblivious applications.

3.6 Computational Experimentation

We conduct our experimentation over the open source MPC Framework VIFF [2]. This tool brings MPC functionality for the multiparty case with an easy user interface (prototypes are written directly in Python) and provides all the basic functionality from the $\mathcal{F}_{ABB}$. Furthermore,we provide results on how such a framework and its characteristics affect the instance sizes we are able to solve. We analyze the performance of our prototypes and compare them with our theoretical results. Additionally, we compare these results between our protocols with and without VIFF, such that we can measure the overhead between our private-preserving protocols and MPC primitives. This gains importance given that some real life applications use trusted third parties to for their secure computations, executing non-secure versions of the protocols and only revealing the final output. Note that in such single-point failure cases, if the trusted party is corrupted, then all private information might be compromised.

VIFF provides security under various security models. In our experimentation we consider only security against passive adversaries under the information theoretic model. VIFF also provides access to two different comparison mechanisms that were introduced by Tomas Toft [64, 45]. Our experimentation was conducted using the second [45]. Our tests were conducted using randomly generated complete graphs under the more classical semi-honest adversarial mode. In this setting, several instances for each of the tested graph sizes were executed and averaged. All prototypes executed outside VIFF use Python as well, and were executed one hundred thousand times and their CPU performance averaged to normalize the noise between executions. We experiment with 3 and 4 computational parties. Finally, we use a single workstation (server) with an Intel Xeon CPUs X5550 (2.67GHz) and 42GB of RAM memory, running Mac OS X 10.7. It has to be noted that all processes had the same amount of CPU power available for their use.

To provide means of analysis, our experimentation included the following prototype implementations:
Standard non-secure version (SNSV). Solves the shortest path problem with a vanilla implementation of the Bellman-Ford and Dijkstra algorithms. Indeed, these algorithms could effectively be used by a trusted third party to compute the shortest path problems. We use them for benchmarking against other implementations.
Secure version with VIFF (VVIFF). These are the implementations of our data-oblivious shortest path protocols over VIFF.
Secure version zero-cost Functionality (ZCOST). We assume the performance cost added by the secure functionalities of our $\mathcal{F}_{ABB}$ to be 0. We build our secure protocols using nothing but Python. The goal is to differentiate between the overhead introduced by the $\mathcal{F}_{ABB}$ itself, and the overhead introduced by everything else in VIFF. Under this paradigm, we measure, once again, the performance of our Bellman-Ford and Dijkstra implementations. We use the data for benchmarking against our algorithm's realistic times obtained with VIFF, and the standard prototype. Another strong motivation comes from how VIFF manages its memory and its strong effect on performance. Indeed, this

is clearly visible when a relatively large amount of operations are performed. Moreover, it can be seen in our experimentation.

3.6.1 Bellman Ford

We conducted our experimentation with 4-vertex to 20-vertex graph instances. Given that we only test complete graphs, we chose a weighted adjacency matrix representation instead of adjacency lists. The vanilla implementation used by the *SNSV* prototype includes the standard notion that in case no change was registered in the iteration the algorithm stops before reaching its worst case complexity. Table 3.2 shows the results of the experimentation with the VVIFF prototype.

Number of vertices		4	8	12	16	20
Execution times (in Seconds)	3 Players	3.5	38	140	350	697
	4 Players	4	38	165	405	809

Table 3.2: CPU Time from Secure Bellman-Ford protocol

Additionally, we extend our trial to 64-vertex graphs with 4096 edges. In this case, the computational time needed to solve such an instance was of 7 hours and 59 minutes. Because of VIFF memory issues, there is a monotonic increase on the CPU time needed to execute the MPC functionality. This explains the spike on the CPU time for such large instances. Figure 3.6.1 shows the ratio analysis and performance results of the experimentation.

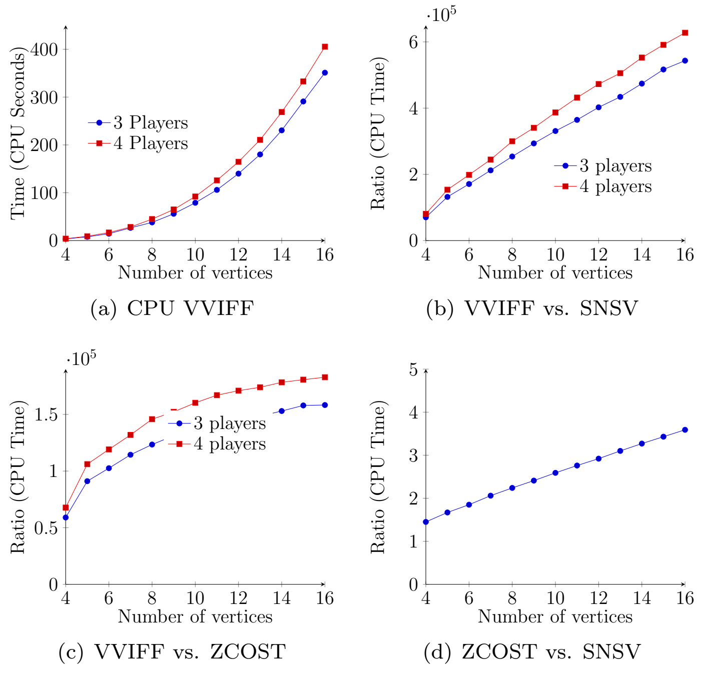
From these, we derive the following conclusions:

- The privacy-preserving Bellman-Ford protocol can solve small to medium graph sizes instances using VIFF in reasonable time.

- While increasing operational cost, the number of the parties involved does not alter the behavior of the algorithm performance.

- Although our secure protocol posses the same complexity bounds of the original Bellman-Ford algorithm, the intuitive adaptation of the stopping condition from the SNSV prototype influences the growth of the ratio between ZCOST and SNSV prototypes. It monotonically increases by a constant factor, approximately smaller than 20%. If such condition were to be implemented it would leak the iteration where the algorithm converges, and allow the adversary to adapt its input accordingly.

- When MPC-primitives are involved, the VVIFF prototype scored $\approx 2 \cdot 10^5$ higher times than its ZCOST counterpart. It has to be noted that although theoretically comparisons can achieve single round complexity, this is not the case for VIFF where a comparison can be ≈ 160 times slower than a multiplication. In smaller instances such operations dominate the spectrum but this is not the case when the sizes of the instances increase.



(a) CPU VVIFF (b) VVIFF vs. SNSV

(c) VVIFF vs. ZCOST (d) ZCOST vs. SNSV

Figure 3.6.1: Bellman-Ford CPU Times and Ratio Analysis

3.6.2 Data-Oblivious Dijkstra

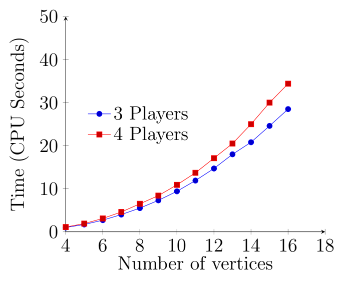
Experimentation was conducted with our main result for the Dijkstra algorithm. Once more, we consider complete graphs and represent them by weighted adjacency matrices, and various sizes of instances were solved using our Data-Oblivious Dijkstra protocol.

Additionally, we were able to run 64-vertex instances, using adjacency matrices, with a total of 4032 edges/matrix entries, taking around 18 minutes. The spike in computing time when working with bigger instances follows the fact of VIFF's difficulty to manage the memory for large graph instances. Table 3.3 shows the results obtained by our VVIFF shortest path prototype.

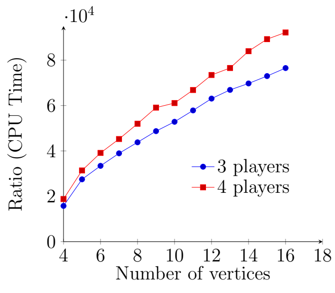
Number of vertices		4	8	12	16	20
Execution times (in Seconds)	3 Players	0.9	5	14	28	48
	4 Players	1	7	17	34	57

Table 3.3: CPU Time from Secure Dijkstra protocol

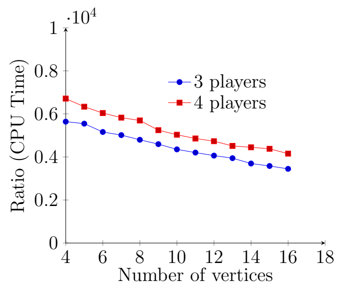
Figure 3.6.2 also shows the CPU time and respective ratios calculated from our different implementations:



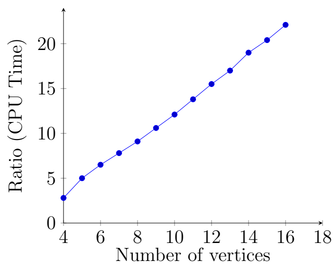
(a) CPU VVIFF



(b) VVIFF vs. SNSV



(c) VVIFF vs. ZCOST



(d) ZCOST vs. SNSV

Figure 3.6.2: Dijkstra CPU Times and Ratio Analysis

From this we can conclude the following:

- We can solve securely, in reasonable time, shortest path problems on complete graphs of sizes up to 64 vertices over VIFF.

- As expected, the number of players, have little incidence on the general behavior, given that in VIFF performance cost increases linearly in the number of players [2].

- Compared to the standard implementation, roughly a factor of $5000|V|$ is needed to securely solve the Dijkstra algorithm on VIFF.

- Roughly an extra factor of $1.4|V|$ is needed when executing MPC-primitives have 0 cost, to solve the problem securely.

- Combining the two previous remarks, we conclude that out of the $5000|V|$ overhead of our MPC implementation, the factor $|V|$ is explained by algorithmic design, a factor 1.4 is due to non-MPC issues like algorithmic changes, and the rest (a factor of a few thousands) is due to the MPC-related VIFF implementation.

- Finally, for larger graphs, the ratio between execution time of VVIFF and ZCOST decrease. This can easily be explained by the fact that it is the number of multiplications that gains importance as opposed to the number of comparisons, and that multiplications are substantially cheaper. A similar phenomena can be appreciated with our Bellman-Ford algorithm, in that case because multiplications and comparisons share their asymptotic complexity, it increases by a sub-linear factor.

A general conclusion is that security in our implementation comes at a very high cost, but not so high as to make the approach completely out of question in practice.

Chapter 4

Securely Solving the Max Flow, Minimum Cost Flow and Other Related Problems

4.1 Introduction

Secure Multiparty Computation (MPC), studies the problem where several players want to jointly compute a given function without disclosing their inputs; this problem was first addressed by Yao [1]. Different adversary models can be considered. A semi-honest setting, where corrupted players try to learn only what can be inferred from the information they have been provided with; or an active setting, where they manipulate the data in order to learn from any possible leakage caused. Several cryptographic primitives for secret sharing and homomorphic encryption, e.g. Shamir scheme [13] and Paillier encryption [22], have been proposed to address the problem.

Applications have emerged naturally in different fields, for instance, where all the secret information is sent by the players to a third trusted party who only reveals the final output. For example, in auctions, the auctioneer can be seen as a trusted third party. We study the scenario where no trusted third parties are allowed.

Since 1982, MPC has been in constant development [65, 3, 66, 29]. At first MPC was regarded as a theoretical research project. Later, the advent of frameworks like Fairplay [29], SEPIA [67] or the open source tool Virtual Ideal Functionality Framework (VIFF) [11], complemented by an improvement in performance bounds, fostered the appearance of real life applications e.g. Damgård et al. [31]. Faster MPC protocols immediately followed [33], widening the spectrum for application development. Classical network flow problems arise in real life applications in several areas e.g. project planning, networking, supply chain management, production scheduling. Combinatorial optimization, dynamic programing and mathematical programming have yielded polynomial-time algorithms for many of these problems (a detailed treatment can be found in Ahuja et al. [38]).

Our central objects of study are the Maximum Flow Problem (MF), the Minimum Mean Cycle problem (MMC) and the the minimum cost flow problem (MCF). We present algorithms that address privacy preserving constraints on these problems and solve them in polynomial time. We also empirically test the performance of our implementations. Finally, we explain our protocols as building blocks to solve more complex problems. For example, a WLAN network constructed by competing agents that want to securely compute, in a distributed fashion, their routing tables and the network flow configuration that supports its maximum traffic volume at the minimum cost possible. The routing algorithms could use our shortest path protocol introduced in Chapter 3, to securely define the routing tables. Moreover, a combination of the max flow algorithm [51] with our minimum cost flow protocol could be used to obtain the desired flow distribution securely. Note that for these types of application the number of vertices e.g. routers, is not necessarily very large.

The contents of this Chapter are taken from the Following Papers:

2013 Securely Solving Simple Combinatorial Graph Problems (Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, Mathieu Van Vyve), *In Financial Cryptography*, pp. 239-257, 2013.

2014 Securely Solving Classical Network Flow Problems (Extended Treatment) (Abdelrahaman Aly, Mathieu Van Vyve), *In CORE Discussion series Vol:2014/57*, 2014.

2014 Securely Solving Classical Network Flow Problems (Abdelrahaman Aly, Mathieu Van Vyve), LNCS 8949 ICISC, 2014.

4.1.1 Our Contributions

We provide algorithmic solutions to three classical network flow problems in a multiparty and distributed setting: the Maximum Flow Problem (MF), the minimum mean cycle using Karp's protocol and the Minimum Cost Flow using the Minimum Mean Cycle Canceling (MMCC) algorithm. To the best of our knowledge, this is the first time the last two problems have been studied under MPC security constraints. Moreover, we introduce polynomial bounds for all three problems relying only on black box operations, discussed in detail in the

following sections. The secret information can be distributed as pleased by the parties. Our work considers input data to be secret except for a bound in the number of vertices of the graphs, hiding the weights and costs of the edges as well as the topology of the graph.
First, we identify the best and more efficient algorithms that are suitable to work with the limited $\mathcal{F}_{ABB}$ functionality. Then, we introduce secure versions of these algorithms, guaranteeing their correctness and provide their complexity bounds. We also report on computational experiments on running implementations using Python and VIFF (given its status as an open source tool, capabilities, performance and availability).
Protocol Design. Performance is highly tied to the capabilities of the $\mathcal{F}_{ABB}$. How crypto-primitives are implemented and their scalability i.e. number of players, determine the general overhead of the protocols. Factors like the communicational round (a simultaneous exchange of information between all parties involved in the computation) complexity of the arithmetic methods, performance of the comparison protocol used and parallelization capabilities, influence the overall process. Just as we do, traditional works abstract these details and focus solely in the algorithm design.
Security and Correctness. The security of our algorithms comes from the fact that we use arithmetic black-box operations only and prevent any information leakage. This implies that the protocols are as secure as the MPC primitives they rely on e.g. information-theoretic secure (see also Section 13). Furthermore the correctness of our algorithms is essentially inherited from the correctness of the classical algorithms from which they are derived. More specifically, we modify the previously known and correct algorithms to avoid, in general, information leakage, while working on secret data, showing that these modifications do not alter their output.
Complexity. We use atomic communication rounds as our main performance unit to determine the complexity (round complexity) of our protocols. Although there exist protocols for comparisons with similar theoretical round complexity i.e. constant round, because of strong differences in real life scenarios performance-wise between comparisons and multiplications motivate tradeoffs, limiting the use of comparisons in favor of more arithmetic operations. i.e. additions and multiplications.

Table 4.1 presents the complexity bounds we have obtained. In all cases, the number of comparisons matches the complexity of their simple implementations on complete graphs. However, we need to introduce additional multiplications to hide the branchings involved in the algorithms.

	Advance Impl.	Simple Impl.	Complete Graphs	Privacy Preserving	Secure Comparisons
MF	$\|E\| \cdot \|V\| \cdot log(\frac{\|V\|^2}{\|E\|})$	$\|V\|^3$	$\|V\|^3$	$\|V\|^4$	$\|V\|^4$
MMC	$\|E\| \cdot \|V\|$	$\|E\| \cdot \|V\|$	$\|V\|^3$	$\|V\|^5$	$\|V\|^3$
MMCC	$\|V\|^2 \cdot \|E\|^3 log(\|V\|)$	$\|V\|^2 \cdot \|E\|^3 log(\|V\|)$	$\|V\|^8 \cdot log(\|V\|)$	$\|V\|^{10} \cdot log(\|V\|)$	$\|V\|^8 \cdot log(\|V\|)$

Table 4.1: Worst-case bounds of Original and Privacy-Preserving algorithms

Prototyping. VIFF, provided the functionality needed to evaluate our protocols and measure their performance. Details on the framework can be found in [2]. VIFF delivers by default security against honest-but-curious adversaries, and can be configured to provide certain protections against active attacks e.g. [44]. Our tests show the effects, in terms of performance, that the implementations of the MPC primitives can have in our protocols. Single rounded primitives, like multiplications can be consider to have a low performance cost (whatever it takes to send on a piece of secret shared or encrypted information towards other players on single round). Depending on the bit-size of the data processed, a more complicated operations such as Toft's comparison method used by VIFF [45], is ≈ 160 times less efficient than a multiplication. Current state of the art comparisons protocols [26] are still significantly more costly than multiplications.

4.1.2 Related Work

Sorting. Various sorting algorithms have been proposed using MPC, for instance Goodrich [68] introduced an approximated data-oblivious version of shell-sort that can be used to securely sort items. Jónsson et al. [69] show how to securely implement Batcher's odd–even mergesort sorting network. More recent work by Hamada et al. [70] introduced practically efficient sorting algorithms using a permutation step before sorting. Later work by Hamada et al. [71] on combinatorial sorting algorithms has been also proposed. Recently a data-oblivious version of a variant of the shell-sort called "zig-zag" sorting [72] by Goodrich has also been presented.
Privacy Preserving Graph Theory Protocols. Blanton et al. [54] introduced a data-oblivious algorithm to solve the maximum flow problem with a slightly bigger asymptotic complexity than our privacy preserving protocol i.e $\mathcal{O}(|V|^5 \cdot log(|V|))$. Their algorithm uses an adaptation of the Breath-First Search (BFS) algorithm to identify and saturate viable paths (path augmentation). We, however, use an adaptation of the FIFO Push/Relabel algorithm, to achieve $\mathcal{O}(|V|^4)$ complexity. Additionally, Aly et al. [51], presents a privacy-preserving adaptation of the Edmonds-Karp algorithm for the MF problem. Such implementation matches the original complexity of the algorithm i.e. $\mathcal{O}(|V|^5)$. The algorithm uses augmenting paths to saturate the graph until a maximum flow is

allocated. This is of course considering complete graphs. As it can be observed one of the main differences with the work hereby introduced is better complexity bounds. This can be explained by the fact that traditionally push/relabel algorithms present better bounds than their augmenting path counterparts in theory and practice.
Linear Programming. Protocols to securely solve linear programming problems have been reported on: [59, 60, 61]. All three solve the problem by using the simplex algorithm. Toft in [60] revises security weaknesses on Li and Atallah [59] propositions and presents termination conditions and methods for the algorithm, given that simplex has no polynomial-time complexity. Catrina and Hoogh [61] method implements simplex as well but includes an optimized support for rational numbers.
Other Constructions. Oblivious Random Access Memory has been used to solve some related problems, namely the shortest path problem e.g. [32, 58]. This is true for the client-server model and the multiparty setting as well. Privacy-preserving protocols to other graph theory problems are also provided. There are also applications that require privacy preserving protocols, involving some level of graph theory and make use of MPC for that matter e.g. [73]. Similar applications can be considered using the contributions of this thesis, where routing processes use MPC to provide privacy.

4.1.3 Overview

Section 4.2 describes the notation we use, as well as the cryptographic primitives. It also serves to introduce "building blocks" i.e. small algorithmic procedures that are regularly used. We begin our treatment of the maximum flow problem on Section 4.3 and introduce a privacy preserving algorithms in Section 4.4. Section 4.5 introduces the minimum mean cycle problem. Section 4.6 then explains the implementation using MPC limited functionality. Section 4.7 gives an overview of the minimum flow problem and the minimum mean cycle-canceling algorithm. In the sections ahead, details on the algorithm are presented. Following this, Section 4.8 specifies the secure formulation of the problem and gives some general remarks, it also provides the body implementation of the algorithm with MPC primitives and contemplate early termination conditions. Section 4.9 shows the results of our computational experimentation.

4.2 Preliminaries

4.2.1 Cryptographic Foundations

As in previous chapters, we make use the formalized notion of security introduced in Chapter 2. Our algorithms provides security against active and passive adversaries with no leakage under the *information theoretic* model. We simulate integer arithmetic using modulo arithmetic over some finite field of size M. The underlying condition is that such an M has to be sufficiently big to avoid overflows. Basic primitives of MPC such that Shamir's secret sharing [13], linear addition and multiplication rely on modulo arithmetic [4, 18]. The field can be defined as $\mathbb{Z}_M$ where M is a sufficiently big prime number. This holds as well for threshold homomorphic encryption mechanisms like Paillier encryption [22, 43].

MPC functionality can be instantiated by many different protocols and primitives' flavors, their use depends on many factors like the adversarial model or the number of parties involved in the computation. We want to abstract this primitive selection process and make it transparent for the algorithm designer by the use of the arithmetic black-fox $\mathcal{F}_{ABB}$ of Damgård and Nielsen [23] [15]. We assume that the $\mathcal{F}_{ABB}$ provides the algorithm designer with secure storage and retrieval of secret inputs over $\mathbb{Z}_M$, as well as secure addition, multiplications. Following [47, 51] we extend the functionality to provide comparisons. In practice, this implies that our protocols will be as secure as the underlying primitives that instantiate the MPC functionality of the $\mathcal{F}_{ABB}$. Moreover the data-oblivious characteristics of the protocols provide no information about the specific instance being solved. Our approach also takes into account the arithmetic circuit paradigm where results of any of such operations can be used as inputs for others.

4.2.2 Notation

We use square brackets $[x]$ to denote any secretly shared value stored in the $\mathcal{F}_{ABB}$. This is a common convention in the field e.g. [70, 26, 71]. Moreover $[\top]$ denotes a sufficiently large constant i.e. (greater than all inputs) that is used by our protocols, its size depends on the application at hand. Furthermore, it has to be sufficiently small to avoid any overflow $\mathbb{Z}_M$. Additionally, some inequality tests provide statistical security on some parameter k. All of these characteristics have to be taken into account when choosing the size of M.

We use the infix notation to represent the secure operations from the $\mathcal{F}_{ABB}$ functionality. Additions can be expressed as follows $[z] \leftarrow [x] + [y]$, where the result of the addition of $[x]$ and $[y]$ is assigned into $[z]$, and stored in the $\mathcal{F}_{ABB}$. Such values can only be made public through the agreement of a majority of parties. The same stands for operations between scalars and secret values. Negative numbers are represented as usual in the natural way: $M - x \quad \in \mathbb{Z}_M$.

Sets are denoted by capital letters e.g. L. Moreover $|L|$ is the number of elements of such set and L_i is the $i-th$ element in the set. Furthermore $[L]$

denotes a set where all its elements are secretly shared, however this does not necessarily imply that the size of such set is private.

Finally we review the sub-routine or building block constantly used by our protocols:

conditional assigment: Overloaded functionality of the assignment operator represented by $[z] \leftarrow_{[c]} [x] : [y]$. Much like in [51], [58], the behavior of the assignment is tied to a secretly shared binary condition $[c]$. If $[c]$ is one, $[x]$ is assigned to $[z]$ or $[y]$ otherwise. The operation can be characterized as follows: $[z] \leftarrow [y] + [c] \cdot ([x] - [y])$. The subroutine can be extended for other mathematical structures i.e. vectors, matrices.

4.2.3 On Network Flows and Matrix Representation

The number of vertices in the graph or at least an upper bound of them are assumed to be publicly known. As in previous chapters, there is no restrictions on how the information is distributed amongst the players. Following [51, 54, 58] we assume our protocols to work with complete graphs, as an instrument to hide the graph structure. This is why an adjacency matrix representation of the graph, using the vertices upper bound as its size, is preferred. Capacities and/or costs of the edges are represented as elements in adjacency matrices. This allows the algorithm designer to decouple the graph representation from its topology. The application designer has to define how the information about the topology is distributed and what should be hidden. For instance, if its known that each player owns at most a single vertex, then, each player has to secretly share a row of a capacity adjacency matrix where he places a $[0]$ at each unconnected vertex position or $[\top]$ if its a cost matrix.

4.3 Maximum Flow Problem

Let us consider the connected directed network $G = (V, E)$ where edge (v, w) has an associated non-negative integer capacity $u_{v,w} \forall (v, w) \in E$. Moreover $f_{v,w}$ is the flow circulating over edge $(v, w) \forall (v, w) \in E$. Vertex s and t are the source and sink of the network. The maximum flow problem refers to finding the maximum flow that can be allocated from vertex s towards vertex t such that the capacities of the edges and balance constraints in the vertices are respected. The problem can be defined as follows:

$$\begin{aligned}
\text{max} \quad & \sum_{(s,w)\in E(s)} f_{s,w} && && (4.1)\\
\text{subject to} \quad & \sum_{(v,w)\in E(v)} f_{v,w} = \sum_{(w,v)\in E(v)} f_{w,v} && \forall v \in V - \{s,t\} && (4.2)\\
& f_{v,w} \le u_{v,w} && \forall (v,w) \in E && (4.3)\\
& f_{v,w} \ge 0 && \forall (v,w) \in E && (4.4)
\end{aligned}$$

An important assumption of the problem is that network G should not contain a directed path from s to t composed solely of edges of infinite capacity. Moreover, we assume that whenever we have an edge $(v, w) \in E$, then (w, v) is in E, this is a non restrictive assumption given that such edges can have zero capacity.
We investigate how to securely solve the maximum flow problem using the push/relabel algorithm such that no information leakage is produced. Push/relabel algorithms are typically seen as more general and powerful than augmenting path alternatives. The algorithm, in its intermediate steps pushes flow through individual edges that still have available residual capacity. This briefly violates the balance constraints of the vertices, the algorithm keeps moving the flow until such constraints are respected. When vertices have a positive excess of flow, these are called active. By convention, source and sink vertices are never active. The algorithm continues pushing flow until no active vertices are left.
Generally speaking, a generic push/relabel algorithm can be defined as follows:

1. Identify active vertices.

2. Select and active vertex v.

3. Push and Relabel flow in vertex v.

4. In case there are still active vertices, *goto* to 2.

Note that a pre-process phase takes place where flow is pushed from the source towards adjacent vertices causing them to have some excess.
We investigate the FIFO variant of such algorithm also called relabel-to-front. It introduces two additional attributes to the vertices, the height (initially defined as the distance of the vertex from the source) and the excess. An adjacent edge is categorized admissible if it goes from a higher to a lower vertex. The algorithm alternatively pushes the excess along admissible edges and increases the height of the vertices until all excess has been pushed to the sink or back to the source.
The problem has several applications in various fields from electricity markets to telecommunications. Further details on the problem and its characterization can be found in [38].

4.4 Privacy-Preserving FIFO Push/Relabel Algorithm

For simplicity, we assume our privacy-preserving adaptation of the Push-Relabel algorithm works with complete graphs represented by a capacitated adjacency matrix. The algorithm can be easily adapted for the case when topology is publicly available. The same stands for the other problems studied in this Chapter. In practice, for a complete graph containing m vertices and $m \cdot (m-1)$ edges, we build the capacity matrix and the flow matrix $[C], [F]$ of dimension $n \times n$ where an entry (i, j) corresponds to the edge between the labeled vertices i and j. The flow matrix is initially empty. Two additional lists maintain the values of excess and height for each vertex and are noted $[e], [h]$ of dimension $1 \times n$. These two lists are initially empty.

The core operation of the original algorithm is the Push/Relabel phase, that is applied to a given vertex. This operation pushes all the excess through incident admissible edges (updating the excesses of incident vertices accordingly). Finally, in case not all the excess has been pushed, then the elevation of the vertex is minimally increased so as to create at least one more admissible edge, and Push/Relabel terminates.

Throughout the algorithm a list L with vertices with positive excess (except the source and the sink) is maintained. At each iteration, one vertex of L is selected and Push/Relabel is applied. The algorithm terminates when the list is empty. In the FIFO implementation, the next vertex of L to be treated is selected in the FIFO order. The original FIFO Push/Relabel algorithm terminates in $\mathcal{O}(|V|^3)$ operations.

Our privacy-preserving protocol for the Push-Relabel phase is presented in Protocol 6. The main differences between this protocol and the traditional Push/Relabel algorithm are as follows : *a*) when Push/Relabel is applied to a vertex with zero excess, no update of the elevation is performed at the end, *b*) in each phase, treat *all* vertices except the source and the sink, in a fixed order agreed between the players, and *c*) during each Push/Relabel operation applied to a vertex i, the order in which the edges (i, j) are considered is fixed and agreed in advance between the players. It is clear that these changes do not modify the correctness of the original algorithm.

Moreover, it can be verified that the relabel-to-front algorithm terminates in maximum $4|V|^2 - 10|V| + 12$ complete phases. Hence, it suffices to execute protocol 6 to such bound to guarantee correctness. From this, we can obtain an "all-cases" complexity of $\mathcal{O}(|V|^4)$, both in comparisons and multiplications. Note that this does not match the FIFO complexity, because we scan all edges at each pass, even when the excess of the tail vertex is zero. The complexity of this algorithm remains lower than the one of the original Edmonds-Karp from [51] Experiments showed that the use of a traditional halting criterion

Protocol 6: A phase of the protocol based on Push/Relabel

```
Input: A complete graph $G = (V, E)$ where $V$ is the list of vertices and
       $E$ the list of edges. A vertex $i$ to be treated, a vector of
       elevations $[h]$, a matrix of residual capacities $[R]$ and a vector of
       excesses $[e]$.
Output: Update of the elevations $[h]$, the residual capacities $[r]$ and
        the excesses $[\mathbf{e}]$.
1  $[\delta] \leftarrow 2|V|$ ;
2  for $j \leftarrow 1$ to $|E|$ do
3      $[\alpha] \leftarrow [h]_i > [h]_j$;
4      $[x] \leftarrow \min([e]_i, [R]_{i,j})$;
5      $[y] \leftarrow [\alpha].[x]$;
6      $[R]_{i,j} \leftarrow [R]_{i,j} - [y]$;
7      $[R]_{j,i} \leftarrow [R]_{j,i} + [y]$;
8      $[e]_i \leftarrow [e]_i - [y]$;
9      $[e]_j \leftarrow [e]_j + [y]$;
10     $[\delta] \leftarrow \min([\delta], [h]_j + 2|V| \cdot [\alpha])$;
11 end
12 $[\alpha] \leftarrow [e]_i > 0$;
13 $[h]_i \leftarrow [h]_i \cdot (1 - [\alpha]) + ([\delta] + 1) \cdot [\alpha]$;
```

at the end of each phase (i.e. nothing has been pushed) results in dramatic running time improvements. However it also demonstrated a huge variability (the algorithm may halt after a single phase), which suggests that a substantial amount of information could be derived from it. Quantifying this information is left for future work, and its impact is likely to depend on the application.

4.5 Minimum Mean Cycle Problem

The Minimum Mean Cycle problem (MMC) is to determine (on a directed graph $G = (V, E)$ with edge costs C) the cycle W with the minimum averaged cost (total cost divided by the number of edges in W). The original description of the problem and algorithms can be found in [74, 75, 76, 77].

Our interest on the MMC problem comes from the fact that it is used as a subroutine to solve the minimum cost flow problem by the minimum mean cycle canceling algorithm [78]. It is also used by other algorithms of the same nature. More details like applications, proofs and algorithms can be found in [38]. The following analysis assumes strong connectivity on G. In case a graph instance does not provide enough edges to fulfill this requirement, edges with a very large cost can be added to the graph.

The algorithm introduced by Karp [77] can be divided in two steps. First, we arbitrarily define a vertex s to be the origin of all paths to all vertices in V. Let $d^k(i)$ be the smallest weighted walk from s to the vertex i that contains exactly k edges. The walk obtained might contain one or several cycles. Then, we calculate $d^k(v)\ \forall v \in V$ with k from 1 to $|V|$. The following shows how to compute this recursively:

$$d^k(j) = \min_{\{i:(i,j)\in E\}} \quad \{d^{k-1}(i) + c_{ij}\}, \tag{4.5}$$

where $d^0(s) = 0$ and $d^0(v) = \top\ \forall v \in \{V - s\}$. Second, we calculate the cost of the minimum mean cycle as:

$$\mu^* = \min_{j\in V} \quad \max_{0\leq k\leq |V|-1} \left[\frac{d^{|V|}(j) - d^k(j)}{|V| - k}\right] \tag{4.6}$$

This expression can be intuitively explained as follows. Let j^* and k^* the indexes achieving μ^*. Then $d^{|V|}(j^*)$ is the cost of a walk containing the cycle W and $d^{k^*}(j^*)$ is the cost of the same walk with the cycle removed. The difference between the two yields the cycle cost. Proofs can be found in [77]. A strictly positive or negative μ^* means that at least a positive/negative cycle is present with μ^* as its mean. A case where the answer is 0 might also mean no cycle was found in the graph. The algorithm can be extended to find the cycle W as part of the answer. Overall algorithmic complexity is $\mathcal{O}(|V||E|)$.

4.6 Privacy-Preserving Minimum Mean Cycle

The privacy-preserving protocol we introduce follows the steps provided by the previous section. Moreover, each step and the whole protocol are designed to be used as sub-routines. As usual, our approach assumes all input data, but an upper bound on the number of vertices, is in secret form, including the adjacency matrix of costs $[C]$, where non-existing edges are represented by $[\top]$-. Additionally, all quantities are integers bounded by some M i.e. bigger than the greatest quantity to be analyzed, but still much smaller than the size of the field $\mathbb{Z}_M$ to avoid overflows, and the edge costs as an adjacency matrix. The final goal of the protocol is to obtain not only the mean cost of the minimum cycle, but the cycle itself as well. We use the function `getmincycle` to refer to the protocol.

Correctness. First, we have to replicate the result of equation (4.5). We select node 1 as the source node s. Implementing the recursion is fairly straightforward as the order in which the edges are scanned does not depend on the input. The more difficult task is to encode the walks. To that end, we define the 4-dimensional matrix $[walk]$ where $[walk]_{i,j,k,l}$ is the number of times the edge (i, j) is traversed by the shortest walk of length k from s to l. Also, because of the specific way we want to use our secure version of the MCC algorithm as a sub-routine, we define an additional argument $[b]$ to the protocol. Specifically, $[b]_{i,j} = 1$ indicates that the edge (i, j) is forbidden, i.e. cannot be part of the solution. The algorithm is detailed as Protocol 7.

Protocol 7: First step of: MMC protocol based on Karp's algorithm

```
Input: A matrix of shared costs $[C]_{i,j}$ for $i, j \in \{1, ..., |V|\}$, a binary
       matrix on viable edges $[b]_{i,j}$ for $i, j \in \{1, ..., |V|\}$.
Output: A matrix of walk costs $[A]_{i,k}$ for $i \in \{1, ..., |V|\}$ and
        $k \in \{0, ..., |V|\}$, a walk matrix $walks_{ij}$ for $i, j \in \{1, ..., |V|\}$
        encoding these walks.
1  $[A] \leftarrow [\top]$; $[A]_{00} \leftarrow [0]$; $[C] \leftarrow [C] + [\top](1 - [b])$;
2  for $k \leftarrow 1$ to $|V| + 1$ do
3      for $j \leftarrow 1$ to $|V|$ do
4          for $i \leftarrow 1$ to $|V|$ do
5              $[c] \leftarrow [A]_{ik-1} + [C]_{ij} < [A]_{jk}$;
6              $[A]_{jk} \leftarrow_{[c]} [A]_{ik-1} + [C]_{ij} : [A]_{jk}$;
7              $[walks]_{\cdot\cdot kj} \leftarrow_{[c]} [walks]_{\cdot\cdot k-1i} : [walks]_{\cdot\cdot kj}$;
8              $[walks]_{ijkj} \leftarrow_{[c]} [walks]_{ijkj} + 1 : [walks]_{ijkj}$;
9          end
10     end
11 end
```

Loop 5-8 checks whether edge (i, j) improves the walk of length k from s to j. This is done by comparing the best one found so far with cost $[A]_{jk}$ to $[A]_{ik-1}$ plus the cost of edge (i, j). Depending on the result, the best costs and walks are updated.

Second, we adapt (4.6) to obtain the value of the minimum mean cycle, as well as the encoding of the cycle. We achieve it by iterating over the matrices $[A]$ and $[walks]$ generated in the *first* step. The only difficulty is to workaround the non-integer division. In place of any costly procedure, we keep track of the numerators and the denominators separately, and compare only the cross multiplication instead. The minimum mean cost cycle is encoded as a $|V| \times |V|$ matrix $[min - cycle]$ where $[min - cycle]_{ij} = 1$ if the edge (i, j) is part of the minimum mean cycle. The rest of the algorithm is a straightforward implementation of (4.6). The details are provided as Protocol 8.

Security. Like with our previous results, no intermediate data is released and the operations are provided by the $\mathcal{F}_{ABB}$, following our definition of security.
Complexity. In total (Protocols 4 and 5), our implementation of MMC requires $\mathcal{O}(|V|^3)$ (Line 5 of Protocol 4) and $\mathcal{O}(|V|^5)$ multiplications or communication rounds (from the conditional assignments of Lines 7 and 8 of Protocol 4). One might ask whether this could not be brought down to $\mathcal{O}(|V|^4)$ by encoding the walks in Protocol 4 as a 3-dimensional matrix holding the predecessor node of

Protocol 8: Second step of: MMC protocol based on Karp's algorithm

```
Input: A matrix of walk costs $[A]_{i,k}$ for $i \in \{1, ..., |V|\}$ and
       $k \in \{0, ..., |V|\}$, a walk matrix $walks_{ij}$ for $i, j \in \{1, ..., |V|\}$
       encoding these walks.
Output: The cost of the minimum mean cycle $[min - cost]$. A matrix
        with the minimum mean cycle [min-cycle]$_{i,j}$ for
        $i, j \in \{1, ..., |V|\}$.
for $j \leftarrow 1$ to $|V|$ do
    [max-cycle], [max-cost] $\leftarrow \emptyset$;
    for $k \leftarrow |V|$ to 1 do
        [a-num] $\leftarrow [A]_{j(|V|+1)} - [A]_{jk}$;
        [a-den] $\leftarrow |V| - k$;
        $[c] \leftarrow$ [k-num] $\cdot$ [k-den] $<$ [a-num] $\cdot$ [k-den];
        [k-num] $\leftarrow_{[c]}$ [a-num] : [k-num];
        [k-den] $\leftarrow_{[c]}$ [a-den] : [k-den];
        [max-cycle] $\leftarrow_{[c]}$ $[walks]_{..|V|j} - [walks]_{..kj}$ : [max-cycle];
        [max-cost] $\leftarrow_{[c]}$ $[A]_{jk}$ : [max-cost]
    end
    $[c] \leftarrow$ [j-num] $\cdot$ [k-den] $>$ [k-num] $\cdot$ [j-den];
    [j-num] $\leftarrow_{[c]}$ [k-num] : [j-num];
    [j-den] $\leftarrow_{[c]}$ [k-den] : [j-den];
    [min-cycle] $\leftarrow_{[c]}$ [max-cycle] : [min-cycle];
    [min-cost] $\leftarrow_{[c]}$ [max-cost] : [min-cost]
end
```

each node. However, reconstructing the walks for the operation performed at Line 9 of Protocol 5 would then need $\mathcal{O}(|V|^5)$ conditional assignments instead of the currently $\mathcal{O}(|V|^4)$. So we prefer to stick with our simple and as efficient approach.

4.7 Minimum Cost Flow Problem

The Minimum-Cost Flow problem (MCF) is of finding a feasible flow in a capacitated directed graph $G = (E, V)$ that minimizes the costs (proportional to the magnitude of the flows). The problem can be modeled as a linear program but there exists more efficient strongly polynomial time combinatorial algorithms, see [38, 79]. The more traditional minimum capacitated cost flow problem can be shown to be equivalent to the transshipment and the minimum-cost circulation (MCC) problem.

Formally, the MCC problem is of finding a capacitated flow in a symmetric graph $G = (E, V)$ of minimum cost. The problem can be modeled as follows:

$$\begin{aligned}
&\text{min} && \sum_{\{v,w\}\in E} C_{v,w} f_{v,w} && && (4.7)\\
&\text{subject to} && f_{v,w} \leq U_{v,w} && \forall (v,w) \in E && (4.8)\\
& && f_{v,w} = -f_{w,v} && \forall (v,w) \in E && (4.9)\\
& && \sum_{v\in\mathbf{E}(w)} f_{v,w} = 0 && \forall w \in V && (4.10)
\end{aligned}$$

Here the graph is assumed to be symmetric, i.e. for every $(v, w) \in E$ there is an edge $(w, v) \in E$. Each edge (v, w) has a maximal capacity $U_{v,w}$ and a cost $C_{v,w}$ per unit of flow. Additionally, all costs are antisymmetric, i.e. $c(v, w) = -c(w, v)\ \forall (v, w) \in E$. The variable f represents the amount of flow passing through an edge. Using this notation, the residual capacity can be formally defined as $r_{v,w} = U_{v,w} - f_{v,w}$.

Constraints (4.8) are the capacity constraints. Constraints (4.9) are the flow antisymmetry constraints. Constraints (4.10) are the flow conservation constraints at each node. This characterization of the problem is the same used by Goldberg and Tarjan [78] for their description of the MCC problem using the Minimum Mean Cycle-Canceling algorithm (MMCC). It can be seen as a variant of the non-polynomial cycle-canceling algorithm proposed by Klein in [80], but where the next cycle to be canceled is chosen by finding the minimum mean cost cycle. The change makes the algorithm strongly polynomial, i.e. its complexity depends only on $|V|$ and $|E|$ and not on other parameters.

The algorithm is based on the finding of Busacker and Saaty [81], which asserts that a circulation with no residual negative cost cycles is of minimal cost. Moreover, the algorithm can be characterized as follows:

1. Initialize the feasible circulation to 0.

2. Obtain the minimum mean cycle W in the associated residual graph.

3. Set $\delta \leftarrow min\{(v, w) \in W r_{v,w}\}$.

4. Augment the flow by δ along the cycle W.

5. If there are still negative cycles *goto* 2.

Basically, we compute the cycle with the minimum negative average cost W in the associated residual graph. Then, we augment the flow along this cycle until an edge reaches its capacity. This process is repeated until no negative cycle is found. Its complexity is $O(|V|^2 \cdot |E|^3 \cdot \log |V|)$.

4.8 Privacy-Preserving Minimum-Cost Flow

The input data are the capacity and cost adjacency matrices $[U]$ and $[C]$, where non-existing edges are represented by $[0]$ on the capacity matrix and by $[\top]$ on the cost matrix. As usual, all input data is secretly shared, except the bound on the number of vertices. We assume all values are integer and of a bounded size much smaller than M to avoid overflows in the field $\mathbb{Z}_M$. The solution is to be provided as the flow matrix $[F]$ and total cost $[totcost]$. The final composition of $[F]$ might leak some details on the graph's topology depending on the answer. The protocol can be used as a sub-routine for more complex applications in case the final output is kept private. Once the MMF problem is modeled as a MCC problem, it is sufficient to securely solve the minimum circulation problem using a privacy-preserving implementation of the MMCC algorithm to obtain a flow of minimum cost.

Adapting the MMCC algorithm. If one wants to avoid any leakage of information, an important difference between a standard implementation and a secure one is that the augmenting flow process has to be repeated as many times as the worst case analysis guarantees, instead of stopping it as soon as no negative cycle is detected. We call each flow augmentation along the cycle a phase/iteration. We use the bound provided by Goldberg and Tarjan on [78]: $|V||E|^2 \log |V| + |V| \cdot |E|$ flow augmentations at most. Note that this is not an asymptotic bound. Given that we also hide the graph structure, $|E|$ has to be replaced by $|V|^2$ in our complexity estimates. Our secure protocol requires to perform that many iterations to guarantee correctness with no leakage. Possible stopping conditions to reduce the number of iterations are considered later in this section.

Protocol 9 shows our privacy-preserving alternative for the MMCC algorithm, which is a straightforward translation of the algorithm outlined above.

Correctness. The initial solution is set to zero at Line 1. The body of the main loop is one flow augmentation phase. It starts by calling our secure implementation of the Min Mean Cycle problem, leaving out saturated edges. Loop 5-9 computes the maximum augmentation possible along the cycle identified. If the cycle has non-negative cost, this augmentation is set to zero at Line 10, before updating the cost of the solution. Then the flow itself is augmented at Loop 12-17.

Security. Following the previous protocols, the current algorithm does not leak intermediate values and uses $\mathcal{F}_{ABB}$ operations to calculate secret data, respecting our definition of security.

Complexity. The most costly operation during one augmentation phase is the call to `getmincycle` with $\mathcal{O}(|V|^3)$ comparisons and $\mathcal{O}(|V|^5)$ communication rounds. The overall complexity is $\mathcal{O}(|V|^8 \log |V|)$ comparisons and $\mathcal{O}(|V|^{10} \log |V|)$ communicational rounds. As mentioned above, one main difference between our secure Minimum Cost Flow algorithm described above and

Protocol 9: Privacy-preserving MMCC

```
Input: $|V| \times |V|$ matrices of shared capacities $[U]_{i,j}$ and shared costs $[C]_{i,j}$.
Output: The $|V| \times |V|$ matrix of flows $[F]$ and the associated total cost $[totcost]$.
1  $[F], [b], [totcost] \leftarrow 0$ ;
2  for $k \leftarrow 1$ to $|V|^5 \log |V| + |V|^3$ do
3      $[cost], [cycle] \leftarrow getmincycle([C], [b])$;
4      $\delta \leftarrow [\top]$;
5      for $(i, j) \in [U]$ do
6          $[r] \leftarrow [U]_{ij} - [F]_{ij}$;
7          $[c] \leftarrow [cycle]_{ij} \cdot ([\delta] > [r])$;
8          $[\delta] \leftarrow_c [r] : [\delta]$;
9      end
10     $[\delta] \leftarrow [\delta] \cdot ([cost] < 0)$;
11     $[totcost] \leftarrow [totcost] + [\delta] \cdot [cost]$;
12     for $(i, j) \in [F]$ do
13         $[c] \leftarrow [cycle]_{ij}$;
14         $[F]_{ij} \leftarrow_c [F]_{ij} + [\delta] : [F]_{ij}$;
15         $[F]_{ji} \leftarrow_c [F]_{ji} - [\delta] : [F]_{ji}$;
16         $[b]_{ij} \leftarrow [U]_{ij} - [F]_{ij} > 0$;
17     end
18 end
```

a standard implementation is that, to guarantee no leakage of information, we have to execute as many iterations as in the theoretical worst case. This makes the practical performance of the algorithm much worse than a standard implementation because, in most practical applications, it is expected that the number of iterations needed to find the optimal solution is much smaller than the theoretical upper bound. Of course, one could easily publicly reveal the outcome of the test performed at Line 10 of Protocol 6 and stop the algorithm if the cost of the cycle is non-negative. But some information would be leaked.

To limit the amount of information leaked, several strategies are possible. One is to open the test every K iterations, with K being a publicly known integer. Another alternative is to multiply the result of the test by a random bit (for $= 1$ with probability p) to statistically hide the result. These two would also be combined. In both cases, the parameters (K and/or p) would control the trade-off between performance and information leaked.

4.9 Computational Experiments

Although we can determine theoretical bounds for our protocols, processing time is also constrained by a variety of phenomena; the way on which the secure arithmetic functionality is implemented and the infrastructure were prototypes are executed affect their running time in real life applications. Therefore, we also want to empirically test the performance of these algorithms. More specifically we want to analyze the following aspects.

Measure the size of solvable instances using currently available MPC framework e.g. (VIFF). The theoretical bounds only give a rate of increase with the size of the instance. They do not say anything about the actual computing time. Also, the scalability on VIFF is a concern. Our interest is to determine what is the size of the instances that can be solved in a "reasonable" amount of time. Moreover, we want to determine the impact that the number of players and the size of the graph instances have on CPU time performance. As it was mentioned VIFF was chosen, given its availability (open source) and easy coupling with larger applications.

Comparison between complexity bounds and practical behavior. Implementing the protocols described is a good way to check and demonstrate that no aspect of the problem has been neglected in our analysis.

Overhead of secure vs. non-secure implementations. Real life applications, in many cases, employ a trusted third party to share and compute their information. It executes a non-secure implementation of an algorithm solving the problem, and then reveals to the players the final output only. In this context, we try to determine the real overhead in performance of an implementation with secure and a non-secure implementation of the protocols studied.

To answer these questions, we have run the following implementations :

Standard non-secure version (SNSV). We provide a vanilla implementation for both problems: the push/relabel algorithm to solve the maximum flow problem and the minimum mean cycle canceling algorithm to solve the minimum capacitated cost flow problem. In both cases such implementations could be used by a trusted third party to compute the answers instead of our private preserving protocols. We benchmark its performance against our other prototypes.

Secure version with VIFF (VVIFF). We use VIFF to implement our privacy-preserving MF and MMCC algorithms, to use them for benchmarking.

Secure version zero-cost Functionality (ZCOST). Performance is somehow influenced by the implementation of the $\mathcal{F}_{ABB}$ functionality itself. This is specially evident with VIFF, where overhead is induced not only by the intrinsic cost of the functionality but for implementation factors like memory consumption. We pursue to analyze the behavior of our algorithms assuming that such functionality have no performance-cost. That way we can differentiate between the overhead that comes from the $\mathcal{F}_{ABB}$ and its implementation and our privacy preserving protocols. We have implemented our privacy-preserving algorithms using Python. We use these results to benchmark the performance of our algorithms against vanilla implementations of non-secure versions and VIFF implementations.

We benefit from VIFF's passive security under the information theoretic model on the multiparty case. VIFF provides access to Shamir secret sharing [13] and BGW multiplications [4] optimized by Gennaro et al. [18]. For comparisons we use the most recent Toft comparison method implemented [45]. VIFF was selected because it is freely and openly accessible. Additionally, for our experiments we use randomly generated complete graphs. All results presented are averaged over 20 instances of the same size with 3 and 4 players.

All trials used the same workstation, an Intel Xeon CPUs X5550 (2.67GHz) and 42GB of memory, running Mac OS X 10.7. Additionally, every single process had the same amount of CPU power and memory available. Execution times obtained from the non-secure implementations are in the order of microseconds. This means that the times can be highly influenced by noise during the tests. To normalize this noise, we execute the standard non-secure implementations one hundred thousand times and then report the averages.

4.9.1 Maximum Flow Problem

For the privacy preserving push relabel algorithm, we evaluate the CPU time it takes for a single Push/Relabel iteration to be executed. Stopping conditions with some leakage can reduce computational times. We can imagine real life applications, where trade-offs between security and performance are put in place. This gives relevance to measure phase times. Full execution times can be extrapolated from such values. All instances represented complete graphs. We experiment with various graph sizes to later derive some conclusions. Table 4.2 shows the experimentation results.

Number of vertices		4	5	6	7	8	9
Execution times (in Seconds)	MF Phase - 3 Players	2.7	5.3	8.2	12.3	16.9	22.1
	MF Phase - 4 Players	3.2	6.1	9.7	14.5	20	26.2

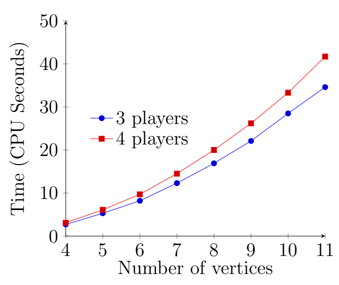
Table 4.2: Execution times per phase Secure Max Flow Algorithm for a complete graph.

Furthermore, figure 4.9.2 shows the results of the ratio analysis based on the various results obtained from our different prototypes.

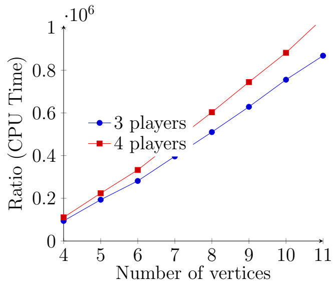
From the results we conclude the following remarks:

- When executed to the worst complexity and no stopping condition is involved, computational cost is relatively "high" for any graph size.

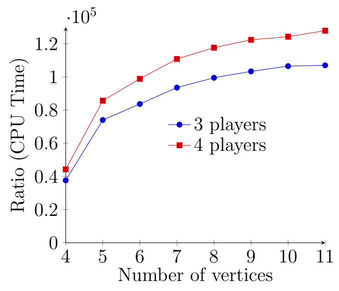
- As with previous results, increasing the number of parties involved in the computation caused some increase in the computational times by some constant factor.



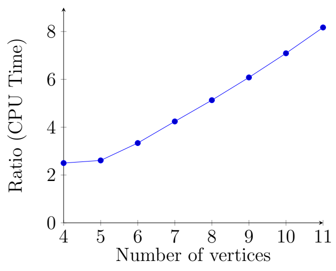
(a) CPU VVIFF-Phase



(b) VVIFF vs. SNSV



(c) VVIFF vs. ZCOST



(d) ZCOST vs. SNSV

Figure 4.9.1: Secure Max Flow CPU Times and Ratio Analysis

- When we compared our VVIFF vs SNSV prototypes, we found that the overhead is around $10^6 \cdot |V|^2$, which is justified by the cost associated to the MPC-primitives and algorithmic adaptations. Moreover, factors like early termination of the SNSV prototype (when a solution is found before reaching worst case complexity) is also influential.

- The influence of comparisons on the behavior of the ratio between VVIFF and ZCOST prototypes tends to be dominated by the multiplications, when the size of the instance grows. this reduces the monotonic increase of the ratio in the cases we have observed.

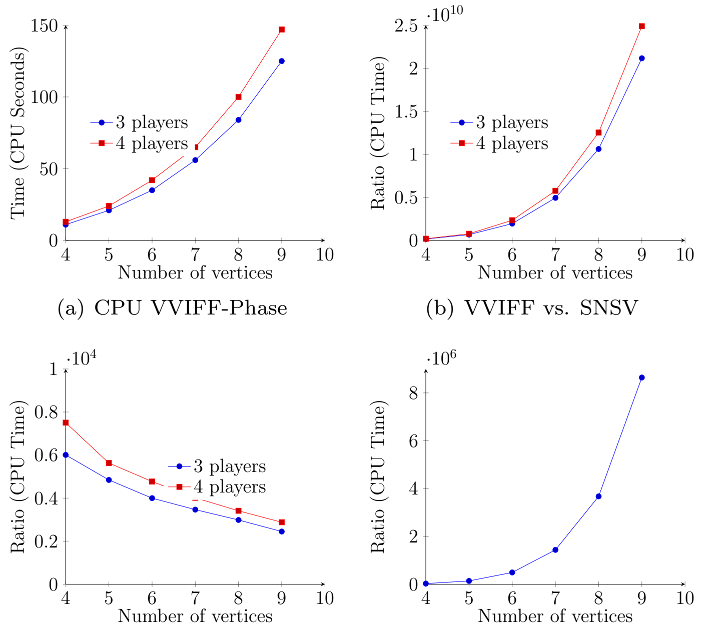
- There is a relatively small monotonic increase of the ratio between the ZCOST and SNSV prototypes. This is due to the early termination conditions that can be met by the SNSV prototype. The adaptations made to avoid information leakage, as well as the increase on the complexity bounds have also influenced the results.

4.9.2 Minimum Cost Flow Problem

For the minimum flow problem, we measure the time a single phase (one iteration of Protocol 9) takes to be executed, given that stopping conditions with some leakage can substantially reduce the number of phases needed e.g. A graph with a single cycle would only take one phase to be completed. To estimate the execution time of the full algorithm, it suffices to multiply this by the known number of phases needed. Our analysis includes the ratio between the time it takes the SNSV prototype to find an answer and the execution time of privacy preserving versions without stopping conditions. The results of these experiments can be found in Table 4.3 and Figure 4.9.2.

Number of vertices		4	5	6	7	8	9
Execution times (in Seconds)	MMCC Phase - 3 Players	11	21	35	56	84	125
	MMCC Phase - 4 Players	13	24	42	65	100	147

Table 4.3: Execution times per phase MMCC Algorithm for a complete graph.



(a) CPU VVIFF-Phase

(b) VVIFF vs. SNSV

(c) VVIFF vs. ZCOST

(d) ZCOST vs. SNSV

Figure 4.9.2: Secure MMCC CPU Times and Ratio Analysis

From these we can conclude the following:

- The fully secure version of our implementation is highly costly in terms of performance even for very small instances. This highlights the necessity of using termination conditions.

- Once again, the influence of the extra player has little incidence on the overall performance time.

- The overhead of our secure implementation versus a standard one is of the order of $2.5 \cdot 10^8 |V|^2$. Note that both algorithms have different complexity functions and vanilla versions of the algorithm typically converge towards an answer before reaching its worst case complexity.

- Again, one can observe that the multiplications absorb a larger fraction of the computing time as the size of the instances increases.

Chapter 5

Conclusions

Strongly polynomial-time algorithms are appealing for MPC implementations because, as the worst-case complexity is polynomial, it is possible to obtain fully secure (i.e. no leakage) and theoretically efficient algorithms and implementations. We have demonstrated this for three classical network problems: Shortest Path, Minimum Mean Cycle and Minimum Cost Flow, as well as for Minimum Mean Cycle Problem. However, our computational experiments demonstrate that the price to pay for such security is very high for the simplest problem (Shortest Path) and extremely penalizing for the more complicated ones.

This research raises several questions for further work. A first one is whether theoretically more efficient algorithms can be obtained for these problems. Another one is related to the development of more efficient MPC platforms compared to the one we used for our computational experiments. Also one could consider other classical optimization problems. Obvious candidates are the Matching problem and Linear Programming. From a theoretical point of view, Linear Programming is probably more interesting as no fully combinatorial theoretically efficient algorithm is known for this problem. The efficient algorithms known (i.e. interior point methods and the volume algorithm) require linear algebra and it is not clear that they can be fully adapted to the general MPC constrains we report on.

More complex applications are also possible, using our protocols as building blocks, for instance: Imagine you have calculated the maximum flow of your network, and now want to obtain the configuration, equivalent to the maximal flow, at the minimal cost possible. As we estated, this can be achieved by obtaining the maximal flow using our MF Push/Relabel secure implementation from algorithm 6, immediately followed by out MMCC protocol described in 9.

Part III

Towards Practically Efficient Applications

Chapter 6

A Modular and Compact MPC Toolkit

6.1 Introduction

This Chapter covers the design and construction of the MPC primitives introduced in part I in a compact, modular MPC Toolkit codenamed "Edge Runtime". Through our experimentation with our protocols and VIFF, we have been able to see the deep impact that the implementation of the MPC primitives has on the performance of the prototypes. To better estimate the behavior of possible real life applications, we need a streamlined set of tools that provide the basic MPC functionality that our algorithms need. Currently, the tools that we have at our disposal are either too general or too specific and their availability and adaptability is limited. The Toolkit has been conceived not only to be a tool to achieve practically efficient times on the algorithms introduced by this thesis, but also to provide the protocol designers a good prototyping tool. It has been conceived to achieve better CPU times and lower memory consumption than its open source counterparts against semi-honest adversaries. Additionally, we test its performance with our secure Bellman-Ford algorithm from Chapter 3 and some atomic operations i.e sharing, multiplications. The content of this Chapter is based on the following Technical and User Reports of the application delivered to the Walloon Region as part of the CAMUS project and others:

2014 Edge Runtime Technical Manual (Abdelrahaman Aly, Mathieu Van Vyve), *CAMUS Report*, 2014.

2014 Edge Runtime User Manual (Abdelrahaman Aly, Mathieu Van Vyve), *CAMUS Report*, 2014.

One of the goals of our implementation is modularity. We have chosen C++ and an Object Oriented Architecture because it allows us to keep low coupling levels and it is easily adaptable for realistic life problems. Moreover, the selected set of primitives and applications implemented are essential building blocks for more complex problems and solutions described in this thesis. The design of the Toolkit was thought to support future extensions and expansions that could include state of the art developments without the necessity of radical changes in the existing code. We have eliminated the burden of an interpreted language like Python or Java and its performance, and we use the available tools in C++ to build the Toolkit, respecting the principles of the Object Oriented Programing (OOP) paradigm. This is not a replacement for more complete frameworks similar to PICCO [19] or VIFF [2]. It is instead a set of streamlined functionality compiled into a single library that, thanks to its modularity and software composition, can be easily used and improved to obtain specific results on custom developed software.

The Toolkit implements standard protocols for MPC and different algorithmic solutions to well known problems like inequality tests. Furthermore, the Toolkit can solve ≈ 16500 multiplications per second and a comparisons in a bit more than 8 milliseconds. To put this in contrast, this is between 8 to 10 times faster than VIFF under the same configuration.

6.1.1 Related Work

Relevant work has been done for two and multiparty computation frameworks that support standard share mechanisms and are secure under various adversarial and communicational models. Although their design provides in many cases robust functionality in terms of security, factors like simplicity, adaptability and component based design have been overlooked. Our general goal is to build a modular MPC application that works efficiently providing the security levels required in a streamlined process. Moreover, we pursue the following differentiating characteristics:

Software Composition. Modular design, where new elements can replace old ones without a code overhaul. We achieve this by following OOP principles and using a component driven architecture.

Simplicity. Only the tasks that are indispensable are built into the application. Our goal is efficiency and not functionality.

Adaptability. We have limited greatly the use of third party libraries. This is specially true for the communications support, where we have designed our own modular components to data transmission and recollection. That way we have unrestricted access at almost every component level of the code.

Some of these frameworks, like Fairplay [28] and TASTY [82], are specialized in two-party computation using, amongst other things, garbled circuits and at times homomorphic encryption mechanisms. Fairplay was later improved and transformed into FairplayMP [29], to solve the multiparty case, although it did not directly implement linear secret sharing. We have focused our attention on the multiparty case (current version supports 3 computational and designed for n players) and the use of standard sharing mechanisms and not garbled circuits in general.

Frameworks like Sharemind [30] or VIFF [2] focus on the multiparty case. Sharemind is considered an efficient tool for the three party case and have implemented some of their functionality directly into Assembly to improve its performance. However, they do not rely on conventional sharing mechanisms to obtain faster results in the three party case. Despite the benefits in terms of performance, these techniques do not automatically work with n parties. This is the main reason why Sharemind is mainly promoted as a three party computational tool. Component adaptability is also limited with Sharemind, given that it is a commercial tool, and the access to its source code is limited. On the other hand, VIFF is an open source tool designed to work for the multiparty case with no restriction in the number of players, secure against passive adversaries and functionality for the active case. It has been used in academic e.g. [51] as well as in commercial environments [31]. This is not only because of its large set of functionalities, but also of its availability and code adaptability (the source code is available online). VIFF is also a Python based application, which means it is interpreted by a Just In Time compiler. We can see how using Python as a platform facilitates code readability and allows better access to the general public, Python is a 4th generation programming language, easy to understand and program with. However, the cost of having these nice properties is performance. The fact of working over an interpreted language has had an impact on VIFF's performance output. VIFF was built to make up for this performance difference with a strong parallelization processing with the use of deferreds and asynchronous scheduling of operations. This, in return, has caused a high memory consumption and added extra processing time to the tasks given its sometimes unnecessary parallelism. Basically, VIFF tries to parallelize all operations although there is no support for physical parallelism for Python deferreds (logically deferreds are given individual threads, but the original Python Virtual Machine can only use a single CPU). Moreover, given that VIFF is designed to not to forget shares (this is necessary because of the architecture devised to solve the asynchronous scheduling), memory use increases monotonically, and the indexation of many small objects stored in memory causes high latency and reduces its performance.

Sharemind and VIFF can be pictured as two opposite design options, where the first one tries to optimize by reducing functionality the other diversifies. Our Toolkit, instead, tries to occupy the practical middle ground. Although this work introduces a version of the Toolkit that works with three parties, we use conventional sharing mechanisms that could be easily generalized to support more players. We have also tried to improve readability by introducing strong coding standards and strictly following the OOP paradigm, but have not sacrificed performance using a more suitable OOP tool like Java or Python. We have centered our efforts instead, on exploiting the properties of OOP while working with C++, a less suitable but undoubtedly more efficient tool. The same arguments can be made about our Toolkit's modularity. Finally, we have reduced our memory footprint to avoid latency problems allowing us to compute millions of operations with little to no variation in memory consumption.

Finally, although our work shares some of the same goals with other multipurpose compilers like PICCO [19] or the ORAM tool introduced by Liu et al. [32], we have followed a different approach. They have chosen to use compilers instead of libraries in C, to empower some specific functionality, like easy and intuitive parallelization or access to ORAM data-structures, but more importantly to enhance readability. We focus on building a compact and simple streamlined process. Although these factors are also important, our approach is centered on software composition, where the pieces can be improved without any general overhaul of the Toolkit. The availability of these frameworks is also a factor. Furthermore, the Toolkit is thought out to be used essentially as an prototyping mechanism where the cost of atomic operations can be evaluated and contrasted. This makes component adaptability an important characteristic as well.

6.1.2 Application Features

The MPC Toolkit is capable of solving functions in a collaborative setting with MPC primitives using Shamir Secret Sharing, BGW [4] and other protocols and applications, in a secure fashion. It currently supports 3 parties and it has a structure in place to support $n \geq 3$ computational parties. It minimizes memory consumption, manages its own communications and uses the powerful GMP (GNU Multiple Precision)library for core field operations on NTL (Number Theory Library). The following is a description of some of its features. A detailed treatment can be found in [83].

Security Model. Our interest in terms of security has been focused on providing a solution for the classic setting of semi-honest adversaries and threshold corruption using the findings first introduced by BGW [4]. In this case perfect security can be achieved as long as the adversary does not corrupt more than half of the computational parties. For the multiplication protocol we use the traditionally improved version from Gennaro et al. [18]. The Toolkit uses, as a sharing mechanism, the Shamir Secret Sharing scheme [13]. We also assume parties are connected through secure and authenticated channels. The task of building the channels is decoupled from the Toolkit itself. In case this is not an alternative for the algorithm designer, the Toolkit supports the use of custom-developed sockets that allow the use of cryptographic mechanisms to protect the communication channels. This is possible thanks to the custom communicational support built specifically for the Toolkit's functionality.

Synchronous Execution Model. It is in our interest to minimize the impact of repetitive tasks on the Toolkit's performance. Memory indexation, for instance, is a factor that has weighed heavily on typical MPC implementations such as VIFF [2]. A synchronous model allows us to avoid the storage of great volumes of shares during processing of MPC tasks, minimizing the use of memory. Moreover, it prevents complicated orchestration mechanisms in order to successfully coordinate the shares handling. Our library adapts the classical clock approach of some synchronous MPC constructions and replaces it by an operation counter that acts much like an authorization flag. Only when all shares of the previous iteration have arrived, the Toolkit is authorized to increment the counter for the next operation. This approach, although beneficial in terms of performance, would also have impact on the security of the applications, it would give the adversaries the ability to exert control over the transmission times and scheduling of their own shares. Moreover. The implications of such a phenomena could incur on waits in case the attacker decides to hold its shares. The Toolkit is designed to process incoming information as fast as possible, which implies that no legacy data from previous iterations is stored implicitly by the Toolkit nor is it expected to do so.

Low Coupling, High Specialization and Wide Software Composition. The Toolkit strictly follows the OOP paradigm to facilitate software composition. Our design was oriented towards low coupling and high cohesion. Moreover, the application is built on layers, this allows to separate functionality by class families that fulfill specific tasks. Principles such as encapsulation are used throughout the toolkit allowing modularity to be a reality. The Toolkit can be seen as some kind of block construction. When some functionality needs to be changed, it is sufficient to replace the block responsible of the task at hand. Polymorphism is also used to reduce coupling between components and classes. C++ is a portable language that respects the OOP principles and provide the tools of the paradigm. Indeed, any method in our classes that parameterizes abstract objects, is enforced to work with pointers. In this way, the coding logic would also respect OOP principles. This forces us to be extremely careful with the management of object instances, not only to avoid memory leaks, but also to prevent accidental deallocations. The principle of extending OOP practices is the corner stone of the application modularity and easy software composition.

6.1.3 Overview

This Chapter is divided as follows: Section 6.2 introduces the Toolkit internal design. Technical characteristics and a general specification of its capabilities can be found in Section 6.3. Section 6.4 is a revision of the functionality provided by the Toolkit. We introduce the results of the computational experimentation in Section 6.5. Finally, we explore future improvements and the Toolkit's development road map.

6.2 Architecture and Design

The MPC Toolkit is a compact collection of 64 files containing several classes written in C++ distributed in 21 different namespaces. They cover various aspects, from the implementation of the Shamir sharing scheme [13] to several comparison methods and code examples. It has a built-in support for numbers of up 63 bits in length. The application was compiled and tested for Mac OS X 10.7. The minimum requirements are 500 KB in RAM Memory and 2 CPU Cores per engine instance.
Furthermore, GMP has been placed at the core of the mathematical operations. This powerful library has been used to compile the NTL libraries that are used to perform the modulo arithmetic.
Following the OOP paradigm, no method nor any functionality has been implemented outside a host class. The 34 classes that comprise the application are structured as follows: a header file to declare the class and a .cpp file that implements it. Both use the class name as its file name. These classes have been logically organized in namespaces, and physically in class groups. Following the same logic than the class structure, namespaces correspond to those of the class groups. We have in total 21 different groups that host 68 headers and .cpp files.
At the core of the Toolkit, there is a portable engine tied to a sharing mechanism. It provides the basic and core MPC functionality. The current version of the Toolkit implements a Shamir sharing engine, although future versions could include different sets of engines. There is a structure put in place to support future implementations. Additionally, this version hosts a series of examples and applications. They typically make use of the engine, either by instantiating it or as a parameter during their instantiation. Figure 6.2 shows the invocation scheme of one method that uses the engine functionality, towards and from the engine, and how application classes use the engine functionality.

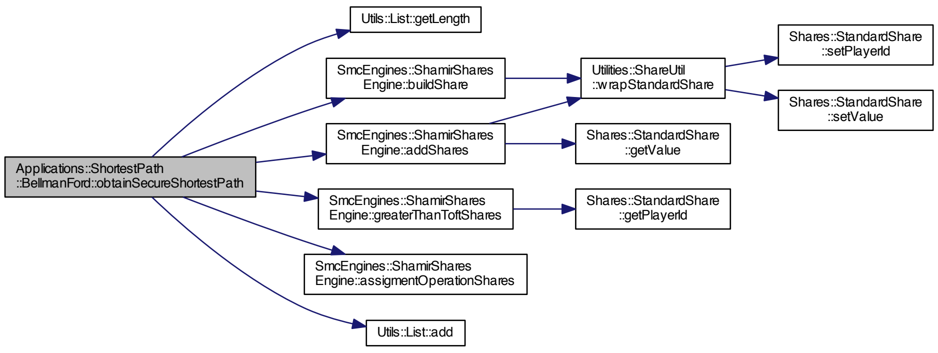

Utils::List::getLength
SmcEngines::ShamirShares Engine::buildShare
SmcEngines::ShamirShares Engine::addShares
Applications::ShortestPath ::BellmanFord::obtainSecureShortestPath
SmcEngines::ShamirShares Engine::greaterThanToftShares
SmcEngines::ShamirShares Engine::assigmentOperationShares
Utils::List::add
Utilities::ShareUtil ::wrapStandardShare
Shares::StandardShare ::getValue
Shares::StandardShare ::getPlayerId
Shares::StandardShare ::setPlayerId
Shares::StandardShare ::setValue

Figure 6.2.1: `SmcEngines:ShamirSharesEngine` Invocation Diagram

The structure corresponds to the Bellman Ford application and its main functionality method hosted by the `::ShortestPath:: BellmanFord` class. The engine then proceeds to call lower level functionality to continue the calling chain.

6.2.1 Architecture and Task Division

The MPC Toolkit was built with scalability, modularity and specialization in mind. Architecturally, the application has been divided in several layers. In accordance with the OOP paradigms, these layers fulfill a specific set of tasks that are divided by classes. Data moves along each of the layers in different forms accordingly. The process can be seen as a series of small factories capable to do some transformations on the data that later is fed to the next layer. These elements or layers can be reorganized at will, to provide with new forms of functionality, as long as their data inputs and outputs are respected.

The Toolkit uses this layer architecture as the basis of its 2 main components. The communicational support provided by the `Listener:: Shares Listener` class, and an MPC engine, in our case the `ShamirSharesEngine`. Both can be seen as stand-alone functionalities that are capable to exchange information. They can be understood as two different interfaces. One in charge of communicating the Toolkit with other computational parties, and the other with the user. Figure 6.2.2 shows the communicational relationship between the Toolkit and its environment.

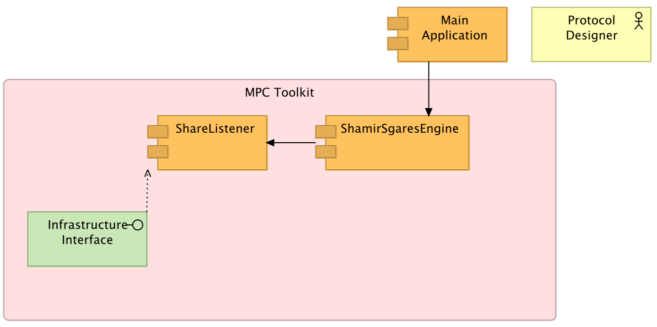


Figure 6.2.2: MPC Toolkit Component Architecture

An additional support section has been put in place as well, to provide services to both components. For instance, both use similar classes for data structures and data flow between different layers, the same is true for utility methods. Because of their different nature and the specialization level, they also have function specific layers. The Toolkit's most basic data unit is `std::string`. More complex objects are "serialized" into character streams and then exchanged in between the engine and the communications. Figure 6.2.3 shows this data movement, as well the different layers that are part of both components.

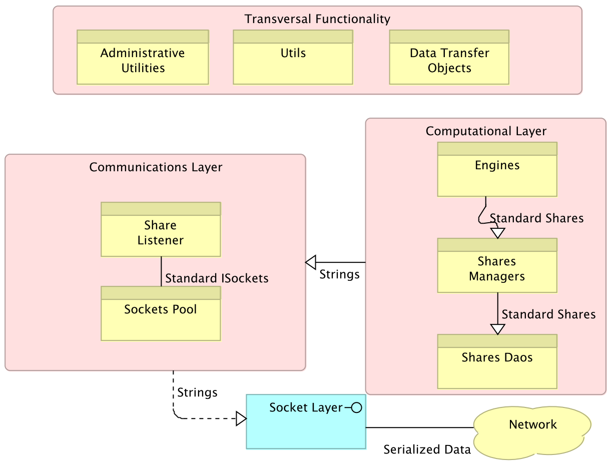


Figure 6.2.3: MPC Toolkit Logical Architecture

Amongst the transversal functionality, the Utilities fulfill a vital role. These are non-instantiated classes by definition, and provide basic and repetitive functionality. From basic arithmetic operations to data conversions. Tasks that are related to specific data types, but are static by definition, are also placed here.

Communicational Component. From the Computational Component's point of view, the Listener layer is in charge of sending the information needed to reconstruct the respective Share Objects to other players. From the side of the computational parties, with whom it is exchanging information, it is a socket transmitting information through a serialized data protocol e.g. UDP, TCP. The engine can ask to send some data, in this case it is delivered as `std::String` objects, following a previously established format. The Listener asks the sockets pool layer to provide a socket and later uses it to exchange information with the parties.

Computational Component. From the perspective of the algorithm designer, the engine is in charge of providing the basic functionality. But the engine is not in charge of the mathematical construction of the shares nor interacting with the Listener. Lower layers take up these responsibilities. From the perspective of the listener, it provides information to the Data Access Objects (DAOs) of the engine. They are in charge of the data transformation from strings to shares and vice-versa. From there, information is transmitted to the data managers to redirect the information in such a way that it can fulfill the functionality requested by the engine.

6.2.2 Information Movement and Operation Counter

The Toolkit has been conceived to work on a synchronous environment. Many things can be said about the advantages or disadvantages of such an approach. This communicational environment allows us to reduce tasks such as memory administration and simplifies the data flow process. It has to be noticed that whatever the approach is, the speed of the computation is going to be given by the slower element on the chain i.e. the slower computational party.

The scheme also raises a new challenge: all parties have to agree on an operation counter. Synchronous does not imply that all parties work at the same speed. At a given moment in time one of the parties could still be processing some previous computation while already receiving the result from concurrent parties. That is why we have added to the Shares an operation identification. This sort of operation counter is added to the share just before it is sent to the computational parties, and increment by one immediately after. The engine, in this case, is responsible to keep track of the counter, which is binded to the engine instance.

The fact that there is no global operation counter in the application, but individual counters per engine instance, facilitates, amongst other things, the coexistence of concurrent engine instances in host applications, but no parallelization inside the engine itself.

However, every operation that requires a communicational round would need to manipulate the counter. To minimize the risk of errors, and avoid code repetition, we have centralized this functionality. Given that communicational tasks have been transferred from the Engine towards lower layers, we have established a single point of communication with them. Any operation with shares that requires a communicational round, calls the `ShamirSharesEngine::shareValue` method. We have included the implementation of the method in the following snippet:

```
Utils::List<Shares::StandardShare> *
ShamirSharesEngine::shareValue(long value)
{

            ..
    this->generator_->generateShares
    (value, players_->getLength(),list);

    //it invokes it directly because saving RAM memory,
    //given that no modification is needed.
    Utils::List<Shares::StandardShare> * aux =
    this->shareManager_->transmitShares
    (list,  Buffers::EngineBuffers::operationCounter_+1);

    Buffers::EngineBuffers::operationCounter_++;

    Utilities::ShareUtil::destroyList(list);

    return aux;
};
```

First, shares are generated by a specialized class and the operation counter is added to the shares that are about to be transmitted. Then, the lower layers have to manipulate the data coming from the listener, and build all the corresponding shares from other computational parties. Once the listener and lower layers have sent and then collected the shares of the current operation counter, they are transmitted back to this method. Then, the operation counter is increased and temporal data destroyed. Finally, the method returns a list containing the shares transmitted towards him by all other computational parties.

Furthermore, to keep with task specialization, methods that serve the purpose of making secret information publicly available, manage a similar but separated process, where shares are not generated.

6.2.3 Communications Support

The communicational component of any MPC implementation is a key element in terms of efficiency. Previous works have relied heavily on third party libraries that provide several configuration alternatives. Our approach, instead, its based on simplicity. Our aim is to reduce the footprint of the processes related to communications, thus, improving performance.

We have taken 2 steps in this direction. The first was to build our own communicational structure without relying on third party platforms. We created the sockets, established the connection and then transmitted our formatted strings. This allows us to eliminate lengthy intermediate processes for data standardization and streaming. Secondly, we exploited the autonomy of the Listener and have placed it on an individual thread. That way the listener is capable of constantly hearing any incoming transmission from the other computational parties, and later process it, without interfering with other computational tasks. An additional advantage of such separation is that it allows us to monitor communicational time individually. The process only waits when shares from other parties have not still arrived.

The Toolkit can support any socket class that implements the library's interface `IStandardSocket`. Socket classes are specialized entities that manage the lower level communications. They specify buffer sizes and transmission protocols. Our class `Sockets:: UdpSocket` implements this interface and provides functionality to connect to other players using UDP as transmission protocol. This construction allows for different socket implementations. Our goal is to empower the algorithm designer with the ability to put in place its own connection mechanisms according to his needs.

Once all computational parties are running, their respective communicational supports, embodied by their Listener instances, will automatically start a synchronization process. They will verify whether all other players are online, and then synchronize their operation counters. After that, the Listeners will signal the end of this process and allow the computational component to start the information exchange and crypto-calculations.

6.2.4 Parametrization

In general, many behavioral characteristics, can be administered by the user through parametrization. Currently, the Toolkit provides the algorithm designer with the possibility to change basic administration settings through the use of a parameters class. Support is included to allow future versions the possibility to have an external properties file that is uploaded to the parametrization class at the beginning of the execution. This is a common practice for software applications of this nature. Parametrization classes upload these values as constants to avoid any modification in runtime. In our case the parametrization can be done by directly changing these parameters in the class `Utilities::Constants`. The following snippet shows an example of such parametrization:

```
    const long Constants::SYSTEM_P=9182237390125665823L
    const int Constants::SYSTEM_L=32;
```

This class is also used as a container of the identifiers of parameterizable functionality and, in general, system constants. This is for consistency throughout the application, and eases readability. For instance, when a functionality in a socket has failed, it uses the constant `Constants::TRANS_FAILURE`, instead of some raw value. If the value of the flag has to change for any reason, it suffices to change the value of this constant instead of directly modifying the code.

6.2.5 MPC Functionality

The toolkit offers a wide range of functionality and applications to the algorithm designer. They depend on a basic set of methods for the common primitives and some general tasks. Each engine implementation has to provide, at least, some implementation of such functionality. Our `SmcEngines::ShamirSharesEngine` is not the exception, table 6.1 shows some of the more important of these methods:

Primitive	**Algorithm**	**For Scalar**
Share Value	Shamir [13]	✓
Reconstruct Share	Interpolation	✓

Table 6.1: Basic MPC functionality provided by the ToolKit

Many of these functionalities are available by direct access through any instance of the engine, since, they have to be constantly used by the algorithm designer. That is the case of the share value method and the reconstruct share method. The first is used when the user wants to share a value using Shamir's sharing scheme. The second provides the functionality to open secret shared values. There are many other methods that provide simpler functionality, like the ability to send shares amongst players without reconstructing them.

6.3 Prototype Capabilities and Technical Characteristics

Third Party Libraries. The Toolkit limits the use of third party libraries. External function oriented components may provide many unnecessary functionalities, that in exchange influence the performance. The main exception is the use of the Number Theory Library (NTL) for modulo arithmetic. The

NTL library was introduced by Shoup in 2001 [84], and since then, it has been improved and used by many applications that need not only of modulo arithmetic but other topics in number theory. NTL can be compiled with the GNU Multiple Precision Arithmetic Library (GMP), a powerful arithmetic library to improve its efficiency. Our core arithmetical processes use the methods and data types provided by NTL (in conjunction with GMP) for the modulo arithmetic. Although we are satisfied with the results we have obtained from NTL in terms of efficiency and precision, the lack of some conversion methods and some type issues, limit some further planned bit size tolerance. For instance, this is the case of the support for unsigned long numbers that would allow us to expand the bit tolerance of the Toolkit to 64 bits.

Parallelism and Concurrent Execution. Current state of the art in several applications has been focused on the reduction of communicational rounds to exploit the advantages of parallelization. This topic has been examined and explored in detail by previous works with different visions. Several support, up to some level, concurrent executions while a number of other limit its scope or unnecessarily extends its use. While it is true that parallelization improves the performance of certain applications, this is not possible in many environments. This is why the goal of this work is not to promote parallelization, but to give the algorithm designer the tools to assess the impact and the distribution of the workload with atomic operations such that it could be used by real time applications. Given the strong object oriented architecture chosen for the Toolkit, parallelization is possible but is left as a responsibility to the developer/final user.

Data-types Handling. Secure shares have their own data type support. The basic share type is called `Shares::StandardShare`. It can be used to interact with the `SmcEngines:: ShamirSharesEngine`, and allows data flow from lower application layers towards the final user applications. It can be extended in the case that different sharing mechanisms with specific needs are implemented. It also provides a clone method that allows users to duplicate the share instance with the same values.

Open data are returned as variables of type `long`. No implicit casting is allowed between secret shared and public data. If the algorithm designer decides to manually create a share, it would have to be built by the method provided by the engine and not be manually instantiated. The `SmcEngines:: ShamirSharesEngine` provides a method to instantiate shares. When used, all players involved in the computation should provide the same input. The following snippet exemplifies this procedure:

```
Shares::StandardShare * zero = this->engine_->buildShare(0);
```

In this case, the parties had to previously agree on the raw value placed in the share. In this case the reconstruction of the zero share would yield 0. No computational round is needed for this procedure.

Memory Access. Memory access and array management with secret share indexes are left at the discretion of the user. Techniques and their applicability to solve the problem vary from different types of applications. Sometimes the user might want to use ORAM strategies e.g. [58] or to avoid completely array management e.g. [85]. Management of arrays with public index access can be conducted as usual on C++. To facilitate readability, we have constructed our own List data-type (`Utils::List`). This is a wrapper of the std::vector, and it includes some typical list functionality, e.g. remove, set. We provide several methods to safely destroy the object and its contents. The list type is also based on generics, which allows it to be used by the user as well as in other scenarios. Following the OOP paradigm, this list is conceived to store pointers only. Other containers i.e. `Utils::Matrix`, with the same characteristics are also provided.

Naming Standards. To improve readability, we have adopted simple naming standards. We have tried to maintain an intuitive approach on naming, such that a collaborative and open environment to further develop the Toolkit can later be put in place. The rules are simple: types including classes, structures and namespaces use camel case style naming and always start with a capital letter. A suffix (e.g. Engine, Manager) has to indicate its belonging to a specific classification. Notice `SmcEngines:: ShamirSharesEngine` for instance. It follows camel case style on both the namespace and the class type itself. The prefix Engine tells the user what kind of class it is. Interfaces however, should always start with an `I`. Adjectives are preferred to name interfaces, because they usually signal a capability to be implemented rather than a type or a functionality.

On functions and methods, it is preferred to use verbs to signal actions on naming. Moreover, they should use camel case and start with a lower case. Variables, depending on the context, should try to use camel case as well and start with a lower case. Constants, on the other hand, should be written in capital letters and use an underscore in case they have composed names. Finally, data members of classes should finish by underscore and have to be private in nature. Getters and Setters should be provided to grant access to them. Other C++ standards are also considered and encouraged. The general goal is to maintain uniformity, that way we also facilitate code maintenance tasks and bug detection.

Protocol Building. This Toolkit can be used by the algorithm designer as a standalone application, using the `main.cpp` file to develop his protocols or as a external library added to the users source code. Following [86], we introduce a step-by-step process on how to build a basic protocol using the MPC Toolkit.

1. Instantiate the players, they could be provided by any I/O mean, including a configuration file if desired:

```
int ip1[]= {192,168,1,1};
int ip2[]= {192,168,1,2};
int ip3[]= {192,168,1,3};
Players::StandardPlayer * p1=
        new Players::StandardPlayer(1,3000,ip1);
Players::StandardPlayer * p2=
        new Players::StandardPlayer(2,3001,ip2);
Players::StandardPlayer * p3=
        new Players::StandardPlayer(3,3002,ip3);
```

2. Instantiate the engine and configure it also indicating who is the local player:

```
SmcEngines::ShamirSharesEngine *engine =
        new SmcEngines::ShamirSharesEngine
        (list->get(player-1),list);
```

3. Proceed to share your secret value or values, you will in response receive the corresponding list of all shared values by other players in shared form. We remind you that the procedure being executed has to be the same for all players, that is how this list is obtained:

```
Utils::List<Shares::StandardShare> *shares;
//first value sharing
shares=engine->shareValue(secret);
```

4. Implement the chosen protocol, in our case, the product between all shared values:

```
prod= engine->multiply(shares->get(0), shares->get(1));
prod=engine->multiply(prod, shares->get(2));
```

5. Finally the result can be opened and used as desired:

```
Utils::List<Shares::StandardShare> * localOpenShares=
        new Utils::List<Shares::StandardShare>(1);
localOpenShares->add(prod);
```

```
response =
        engine->reconstructShares(localOpenShares);
std::cout<< "The␣Result␣is:␣" <<response[0]<<"\n";
```

6. If the secret value and the identification of the computational party are introduced by the initial arguments of the application, then all players have to access the directory where the executable is located and run the following command:

```
bellman-secure-55:toolkit aaly$ ./edgeRuntime 3 5
```

Here, the first number stands for the player id and the second the secret to be processed (in this case, multiplied by the other shares).

Program Flow and Control Sentences. As expected, user applications are allowed to use any control sentence on public available data. This is not the same for private shares. Although they can interact with any control sentence, they should not be used to control the program flow. No method for implicit transformation of private data is provided by the Toolkit. In other words, private data has to be explicitly open to direct the program flow. It has to be taken into account that our approach on the Toolkit is to treat private shares variables as pointers to the memory. Given that logical operations can be performed with pointers, this could be misleading. Take for example the following snippet:

```
Shares::StandardShare * c=  engine->multiply(a,b);
Shares::StandardShare * d=  engine->multiply(a,b);
if(c==d)
{
 std::cout<<"They␣are␣Equal"<<endl;
}
else
{
 std::cout<<"They␣are␣Different"<<endl;
}
```

In this case, what is being compared by the logic operation are the pointer addresses and not their intrinsic values. The output in this case would be:

```
"They␣are␣Different"
```

In case the algorithm designer desires to implement such an if, he must first open `c` and `d`. The same stands for any other control sentence.

6.4 Engine Functionality

Our efforts have been centered on providing a set of basic functionality tools on the engine such that the algorithm designer could take it as a starting point on building its own application. Moreover, in some cases we have implemented more than one protocol per functionality. This was necessary for 2 reasons: to keep up with the state of the art, and to provide alternative solutions with different security levels. This gives the algorithm designer a variety of flavors to choose from. Its worth to notice that typically in our benchmarks and computational tests, we have used the best performing methods implemented.

Moreover, many of these functionalities are designed to work not only on Share to Share operations but with scalars (of type `long`) whenever possible. To improve readability, we have overloaded the invocation methods of the functionality. Take for example the following snippet of code:

```
Shares::StandardShare * c=  engine->multiply(a,b);
```

The functionality used by this method depends solely on the types of a and b. The correct overload will take up the method call on either case (Share to Share or Share to Scalar).

Additionally, we have maintained standardized code patterns to access these functionalities. Non overload versions of each of these methods can also be invoked, for instance:

```
Shares::StandardShare * c=  engine->multiplyScalar(a,b);
```

In this case, the compiler will expect b to be an scalar. The same is true if the operation is Share to Share:

```
Shares::StandardShare * c=  engine->multiplyShare(a,b);
```

None of these methods is static by definition. This means they can only be accessed once the engine is instantiated. Additionally, they guarantee data integrity, which means its functionality does not alter the content of the object. As it might be expected from our object oriented approach, function parametrization are passed by pointers. This means they might be referenced by other variables at the same time. This is why it is important, to ensure that none of their content is manipulated during the execution of the functionality.

The user has to be aware of the fact that each of these operations create a new `StandardShare` instance. As mentioned before, memory has to be managed carefully in a C++ environment to avoid any possible memory leak. To help the algorithm designer with this task, we have included a memory safe invocation for some of the most used and most basic functionality. This invocation destroys one of the parameters of the function and returns a new instance with the result of the operation. This is helpful in self assigned operations, where the variable on the parameter is immediately assigned to the answer. The following is an example of such scheme:

```
Shares::StandardShare * c=  engine->multiply(a,b);
c=  engine->multiplyTo(c,b);
```

In this case c is the shared value of $a \cdot b$ and then immediately after is multiplied by b again. Without this formulation we would have lost track of the first assigned instance of c, which constitutes a memory leak. We avoid this scenario thanks to `multiplyTo` implicit functionality. It destroys c just before assigning it again. We can see `multiplyTo` code's in the following snippet:

```
    Shares::StandardShare *
    ShamirSharesEngine::multiplyTo
    ( Shares::StandardShare * a, Shares::StandardShare *b )
    {
        //any normalization here
        Shares::StandardShare *result =this->multiply(a, b);
        delete a;
        return result;
    };
```

Using a similar principle of that of encapsulation, our method wraps such functionality, avoiding a possible memory leak point and facilitating readability. We have provided also overloaded versions of the same method for when scalars are involved.

6.4.1 Arithmetic Operations

Table 6.2 shows the arithmetic functionality included in the toolkit. It also shows whether or not the functionality can be used between Shares and Scalars. When standard conventions for negative numbers are adopted i.e. dividing the field in half to differentiate positive from negative, special addition overloads can be used, namely subtraction. The overloads of the functionality include all the variants explained in the previous section.

Primitive	**Algorithm**	**Scalar**
Addition	No Round	✓
Multiplication	Gennaro et al [18]	✓
Conditional Assignment	Single Round	✓
Fan In Multiplication	Successive Multiplications	N/A
Mod	Catrina and Hoogh [27]	N/A
Power	Successive Multiplications	N/A

Table 6.2: Secure Arithmetic Operations on MPC ToolKit

We have also included some naive implementations of some functionality that might be used by some more complex applications either in the engine or by the algorithm designer. These are the Fan In Multiplications and Exponentiation. In both cases a share is multiplied by its corresponding value as many times as indicated by a public `long` number.

For the case of the conditional assignment. It replicates the C++ ternary operator $z = c?a : b$. Such functionality uses this typical construction: $[z] \leftarrow [b] + [c] \cdot ([a] - [b])$. The parameter $[c]$ is a binary flag that selects the value to be assigned to $[z]$.

Finally, the mod operation allows us to obtain the modulo of a secret share when divided by a power of 2. The exponent is publicly available, in this case the divisor is public. The method is later used by the the comparison method from the same authors.

6.4.2 Bitwise Operations

Table 6.3 contains a revision of the main methods provided for secure bitwise operations in the Toolkit:

Primitive	**Algorithm**	**Scalar**
Addition	Damgård et al [25]	✓
Xor	$[a] + [b] - 2 \cdot ([a] \cdot [b])$	✓
Comparison	Damgård et al [25], Adp.	✓
Random Bit	Damgård et al [25]	✓
Bitwise Random Number	Damgård et al [25]	N/A
Bit Decomposition	Damgård et al [25]	N/A

Table 6.3: Secure Bitwise Operations on MPC ToolKit

We have included a variety of bit related functionality. Some applications like comparisons are dependent on bitwise operations, such as bit decomposition of shares and random numbers. Many others might need to use, for instance, the bit generation sub-routine, when the toss of a coin is needed.

On the random bit generation, we have also included an additional naive implementation of such a method, where each player generates a random bit and then they are `xor` against each other. This methodology, under our configuration, employs the same number of communicational rounds than its Damgård counterpart. That would no longer be the case with either a PRSS [87] implementation or when more than 3 players are involved.

On the bit addition and bit decomposition, we have implemented subroutines to help with this process, namely the carry bit (secure) amongst others. These methods are also available at any engine instance. Furthermore, there are other different mechanisms that can be implemented to securely obtain the carry bits for an addition computation. A support scheme for more methods has been put in place as well. Using the parameterization principles of the Toolkit, the addition of shares can identify the carry mechanism the algorithm designer has selected. The following snippet provides a call example of the method with such parameterization:

```
*bitwiseNumber=this->bitwiseAdditionShares
(*bitwiseNumber, bitShares->get(i),
Utilities::Constants::SHAREMIND_CARRY);
```

The MPC Toolkit provides by default a `xor` gates chain to generate the carry vector. In case the algorithm designer would want to add additional flavors to such computation, it would suffice to add a new type on the parameters class (`Utilities::Constants`) and add its invocation to the bitwise addition method, linked to the corresponding parameter.

6.4.3 Logical Operations

Logical applications in several flavors have been incorporated into the Toolkit as well. The table 6.4, showcase the main implementations we report on:

Primitive	**Algorithm**	**Scalar**
Equality Test / Zero Test	Catrina and Hoogh [27]	✓
Equality Test / Zero Test	Limpaa and Toft [26]	✓
Inequality Test	Damgård et al [25]	✓
Inequality Test	Catrina and Hoogh [27]	✓
Inequality Test	Limpaa and Toft [26]	✓

Table 6.4: Secure Logic Operations on MPC ToolKit

These applications are all available on the engine. Moreover, we have included several flavors with different security approaches and performance. We provide 2 logic tests: zero test (`EQZ:`$[a] == 0$) and less than zero test (`LTZ:`$[a] < 0$). Equality tests and inequality tests can be built from these basic two tools without any additional computational round. Trade-offs on performance and security allows us three basic flavors on comparisons. We have expanded the inequality functionality of the application by providing methods to access all inequality tests using Catrina and Hoogh method.

Furthermore, table 6.5 shows how Catrina and Hoogh use the `LTZ` test to compute other inequalities and the inequality tests provided by the Toolkit. We use the same principle with Catrina and Hoogh inequality test.

Primitive	**Algorithm**	**Scalar**
$[a] < [b]$	$\texttt{LTZ}(a-b)$	✓
$[a] > [b]$	$\texttt{LTZ}(b-a)$	✓
$[a] \leq [b]$	$1-\texttt{LTZ}(b-a)$	✓
$[a] \geq [b]$	$1-\texttt{LTZ}(a-b)$	✓

Table 6.5: Secure Catrina and Hoogh Comparison Formulations on MPC ToolKit

It has to be noticed that the Damgård inequality test, provides perfect security something that Catrina and Hoogh's method does not. The latter provides statistical security that can be adjusted to the discretion of the algorithm designer. On the other hand, Damgård's inequality test is relatively slow in comparison, because of the bit decomposition involved in the process. This is the main motivation to use Catrina and Hoogh's method to provide this extra functionality. In their case, the security parameter k and bit size of the input l would dictate the terms of the statistical security. Default values in our application are $k = 29$ and $l = 32$. They can vary across the 63 bits available for them. A secure implementation of Limpaa and Toft method with statistical security has been provided as well. This method, to the best of our knowledge, is the latest approach to compute the inequality test. Both equality tests provide statistical security and are regulated by the same parameters.

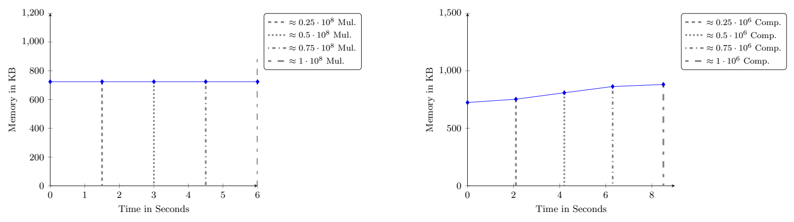
6.5 Computational Experimentation and Network Benchmarking

We provide the results of our computational experimentation with the MPC Toolkit. We report on, stress, performance tests and network benchmarking. We also have included experimentation with user program implementations i.e. the secure Bellman-Ford formulation introduced in previous chapters. Our tests are aimed to evaluate the behavior of the Toolkit over aspects like memory usage and CPU Time, over extended periods of time. We analyze the results and present some observations based on our experimentation. We run our tests on an Intel Xeon CPUs X5550 (2.67GHz) workstation with 42GB of memory, with Mac OS X 10.7. All our tests used a single engine running on standalone applications with 2 threads assigned to each computational party and unlimited access to the RAM. All parties ran their applications on the same machine (except our latency tests for network benchmarking) and the same amount of resources was available for them at any given moment in time.

Basic algorithmic elements e.g. multiplications and comparisons, need communicational rounds for their execution. Typically, the cost of other primitives like addition, are consider to be "free" given that no information exchange is needed to perform them. This is a clear indicative that the overall performance of any MPC application relies heavily in the cost of executing a communicational process or round. Complex applications e.g. comparisons are made entirely of a mixture of this basic "costless" primitives with elements in need of at least one communicational round e.g. multiplications or to open shares. Typically, because of complexity and implementation issues a comparison needs more computational time than a multiplication.

Our goal is to analyze what is the behavior of these two primitives in our MPC Toolkit. Not only determine how fast these primitives can be executed, but how parameters like memory consumption change with time. Taking aside the fact that RAM is not an infinite resource, it is clear that bad memory management has a high impact on performance as well.

We have executed large batches of these operations (10^8 multiplications and 10^6 comparisons) and compiled the results in Figure 6.5.1. We have selected our implementation of Gennaro et al. [18] for multiplications and Catrina and Hoogh [27] comparison method for our experimentation.



(a) Multiplications ($\approx 10^8$) (b) Comparisons ($\approx 10^6$)

Figure 6.5.1: MPC Toolkit Multiplication and Comparison Life Cycle

The experimentation on multiplications is interesting because it only needs a single computational round. Besides that, it uses little algorithmic elements that we can consider negligible in terms of performance. Moreover, it is an essential building block for more complex applications and it is continuously used in the comparisons primitives we have implemented. Comparisons, are more complex protocols, that need several intermediate steps to be built over the multiplication, addition and sharing primitives. Comparisons are also essential for more complex constructions and user programs.

The results showed that our engine is capable to execute ≈ 16500 multiplications per second. The average RAM consumption never exceeds a Megabyte. Moreover, it can be seen that memory consumption was stable thought our tests and the velocity on which multiplications were being solved was constant throughout the experiment. On comparisons, the engine solved 117 per second, in other words a comparison each 8.4 milliseconds. To put that in contrast a comparison under the same configuration and environment took around 9 times more (70 milliseconds) in VIFF. This is also true for multiplications where our results were 9 to 10 times faster. RAM memory consumption when the comparisons were tested registered a little but monotonic growth, smaller in average than 200 Kilobytes at the end of the test batch. This can be explained by the following: memory fragmentation. Because of the Object Oriented Architecture we adopted, thousands of objects are allocated and then destroyed by the application each second, in the case of multiplications, these objects are in large majority of the same size. With comparisons, however, that is no longer the case, several different sized objects are needed during the computations, specially data containers of different sizes. This causes a slow but steady memory fragmentation in smaller and "un-allocable" memory spaces. Hence, this small growth in memory consumption after 8540 seconds that took for the experiment to be completed (a bit more than 2 hours). Other possible causes for this increase include a sufficiently small memory leak for the increase that has not yet been detected.

Network Latency. We have performed additional tests to measure the impact of latency in Local Area Networks (LAN) using Ethernet and Wireless connections. Our benchmarking included, in this case, atomic operations i.e. multiplications and comparisons, that are the basic building blocks of the Toolkit that require exhaustive information exchange. We carry out this experiment using two standard computers with different processing capabilities running the same operative system i.e. OS X 10.10.3. A machine equipped with a processor with 2 cores and 2 GB in RAM memory (machine `A`) and another machine with 4 cores and 8 GB in RAM (machine `B`). As expected running times are faster on the latter. Finally, to determine the impact of the latency, we first execute each operation batch with two client process (two parties) in our 2 processor machine, later, we execute the same batch having one process (one party) in our 2 core machine. That way we can measure any increment on running times caused by the newly added communicational costs, while the other parties run in our 4 core machine. Note that the speed of the system is given by the slowest processing party plus the latency of each connexion. Our experimentation on Ethernet included Gigabit connections through a smart switch. Additionally, some of our tests where carried out on Wireless networks using standard 802.11 routers.
Multiplications. Several operations batches comprising 20000 multiplications were executed using the aforementioned configuration (Ethernet connections). We have tabulated some of the results on table 6.6.

	Mach. `A` - 2 Parties		**Mach. `A` - 1 Party**		**Margin**
Mult.	CPU Time	Com. Time	CPU Time	Com. Time	
2000	35.52	35.38	37.83	37.5	6.00%
2000	29.81	29.61	40.15	39.79	34%
2000	30.25	30.02	38.92	38.59	26.56%
2000	30.53	30.28	36.56	36.267	19.74%
2000	30.21	29.96	39.2	38.86	29.68%

Table 6.6: CPU Time in Seconds from Multiplications over LAN Ethernet

Moreover, tests were also carried out with Wireless connections. The experimentation on the datasets shown in these case, an increment in time of $\approx 27.4\%$ in average for each new Wireless connection.

Comparisons. Following our experimentation with multiplications, we executed batches of 100 comparisons each, and measured computational times using the same configuration. Our results are shown in table 6.7:

	Mach.A - 2 Parties		**Mach. B - 1 Party**		**Margin**
Mult.	CPU Time	Com. Time	CPU Time	Com. Time	
100	21.86	21.43	32.17	31.72	32.46%
100	21.32	20.95	26.04	25.63	18.28%
100	19.35	18.86	22.32	21.95	14.05%
100	20.35	19.97	23.6	23.236	14.56%
100	20.18	19.79	23.045	22.66	12.65%

Table 6.7: CPU Time in Seconds from Comparisons over LAN Ethernet

Our Experimentation using Wireless connections shown an increment of around $\approx 32.4\%$ computational time per each new connection. Note that because of the transmission protocol selection i.e. UDP, package loss is frequent on Wireless connections, which in this case was a constant challenge during the different tests.

From these benchmarking we can conclude the following:

- Given that computational time, is highly dependent on the speed on communication transmission, as the data suggests, the results shown considerable variations between different test batches.

- Given the high susceptibility to latency on the network, the increase in CPU time of adding additional Ethernet link is around 30%.

- A possible configuration to avoid the added administration and transmission costs of LAN connections is to use machines with exclusive transmission channels between parties i.e. individual network cards connected through optic fiber. It is our believe that these kind of dedicated connections would provide similar results than our tests in a single machine i.e 16500 multiplications per second. Sadly because of the cost of such infrastructure we were not able to test such set-up.

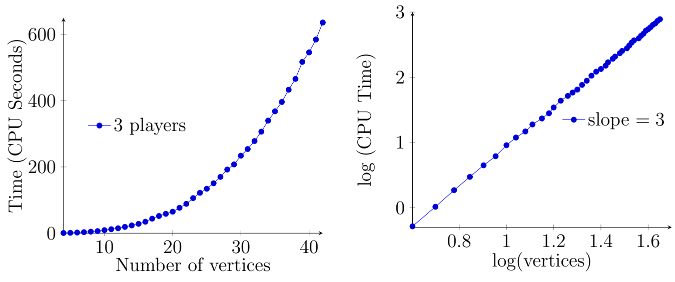
- Additionally, we have experimented in this specific setting with the Bellman-Ford MPC algorithm introduced in previous chapters. The results show that it took 14.18 seconds to solve a 4 vertices graph with one Ethernet LAN connection, and 15.48 seconds with 2 connections. These results are similar to the times obtained by previous latency experimentation.

Bellman-Ford We provide experimental results on user programs, specifically the secure Bellman-Ford protocol by Aly et al. [51] introduced in previous chapters. The algorithm is an iterative construction of complexity $\mathcal{O}(|V|^3)$ that makes use of a comparison and multiplications to find the shortest path of a complete graph of size $|V|$. We have omitted the source code for facilitate readability. Instead, we report on full results of our experimentation including running times for relatively large graph instances. Table 6.8 shows the results on different graph instances.

Vertices	CPU Time (Toolkit)	CPU Time (VIFF)	Communicational Rounds
4	0.51856 Sec.	3.5 Sec.	7815
64	2240.67 Sec.	≈ 8 Hours	335 46 495
128	17823.5 Sec.	N/A	301 965 949
256	≈ 40 Hours	N/A	≈ 2181 038 080

Table 6.8: Secure Logic Operations on MPC ToolKit

Note that we extrapolate the values that correspond to the graph of 256 vertices. A similar experiment was carried out by Aly et al. [51], on $VIFF$ and reported in this thesis as well. Under a similar environment i.e. same machine and security model, our solution outer perform VIFF's by a 8:1 ratio in small instances (some tens of vertices). This is not true for bigger instances where the differences widen. A clear example is the 64 vertices graph instance. With our MPC Toolkit the problem was solved in a bit more than 37 minutes, meanwhile, the same instance, with VIFF took more than 8 hours to be solved (1 : 16 ratio). On the other hand the 4 vertices graph instance was solved by VIFF in 3.5 seconds versus the 0.5 seconds for the MPC Toolkit (1 : 8 ratio). This variation on behavior is highly tied to a memory consumption issue. As mentioned before, because of VIFF issues with big data instances, memory indexation starts to affect its running times at the longterm. This is also visible with the 64 vertices graph experiments. At the end of the life-cycle of both, the MPC Toolkit used less then 1 Megabyte while its VIFF counterpart already surpassed half a Gigabyte of use. Figure 6.5.2 shows the results obtained by our MPC Toolkit with different graph sizes.



(a) CPU Time (b) Logarithmic Scale

Figure 6.5.2: MPC Toolkit Bellman-Ford Test CPU Times

From the results we make the following comments:

- Memory does not limit the performance of the MPC Toolkit. Moreover, our trials showed that memory increases slightly (probably because of fragmentation issues) during long runs on specialized primitives i.e. comparisons, and the increment never surpassed the $200KB$ after a million of comparisons. On basic operations, however, no such increase was detected.

- There is a clear difference on performance in comparisons with other open source frameworks like VIFF. The cost of a communicational rounds have been improved by the use of a non-interpreted language. Further improvements could be obtained by the use of lower level languages i.e. Assembly.

- A single MPC Toolkit engine can solve in a fraction of a millisecond (0.6 mill.) a communicational round with limited CPU capacity. Although our testing environment provided us with better computational power and RAM memory than a typical personal computer, it is also true that it is possible to find better CPU benchmarks on commercial computers. This is also true for server lines and cloud computing. The Toolkit could benefit from this extra CPU power and produce better results. In terms of RAM memory, however, given the engine's low consumption, its has little impact whatever the environment.

- The "free" and "costless" operations like secure addition and other algorithmic processes actually present a cost although small to the overall performance of the application. This can be inferred from our experimentation: In average the MPC Toolkit was able to solve 16500 communicational rounds when little processes were involved i.e. multiplications. But when we did the same experiment with comparisons this number was reduced to 15200 rounds per second. That is because comparisons have more complex algorithmic processes involved between communication rounds.

6.6 Road Map and Future Work

A collaborative scheme, and open source licensing could open the possibility of incremental functionality iterations. A natural next step includes support for active players, as well as improvements on random number generation through the use of PRSS [87]. We have compiled the following list with general goals for the application road map and future work:

- Incorporate PRSS [87] to the basic functionality of the engine. This would reduce the communicational rounds needed to generate random numbers, as well as other advantages concerning share instantiation.

- Introduce support for VSS [14] protocols. This would allow to pave the way to have support for security against active adversaries under the private channel model. When incorporated, an additional engine that supports on VSS will have to be built, and basic arithmetic primitives recreated. Shared functionality, like comparisons could be re-utilized.

- Include exception/error control in the functionality. We have seen that in complex applications a clear guide on where and why errors occurred can facilitate the work of the algorithm designer. This can quickly accelerate development times, and ease the bug detection.

- At the moment the application has been tested with 3 players. Changes for bigger sets of players does not introduce structural changes on the application and its architecture.

- Augment the bit-size support of the application. Although 63 bit size numbers are sufficient for many applications this is not true fro all. An expansion on the bit-size tolerance, would augment the toolkit scope and reach. This would forcibly mean to abandon NTL as the mathematical core of the application and adopt stable (maybe commercial) ring arithmetic implementations that can manage efficiently bigger data-types. The application should allow the algorithm designer to choose the arithmetic core to use.

- Introduce parametrization by properties file. The Toolkit would be able to upload the application parameterization using an external file in coordination with other players.

- Provide a variety of default socket alternatives. Hence, the socket pool support has to be improved to give support to classified sockets.

Chapter 7

Practically Efficient Secure Auctions with Transmission Constraints

7.1 Introduction

Auctions have a deep history in our economic system. They have long been regarded as an intelligent allocation mechanism. In recent years, with the advent of larger scale markets e.g. online commerce and commodities, factors like secrecy, integrity and fairness have grown to be an essential need of the process. Parties need adequate incentives to bid truthfully, without the risk of loosing competitive advantages in future interactions. This problem arises in many market settings where a variety of auctions mechanisms have been proposed. This is the case of the problem at hand, which was inspired by day-ahead electricity markets. Our aim is to solve the problem where a commodity is to be transported between the different markets up to a given capacity limit. In our setting, buyers, sellers, markets and control agencies may have to interact in a competitive environment where information about prices and volume can reveal much more of what any party is willing to disclose. Simple procedures like sealed-bids have evolved to provide a higher level of confidence to all the players involved in the process. But with the rise in consumption and production, the need for better privacy-preserving auction protocols has become more relevant.

Traditional solutions include a neutral third party in charge of all computations and responsible to exert secrecy, integrity an fairness in his own processes. However, such a third party is in general hard to find, and would concentrate all attacks making it especially vulnerable, a single failure point.

We report on a mechanism where this third party can be replaced. We seek to maintain the basic properties from previous auction methods and change the centralized confidence scheme for a mechanism that makes all players part of this process. Having in mind practicality, we suggest adaptations and novel techniques to improve performance.

Our *virtual third party* uses Secure Multiparty Computation (MPC) and can be composed by any subset of players. MPC is a secure mechanism that allows several players to compute a function in a distributed environment. From Yao's original result in 1982 [1], to the current state of the art, secure multiparty computation has evolved from a theoretical object of study, to a field that is used in real life applications. MPC offers a variety of techniques, primitives and applications that provide security under diverse models, and in a distributed environment.

Our aim is to merge these two horizons, not only providing some algorithmic tools, but the analytical means to improve their performance. This is why our prototyping implements the necessary MPC primitives using conventional methodologies i.e. Object Oriented Programming (OOP), with a 3rd generation language.

The content of this Chapter is based on the following:

2015 Practically Efficient Secure Auctions with Transmission Constraints, (Abdelrahaman Aly, Mathieu Van Vyve).

7.1.1 Our Contribution

We introduce a novel greedy algorithm and its secure formulation, for auctions with several geographical markets where exchange between them is possible. We present a series of mechanisms and some adaptations needed by our protocols. Later, we analyze and introduce variations and trade-offs of these building blocks to obtain efficient times, addressing the privacy-preserving protocol implementation and its security and performance constraints. Additionally, we report on the computational experimentation on electricity markets, running our secure algorithm against real life data. To the best of our knowledge, this is the first time the problem of secure auctions with transmission constraints, has been addressed in detail. We focus our attention in the following:

Algorithm Design. We provide a polynomial time algorithm for the problem at hand and its respective adaptation for an MPC environment. We introduce proofs of correctness and security analysis. Given that in real life cases changes in the number of markets is less common than changes in the amount of bids, our efforts were centered on maintaining complexity growth linear in the number of bids. The result is a simple algorithm capable of managing bigger sets of bids with little effect on its performance.

Prototype. Our prototyping introduces the advantages of a modular application

and allow us to present an analysis and drive conclusions about the workload's distribution of the application throughout its lifetime. We discuss bottle-necks and possible improvements. Although perfectible, it is a tool from which the algorithm designer can devise further improvements based on the application at hand.
Complexity and Correctness. As similar works in the field, we use communicational rounds (exchange of messages between parties involved in the computation) as a complexity measurement unit in our secure protocol. It has to be noticed that our work centers over the possibility of practical use, which is why our work minimizes the use of comparisons. This is because although bounds in some cases are theoretically close, multiplications and comparisons have rather different performance outputs. This is explained in detail in further sections. Even if asymptotic bounds are maintained, little changes on the number of comparisons and multiplication can have an important impact on overall performance of any prototype. We also offer a correctness analysis of our algorithm, and adaptations associated to the building blocks we use in our secure protocol.
Alternative Formulations. This problem can find a general solution on the minimum cost flow problem. Known secure solutions of this problem are expensive in terms of performance as shown in Chapter 4. It would also require an elaborated construction where vertices represent bids. Given that the complexity function from Chapter 4 grows in the number of vertices of the graph. A relatively large number of bids would cause an explosion in terms of performance (polynomial growth). As mentioned, we propose a specialized greedy algorithm and its secure counterpart that, in our case, have linear growth in the number of bids.

7.1.2 Related Work

Secure Auctions have been studied from different perspectives, both in terms of security and configuration. In all cases questions on topics like performance, fairness and integrity have been raised. In this section we cover some of the works with similar characteristics and explore their differences with our contributions.

Auctions with Secure Multiparty Computation. Work on the field of efficient real life auctions with secure multiparty computation, comes from Bogetoft et al. [31]. A descriptive work on a real life case with MPC. In their setting, Danisco, the only sugar beet processor of the danish market, and several thousand farmers settled clearance market prices in a secret and distributed fashion using MPC. Their case, which takes into account a single market, provides a secure protocol for such a problem, without transmission constraints. In this paper, we explore a different setting, where several markets can exchange commodities up to an operative limit. This case cannot find a solution with the results introduced by [31]. Moreover, our setting is more realistic for other types of markets e.g. European electricity markets. Additionally, they built their protocols using VIFF [2], which proved to be reliable for the size of their problem. In our case, however, previous experimentation on equivalent problem formulations (Chapter 4), suggest that this is not true in our case. This is the reason why our prototype does not makes use of existing MPC frameworks e.g. VIFF. We explore the behavior of a dedicated application, using the flexibility of C++ and OOP. That way we are capable to use a compact set of secure MPC primitives that provide security and are reasonably fast in more realistic scenarios, as shown in our computational experimentation and in Chapter 6.

Secure Auction Mechanisms with Secret Sharing. Several authors have studied the properties of secure auctions with secret sharing e.g. [88, 89, 90, 91]. These works explore several different auction mechanisms in various environments. Recently, Nojoumian and Stinson [92], introduced algorithms for second-price and combinatorial auctions. Their protocols offer security against active and passive adversaries, using amongst others, Shamir secret sharing [13] and a verifiable secret sharing schemes (VSS). They model their auction problems as graphs, and device theoretically efficient algorithms. It has to be noticed that no experimentation is reported.

Secure Second Price Auctions. Work has been done on cryptographic alternatives to guarantee security in second price auctions. Recent work, introduced by Catane and Herzerg [93] presents some answers for secure second price auctions. They provide a requirement framework for second price auctions and introduce an auction scheme that makes use of some known cryptographic principles. They achieve this by trusting computations to a supervisor entity and using randomization. Their goal however is to keep the bids secret from other players. Our decentralized approach allows us to get rid of the supervisor as a central entity on the security scheme. Given that a supervisory entity can be enforced by the environment, our protocol allows the participation of a control entity for instance in the process. In our setting, when a supervisory entity is involved in the process, its input would be necessary, but not sufficient to actually leak any information to such entity. Our privacy-preserving protocol provides security and fairness without relying on any third party. Finally, their approach does not take into account the transmission exchanges that are essential for our model. Similar to [31], this solution would work for one market but needs to be adapted for a multi-market scenario.

7.1.3 Overview

The paper is organized as follows: Section 7.2 introduces the problem definitions. Section 7.3 presents a different network flow formulation of the problem and introduces our polynomial algorithm and its correctness proof. In Section 7.4, we describe the security constraints, building blocks and technical tools for later use in our secure protocol. We also explore the possible trade-offs of these mechanisms in terms of security and performance. Section 7.5, makes use of previously presented techniques to securely solve the problem. In this section we provide an analysis on complexity, security and correctness. Experimentation and prototyping are described in Section 7.6. We also explore various workload distributions for realistic applications.

7.2 Problem Overview

We first proceed by introducing the problem definitions i.e. the design of the auction, goals and notation.

7.2.1 Auction Mechanism

The treatment of our secure auction is as follows: a reverse auction scheme with several sellers or bidders. Markets or auctioneers adjudicate orders to supply and demand bids that maximize social welfare. A control agency may be part of the process, to supervise and guarantee the integrity of the result. The security follows from the use of Secure Multiparty Computation with no leakage for information theoretic security. We provide a security analysis.

Furthermore, a secure implementation of the auction would be aimed at protecting the interests of all players involved.

Individual interests and involvement level are the following:

Markets and Transmission Network: The set of markets and the capacity of the transmission network are assumed to be public. The transmission network is represented by a capacitated network flow, i.e. pairs of markets are binded by bidirectional transmission lines. Thus giving an upper limit to the flow i.e. capacity. Notice that in this case, markets are geographically separated.

Sellers/Buyers or Bidders: The set of players interested in acquiring or selling auctioned bids i.e. negative or positive quantity. Each Bidder can submit non-related bids to several markets. Bids are composed by a certain quantity Q and a price P. All bids are enclosed and final i.e. no re-bidding is allowed. The bid placed by the player can be partially or totally adjudicated to the Bidder depending on what maximizes social welfare. One of their interests is the secrecy of the information contained on each bid towards any other player e.g. other bidders and markets, for as long as the auction takes place. Their concerns are also correctness (the result of the auction is correct) and fairness (all players receive the same information at the same time).

Automated Auctioneer: Is the proxy entity in charge of managing the auction. Our work proposes that the role of the auctioneer is to be taken by the computational parties representing markets, bidders and control agencies, in a distributed and secure fashion. A virtual ideal functionality capable of determining the set of accepted supply bids and rejected demands guaranteeing correctness, without disclosing nor operating over secure data.

Control Agency: Is a regulatory entity or any institution trusted by the Markets and Bidders alike. By the parties choosing, or environmental enforcement, it participates to add confidence to the process. Because of the nature of MPC, our secure protocol allows active participation of the Control Agency as a computational party, their presence would be necessary for the correct and secure operation of the protocols in conjunction with the model. It has to be noticed that in some configurations its presence might not be required.

On Computational parties. It is possible to have as many computational parties as considered necessary by the algorithm designer to guaranty security and bring confidence to the process. Although many of the building blocks require a minimum of three parties, the algorithm itself can be adapted to be used with two-party computation. As estated many auctions require the presence of an external supervisor. A basic configuration would include a computational party representing the bidders, another the markets, and a third one for the supervisor or control agency. If a larger number of computational parties are considered, the trade - off in this case is performance.

7.2.2 Problem Definition

Similar to [94, 95], suppliers and consumers first submit bids that are binded to a specific market i.e. (individual day-ahead markets). All bids are composed of a quantity and a unique associated price. The goal is to determine which bids to accept to maximize social welfare. This has to be done in realistic times (if possible) with the sufficient capacity that satisfies the accepted demand bids and external demand coming from adjacent markets. As previously mentioned, markets are interconnected by a transmission network with capacitated transmission lines with zero cost.

We keep the information contained in all bids secret from other players until the end of the auction process. Additionally, we seek to eliminate the need of any trusted auctioneer (Typically, a single entity with access to the secret information included in the bids and who decides the outcome of the bidding e.g. power exchange in the European market) carrying the process. In this case, the responsibility of the execution of the auction itself is moved unto the control of the players (including the control agency) involved in the process. Detailed treatment on working solutions for these markets can be found in [96, 97].

Input Data. The problem considers all information contained in bids to be private. This includes the prices, production quotas, and the markets to which they are linked. Much like in real life, the topology and the capacities of the transmission network are assumed to be public. Data is formulated as integer values over a finite field $\mathbb{Z}_q$ where input values are much smaller than q such that no overflow occurs. Its size is tied to the application in hand. Bids can be partially adjudicated as well. Note that when they are secretly shared we can not differentiate between a demand bid and a supply bid.

7.2.3 Problem Formulation

Consider the set of all markets M, the set L containing all transmission lines between markets in M. D is the set of all the Demand Bids where $d_m \forall m \in M$ is the set of demand bids of market m. K is the set of all Supply Bids where $k_m \forall m \in M$ is the set of supply bid of market m. Parameters P_i, Q_i correspond to the price and volume of the bids ($Q_i < 0$ for supply bids and $Q_i > 0$ for demand bids) $\forall i \in K \cup D$. The variable $q_i \forall i \in K \cup D$ is the accepted quantity of bid i. Variables $f_{v,w}$ stands for the flow through the line $(v, w) \in L$, with capacity $C_{v,w}$. The problem is defined as follows:

$$\max \sum_{i \in D} P_i q_i - \sum_{i \in K} P_i q_i \tag{7.1}$$

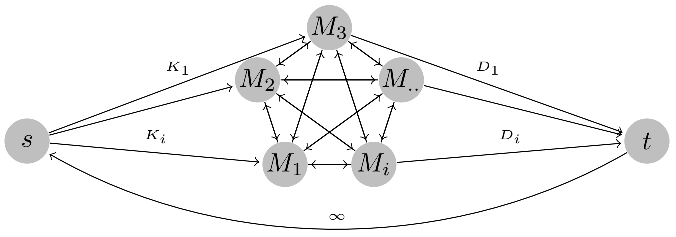
$$\text{s.t.} \sum_{i \in k_m} q_i + \sum_{j:(j,m) \in L} f_{j,m} = \sum_{i \in d_m} q_i + \sum_{j:(m,j) \in L} f_{m,j} \quad \forall m \in M \tag{7.2}$$

$$0 \leq f_{v,w} \leq C_{v,w} \quad \forall (v, w) \in L \tag{7.3}$$

$$0 \leq q_i \leq |Q_i| \quad \forall i \in K \cup D \tag{7.4}$$

7.3 Network Flow Formulation

The problem (7.1) - (7.4) can be modeled as a minimum cost capacitated network flow problem on graph $G = (V, A)$ as shown by figure 7.3:



Using this kind of characterization, we model the problem as follows: consider that $V = M \cup \{s, t\}$ where s and t are artificial source and node vertices. For each bid (supply and demand) $i \in K \cup D$, there is an arc s, m where m is the market of the bid i, with capacity $|Q_i|$ and cost P_i. Set S contains all edges exiting s. For each pair of markets (m, m') there is an edge with capacity $C_{m,m'}$ associated to the capacity of the corresponding line $L_{m,m'}$ and no cost. Moreover, for each market m, edges that represent its demands can be replaced by an edge (m, t) with capacity $T_m = \sum_{i \in d_m} Q_i$ and zero cost. For simplicity, let us consider B to be the set of all bids $K \cup D$.

7.3.1 Greedy Algorithm

As it was previously noted, we can identify the sets of accepted and rejected bids that would satisfy global demand by solving this Minimum Cost Flow (MCF) problem. Secure protocols to solve the MCF problem have been studied in Chapter 4. Their results provide polynomial computational bounds. Although, the protocol is theoretically efficient, in practice, its applicability seems to be limited not only by the high degree of the polynomial from the complexity function, but also for the MPC framework used for its testing. For instance, the experimentation shows that with their privacy-preserving algorithm running over VIFF [2], it would take around a year to solve the problem with perfect security in a 10 vertex complete graph. On this setting, each bid has to originate from an individual source vertex (instead of a unique source) connected to his respective market through an edge of the capacity of the bid. In a realistic settings, like the ones we report on our experimentation, with around ≈ 1500 bids in average, that would compromise its overall performance.
A more practical approach is taken into account. An algorithm, greedy in nature, that uses the less expensive bids from set B first, and allocates sufficient flow to satisfy all demand and obtain an optimum, one edge at the time. We achieve this, thanks to a series of iterative sorting and max flow sub-routines. The following algorithm shows its construction:

```
1. $\nu \leftarrow 0$
2. $B \leftarrow$ sort-price:B
3. $c_i \leftarrow 0 \quad \forall i \in S$
4. for all:   $i \in B$ :
5.       $c_i \leftarrow |Q_i|$
6.       $\nu' \leftarrow$ maxflow:  G(V,A)
7.       $c_i \leftarrow \nu' - \nu$
8.       $\nu = \nu'$
9. End
```

Algorithm: 4: Iterative Greedy Algorithm for Auctions with Transmission Constraints

First, we sort the set of all bids B in function of their price and set the capacities of edges in S to 0. Second, we restore the capacity of the edge associated to bid i to its original value $|Q_i|$ and calculate then max-flow on G. We then set the capacity of such edge to the flow variation with respect to the max-flow calculated in the previous iteration. We repeat this process for all bids in the order of prices. Once this process is completed, the volume provided by demand bids is then automatically rejected and accepted for the supply bids.

7.3.2 Correctness

We now prove that the greedy algorithm described above is correct. To do this let us disaggregate each bid as a collection of bids of capacity 1 e.g. 1 Mw each and with identical prices. This obviously does not modify the problem. For a given set of bids I, let $r(I)$ be the maximum amount of demand that can be satisfied using the bids of I only. This can be seen as a maximum flow problem on the graph G.

Proposition 1. *The set function $r : 2^S \to \mathbb{R}^+$ is the rank function of a matroid.*

Proof We use a characterisation of Whitney [40] for a function to be the rank function of a matroid:
(a) $r(\emptyset) = 0$.

(b) $r(I) \leq r(I+i) \leq r(I)+1$ for $I \in S$ and $i \in S \setminus I$,

(c) for all $I \subseteq S$, $i, j \in S \setminus I$, if $r(I+i) = r(I+j) = r(I)$, then $r(I+i+j) = r(I)$.

The set of arcs associated to the bids themselves is a cut separating the source from the sink in the associated max-flow problem so $r(J) \leq |J|$ for any J, proving (a). Moreover (b) comes from the fact that adding one bid i to I amounts to increase the capacity of one arc by one unit in the associated max-flow problem. Therefore the capacity of any cut increases by at most 1, and the size of the minimum cut will certainly increase, but by one unit at most.

We now prove (c). Let S^I denote the set of nodes containing the source s defining a minimum cut associated with the max-flow problem of computing $r(I)$. In other words, $r(I) = c(\delta^+(I))$.

Note first that since $r(I+i) = r(I)$, there exists S^{I+i} such that the associated cut does not contain (s,i) the arc associated to the bid i. Similarly there exists S^{I+j} such that the associated cut does not contain (s,j) the arc associated to the bid j. This implies also that $\delta^+(S^{I+i} \cup S^{I+j})$ does not contain the arcs (s,i) and (s,j).

By submodularity of cut functions in directed graphs (Theorem (2.4.4), section 2), we obtain that $r(I+i) + r(I+j) = c(\delta^+(S^{I+i})) + c(\delta^+(S^{I+j})) \geq c(\delta^+(S^{I+i} \cup S^{I+j})) + c(\delta^+(S^{I+i} \cap S^{I+j}))$.

Since $\delta^+(S^{I+i} \cup S^{I+j})$ is an $s-t$ cut that does not contain (s,i) and (s,j), if $c(\delta^+(S^{I+i} \cup S^{I+j})) \leq r(I)$, statement (c) holds (the strict inequality case is ruled out by (b)). If $c(\delta^+(S^{I+i} \cup S^{I+j})) > r(I)$ then $c(\delta^+(S^{I+i} \cap S^{I+j})) < r(I)$. But this would contradict the minimality of S^I since $\delta^+(S^{I+i} \cap S^{I+j})$ is an $s-t$ cut. □

The last proposition directly implies that we can solve the auction problem greedily: it suffices to use the cheapest supply bids first, as long as the transmission network allows the use the bid to satisfy some demand, and until all demand is satisfied.

7.4 Cryptographic Preliminaries

7.4.1 Security Model

Ben-Or et al. [4] showed, amongst other things, how (with Shamir's secrete sharing for passive adversaries or Verifiable Secret Sharing (VSS) [14] for active adversaries) every functionality can be computed under the *information theoretic* model. However, that does not necessarily imply efficiency in terms of performance. In our secure algorithms, variations can be included, to accelerate some functionality, at the price of providing statistical security and/or some leakage. Moreover, changes in the communicational or adversarial models would yield different security levels as well. Our protocols follow the same line of thought. Our privacy-preserving protocols can achieve the same level of security, than the underlying primitives (our algorithms have no leakage). A careful sub-routine selection can yield statistical security, with a significantly improvement in terms of performance. We study both aspects of the implementation of our secure protocols.

7.4.2 Basic Building Blocks

Secret Sharing. Important contributions on secret sharing have been proposed during the last 4 decades. Threshold secret-sharing schemes like Shamir's secret-sharing [13], provided an important tool to future results like BGW [4]. Shamir's Secret sharing, is a scheme that allows n parties to share information amongst each other, to later be reconstructed by a subset of the players. Several MPC protocols use Shamir secret sharing as a viable tool for multi-player environments, this is our case. Note that many protocols can rely on some kinds of homomorphic encryption e.g. Paillier encryption [22], instead of traditional sharing mechanisms, this is specially true for the two-party case. For a detailed treatment of Secret Sharing Schemes we refer the reader to Beimel survey paper [20] on the topic.

Multiplications. Unlike addition, multiplication on secret shared information has a communicational cost associated. Ben-Or et al. [4] showed the difference between secret addition and multiplication on shares. Indeed, secret additions on shares (Shamir's secret-sharing) are "cheap", they do not require any additional information exchange between the players. Indeed, they can be computed locally, this is not the case for multiplication. In their paper Ben-Ot et al. came across an interesting formulation to solve the problem. The method was later improved by Gennaro et al. [18]. These mechanisms allow us to bring multiplications with perfect security (tied to the communicational model) into the scene.

Comparisons. Methods for secure comparisons like [25], for the equality and inequality test have been introduced through the years. Recent work by Limpaa and Toft [26], introduced methods with sub-linear complexity on their online phase. Moreover, amongst the variety of existing protocols, there are methods that bring both, perfect and statistical security with efficient bounds and simple algorithms, this is the case of Catrina and Hoogh [27] inequality test. It uses a secure modulo operation and is capable to solve the inequality test problem with an intuitive protocol. The overall complexity of their method is constant. Catrina and Hoogh method is later used in our prototype. For equality test, however, we have used the Limpaa and Toft protocol (based on

the hamming distance).
Although, theoretically speaking, these methods can achieve constant complexity bounds, in practice, they are typically much slower than multiplications. If we take for example Catrina and Hoogh's inequality method ($\mathcal{O}(1)$) to compare 2 secretly shared numbers of 32 bits each. To use up to 4 computational rounds we would need to parallelize 32 threads per player. This might be proven to be a challenge on some limited environments. To achieve practical results in terms of performance, is still important to minimize the use of comparisons.

7.4.3 Complex building Blocks

Our privacy-preserving protocol requires to adapt existing secure applications for solving more complex combinatorial problems. The methods used to solve these problems have to guarantee correctness and security while at the same time minimize their impact over the performance. This includes a practically efficient vector shuffling protocol, sorting and max-flow mechanisms.

Vector Permutation Mechanism. Securely permuting a vector, implies that for any vector E, the resulting configuration is uniformly distributed in the space of all permutations $|E|!$.
The state of the art describes several mechanisms for vector permutation also known as oblivious shuffle. Leur et al. [98] introduced several permutation mechanisms that work with secret sharing, amongst them the multiplication of the vector for a zero/one matrix (permutation matrix M_π where $\pi([E])$ is a permutation of vector $[E]$). In other words $\pi([E]) = M_\pi \times [E]$. The overall complexity of the method: is $\mathcal{O}(|E|^2)$ and can grant perfect security. They also offer other alternatives, amongst them, the use of sorting methods instead of the matrix multiplication, to improve complexity. There exists other sorting methods, with better asymptotic complexity, namely the oblivious shuffling by Keller and Scholl [58] which introduced the use of the Waksman network. These last two mechanisms are indeed asymptotically faster than the matrix multiplication mechanism, in practice they have to face some challenges. For example, the shuffle by the sorting method introduced by Leur et al. [98] depends on your sorting capabilities. It is true that current state of the art on sorting methods offer faster bounds, but they depend on building blocks like comparisons that can greatly affect performance. That is not the case for the Keller and Scholl method, instead, the speed of the algorithm depends on the way the Waksman network is built and the aggregation mechanism, suffering of many of the same issues than Leur et al protocol. Additionally, in all the permutation methods described above, the protocols have to be executed several times to guarantee secrecy.
One advantage of the sorting approach is the use of sorting networks, that are

by nature, data-oblivious. Similar to Leur et al. method, we propose to make use of sorting network techniques where instead of comparison gates, we use exchange gates (`exGate`). An exchange gate can be defined as a probabilistic switch mechanism, in our case each gate uses the random bit generation method of [25], instead of the comparison to decide the switch. Moreover, an *exGate* network i.e. a sorting network composed of `exGates` where the space of all gate combinations is $2^{|[E]| \times log(|[E]|)}$.
Note that $log(|[E]|!) = \theta(|[E]| \times log(|[E]|)$ and $\frac{|[E]| \times log(|[E]|)}{2} \leq log(|[E]|!) \leq |[E]| \times log(|[E]|)$. Thus, it can be said that $|[E]|! = \theta(2^{|[E]| \times log(|[E]|)})$. Czumaj et al. [99] have shown how to build such a network with $\frac{1}{2}$ probability `exGates`. The result is a permutation with (almost) uniform probability in the space of all possible permutations. Such an `exGate` network for a sufficiently large $[E]$ would be able to provide a gate combination per each vector permutation, or a close approximation.
On the practical implementation of such a mechanism: to achieve perfect security, we would require an exact match between the size of the space of all permutations and gate combinations. Instead, statistical security with negligible probability can be achieved using an exchange network. There are sorting networks with $\mathcal{O}(|E| \times log(|[E]|))$. e.g. AKS. Its usability, however, is relative given the weight of its constants. A more standard sorting network e.g. Batcher even-odd merge sort, Bitonic sort, should provide enough confidence on a practical level. Furthermore, the use of a randomized sorting network such as [68], with a complexity of $\mathcal{O}(|E| \times log(|E|))$ could also be considered for real-life applications. Note that the latter method would not be able to reach some smaller sub-set of permutations of E.
An additional option also arises. The use of simplified mechanisms that are capable to choose uniformly a random permutation amongst a sub-set of the set of all possible permutations. For instance, an example could be, if $|[E]|$ is a power of 2, the use of a sorting network where each element is exchanged with a different element at least $log(|[E]|)$ times. Such a step is usually called, in sorting networks theory, a Merge step. This will guarantee that we have selected a random permutation, but only amongst a subset of all the permutations of the vector. As with the randomized shell sort, this of course would leak the sub-set of possible configurations from which this permutation comes.
To conclude, we propose the following three options in terms of an `exGate` network. *i)* Get an (almost) uniform distribution. To get a close approximation through the use of AKS or Czumaj et al construction. This might also be unrealistic for many data configurations. Constraints related to performance are also present. *ii)* A more realistic approach, using sorting networks and being subject to the distribution it provides for its solutions e.g. Batcher's odd-even merge sort. *iii)* The use of a merge step, this in case performance is a bigger constraint that the leakage produced. The decision on which method to use, belongs solely to the algorithm designer and its tied to the application

implementing them.

Sorting Mechanisms. Sorting protocols are necessary building blocks of various complex solutions, including ours. Efficient secure sorting algorithms have been studied for several years, yielding interesting results.
In fact, various protocols for data-oblivious sorting have been proposed to answer this question. Naive approaches can achieve $|V|^2$ complexity e.g. bubble sort. As mentioned before, work by Goodrich [68] introduces a data-oblivious randomized shell-sort protocol. Although, with a relative low complexity bound for a sorting network ($\mathcal{O}(|V| \times log(|V|))$), it does not guarantee correctness. Other data-oblivious approaches have used the same principle, an oblivious construction for the Batcher even-odd sorting network was introduced by Jónsson et al. [69] with a higher complexity bound than the Goodrich method but guarantees correctness. In the literature, however, data-oblivious sorting algorithms tend to report higher complexity bounds than their data dependent counterparts.
Hamada et al. [71] introduced a different approach to the problem. They suggested a mechanism with which non data-oblivious methods could still be used keep and maintain the same asymptotic bounds. The approach called *shuffling before sorting* allows to transform data dependent sorting algorithms into secure versions of themselves with little adaptation. The technique requires the algorithm designer to secretly shuffle the input data before the sorting procedure. Secure comparisons, are used just as in their data-oblivious counterparts, but in this case, their output is revealed. This allows the protocol to choose the data-flow that suits the original method instead. Results show not only competitive theoretical bounds but also practical efficiency. They additionally report on an intuitive adaptation to quick-sort using this technique, obtaining practical running times is a goal of ours with our current work, which makes this technique suitable for our protocols.

Max Flow Mechanisms. Network flow problems have been recently studied by [51, 19, 85] and reported in this thesis. The most efficient method with a $\mathcal{O}(|V|^4)$ is based on the push-relabel algorithm for the max-flow problem and was introduced in Chapter 4. It also suggests the use of stopping conditions (some leakage) to accelerate performance.
Our problem contemplates public knowledge of the transmission network configuration. This can also be contemplated by the original problem introduced by this dissertation to accelerate its performance. Our max-flow algorithm can be adapted to ignore non-existing edges. At its core, the algorithm there is a push/relabel phase where all edges are considered. A flag signaling edges to be ignored and others to be considered would suffice. Moreover, this flag evaluation can be done publicly. Thus, this adaptation is also present in our prototype, we also make use of a stopping condition. Indeed, we use the tracked

non-existence of excess to be allocated on any vertex, as the tool to determine when to stop the algorithm. We are aware that this may cause some leakage (number of iterations were necessary to allocate the flow). This is at the very end a trade-off. Perfect security can be achieved running the algorithm to its complexity bound as shown by the authors, however, the price to pay is performance. The stopping condition can be altered to assert the condition after a certain number of iterations to bring more confidence to the process. This last approach also has to be taken into account by the algorithm designer in accordance with the application's necessities.

7.5 Secure Auction Mechanism

We extend the results of section 7.3 and introduce a secure variant of algorithm 7.3.1. We assume the configuration of the transmission network to be public, and all inputs to be integer.
First, all information is gathered in shared form, randomly permuted and loaded by the parties in charge of the computation. Our protocol uses amongst others, the *shuffle before sorting* technique introduced by Hamada et al. [70] to do an initial bid sorting, and the Max Flow protocol from Chapter 4. Our protocol complexity grows linearly with respect of the number of bids $|B|$ and polynomially by the number of markets $|M|$.

7.5.1 Notation

Our protocol uses the traditional square brackets notation employed by several secure applications in distributed environments e.g. [25, 60, 51]. For instance, a secure assignment and secure addition are denoted by the use of the infix notation and the corresponding square brackets e.g. $[z] \leftarrow [x] + [y]$. The same treatment is extended to any other operation. Vectors are denoted by capital letters e.g. E where $|E|$ denotes the number of elements in E and E_i is the i-th element and $|E_i|$ is the absolute value of the i-th element. To represent negative numbers we use the typical approach of using the lower half of the field for positive values and the upper half of the field for negative values. It has to be noticed that on shared form a negative value is indistinguishable from a positive one. And that in our approach all information related to the bids is kept secret including whether or not it is a supply or demand bid.

On the bid tuples (price and quantity). The algorithm requires additional data for tractability. A market identifier m and a bid identification b. The collection of data from a tuple is completed with its price p and volume q (negative for supply bids and positive for demand bids). The vector of bids is defined as follows: $([b], [m], [p], [q])^{|B|}$. Moreover, the weighted adjacency matrix N is used to define the graph G.Finally note that for simplicity reasons, we have aggregated the edges originated in s towards the same market $m \in M$ into a single edge (s, m).

Furthermore, we provide the definitions of 2 sub-routines we made constant use of, throughout our protocol. These definitions contribute to an easy reading and simplify the expressions. Both of these sub-routines can be built using the elements enumerated in section: 7.4.

- conditional assignment : This functionality serves as a replacement of a flow control instruction for branching. Although branching on encrypted data is not possible, the functionality can be emulated for assignment tasks. Following [85] we represent the operator by : $[z] \leftarrow_{[c]} [x] : [y]$. Where much like in previous works e.g. [85, 32, 51, 60] $[z]$ would take the value $[x]$ if $[c]$ is 1 and $[y]$ otherwise. This can be achieved simply by doing the following $[z] \leftarrow ([x] - [y]) \times [c] + [y]$.

- market identification : Part of the data that composes a bid is the identification of the market it belongs to. Users are required to input a single identification tag. Later, this tag has to be transformed into the vector $Z_i \quad \forall i \in B$ of size $|M|$, a zero-one list where its element $Z_{i,m} = 1$ if and only if B_i is binded to market $m \in M$. It serves to provide a reusable mechanism to evaluate the market identification, and reduce the amount of inequality tests. This transformation can be achieved following protocol:

Protocol 10: vector transformation for market identification

```
Input: vector of all markets $M$, bid $[i] \in B$.
Output: zero-one vector $[Z] of size |M|$
1 for $i \leftarrow 1$ to $|M|$ do
2     $[Z]_i \leftarrow m_i == [m]_i$;
3 end
4 return $[Z]$;
```

7.5.2 Secure Auction with Transmission Constraints

Prerequisites. Data is presented to the computational parties: the Secret Shared and permuted vector of all Bids. The number of Bids or at least an upper bound on the size of the vector is assumed to be public. We assume the topology and capacities of the transmission network to be public.

1. Bids are sorted in ascending order according to their price.

2. Graph $G = (V, A)$ is processed as follows: Set the capacity of edges in S to 0. Capacities of edges from G towards t are fixed to the following: $\sum_{i \in [d]_m} [Q]_i \quad \forall m \in M$. To achieve this, all bids have to be explored and its volume is added in case it is a demand bid. At this stage matrix Z_{ij} where vector $Z_i \forall i \in [B]$ is produced using the market identification protocol 10.

3. Evaluate the viability of each of the bids from the recently sorted vector $[B]$ in ascending order. We do that by setting the capacity of the corresponding edge to the value of $|Q_{B_i}|$, and then calculating the max flow on the graph $G = (V, A)$ to determine whether the bid can improve the solution. If that is the case the amount of flow that can be allocated from the bid is stored and added to the capacity of the edge connecting its source to the transmission network. That way subsequent iterations can take this value into account. We repeat the process $\forall i \in B$. Protocol 11 shows a detailed description of this procedure.

4. The bids then are permuted randomly, to hide their order. This step is necessary to avoid leaking the result of the initial sorting from step 1.

Protocol 11: Implementation of secure auction.

```
Input: Adjacency Matrix $[N]_{ij}$. Vector of Bids $[B]$. Matrix of market
       identification $[Z]_{ij}$ $\forall i \in [B]$ and $\forall j \in M$
Output: Flow Matrix $F$, the list of bids and they accepted capacities
1  $[\nu] \leftarrow [0]$
2  for $i \leftarrow 1$ to $|[B]|$ do
3      for $j \leftarrow 1$ to $|M|$ do
4          $[N]_{sj} \leftarrow_{[Z]_{ij}} [N]_{sj} + |[Q_{B_i}]| : [N]_{sj}$;
5      end
6      $[\nu'] \leftarrow$ maxflow($[N]$);
7      $[\phi] \leftarrow ([\nu] - [\nu])$;
8      for $j \leftarrow 1$ to $|M|$ do
9          $[N]_{sj} \leftarrow_{[Z]_{ij}} [N]_{sj} - |[Q_{B_i}]| : [N]_{sj}$;
10         $[N]_{sj} \leftarrow_{[Z]_{ij}} [N]_{sj} + ([\phi]) : [N]_{sj}$;
11         $[Q_{B_i}] \leftarrow_{[Z]_{ij}} ([\phi]) : |[Q_{B_i}]|$;
12     end
13     $[\nu] \leftarrow [\nu']$;
14 end
```

Analysis. On the prerequisites, several parties constantly submit bids in shared form, we believe it is safe to assume this will not occur simultaneously. Precomputed permutation matrices can be generated. A simple vector multiplication of the corresponding row of the matrix would suffice in this case to place the incoming data in their corresponding permuted position in the vector. Once all data is received, the existing vectors can be easily combined, the result is a single permuted vector. In case this approach is not feasible, the algorithm designer could make use of one of the suggested permutation mechanisms instead.
Permuted bids allows us to make use of Hamada et al [70] technique of

`shuffling before sorting`. This improves considerably the performance of sorting protocols and allows them to achieve $\mathcal{O}(|E| \times log(E))$ complexity.
Step *3.* can be seen as a preprocessing step. where data is transformed to fit to the necessities of the next phases, where volume is selected. Moreover, it synthesizes processes that would be constantly repeated otherwise. Given that our emphasis is to achieve practically efficient times, this constitutes an important gain in terms of performance.
We called these previous steps: Data Generation and Preprocessing Phase. Step *3.* however, serves as an evaluation and allocation mechanism. We called this step the Iterative Augmentation phase. It can be seen as some heuristic tool that allows us to identify the impact of the bid on the result. Note that the public transmission network also contributes to accelerate the process. As mentioned in previous sections, known algorithms for the max flow problem like the one introduced by this thesis can improve running times when the topology of the graph is public. Stopping conditions, however, because of the leakage associated to them, are left at the discretion of the algorithm designer.
Protocol 11 let us explore the inner works of this step into detail. *Line 2* allows us to explore all previously sorted bids in order. *Lines 3 to 5* augment the corresponding edge capacity from the source to the corresponding market with the volume of the bid. *On Line 6*, ν stores the maximum amount of flow that can be allocated with the new volume. On the final section of the protocol (*Lines 7 to 13*) the difference between previous and present flow gap is calculated. Moreover, the flow added to the graph at the beginning of the iteration is replaced by the gap variation. This value has to be stored as well as the amount of capacity assigned to the bid and the value of the maximum flow for future iterations.
Finally, the last and *4.* step is also called the presentation phase. At this stage, data can be edited at will by the algorithm designer. What information is taken to later be presented depends solely on the application's nature. The permutation, although capable to hide the sorting should be ignored in case the final answer also contemplates to open the prices of the bids as well. This is because any party could later sort the bids accordingly.

Complexity. Step *1.* depends entirely in the selection of the sorting algorithm. Hamada et al. technique allows us to use non-oblivious sorting mechanisms with lower complexity bounds. Quick-sort, or merge-sort are suitable options. This phase requires as many rounds as the method selected. i.e. $\mathcal{O}(|E| \times log(|E|))$ or in our case $\mathcal{O}(|B| \times log(|B|))$. On step *2.* however,complexity does not only depend on the size of the bids vector. Protocol 10 has to be executed with all bids i.e. $\mathcal{O}(|B| \times |M|)$. Step *3.* calculates the max-flow of the extended graph $G = (V, A)$. Previous sections analyze algorithms to the max-flow problem suitable to our work. To the best of our knowledge, the

method introduced by Aly et al. offers the best asymptotic bound to the secure problem formulation i.e. $\mathcal{O}(|V|^4)$. When this method is used, the complexity of this step can be defined as follows: $|B| \times (|M| + |M| + \mathcal{O}((|M| + 2)^4))$ or simply like: $\mathcal{O}(|B| \times |M|^4)$.
Lastly, step *4.* its also tied to the method for the permutation. Its round complexity can vary from $\mathcal{O}(|B| \times log(|B|))$ to $\mathcal{O}(|B|^2)$. In any case the general complexity of the protocol is dominated by step *4.*
In our secure protocol the number of markets influence performance greatly. The impact of having more markets to analyze has a polynomial impact in the number of operations. Whereas the size of the bids only lineal. It can be also seen why we allow this phenomena in our algorithm. This is primarily because of the impact that a polynomial algorithm could have over large sets of bids (In our study case we saw ≈ 1500 bids per instance). In comparison, a smaller number of markets are involved in the computations. This can be translated in overall improved performance, specially when variations in the number of bids are common, but the number of markets is relatively small in comparison and the set stays the same size for extended periods of time.

7.5.3 Security and Correctness

The protocol itself can achieve perfect security, this is because our privacy-preserving algorithm can be implemented with no leakage. In this case, its security depends solely on the primitives that implement the sharing, secure arithmetic operations, comparison algorithms and the security model. However, trade-offs in performance have to be considered. This is the case of the stopping conditions for the maximum flow calculation, where some leakage is produced on the number of iterations needed to move the newly added flow, as well as, the vector permutation. The impact of such leakage has to be evaluated by the protocol designer. Furthermore, some complex building blocks, in this case some comparison protocols, only offer statistical security. When that is the case, security is given as a function of parameters k and l. Where k is a security parameter on the bit size of the input and l is the inputs bit size. This is because in general terms, randomized elements of the solutions are chosen up to a threshold, smaller than the finite field they were generated upon. The word size of the threshold depends on both parameters. That is why the bigger the size of k the better in terms of security. Furthermore our secure protocol is an implementation of algorithm 7.3.1, and thus guarantees correctness.

7.6 Computational Experimentation

We have tested our protocol with the MPC Toolkit reported on Chapter 6. The library implemented all the primitives and building blocks we report on,

including the underlying MPC crypto-primitives. It also provide our own communicational support and use NTL (Number Theory Library) [84] and GMP (GNU Multiple Precision Library) libraries for the underlying modulo arithmetic.
An Advantage of "from the scratch" implementations is that they can satisfy punctual necessities, in this case the use of C++ and an object oriented architecture. This allows us to keep low coupling levels and easy adaptability to real life problems. Moreover, the library provides us with the primitives that are essential building blocks for more complex problems e.g. sorting, max-flow and oblivious permutation. Future extensions could include state of the art developments without the necessity of radical changes on the existing code thanks to the properties of encapsulation. This is not a replacement for more complete frameworks e.g. PICCO [19], SPDZ [33] or VIFF [2]. But it is indeed a set of libraries of easy access that can be intuitively improved or modified to obtain specific results with custom developed software.
We run tests with realistic data of electricity markets. We estimate the number of operations and the time needed by the collection of tools we report on to securely solve instances of the auction. Once the answer is found, we quantify the results and propose strategies to improve running times.

7.6.1 Prototype Capabilities and Technical Characteristics

Our prototype makes use of the building blocks and secure protocols we report on, as well of other well known results. Table 7.1 showcases the different versions of primitives and functionality used as building blocks in our application:

Building Block	**Algorithm**
Sharing	Shamir Secret Sharing [13]
Multiplication	Gennaro et al [18]
Equality Test	Limpaa and Toft [26]
Inequality Test	Catrina and Hoogh [61]
Random Bit Gen.	Damgård et al [25]

Table 7.1: List of Primitives used by the Secure Auction protocol implementation

These are considered core functionalities. The architecture from the library is to provide a basic and decoupled processing unit similar to a small engine. This small core implements the functionalities from table 7.1, amongst others, note that this is also the functionality used in our tests. Furthermore, it separates computational and cryptographic tasks from communicational tasks. Each engine runs these two sets of tasks in different threads that communicate with each other to coordinate. Basic conditions to obtain the best performance from the engine, include 2 CPU threads and $\approx$ 500 KB in RAM for the basic use of the primitives. Our configuration gives each computational party 2 similar CPU threads with unlimited access to a memory pool of up 42 GB with each player having a single engine's instance.

About the more complex elements of our protocol namely secure sorting and secure max-flow: On the first, thanks to Hamada et al. technique introduced in [70], it sufficed to implement the quick-sort adaptation. On the latter, we used the secure adaptation of Aly et al. protocol [51] mentioned by previous sections (public topology with stopping condition in this case when no excess is left to allocate in the graph). Finally, on permutations, two flavors were implemented. We use an `exGates` exchange network, with either a merge network or batcher even-odd merge-sort network.

On Security. The library and prototype were built under the private channel model. Depending on the functionality used, the library provides statistical and perfect security against semi-honest adversaries with minority coalition. For instance, the inequality test used in our tests brings statistical security meanwhile addition and multiplication perfect security.
As mentioned, statistical security for such method is a function of parameter k and the bit-size of the input by parameter l. The prototype was pre-configured to use $k = 29$ and $l = 32$. However, because of technical issues, shares themselves can only use up to 63 bits. This means in practice that under the scenario where only primitives with perfect security are used, the size of l could grow up to 63 bits.

7.6.2 Numerical Results

We experiment with realistic data from electricity markets. Our study case was composed of a total of 1945 bids (demand and supply). The origin of the data is one hour of a typical day trade from the Belpex market (12 pm). Additionally, the transmission network considers the existence of 2 and 4 markets interconnected amongst them by bidirectional lines under the same restrictions than the real life problem. Finally, we measure the time, memory and workload distribution it takes our protocol to find an answer from the start of the protocol.
We run our tests on an Intel Xeon CPUs X5550 (2.67GHz) and 42GB of memory, running Mac OS X 10.10. All processes have the same computational power at their disposal (memory and CPU power). We consider the case with 3 computational parties. Table 7.2 shows the average amount of operations of a single test case.
From these tests, $\approx 21 \times 10^6$ communicational rounds were dedicated to randomization processes for the comparison mechanisms e.g. random bit generations. The use of well studied results like PRSS [87] would limit the use of these communicational rounds and in general terms would allow us to achieve

even better computational times. Furthermore, Catrina and Hoogh comparison method depends on the computation of l random bits for its calculations. An offline phase can be considered where this random numbers are pre-computed before the bids arrive to the server, and then distributed to the computational parties for its use. In this case the Secure Auction Mechanism would be executed in an online phase that no longer has to care about random number generation improving the performance specially of comparisons. Table 7.2 shows our numerical results and estimated the impact of the use of online/offline phases.

Markets and Perm. Method		Com. Rounds	Comparisons	CPU Time.	Online Phase
2 Markets	Batcher	$\approx 31.4 \cdot 10^6$	226021	2056 s.	613 s.
	Merge	$\approx 31.4 \cdot 10^6$	226021	2049 s.	606 s.
4 Markets	Batcher	$\approx 71.9 \cdot 10^6$	537627	4702 s.	1276 s.
	Merge	$\approx 71.9 \cdot 10^6$	537627	4694 s.	1268 s.

Table 7.2: Overall Results

When we simulate an environment where these bids are distributed amongst 4 different markets instead of 2, the performance test shows an increase of around twice the number of rounds and comparisons. The same follows on computational time, taking in average around 4700 seconds to complete execution.

On memory, the application did not surpass in average the 2.5 MB per execution. During its life-cycle, some increment in memory consumption levels was registered during data generation phases. The phenomenon is specially evident in the data preprocessing phase. This is of course, because some data is generated and stored for later use. Less significantly changes are also present. This is explained by the continuous fragmentation of the memory throughout the evaluation and selection phase (Iterative Augmentation), where objects of different sizes are continuously created and then destroyed. Nonetheless, these noise variations never grow more than 100 KB in average. Figure 7.6.1 shows the typical behavior of the application during its life-span.

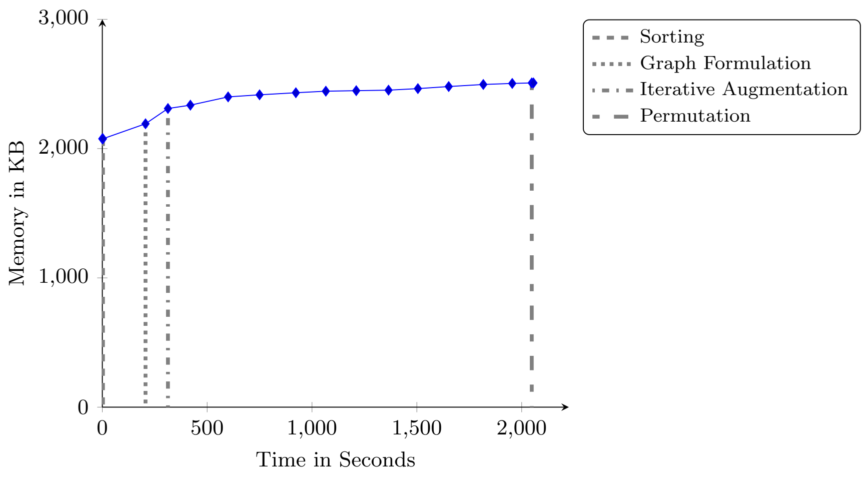


Figure 7.6.1: Secure Auction Protocol Life Cycle

Finally, we found that the bottleneck of the application is communications. In average, data transmission and related tasks are responsible of the 1705 seconds (83%) of the total computational time. Only a fraction of the time, 351 seconds (17%), was dedicated to other tasks e.g. share generation, basic arithmetic operations and other algorithmic and programmatic tasks.

From these assertions we can estate the following:

- Realistic computational times were indeed achieved for the data in question, with limited computational power. With the online/offline case, an hour of trade was solved in less than an hour of computations for all cases. Given that many of the processes of our protocol are sequential in nature, a computer with a better benchmarked CPU under the same basic configuration would yield better results.

- Memory is not a decisive factor in this case. Memory increases monotonically during the execution because of noise, but in a reduced proportion. As it was previously estated this change is smaller than 100 KB, product of the continuous memory allocation and deallocation of different sized objects. In general terms memory use only grows significantly when information is stored for later use.

- The process can be further accelerated by pre-computing the random values that are needed by the protocol. In our case $\frac{3}{4}$ of communicational rounds that are used for comparisons are dedicated to randomization processes. Even with the use of PRSS, these operations would represent an important proportion of the workload. This is why an offline phase where these values are preprocessed could be proven useful. For instance, to have dedicated servers calculating in share form random bits and numbers and store them, such that they can just be fetched when any online process needs them. This would imply a reduction in the 2 markets case of ≈ 1450 seconds in average. This would allow us to solve the problem in ≈ 610 seconds in average, a little more than 10 minutes. When 4 markets are consider instead, the times are reduce to ≈ 1270 which is little more than 20 minutes.

- Moreover, even though we have put in place a light and dedicated communications setting, the prototype looses performance because of the cost that data transmission implies. The cryptographic primitives in our case are not dragging the bulk of the workload to their side. In fact it was 4.8 more expensive to transmit the data than to generate, reorganize and calculate it.

Chapter 8

Conclusions

Our experimentation has shown that cryptographic procedures are no longer the bottleneck in performance. Communications have taken the major hurdle performance-wise. Questions about future work on the topic raise the need to minimize the use of communications. The study of specific cases where changes of workload distribution can accelerate the performance can be assessed as well.

Polynomial time algorithms for auctions with transmission constraints, combined with MPC could be possible in realistic scenarios, although more extensive experimentation is needed to evaluate its applicability in real environments. The same stands for other combinatorial problems that arise in these settings. Secure protocols with no leakage and polynomial bounds can be obtained, although relaxations, i.e. some leakage, in some cases, are necessary to achieve realistic times. Trade-offs between security and performance are necessary. But in realistic settings, only the algorithm designer is capacitated to determine when and were to apply such changes.

When MPC is involved, note that much work is still needed to be able to provide real solutions on electricity markets. The tools and current state of the art on MPC have allowed us to work towards practical implementations. Currently, our applications needs little more than half an hour for the two markets case and less one hour and a half with 4 markets. This is reduced to a bit more than 10 and 20 minutes respectively, when offline/online phases are considered. The performance in our case is strongly tied to the size of the market.

Finally, we can see that well thought systems that use our protocols must be well adapted to the circumstances of the application. A good design would take into account online and offline phases, leakage levels or computational players to maximize the security and guaranteeing practically efficient running times.

Bibliography

[1] Yao, A.C.C.: Protocols for secure computations (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science, IEEE (1982) 160–164

[2] Geisler, M.: Cryptographic protocols: theory and implementation. PhD thesis, Aarhus University Denmark, Department of Computer Science (2010)

[3] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC, ACM (1987) 218–229

[4] Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC, ACM (1988) 1–10

[5] Beaver, D.: Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. Journal of Cryptology **4**(2) (1991) 75–122

[6] Beaver, D.: Foundations of secure interactive computing. In Feigenbaum, J., ed.: Advances in Cryptology — CRYPTO '91. Volume 576 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1992) 377–391

[7] Goldreich, O.: General cryptographic protocols: The very basics. Secure Multi-Party Computation **10** (2013) 1

[8] Asharov, G., Lindell, Y.: The bgw protocol for perfectly-secure multiparty computation. Secure Multi-Party Computation **10** (2013) 1

[9] Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols. In: STOC, ACM (1988) 11–19

[10] Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority. In: Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing. STOC '89, New York, NY, USA, ACM (1989) 73–85

[11] Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.: Asynchronous multiparty computation: Theory and implementation. In Jarecki, S., Tsudik, G., eds.: Public Key Cryptography – PKC 2009. Volume 5443 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 160–179

[12] Kushilevitz, E., Lindell, Y., Rabin, T.: Information-theoretically secure protocols and security under composition. In: Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing. STOC '06, New York, NY, USA, ACM (2006) 109–118

[13] Shamir, A.: How to share a secret. Commun. ACM **22**(11) (1979) 612–613

[14] Chor, B., Goldwasser, S., Micali, S., Awerbuch, B.: Verifiable secret sharing and achieving simultaneity in the presence of faults. In: Foundations of Computer Science, 1985., 26th Annual Symposium on. (Oct 1985) 383–395

[15] Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: CRYPTO. Volume 2729 of LNCS., Springer (2003) 247–264

[16] Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: EUROCRYPT. Volume 6632 of LNCS., Springer (2011) 169–188

[17] Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO. Volume 7417 of LNCS., Springer (2012) 643–662

[18] Gennaro, R., Rabin, M.O., Rabin, T.: Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing. PODC '98, New York, NY, USA, ACM (1998) 101–111

[19] Zhang, Y., Steele, A., Blanton, M.: Picco: a general-purpose compiler for private distributed computation. In: Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security. CCS '13, New York, NY, USA, ACM (2013) 813–826

[20] Beimel, A.: Secret-sharing schemes: A survey. In Chee, Y., Guo, Z., Ling, S., Shao, F., Tang, Y., Wang, H., Xing, C., eds.: Coding and Cryptology. Volume 6639 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 11–46

[21] Resitad, T.I.: A general Framework for Multiparty Computations. PhD thesis, Norwegian Univeristy of Science and Techonology, Faculty of Information Technology, Mathematics and Electrical Engineering Department of Telematics (2012)

[22] Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: EUROCRYPT. (1999) 223–238

[23] Damgård, I., Jurik, M., Nielsen, J.B.: A generalization of paillier's public-key system with applications to electronic voting. P Y A RYAN (2003) 3

[24] Cramer, R., Damgaard, I., Nielsen, J.B.: Secure multiparty computation

[25] Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: TCC. (2006) 285–304

[26] Lipmaa, H., Toft, T.: Secure equality and greater-than tests with sublinear online complexity. In: ICALP (2). (2013) 645–656

[27] Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: SCN. (2010) 182–199

[28] Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay&#8212;a secure two-party computation system. In: Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13. SSYM'04, Berkeley, CA, USA, USENIX Association (2004) 20–20

[29] Ben-David, A., Nisan, N., Pinkas, B.: Fairplaymp: a system for secure multi-party computation. In: CCS, ACM (2008) 257–266

[30] Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A Framework for Fast Privacy-Preserving Computations. In: Proceedings of the 13th ESORICS. Volume 5283 of LNCS., Springer (2008) 192–206

[31] Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M., Toft, T. In Dingledine, R., Golle, P., eds.: Financial Cryptography and Data Security, Berlin, Heidelberg, Springer-Verlag (2009) 325–343

[32] Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.: utomating efficient ram-model secure computation. In: 35th IEEE Symposium on Security and Privacy. (2014)

[33] Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.: Practical covertly secure mpc for dishonest majority – or: Breaking the spdz limits. In Crampton, J., Jajodia, S., Mayes, K., eds.: Computer Security – ESORICS 2013. Volume 8134 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 1–18

[34] Korte, B., Vygen, J.: Combinatorial Optimization: Theory and Algorithms. 4th edn. Springer Publishing Company, Incorporated (2007)

[35] Lawler, E.: Combinatorial Optimization : Networks and Matroids. Dover Publications (March 2001)

[36] West, D.B.: Introduction to Graph Theory (2nd Edition). 2 edn. Prentice Hall (September 2000)

[37] Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. 2nd edn. McGraw-Hill Higher Education (2001)

[38] Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)

[39] Whitney, H.: 2-Isomorphic graphs. Am. J. Math. **55**(1–4) (1933) 245–254 MR:1506961. Zbl:0006.37005. JFM:59.1235.01.

[40] Whitney, H.: On the abstract properties of linear dependence. American Journal of Mathematics **57** (1935) 509–533

[41] Schrijver, A.: Combinatorial Optimization: Polyhedra and Efficiency. Number v. 2 in Algorithms and Combinatorics. Springer (2003)

[42] Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3) (1987) 596–615

[43] Cramer, R., Damgård, I., Nielsen, J.: Multiparty computation from threshold homomorphic encryption. In Pfitzmann, B., ed.: Advances in Cryptology — EUROCRYPT 2001. Volume 2045 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 280–300

[44] Orlandi, C.: Lego and other cryptographic constructions. Technical report (2009)

[45] Toft, T.: Primitives and Applications for Multi-party Computation. PhD thesis, Department of Computer Science, Aarhus University (2007)

[46] Nishide, T., Ohta, K.: Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In: Public Key Cryptography. (2007) 343–360

[47] Toft, T.: Secure data structures based on multi-party computation. In: PODC, ACM (2011) 291–292

[48] Kruger, L., Jha, S., Goh, E.J., Boneh, D.: Secure function evaluation with ordered binary decision diagrams. In: ACM CCS, ACM (2006) 410–420

[49] Barni, M., Failla, P., Kolesnikov, V., Lazzeretti, R., Sadeghi, A.R., Schneider, T.: Secure evaluation of private linear branching programs with medical applications. In: ESORICS. Volume 5789 of LNCS., Springer (2009) 424–439

[50] Barni, M., Failla, P., Lazzeretti, R., Sadeghi, A.R., Schneider, T.: Privacy-preserving ECG classification with branching programs and neural networks. IEEE TIFS **6**(2) (June 2011) 452–468

[51] Aly, A., Cuvelier, E., Mawet, S., Pereira, O., Van Vyve, M.: Securely solving simple combinatorial graph problems. In: Financial Cryptography. (2013) 239–257

[52] Launchbury, J., Diatchki, I.S., DuBuisson, T., Adams-Moran, A.: Efficient lookup-table protocol in secure multiparty computation. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. ICFP '12, New York, NY, USA, ACM (2012) 189–200

[53] Brickell, J., Shmatikov, V.: Privacy-preserving graph algorithms in the semi-honest model. In: ASIACRYPT. Volume 3788 of LNCS. Springer (2005) 236–252

[54] Blanton, M., Steele, A., Alisagari, M.: Data-oblivious graph algorithms for secure computation and outsourcing. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. ASIA CCS '13, New York, NY, USA, ACM (2013) 207–218

[55] Wang, X., Nayak, K., Liu, C., Shi, E., Stefanov, E., Huang, Y.: Oblivious data structures. Cryptology ePrint Archive, Report 2014/185 (2014) `http://eprint.iacr.org/`.

[56] Lu, S., Ostrovsky, R.: Distributed oblivious ram for secure two-party computation. Cryptology ePrint Archive, Report 2011/384 (2011) `http://eprint.iacr.org/`.

[57] Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. CCS '12, New York, NY, USA, ACM (2012) 513–524

[58] Keller, M., Scholl, P.: Efficient, oblivious data structures for mpc. IACR Cryptology ePrint Archive **2014** (2014) 137

[59] Li, J., Atallah, M.J.: Secure and private collaborative linear programming. In: Collaborative Computing: Networking, Applications and Worksharing, 2006. CollaborateCom 2006. International Conference on. (Nov 2006) 1–8

[60] Toft, T.: Solving linear programs using multiparty computation. In: Financial Cryptography. Volume 5628 of LNCS., Springer (2009) 90–107

[61] Catrina, O., de Hoogh, S.: Secure multiparty linear programming using fixed-point arithmetic. In: ESORICS. (2010) 134–150

[62] Maurer, U.: Secure multi-party computation made simple. Discrete Applied Mathematics **154**(2) (2006) 370 – 381 Coding and Cryptography.

[63] Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority. In: Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing. STOC '89, New York, NY, USA, ACM (1989) 73–85

[64] Toft, T.: Secure integer computation with applications in economics, phd progress report. Technical report (2005)

[65] Yao, A.C.: How to generate and exchange secrets. In: FOCS, IEEE (1986) 162–167

[66] Lindell, Y.: Secure computation without agreement. In: Composition of Secure Multi-Party Protocols. Volume 2815 of LNCS. Springer Berlin / Heidelberg (2003) 45–79

[67] Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: Sepia: privacy-preserving aggregation of multi-domain network events and statistics. In: Proceedings of the 19th USENIX conference on Security. USENIX Security'10, USENIX (2010)

[68] Goodrich, M.T.: Randomized shellsort: A simple data-oblivious sorting algorithm. J. ACM **58**(6) (December 2011) 27:1–27:26

[69] Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. IACR Cryptology ePrint Archive **2011** (2011) 122

[70] Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: ICISC. (2012) 202–216

[71] Hamada, K., Ikarashi, D., Chida, K., Takahashi, K.: Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. IACR Cryptology ePrint Archive **2014** (2014) 121

[72] Goodrich, M.T.: Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n) time. In: Proceedings of the 46th Annual ACM Symposium on Theory of Computing. STOC '14, New York, NY, USA, ACM (2014) 684–693

[73] Gupta, D., Segal, A., Panda, A., Segev, G., Schapira, M., Feigenbaum, J., Rexford, J., Shenker, S.: A new approach to interdomain routing based on secure multi-party computation. In: Proceedings of the 11th ACM Workshop on Hot Topics in Networks, ACM (2012) 37–42

[74] Lawler, Eugene L.: Optimal cycles in doubly weighted directed linear graphs. Theory of Graphs International Symposium, Paris New York (1966)

[75] Lawler, E.L.: Optimal cycles in graphs and the minimal cost-to-time ratio problem. In Marzollo, A., ed.: Periodic Optimization. Volume 135 of International Centre for Mechanical Sciences. Springer Vienna (1972) 37–60

[76] Dantzig, G., Blattner, W., Rao, R., Dept, S.U.O.: Finding a Cycle in a Graph with Minimum Cost to Time Ratio with Application to a Ship Routing Problem. Defense Technical Information Center (1966)

[77] Karp, R.M.: A characterization of the minimum cycle mean in a digraph. Discrete Mathematics **23**(3) (1978) 309 – 311

[78] Goldberg, A.V., Tarjan, R.E.: Finding minimum-cost circulations by canceling negative cycles. J. ACM **36**(4) (October 1989) 873–886

[79] Wayne, K.D.: A polynomial combinatorial algorithm for generalized minimum cost flow. In: Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing. STOC '99, New York, NY, USA, ACM (1999) 11–18

[80] Klein, M.: A primal method for minimal cost flows with applications to the assignment and transportation problems. Management Science **14**(3) (1967) pp. 205–220

[81] Busacker, R., Saaty, T.: Finite graphs and networks: an introduction with applications. International series in pure and applied mathematics. McGraw-Hill (1965)

[82] Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: Tasty: tool for automating secure two-party computations. In: ACM Conference on Computer and Communications Security, ACM (2010) 451–462

[83] Aly, A., Van Vyve, M.: Edge runtime technical manual. (Oct 2015)

[84] Shoup, V.: Ntl: A library for doing number theory (2001)

[85] Aly, A., Van Vyve, M.: Securely solving classical network flow problems. In Lee, J., Kim, J., eds.: Information Security and Cryptology - ICISC 2014. Volume 8949 of Lecture Notes in Computer Science., Springer International Publishing (2015) 205–221

[86] Aly, Abdelrahaman and, V.V.M.: Edge runtime users manual. (Oct 2015)

[87] Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In Kilian, J., ed.: Theory of Cryptography. Volume 3378 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 342–362

[88] Franklin, M.K., Reiter, M.K.: The design and implementation of a secure auction service. In: Proceedings of the 1995 IEEE Symposium on Security and Privacy. SP '95, Washington, DC, USA, IEEE Computer Society (1995) 2–

[89] Harkavy, M., Harkavy, M., Tygar, J.D., Tygar, J.D., Kikuchi, H., Kikuchi, H.: Electronic auctions with private bids (1998)

[90] Kikuchi, H., Hotta, S., Abe, K., Nakanishi, S.: Distributed auction servers resolving winner and winning bid without revealing privacy of bids. In: Parallel and Distributed Systems: Workshops, Seventh International Conference on, 2000. (Oct 2000) 307–312

[91] Peng, K., Boyd, C., Dawson, E.: Optimization of electronic first-bid sealed-bid auction based on homomorphic secret sharing. In Dawson, E., Vaudenay, S., eds.: Progress in Cryptology – Mycrypt 2005. Volume 3715 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 84–98

[92] Nojoumian, M., Stinson, D.: Efficient sealed-bid auction protocols using verifiable secret sharing. In Huang, X., Zhou, J., eds.: Information Security Practice and Experience. Volume 8434 of Lecture Notes in Computer Science. Springer International Publishing (2014) 302–317

[93] Catane, B., Herzberg, A.: Secure second price auctions with a rational auctioneer. In: The 10-th SECRYPT International Conference on Security and Cryptography. (2013)

[94] Madani, M., van Vyve, M.: A new formulation of the european day-ahead electricity market problem and its algorithmic consequences. CORE Discussion Papers 2013074, Université catholique de Louvain, Center for Operations Research and Econometrics (CORE) (2013)

[95] Madani, M., Van Vyve, M.: Computationally efficient {MIP} formulation and algorithms for european day-ahead electricity market auctions. European Journal of Operational Research (0) (2014) –

[96] COSMOS: Public Description. CWE Market Coupling algorithm (jan 2011)

[97] EUPHEMIA: Public Description. PCR Market Coupling algorithm (oct 2013) (Avalaible online: http://static.epexspot.com/document/27917/Euphemia).

[98] Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In Lai, X., Zhou, J., Li, H., eds.: Information Security. Volume 7001 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 262–277

[99] Czumaj, A., Kanarek, P., Kutylowski, M., Lorys, K.: Delayed path coupling and generating random permutations via distributed stochastic processes. In: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '99, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (1999) 271–280