An Introduction to Secret-Sharing-Based Secure Multiparty Computation

Daniel Escudero
daniel@escudero.me

June 4, 2023

Contents

Introduction 2

I MPC Fundamentals 14

1 The Theory of Multiparty Computation 15
1.1 A General Introduction to MPC . . . 15
1.1.1 Adversaries and their Power . . . 16
1.1.1.1 Possible Corrupted Sets . . . 17
1.1.1.2 Type of Corruption . . . 20
1.1.2 Privacy Guarantees . . . 21
1.1.3 Output Guarantees . . . 22
1.1.4 Different Functions to be Computed . . . 23
1.1.4.1 Public-Output vs Private-Output . . . 23
1.1.4.2 Reactive vs Non-Reactive Functionalities . . . 24
1.1.4.3 General vs Special-Purpose MPC . . . 24
1.1.5 Efficiency Metrics . . . 27
1.2 Simulation-Based Security . . . 27
1.2.1 High-Level Idea . . . 27
1.2.2 Interactive Agents . . . 29
1.2.2.1 Relevant Interactive Agents in the UC Framework . . . 29
1.2.3 Interactive Systems . . . 32
1.2.3.1 Relevant Interactive Systems in the UC Framework . . . 33
1.2.3.2 Parameterized Interactive Systems . . . 34
1.2.4 Security Definition . . . 34
1.2.4.1 Perfect Security . . . 35
1.2.4.2 Statistical Security . . . 36
1.2.4.3 Computational Security . . . 36
1.2.5 The Composition Theorem . . . 37
1.2.6 Some Basic Functionalities . . . 39
1.2.6.1 Underlying Communication Resource . . . 39
1.2.6.2 Arithmetic Black Box Model . . . 40
1.3 Fundamental Results . . . 41
1.3.1 Results for $t < n/3$ . . . 41
1.3.2 Results for $t < n/2$ . . . 42
1.3.2.1 The Case of a Passive Adversary . . . 42
1.3.2.2 The Case of an Active Adversary . . . 43
1.3.3 Positive Results for $t < n$ . . . 44
1.3.4 Summary of Main Results . . . 44

2 Secret-Sharing-Based MPC **46**
2.1 Linear Secret-Sharing Schemes . . . 47
2.2 MPC based on Linear Secret-Sharing Schemes . . . 49
2.2.1 The Case of an Active Adversary . . . 49
2.2.2 Offline-Online Paradigm . . . 50

II Honest Majority **52**

3 Shamir Secret-Sharing **53**
3.1 Secret-Sharing and d-Consistency. . . . 55
3.2 Error Detection/Correction . . . 56
3.2.1 Error Detection . . . 58
3.2.1.1 A more intuitive view. . . . 59
3.2.2 Error Correction . . . 59
3.2.2.1 Efficient error-correction. . . . 59
3.2.2.2 A more intuitive view. . . . 60
3.3 Error Correction/Detection in the Context of MPC . . . 60
3.4 Reconstructing Secret-Shared Values Efficiently . . . 61

4 Passive and Perfect Security for Honest Majority **63**
4.1 A First Protocol . . . 63
4.2 A More Efficient Protocol . . . 66
4.2.1 Using Double-Sharings for Secure Multiplication . . . 67
4.2.1.1 A small optimization. . . . 68
4.2.2 Producing Double-Sharings Efficiently . . . 68

5 Active and Perfect Security for Two-Thirds Honest Majority **70**
5.1 Actively Secure Multiplication for $t < n/3$ . . . 70
5.2 Instantiating the Offline Phase . . . 71
5.2.1 Hyper-Invertible Matrices . . . 71
5.2.2 Generating Double-Sharings . . . 72
5.3 Actively Secure Input Phase . . . 73

6 Active and Statistical Security for Honest Majority **75**
6.1 Reconstructing Secret-Shared Values . . . 75
6.2 Preprocessing Phase . . . 76
6.3 Online Phase . . . 79
6.4 Verification Phase . . . 80

III Dishonest Majority **82**

7 Passive Security for Dishonest Majority **83**
7.1 Additive Secret-Sharing . . . 83
7.2 Protocols for Secure Multiplication . . . 84
7.2.1 Product-to-Sum Conversion . . . 84
7.2.2 Product-to-Sum Conversion Based on Homomorphic Encryption . . . 85
7.3 Preprocessing Model . . . 86
7.4 Offline Phase . . . 86
7.5 Online Phase . . . 87

8 Active Security for Dishonest Majority **88**
8.1 Integrity via Pairwise MACs 89
8.1.1 Reconstructing secret-shared values 89
8.1.2 Local Operations 90
8.1.2.1 Addition/Subtraction. 90
8.1.2.2 Canonical shares of public values. 91
8.2 Integrity via Global MACs 91
8.2.1 Reconstructing Secret-Shared Values 92
8.2.1.1 Partial openings. 92
8.2.1.2 Commit-and-open. 92
8.2.1.3 Reconstruction. 93
8.2.2 Local Operations 94
8.2.2.1 Addition/Subtraction. 94
8.2.2.2 Canonical shares of public values. 94
8.3 Online Phase 94

Bibliography **96**

Introduction

A big part of the theory and practice of cryptography is devoted to the study of different technologies deployed in our world today. Standard topics include the task of encryption, which relates to hiding information, but it is also common to consider digital signatures and message authentication codes to ensure integrity, enable authentication and authorization, and many other subjects relevant for today's infrastructure. For centuries, these were essentially the main tasks associated with the idea of securing information and communication, and this continues being the case today—of course, with the added complications of digital and worldwide-distributed technologies. Research into correct implementations and deployments of these tools, possible attacks, improvements, enhancements in user experience, adaptation to modern more technologies and scenarios, and other relevant questions, is of high importance.

However, a big portion of the field efforts is devoted not to current systems or natural questions like simply hiding information or ensuring integrity. In recent decades, researchers in cryptography have been pushing the barriers of science by considering new problems and challenges that were not even imagined in previous times. A good example is the idea of *zero knowledge proofs*, which provides us with the ability of proving properties of certain given data, without revealing the data itself. For instance, an individual could be able to prove to a third party that the balance in his or her bank account is above a certain threshold without revealing the exact amount, or it could be able to prove knowledge of the answer to a given question or challenge, without announcing the answer itself. This may sound counterintuitive and perhaps even impossible, but such techniques have been under research for decades already, and it is fair to say that today they are crossing the boundary towards the real world due to their wide range of applications, most notably within the blockchain and cryptocurrency domains (which, on their own, are interesting applications of cryptographic techniques that are gaining popularity quite rapidly). This is because, in simple terms, zero knowledge proofs have the potential of enabling users to prove that certain transaction was made, without revealing the specifics of the transaction itself, among several other use cases. For more information on this, see for example [29].

Another interesting concept that at first glance seems impossible to instantiate is the idea of *computing on private data*. As we have previously commented on, it is natural to expect certain type of information to be kept hidden, such as monetary transactions, balances, highly-sensitive information, login credentials, medical records, private conversations, and many, many others. However, it is commonly the case that, although this data has to be kept private from unauthorized agents, it is ultimately revealed in one way or another to be processed by a legitimate party, and in general, the situation in which data has to be kept hidden perpetually is rare. In many cases, however, it is not the full hidden data that is needed to be retrieved, but rather, a derived property from this data. As an example, consider a set of encrypted medical records from a hospital. In order to compute the proportion of users having certain medical condition, the hospital (or the party executing this analysis) would need to decrypt all records and then perform the computation, potentially revealing much more information than simply the intended proportion.

In today's world, more and more information is collected every single day. With more devices recording data every single minute, and more information about individuals being stored as they interact with new services and technologies, information has never been so plentiful. However, not all of it is readily available due to security or privacy concerns, as illustrated by the different examples above. Researchers in cryptography have studied the task of performing computation over hidden or sensitive data in order to remove this obstacle, and enable advances in multiple disciplines without violating the notion of securing information. To achieve this, multiple ideas applicable to different settings have been proposed. For example, *functional encryption* enables users to encrypt their data, and derive special "keys" that can be used to decrypt not the entire information, but derived data like, for example, only a small piece of the full information or operations carried out on these.

Homomorphic encryption on the other hand promises its users to be able to encrypt, or completely hide data, while still being able to perform certain computations on these. The result of this process leads to an encryption of the result of the computation, which can then be decrypted. This way, throughout the whole procedure the data is always hidden, and the only value that is revealed is the final output. Naturally, this is a very powerful tool with countless applications, and it has received enormous attention from the cryptographic community. Unfortunately, in spite of inspiring recent breakthroughs like the first actual construction of such a scheme [26], and in spite of multiple works proposing new constructions and improving over previous ones, the efficiency of these techniques is still too low for a wide range of relevant applications. However, many use cases are already within reach, and the field evolves in a very rapid manner, so a more widespread adoption of these tools can be around the corner.

Another tool that aims at enabling data analysis and aggregation on private data, that is gaining a lot of popularity in recent years due to its simplicity, is *differential privacy* [23]. The main idea behind this technique lies in adding small "noise" to the data in such a way that information about individuals cannot be discerned from the published records, but overall, certain aggregate data can still be computed. The result will be only approximate, as it contains some errors derived from the noise added to individual records, and there is a trade-off between the level of privacy provided and the precision of the result.

Many other advanced techniques in cryptography with several relevant applications are of interest to researchers, and some of them are slowly making their way into our real world. Of particular interest to us is *secure multiparty computation*, which is proposed as an alternative tool instantiate the dreamful task of performing computations on hidden data. We discuss this set of techniques below.

Secure Multiparty Computation

Consider the following scenario. There is an individual, *Alejandra*, applying for a loan at a *Bank*, and for doing so she needs to present a lot of information about her, like her income, savings, investments and possessions. Alejandra is a very important businesswoman, and she does not want all of this information to be held by the bank in the case she gets rejected, so she asks the bank to publish the algorithm they use to determine whether she applies for the desired credit so that she can check her situation first. On the other hand, the bank developed one of the most advanced credit analysis algorithms, and it is not willing to do this as that might imply they lose an important asset.

Imagine also this setting. A gender-equality campaign promoted by an NGO in certain city aims at determining whether men and women are being paid equally, in average, across several companies in the area. For doing so, the NGO would like these companies to send the information regarding salaries and benefits so that they can compute the desired statistics. However, although the NGO is only interested in learning averages across all companies, and although these might agree with the idea of calculating such figures, they do not feel comfortable sending all this information to a third party such as the NGO, which could harm their operations.

Both of the examples above constitute cases of computation on data that is intended to be kept private, and these can be brought together under the following description. There are n parties, $P_1, \ldots, P_n$, each P_i having an input x_i that they wish to maintain secret. Furthermore, there is a function $f(\mathtt{X}_1, \ldots, \mathtt{X}_n)$, and the parties want to learn the result $z = f(x_1, \ldots, x_n)$. In the first example, $n = 2$, x_1 is the set of records from Alejandra, x_2 is the bank's algorithm and $f(\mathtt{X}_1, \mathtt{X}_2)$ is the function that applies algorithm $\mathtt{X}_2$ to $\mathtt{X}_1$, and in the second case n is the number of companies, each x_i is the information regarding employee salaries of the i-th company, and f is the function that determines average income based on gender.

As in general with the problem of computing on hidden data, the tasks described above seem hard to solve without the involved parties willing to share their data. However, as we have mentioned already, researchers in cryptography have been working on different technologies to enable this type of computations. The parties in the scenarios above could in principle resort, for example, to homomorphic encryption techniques, encrypting their inputs and computing on the corresponding ciphertexts, but for reasonably meaningful functions $f(\cdot)$ this could be simply too inefficient. Alternatively, it is possible that differential privacy techniques can help, specially in the second scenario where the NGO wishes to compute different statistics on data coming from different parties, but this could potentially affect the precision of the study.

A different approach, called *secure multiparty computation*, or MPC for short, aims at developing much more efficient solutions to the problem above that do not undermine precision or correctness. Observe for example the following in the Alejandra vs Bank scenario. Alejandra does not trust the bank to have her information completely, so she might send the bank a "hidden" version of her data. If homomorphic encryption is used, then the bank would be able to apply its analysis algorithm internally, but as we have already mentioned this could place an insurmountable computational barrier. Alternatively, we may notice that Alejandra, also interested in the output of the computation, can lend a hand to *interactively* compute the result together with the bank. A similar observation holds in the second scenario: although all the different companies could simply encrypt their records and send these to the NGO for computation,[1] we may notice that if the companies are willing to collaborate to jointly compute the function, savings in efficiency could be achieved

In contrast to other approaches like homomorphic encryption, which work by hiding data completely from anyone not holding a special key and allow anyone to compute on the hidden data, MPC leverages the fact that there are multiple parties in the computation, and these entities can together compute the desired function. This is achieved via an interactive protocol executed by the parties that guarantees privacy of the data throughout the whole computation. We will discuss in Chapter 1 the details of what these guarantees exactly mean, but for now, it suffices to say that if a function is computed using an MPC protocol, then all of its inputs remain private and only the result of the computation is revealed. Unlike the case of homomorphic encryption, the computation is carried out by the parties holding the inputs themselves and it cannot be delegated to any other entity holding "encryptions".[2]

Secure multiparty computation was introduced in 1982 when Yao presented the concept of *Garbled circuits* [41], which is a particular way of securely evaluating a function using MPC. Since then, many different approaches have been proposed by cryptography researchers in the literature, leading to a fruitful line of exploration that has produced many interesting theoretical and practical results. Today, we have a solid understanding in regards to what type of protocols with what form of security can exist, and many of the state-of-the-art techniques can be used already for a wide range of applications and use cases. For example, the gender-equality study described above actually happened in Boston [35].

Several other applications of MPC have reached the realm of our real world, such as the Danish sugar beet AUCTION in 2008 [8], the tax fraud detection process in Estonia in 2015 [7], and many more. Furthermore, many other relevant use cases are considered regularly by researchers, and several prototypes are already under development. These applications include, for example, custody of cryptographic material, training/evaluating ML models.[3], securing databases, secure statistics, e-voting, and many more. Finally, what is also interesting is that these technologies are getting attention from institutions, organizations, and companies beyond academia, with some notable examples being Google, VISA, Facebook, IBM, Intel and Microsoft. Moreover, many start-ups and well-established companies are aiming at developing products based on secure multiparty computation, such as Sharemind, Galois, Cape Privacy, Unbound tech, Partisia and Inpher.

[1]It is worth noticing that for the situation in which the computation is simple additions, which is potentially the case in this example, homomorphic encryption is actually quite efficient.

[2]This is not a real limitation in MPC since secure computation can still be outsourced, albeit with different security guarantees as in homomorphic encryption.

[3]See for example the blog post *Privacy-Preserving Training/Inference of Neural Networks, Part 2*. `https://bit.ly/3eRKlgM`

Security of MPC Protocols

In an MPC protocol the parties involved send messages to each other according to some specified set of rules. These rules instruct the parties on what to do at each step of the interaction, and they depend on the information known by the given party such as the secret inputs they hold, some internal randomness, the messages they have received in previous steps of the protocol, etc. For example, a rule for some party P_i might look like "after receiving a message m from party P_{j_0}, sample some random value and add it to m, multiply by your internal input x_i, and send the result to party P_{j_1}". Ideally, at some point of the interaction, certain rule will instruct each party how to obtain the intended result of the computation.

Intuitively, an MPC protocol is secure if, after the execution of the protocol, the output $z = f(x_1, \dots, x_n)$ is learned by the parties and nothing about their inputs $x_1, \dots, x_n$ (besides the output z itself) is revealed. Formalizing this notion, which is necessary in order to be able to reason mathematically about these constructions, turns out to be highly non-trivial. In fact, the development of an adequate model to argue about MPC protocols, on its own, constitutes one of the biggest achievements in the theory of MPC itself. We will provide in Chapter 1 a more complete intuition on how security is defined in MPC, together with a rather detailed but still high-level description of the *simulation-based security* paradigm, which permeates the theory of secure multiparty computation. For now, we will be content with the overall idea discussed in the paragraph below, which will enable us to introduce some basic terminology.

At a high level, the goal of an MPC protocol is to ensure that each party learns nothing in addition to the output z after their participation in the interaction. However, it might be the case that, by working together, certain parties can jointly learn additional information about the inputs from the remaining participants, which is an undesirable outcome. To handle this, we require MPC protocols to guarantee that, even if certain subset of parties collude, the privacy of inputs from the remaining parties are kept private. To model the idea of a subset of parties working together toward breaking the privacy of the other parties, we consider an *adversary* that *corrupts* a given subset of the parties. We can think of this adversary as a "coordinator" that is in charge of controlling some with the parties, or alternatively, we can think of the corrupt parties as simply being different identities of the same underneath entity.

It is reasonable to expect that, the more parties that are corrupted, the easier it is for the adversary to break the protocol. Given this, we measure the security of a protocol by the amount t of corrupt parties that it can tolerate before it breaks.[4] Some protocols will be able to tolerate an adversary that corrupts all-but-one parties while preserving the privacy of the non-corrupted—also known as *honest*—parties. However, other protocols will be designed to be secure as long as the adversary corrupts at most a minority of the parties, so $t < n/2$, providing no guarantees if an adversary corrupts more than $n/2$ parties. The latter setting is known as *honest majority*, while in contrast the former scenario is referred to as *dishonest majority*. As we will see throghout this text, the tools, techniques and ideas

[4]As we will see in Section 1.1, there are other ways of quantifying the adversary's power, but the current approach of simply counting the amount of corrupted parties is already very useful and intuitive,, on top of being widely used and quite standard.

used in each of these two settings are typically very different, and the types of guarantees that can be achieved in each case also vary.

Several other factors affect the security of an MPC protocol, like the behavior assumed by corrupt parties (where terms like *passive* and *active corruptions* appear), or "how private" are the inputs from honest parties towards the adversary (where the concepts of *perfect*, *statistical* or *computational security* show up). These different considerations are deferred to Chapter 1, and for now, it suffices to keep in mind that the security of MPC protocols is based on how hard it is for a set of colluding parties, coordinated by an adversary, to break the privacy of the inputs from the remaining parties.

Secret-Sharing-Based MPC

There are quite a few paradigms for designing secure computation protocols. For example, since the introduction of the idea by Yao [41], Garbled circuits have been improved in several directions and today they are much more efficient and promising than their previous counterparts. These techniques are useful when the parties are geographically separated and have a high latency connection between them given that it requires a small amount of communication rounds, although it typically demands high bandwidth. Another method consists of using homomorphic encryption techniques, which is again particularly well-suited for highly distributed computations. However, as we have discussed already, this tends to impose a computational burden. Finally, a very popular approach, which is the main technique for secure multiparty computation we will focus on in this document, consists of making use of *linear secret-sharing schemes*, which enable the parties to hold distributed versions of the intermediate results of the computation, maintain this invariant throughout the evaluation process, until the result is reached.

Arithmetic circuits. Importantly, a common and central idea across all of the techniques mentioned above consists of first representing the desired computation as an *arithmetic circuit*, and then designing a method to process such circuit securely. An arithmetic circuit is a representation of a function that consists of wires and gates. The input wires are fed with the inputs to the computation and then processed through gates to obtain the values for the internal wires, which account for the intermediate results of the computation, until the output wires are reached, where the result of the computation is obtained. More details are discussed in Chapter 1, but for now, it suffices to know that the gates represent operations over a ring, an algebraic structure admitting an addition and a multiplication operation. These constitute the allowed intermediary processes that can be applied to the data, which is itself represented as elements over this ring.

A typical choice of ring is the set of integers modulo a prime p, which constitutes what is called a *field*, where every non-zero element admits a multiplicative inverse. For example, the set $\{0, 1\}$ with the operations `AND` and `XOR` constitutes precisely a field (integers modulo 2), and arithmetic circuits defined over this structure, also known as binary circuits, are highly important in many use cases.[5] On the other hand, for applications involving integer arithmetic, it is common to consider integers modulo a large prime p given that, if the integers used in the computation can be guaranteed to be smaller than p, then reduction modulo p does not play any effect and the resulting arithmetic becomes in essence simple integer arithmetic, which is useful in numerous scenarios. In theory, arithmetic circuits over finite fields can be shown to be sufficient to represent *any* desired computation, so designing MPC protocols to securely compute arithmetic circuits is in principle enough to securely compute any functionality.[6]

Linear secret-sharing schemes. Intuitively, a secret-sharing scheme is a method to "split" a secret into shares in such a way that (1) if not enough shares are held, then nothing is learned about the underlying secret, but (2) certain minimum number of shares together completely determine the secret. In the context of MPC, secret-sharing schemes are useful since they enable the parties to hold shares of a secret in such a way that an adversary corrupting t parties—and hence learning their t shares—cannot learn anything about the hidden secret, but with the cooperation of some honest parties this secret could be recovered.

To securely compute a given function represented as an arithmetic circuit using a secret-sharing scheme, the overall procedure is the following. First, each party P_i, holding an input x_i, distributes shares of this value towards the other parties. The privacy properties of the secret-sharing scheme ensures that the adversary, who controls an unknown subset of t parties, cannot learn this value x_i for an honest P_i.

Now, after all the inputs are held in shared form by the parties, the next step is to execute the different steps in the circuit, obtaining shared versions of all intermediate values in the computation, until eventually the parties obtain shares of the output of the circuit, point in which the parties use the reconstruction properties of the secret-sharing scheme to learn the result. For example, if two secret-shared inputs are set to be added together, the parties execute certain procedure that enables them to obtain, from the shares of the inputs they hold initially, new shares whose underlying secret is the sum of the two original secrets. A similar situation holds for the multiplication of two shared values. A *linear* secret-sharing scheme allows the parties to process addition gates locally—that is, without interaction—by simply adding together the shares of the two secrets under consideration. Multiplication on the other hand requires more care, and in fact, it is not an overstatement to say that most of the hardness of designing a secret-sharing-based MPC protocols lies in obtaining a secure and efficient method to multiply secret-shared values.

[5] In addition, the reader might be familiar with binary circuits having gates like `AND`, `NOT`, `NAND`, etc. since they are quite common in low-level computer architectures.

[6] However, several relevant computations lack a "nice" representation as an arithmetic circuit, so this approach is far from practical. Fortunately, it serves as a central building block. We will have much more to say in this regard in subsequent sections, but for the purpose of this introduction it is sufficient to consider the task of securely evaluating arithmetic circuits as our main goal.

The Purpose of this Document

We see then that secret-sharing-based MPC consists of the parties first secret-sharing their inputs, then proceeding in a “gate-by-gate” fashion to obtain shares of all intermediate wires/values of the computation until shares of the outputs, that can be reconstructed, are reached. This is a general recipe that leaves many questions up for consideration; things like: what exact secret-sharing scheme to use? How do the parties secret-share their inputs and reconstruct the outputs in a way that guarantees correctness? How do the parties process the different gates in the circuit? The goal of this document is to provide detailed answers to these and many more questions, which ultimately results in the specification of several of MPC protocols and techniques. These questions (and their potential answers) depend heavily on multiple factors such as the type of security desired, the power assumed by the adversary, the networking model, the efficiency metric being optimized for, among others.

Secure multiparty computation is a very exciting field, combining ideas from a wide range of disciplines, gaining more and more traction as more applications and use-cases become within reach, and getting attention by researchers and practicioners from all sorts of regions from both academia and industry. Things change shape constantly, optimal protocols become “outdated” almost yearly, and it is very hard to keep track of state-of-the-art techniques. These different aspects, together, make it very hard for new comers to learn about the basics of the field. Furthermore, even if one could specify a set of essential and fundamental reading resources like research papers, books, workshops and schools, a challenge that remain is learning the necessary theory of simulation-based security, in order to be able to appreciate and follow security proofs in this domain.

Motivated by the above, this document serves as a general guide to multiparty computation based on secret-sharing, focusing more on practical aspects of the techniques and constructions rather than their theoretical grounds. As we have already hinted at, simulation-based security is a rich and fruitful theory that deals with the task of properly formalizing MPC protocols, and we do discuss this framework along with some of its core concepts in Section 1.2. However, we clarify that, at least with the early versions of this document, the main intention of this text is to provide a solid understanding of some of the most relevant techniques to achieve MPC based on secret-sharing primitives, so the reader can expect more emphasis on the actual tools and methods rather than the (highly important!) “clutter” that surrounds research papers in order to formally argue about these constructions. Our motto is that simply understanding the *how* of secret-sharing-based computation can be done in a relatively clean, simple and self-contained way, if one is willing to relax the *why* these techniques work by being content with intuitive correctness and security arguments.

This text arises as a by-product of the author’s PhD Thesis, which can be found in this link: https://www.escudero.me/pdfs/phd_thesis.pdf. The thesis introduces new techniques to design MPC protocols when the underlying algebraic structure is the ring of integers modulo a power of two, and presents formal and full-fledged proofs of their security. Some of the approaches followed in the thesis resemble the traditional and standard techniques presented here, and a reader who is interested in guided formal simulation-based proofs is invited to visit that resource (at least until these full-fledged proofs make it to this document). Finally, we remark that this document is work in progress, and it will be ideally updated on a regular basis. For an up-to-date version, check the IACR Eprint repository and/or the author's website: escudero.me.

Organization of the Document

This text is divided into three main parts, as follows:

Part I: MPC Fundamentals. This first part presents the necessary tools to be able to reason and argue about MPC protocols and their security in a formal and rigorous manner. It is made of the following chapters:

Chapter 1: The Theory of Multiparty Computation. This chapter presents some basic concepts in MPC that are necessary to formally argue about protocols and their security. This includes the description of the simulation-based security model considered in MPC, some essential impossibility results, and some general approaches to the design of MPC protocols. The contents of this chapter are not tied to secure computation based on secret sharing.

Chapter 2: Secret-Sharing-Based MPC. Here, an overview of what a linear secret-sharing scheme is, and how it can be used to achieve secure multiparty computation, is provided.

Part II: Honest Majority. Secure multiparty computation techniques can be categorized in different ways depending on a wide range of factors like the intended security, the network conditions, the power of the adversary, among others. In this text, however, we take as a main distinguishing factor whether the adversary corrupts a minority or a majority of the parties, since techniques for each one of these two cases tend to be substantially difference. This part deals with the first case in which there is a corrupted minority, or, in other words, an honest majority. This part includes the following chapters:

Chaper 3: Shamir Secret-Sharing. An essential construction of a linear secret-sharing scheme, called Shamir secret-sharing, is provided in this chapter.

Chapter 4: Passive and Perfect Security for Honest Majority. This chapter presents a perfectly secure protocol with passive security that is secure against an adversary corrupting t parties, where $t < n/2$.

Chapter 5: Active and Perfect Security for Two-Thirds Honest Majority. In this chapter we consider a protocol for active security with abort that still attains perfect security, at the expenses of tolerating t parties with a smaller bound of $t < n/3$

Chapter 6: Active and Statistical Security for Honest Majority. We return our attention to the maximal adversary honest majority setting ($t < n/2$) and present an actively secure protocol with abort that achieves statistical security.

Part III: Dishonest Majority. In this part we discuss techniques in the setting in which the adversary corrupts potentially all but one party, so $t < n$, which is also known as the dishonest majority setting. This includes for example the case of two parties and one corruption. Currently, all the protocols considered in this part are set in the preprocessing model where certain correlation data is assumed, and no attempt to instantiate this phase is done at the moment.

This part includes the following chapters:

Chaper 7: Passive Security for Dishonest Majority. This chapter presents a protocol for MPC in the dishonest majority setting (in the preprocessing model).

Chapter 8: Active Security for Dishonest Majority. This chapter extends the protocol from Chapter 7 to achieve active security with abort making use of message authentication codes, or MACs.

Other Learning Resources

As we have already mentioned, the field of secure multiparty computation changes rapidly, and partly because of this, and also due to its relative young age, the area does not have a condensed and well-established reduced set of to-go resources, at least when compared to other disciplines. However, we can find a wide range of reading material such as tutorials, books, courses, videos, blogs, etc. Instead of listing these, we refer the reader to Drago Rotaru's *Awesome MPC* compendium: `https://github.com/rdragos/awesome-mpc`, which does an outstanding job at collecting relevant learning resources in the domain of secure multiparty computation.

We remark also that an excellent resource that closely resembles our "less-formal-more-practical" approach is *A Pragmatic Introduction to Secure Multi-Party Computation* by David Evans, Vladimir Kolesnikov and Mike Rosulek.[7] We note however that these resources are complementary to each other: *Pragmatic MPC* focuses more on dishonest majority techniques and Garbled Circuits, while the current focus on this text is on secret-sharing-based protocols.

Future Additions

This is a work-in-progress document, and as such it is bound to drastically change in subsequent iterations. The following are some of the planned additions/modifications. This list is intended to change as the document is extended.

- *Concrete constructions for a small number of parties.* Although protocols supporting any number of parties exist, in many cases, when the number of parties is small (say,

[7] `https://securecomputation.org/docs/pragmaticmpc.pdf`

between 2 and 5 parties), more efficient and usually simpler protocols tailored for these settings can be designed. It is our goal to include some of these protocols in the future.

- *Building blocks for computations beyond arithmetic circuits.* Although one can learn a lot by restricting to arithmetic circuits, in practice several meaningful computations are not described as such. Something so seemingly simple as performing scientific computation over the reals, evaluating mathematical functions such as tangents and logarithms, or even branching or bit-decomposing values, can be a daunting task when performed in MPC. Our goal is to include several techniques in the literature to deal with these—more practically oriented—challenges.

- *Concrete applications of MPC.* We would like to include several meaningful and efficient applications of MPC.

- *Instantiating the preprocessing for dishonest majority MPC.* With a few exceptions, most of the current document is concerned with information-theoretic constructions rather than computationally-secure ones. Techniques in the latter domain tend to be of an entirely different nature, given that they involve computational assumptions and therefore make use of heavy mathematical machinery to be able to define hard computational problems and derive meaningful cryptographic constructions from them. It is our objective to include a reasonably comprehensive set of techniques and results in this direction.

- *Garbled circuits.* As we have already mentioned, garbled circuits are a different approach to achieve (computationally secure) multiparty computation. Although it tends to differ from secret-sharing-based MPC, in several cases mixing the two leads to very useful results. It is in our plans to include a discussion on this in the future.

- *Comprehensive introduction to simulation-based security and more formal proofs.* Although we have already stated that the goal of this work is to focus on more practical aspects of secure multiparty computation, emphasizing techniques and results rather than theoretical frameworks and proofs, we still believe it is highly important for the reader to be aware of the existence of the notion of simulation-based security, and due to this we discuss this topic briefly in Section 1.2. However, we currently do not provide any formal proofs of the constructions presented in this work.

  For the interesting readers we plan to include in the future some sections that further expand on the topic of formal security proofs, possibly including different simulation-based frameworks, impossibility results, full-fledged guided proofs of many essential constructions, among others. Nevertheless, this would be done in separate sections in order to not disrupt the main purpose of the text, which is to provide new researchers and practicioners with a simple and relatively comprehensive resource for learning about how the different techniques in MPC work.

Other minor TODOs include the following:

- Include a detailed list of preliminaries, and if possible, a section with elementary background.

- Include more figures and graphs to better illustrate some concepts and ideas.
- Add more references, "further reading" subsections for each relevant part, as well as historical notes.
- Include small subsections with high level explanations and pointers on more advanced topics.
- Although this document has been reviewed thoroughly, the fact that it comes from a larger text may make some sections incoherent. This will be fixed in subsequent iterations.

General Preliminaries and Notation

Most of the notation used in this paper will be introduced "on the fly", that is, it will be presented in each relevant section where it is used for the first time. However, some general notation that we can introduce from now consists of the following:

- The set $\mathbb{Z}/M\mathbb{Z}$ denotes the ring of integers modulo M, whose representatives are taken over the set $\{0, \dots, M-1\}$.
- All vectors, denoted by bold lowercase letters like $\boldsymbol{x}$ and $\boldsymbol{y}$, are column vectors by default. This is particularly relevant when dealing with multiplications with matrices.
- $x \in_R A$ means that x is sampled uniformly at random from the set A.
- For a positive integer ℓ, $[\ell]$ denotes the set $\{1, \dots, \ell\}$.
- For a k-bit integer x, we denote by $(x[k-1], \dots, x[0])$ its bit decomposition, that is, $x[i] \in \{0,1\}$ for $i \in \{0, \dots, k-1\}$ and $x = \sum_{i=0}^{k-1} x[i] \cdot 2^i$.

Part I

MPC Fundamentals

Chapter 1

The Theory of Multiparty Computation

The goal of this initial chapter is to introduce the reader to some of the most relevant existing theoretical concepts in the area of secure multiparty computation. First, in Section 1.1, we present a general and high level introduction to some of the most important concepts and ideas in MPC, like the notion of an adversary, and different settings and goals that are typically considered in MPC. Then, in Section 1.2, we present the idea of simulation-based security, which is the formal mathematical machinery necessary to properly define the concepts discussed in Section 1.1. This tool allows us to approach the task of securely computing a given function from a mathematical point of view, enabling us to obtain precise and explicit security results. Finally, we discuss in Section 1.3 some of the most relevant results in the theory of secure multiparty computation, which are related to the types of security notions that can be achieved in the three main distinctive settings: two-thirds honest majority, honest majority, and dishonest majority.

1.1 A General Introduction to MPC

In secure multiparty computation we consider a setting where n parties $P_1, \ldots, P_n$, each P_i having an input x_i, want to securely compute a given function $f(\mathtt{X}_1, \ldots, \mathtt{X}_n)$, in such a way that only the value $z = f(x_1, \ldots, x_n)$ is learned, and nothing else about $x_1, \ldots, x_n$ is revealed. This is intended to be achieved by means of an *MPC protocol*, which is a set of rules that the parties execute, involving some local computation plus some exchange of messages. These rules depend on the inputs of each party, and they are typically randomized, meaning that they usually depend on some random bits sampled by each party as well.

Ideally, nothing should be learned about the inputs $x_1, \ldots, x_n$, except perhaps from what is leaked about the output z.[1] Towards formalizing this notion, it is useful to think of an *ideal world* in which there is a *trusted third party* who receives the inputs from the parties, computes the result z, and sends this to all the participants, promising to perform the correct computation and not to leak absolutely anything else besides z. The goal of a secure multiparty computation protocol is to instantiate this type of scenario without

[1]Notice that it might be the case that the output z reveals a lot of information about some of the inputs, which is obvious for instance if the function f is defined as $\mathtt{X}_1 = f(\mathtt{X}_1, \ldots, \mathtt{X}_n)$. This is acceptable in the context of MPC given that the only goal is to protect the inputs $x_1, \ldots, x_n$, *except possibly* for what is leaked by the output z itself.

the presence of a trusted third party, only involving communication among the parties holding the inputs themselves. In other words, a protocol must match the behavior of the ideal world in what we call the *real world*. As we will see in Section 1.2, this ideal/real world paradigm is not only useful to understand at a high level what the guarantees of a secure multiparty computation protocol should be, but it also serves as the core idea behind a proper mathematical formalization, which enables the use of rigorous definitions and theorems.

1.1.1 Adversaries and their Power

In a wide range of notions and constructions across all of cryptography, it is very common to formalize the idea of something being "secure" by considering the idea of an *adversary*, which is an entity who tries to break whatever property we are trying to protect, and should not be able to succeed with reasonable probability. This adversary is simply an algorithm, a mathematical object that can be formally defined. For example, in the case of encryption it is common to define security (at least, one particular notion among several other variants) by requiring that no (typically efficient) adversary can win at a "game" that is supposed to represent a real-world scenario where an attacker gets to interact with an already-deployed encryption scheme. In this game, the adversary gets to choose two different messages, and it gets an encryption with an unknown random key of one of these two plaintexts, chosen at random. It is the adversary's goal to determine which of the two messages was encrypted. If no adversary, which in essence means no algorithm, is able to significantly win at this game, then, intuitively, it must be the case that the encryption scheme is good at its job of hiding data, since encryptions of different messages look indistinguishable.

Many other notions in cryptography, like the security of digital signatures or key exchange mechanisms, are formalized via adversaries attacking the system, and secure multiparty computation is naturally no exception. Consider an execution of a secure multiparty computation protocol where n parties $P_1, \dots, P_n$ engage in a set of communication and computation rules, exchange messages, and return a result at the end of this interaction. Who should be the adversary in such scenario? As the name implies, an adversary is a "rival" whose aim is to break a given security property we are trying to maintain, in this case, the fact that the inputs of the parties remain private, except from the output z. For example, we can consider one of the parties in the execution of the protocol to be the adversary, and what we could require is that this party, after the interaction with the other participants, this "adversarial party" does not learn anything about the other parties' inputs, besides what is leaked by the output of the computation.

Let us assume that a given secure computation protocol satisfies the notion that no party, regarded as the adversary, can learn anything about the inputs from the other parties, as considered above (assume for now that we can appropriately define the idea of "not learning more than what is leaked by the output", which is achieved via simulation-based security as considered in Section 1.2). A natural question is, what would such notion reflect in practice? In principle, it is very powerful: if any of the parties behaves as an adversary, trying to learn anything from the other parties' inputs (besides what is leaked by the output), this party will fail at doing so. However, it fails to capture a very important "attack" that could easily happen in a practical scenario, and it has to do with the possibility of a *collusion attack*. Imagine that, among the n parties participating in the protocol, there are two which, for different reasons, might benefit from learning the inputs from the remaining parties. These two parties may decide to trust each other and *collude*, that is, work together, perhaps by sharing out-of-band messages to each other, if this somehow helps them in their task of learning information they are not allowed to gather. Alternatively, if by joining the internal data of different parties it is possible to break privacy, then an external attacker that breaks into several of the participating machines can learn private information.

In light of the scenario described above, which could easily appear in practice, it becomes important to somehow incorporate into our adversarial definition the idea of parties colluding, working together, sharing information to each other, in order to break the privacy of the remaining parties. One could in principle achieve this by considering different adversaries that somehow communicate to each other, but it turns out to be a much simpler way if, instead of following this approach, we consider a *single* adversary, as before, that, instead of simply playing the role of an individual party, it completely controls a given set of parties. This attacker plays a similar role as the "hacker" described in our previous example, and it also serves to model the case in which two or more parties collude voluntarily, since the strategy they follow in their collusion process can be modeled as an algorithm, which can be ultimately regarded an adversary on its own. Finally, notice that this notion strictly generalizes the one we considered initially above, where a protocol was secure if no single party acting as an adversary could violate privacy. This case corresponds to the scenario when an adversary corrupts a set containing only one single party.

With this idea of what the adversary role is in the execution of a secure multiparty computation protocol, we proceed to describe and categorize many of the possible different variants that such adversary can present. Before we do this, however, we introduce some notation that will be used throughout this work. Suppose that the n parties participating in a given protocol execution is $P_1, \dots, P_n$. The set of indexes in $[n]$ corresponding to corrupted parties is denoted by $\mathcal{C}$, while the set of indexes corresponding to the remaining parties, which are also called *honest parties*, is denoted by $\mathcal{H} = [n] \setminus \mathcal{C}$.

1.1.1.1 Possible Corrupted Sets

In our first naive notion of security a protocol was considered secure if *any* adversary, corrupting *any* single one of the parties $P_1, \dots, P_n$ participating in the protocol, cannot learn anything about the remaining parties' inputs. Observe that in this case the protocol should remain secure no matter what party is corrupted. If, for example, we only require security against adversaries corrupting one of the parties among $P_2, \dots, P_n$ (so P_1 is never corrupted), a simple and trivial protocol would consist of the parties sending their inputs to P_1, who computes the function and returns the result. This satisfies the security notion since we only require that the adversary does not learn any extra information about the honest parties' inputs, which is trivially guaranteed since P_1 is always honest. We see then that, depending on which parties are allowed to be corrupted by the adversary, different protocols might exist.

As we have mentioned in previous paragraphs, in our more general context the adversary is not restricted to corrupting one single party but rather it can corrupt a set of parties potentially having size greater than one. However, in order to define security, we require the protocol to protect the privacy of the honest parties' inputs in the event in which the adversary corrupts a given set of parties. A crucial question that appears under this consideration is then the following: which sets of parties can the adversary corrupt? In the single-corruption case from above these possible sets are $\{P_1\}, \dots, \{P_n\}$, and as before, if the collection of possible sets the adversary can corrupt is too simple (e.g. all the possible sets miss one specific party, which means that this party is always honest), the task of designing secure multiparty computation protocols may become trivial.

In general, the collection of possible sets the adversary can corrupt is a security property of a given protocol. Such collection, which is simply a set of subsets of $[n]$, is called an *adversarial structure*, and different secure multiparty computation protocols are designed with the goal of withstanding corruptions from different adversarial structures. Below we consider some relevant adversarial structures. Before we do this, however, we note that, if a set B is part of a given adversarial structure, then we must include every subset $A \subseteq B$ into the structure as well, given that a protocol cannot be secure against corruptions in A if it is already insecure when the adversary corrupts a smaller subset. Given this, we define adversarial structures as antimonotone collections of subsets of $[n]$, meaning that if B is in the collection, every set $A \subseteq B$ has to be part of it too.

$Q2$ and $Q3$ Adversarial Structures. The ability to consider general adversarial structures is very useful in scenarios in which there is a lot of asymmetry among the "importance" of the different parties. For example, consider a setting with three parties P_1, P_2, P_3, and suppose for simplicity in the argument that they have no reasons or motivations to voluntarily collude. However, there is still the concern that an attacker breaks into some of these machines. Suppose now that P_1 is very well protected, but P_2 and P_3 have a weaker safeguards. In such a setting we might consider a protocol that withstands the adversarial structure $\{\{P_1\}, \{P_2\}, \{P_3\}, \{P_2, P_3\}\}$. This way, if the adversary wants to break the system then it has to corrupt a set of parties outside this structure, so either P_1 *and* one of P_2 or P_3. Since $\{P_2, P_3\}$ is part of the adversary structure, even if the attacker breaks into the two weaker machines P_2 and P_3, it cannot still breach privacy.

There is a long and important line of study into how to design secure multiparty computation protocols for general adversarial structures, starting with the work of [15]. However, among all possible adversarial structures, there are two types that are particularly important. We will make more explicit the relevance of these two particular structures later in Section 1.3.

$Q2$ structures. An adversarial structure is $Q2$ if, for every A_1 and A_2 in the structure, $A_1 \cup A_2 \neq \{P_1, \dots, P_n\}$.

$Q3$ structures. An adversarial structure is $Q3$ if, for every A_1, A_2 and A_3 in the structure, $A_1 \cup A_2 \cup A_3 \neq \{P_1, \dots, P_n\}$.

Threshold Adversarial Structures. In settings in which there is more "symmetry" among the parties, and there are no obvious reasons to put more weight into how easy it is for one party to get broken into, or into how likely it is that a given party colludes, with respect to another party, a natural adversarial structure is the *threshold structure*. This measures the adversary's capabilities by how many parties it can corrupt simultaneously, without making any distinction among the parties whatsoever. More concretely, a threshold adversary structure is parameterized by a value $0 < t < n$, and it consists of all the subsets of $[n]$ of size at most t. Intuitively, a protocol that is secure against such structure guarantees privacy of the honest parties' inputs as long as the adversary does not corrupt more than t parties.[2]

Below we discuss three types of threshold adversarial structures, depending on their threshold value t. The main importance of these distinctions will be made clear in Section 1.3, when we discuss several fundamental results in the feasibility of secure multiparty computation protocols depending on each of these cases.

Honest majority, $t < n/2$**.** In this case the adversary is assumed to corrupt at most $t < n/2$ parties, so, no matter what set of corrupted parties is chosen, the set of honest parties constitute a majority. It is easy to verify that the resulting adversarial structure is $Q2$.

Two-thirds honest majority, $t < n/3$**.** Now the adversary is assumed to corrupt even less parties, at most $t < n/3$. Here the set of honest parties is always at least two-thirds the total number of parties. It is easy to verify that the resulting adversarial structure is $Q3$.

Dishonest majority, $t < n$**.** This is the scenario where the no special bound on t holds, so t can take the largest possible value, $t = n - 1$. In this case the adversary can corrupt any set containing all but one party, and a protocol secure in this setting would still guarantee privacy to this remaining party. This is the strongest possible setting: intuitively, each party knows individually that their inputs are secure, even if all the other parties collude. This is not the case with any of the previous scenarios (and in general, if $t < n - 2$), since in these cases the adversary can break the privacy of an individual party's input by corrupting a set with at least $t + 1$ parties.

We remark that, throughout this document, our only focus lies in threshold adversarial structures.

Remark 1.1 (Maximal vs non-maximal adversaries)**.** *Intuitively, when designing secure multiparty computation protocols in any of the threshold settings listed above, it is better to consider the maximum possible value of t that respects the given bound. For example, in the honest majority setting where $t < n/2$, the best is to choose t as the largest integer that respects this bound, i.e. $t = \left\lceil \frac{n}{2} - 1 \right\rceil$, since in this case the resulting protocol tolerates the largest number of corruptions while still falling within the honest majority setting.*[3]

Motivated by the above, it is quite common, and in fact, we do so in several opportunities in this document, to assume for simplicity that the adversary corrupts exactly t parties, and that t is as large as possible while respecting the bound under consideration. At an intuitive level, this should be without loss of generalization given that, if a protocol is secure against an

[2]Notice that t lies between 1 and $n-1$. If $t = 0$ then there are no corruptions and the task of secure multiparty computation becomes trivial. Also, if $t = n$, then all parties are corrupted so there are no honest parties' inputs to protect the privacy of.

[3]It is important to mention that having a gap between t and $n/2$ (or $n/3$) is sometimes useful as it allows the use of*packed secret sharing*, a technique to improve efficiency of MPC protocols in these scenarios [18, 24].

adversary that corrupts exactly, say, $\lceil \frac{n}{2} - 1 \rceil$ parties, then the protocol should remain secure even if the adversary corrupts less, since this means that now the adversary is less powerful.

Unfortunately, although such reasoning makes a lot of sense, the mathematical model under which the concept of MPC is formalized, which is discussed in Section 1.2, does not satisfied this property. More precisely, there are protocols that are secure against $\lceil \frac{n}{2} - 1 \rceil$ corruptions, but an adversary corrupting less than this amount can somehow break the protocol. This counter-intuitive nuisance is hardly an issue in practice, but it is important to be aware of it. We revisit this issue when we assume a maximal adversary in this document.

1.1.1.2 Type of Corruption

Our current corruption model is intended to represent a set of parties colluding, exchanging messages out-of-band, sharing their internal state, and possibly coordinated by an attacker, which is in fact how the proper adversarial model is formalized. Recall that a secure multiparty computation protocol is in essence a set of computation and communication rules that the parties have to follow in order to securely compute a given function. So far, although the adversary is able to see all the internal state of the corrupt parties, including messages received and sent by these, we have implicitly assumed that the corrupt parties follow the rules specified by the protocol faithfully. This corresponds to the notion of a *passive* of *semi-honest* adversary, and it is intended to reflect a setting in which all of the parties are assumed to follow the protocol instructions, but even if an attacker is able to access the internal information from a given subset of the parties (within certain adversarial structure), the protocol should guarantee privacy of the inputs from the remaining parties.

Unfortunately, it is in principle not possible for the parties to somehow verify that the other participants are following the protocol specification faithfully. This is because, ultimately, all the different parties see from other participants are messages which, although depend on their private inputs, are supposed to reveal nothing about these values from the security definition itself. From this, if an adversary can gain additional information by modifying the *behavior* of the corrupt parties during the execution of the protocol, perhaps in a way in which such misconduct goes undetected towards the honest parties, a need to protect secure multiparty computation protocols against such actions appears.

An adversary with the more advanced and realistic capabilities described above is referred to as an *active* or *malicious* adversary, and protocols that are secure against such type of adversarial behavior are the most ideal to deploy in much of the potential use cases for secure multiparty computation, given that it prevents corrupt parties from causing any harm during the execution of the protocol, even if they internally deviate from the specified rules, which in principle they would be able to do without being detected. However, although these protocols are stronger than their passively secure counterparts, they are naturally much harder to construct, plus they tend to add certain overhead in terms of performance.

So, to summarize:

Passive/semi-honest corruption. An adversary is said to be passive if the behavior of the corrupt parties during the protocol execution is exactly as specified by the protocol description. The adversary sees the internal state of the corrupt parties, including in/out messages, input and internal random coins, but it cannot alter their behavior.

Active/malicious corruption. An adversary is said to be active if it controls the corrupt parties completely, including possibly modifying their actions during the execution of the protocol.

In this work we will consider both passive and active adversaries.

1.1.2 Privacy Guarantees

Recapping what we have seen so far, our intuitive security definition for secure multiparty computation protocols requires that an adversary, corrupting (either passively or actively) a set from an adversarial structure, which is simply a collection of possible corruption sets, learns nothing from the honest parties' inputs, which, as will be made more precise in Section 1.2, is formalized by requiring that the protocol execution somehow looks "close" to an ideal world in which a trusted third party is used. In this section we explore in a bit more detail what the concept of these two executions being "close" means. The description here is rather verbal and intuitive, and it is only made more precise in Section 1.2 where we properly define the idea of simulation-based security.

Perfect security. In this case the real and ideal executions follow the *exact same* distribution, so, from the point of view of the adversary, nothing is learned about the honest parties' inputs from the protocol execution. This is regardless of the computational resources available to the adversary.

Statistical security. This is a slightly weaker notion, and it is also called unconditional security. In this case the real and ideal worlds have *statistically close* distributions, meaning that the distributions from the real and ideal worlds may not be the same, but are *very* close. More precisely, by controlling certain parameter of the given construction, it is possible to shorten the gap between these two distributions by any desired amount. To illustrate what this type of security entails, it is useful to think, as an example, of an MPC construction that achieves the following: the real and ideal worlds follow the exact same distribution, except that there is a very small chance (regardless of its computational powers) in which the adversary can completely break privacy of the honest parties' inputs in the real world. In this case, security would be statistical.

Computational security. The two security notions above are very powerful, but as we will see in Section 1.3, they are not always possible to achieve. As a result, and motivated by practice, it is useful to restrict security to only *efficient adversaries*, that is, adversaries that make use of a bounded amount of resources, which are formally modeled as algorithms running in polynomial time (in terms of a security parameter). In a computationally secure protocol, the distributions from the real and ideal worlds are indistinguishable, but only as long as the adversary is not infinitely powerful, which is enough for practical use. As an example, consider an MPC protocol in which some party has to send an encryption of its input using a secret key. Although this might be hard to break for an adversary not knowing the secret key, an attacker with infinite resources can brute-force the ciphertext to recover sensitive information.

The first two notions, perfect and statistical security, are commonly referred to as *information-theoretic security*. It is typically the case that protocols satisfying these notions of security tend to be more efficient than computationally secure ones, given that they are usually simpler and do not rely on certain parameters that aim at making a given computational problem hard. However, as we have mentioned, information-theoretic security is not always possible to achieve.

1.1.3 Output Guarantees

Recall that, to define security of secure multiparty computation protocols, we have resorted to comparing the real world, where the execution of the given protocol takes place, to an ideal world where the desired function is computed by a trusted third party, who receives the inputs from the parties and promises to reveal only the output. Defined in this way, the parties have the guarantee in the real world that their inputs are as protected in the actual interaction as in the ideal world, where only the output is revealed. However, another subtle property of the trusted third party is that, as described above, it *always* returns the correct output of the computation to all of the parties. Unfortunately, as we will see in Section 1.3, this notion, which is called *guaranteed output delivery* in the literature, is not always possible to achieve in the real world. In some settings, for example, the best that can be achieved is that if the parties obtain a result, it is guaranteed to be the correct one, but it could be the case that no party obtains any result at all.

From the above, it becomes necessary to relax the requirements in the ideal world regarding the output of the computation. To this end, we present the following three notions related to this. We remark that, in any of the three concepts below, whenever the parties obtain an output it is guaranteed to be the *correct* one.

Guaranteed output delivery. As described above, in this case all the parties are guaranteed to obtain the output, regardless of the actions that the corrupt parties perform.

Fairness. In this case it could happen that the adversary causes the honest parties to not obtain the output, which is referred to as causing the parties to *abort*. However, if this happens, then the corrupt parties (and hence the adversary) are guaranteed to also *not* learn the output.

Security with abort. In this scenario it can be the case that the adversary denies the honest parties from learning the output, while the corrupt parties may still learn the result. Furthermore, we identify two possible variants: in *security with unanimous abort* all the honest parties are guaranteed to either receive the output or abort altogether, and in *security with selective abort*, which is even weaker, the adversary can choose which honest parties receive output and which honest parties abort.

If the adversary is passive, the corrupt parties will behave exactly as if they were honest parties, following the protocol specification, so in this case it always holds that the protocol terminates with the correct output, which in particular means that, trivially, guaranteed output delivery is obtained.

Additionally, we note that, although the most desired property is guaranteed output delivery, which ensures availability of the output under all possible attacks, the notion of fairness is already very useful in practice, as the adversary does not learn or gain anything from stopping the (honest) parties from learning the output. For example, if the computation under consideration is distributed voting, a protocol that simply satisfies security with abort may allow the adversary to first learn the outcome of the voting, and decide to deny the honest parties from learning this if desired. A protocol with fairness, however, may simply disrupt the computation (which is of course a problem of a different nature on its own), but the adversary cannot decide to cause an abort depending on the output of the computation.

1.1.4 Different Functions to be Computed

Another important consideration when designing secure multiparty computation protocols is what type of computations they are intended to operate with. In this section we provide a general discussion on the topic, differentiating between several important types of computations.

1.1.4.1 Public-Output vs Private-Output

So far, our description of the function to be computed has been $f(\mathtt{X}_1, \dots, \mathtt{X}_n)$, with z being the output of applying this function to the inputs of the parties, $x_1, \dots, x_n$, which is the value that all parties learn at the end of the execution of a given secure multiparty computation protocol. This scenario is referred to as the *public-output* setting, since there is only one result, that all parties learn equally. Alternatively, we may consider the case where each party receives a different result. In this case, we regard the function as producing n different outputs $(z_1, \dots, z_n)$, where each party P_i is intended to learn (only) z_i. Clearly, this scenario, called the *private-output* setting, is a generalization of the public-output case (by taking $z_i = z$ for $i \in [n]$), but it is fortunately not much harder to achieve. The reason for this is that, given an MPC protocol for public-output functions, each party can simply provide as an additional input a secret-key only known to this participant, and the function to be computed can be modified so that all the outputs are returned to all parties, except that output z_i is "encrypted" under the key provided by party P_i. This way, only this party can recover its corresponding output.

In this document we sometimes consider the public-output case (specially when computing arithmetic circuits), and in some other occasions we consider the private-output setting (when in the context of the arithmetic black box model). See Section 1.2.6.2 for details.

1.1.4.2 Reactive vs Non-Reactive Functionalities

A *reactive* functionality is one that enables the parties to learn "partial results" of the computation, and continue the process in a way that perhaps depend on these intermediate results, plus possibly new inputs. A good example of this type of computations is given by, for example, a commitment scheme, which enables a party to commit to a given value without revealing its contents, to do so at a later stage without the ability of announcing a different value to the one committed earlier. On the other hand, in a *non-reactive* functionality the parties simply provide their inputs at the beginning of the protocol, and obtain the result at the end of the execution. There are many relevant applications that can be phrased as non-reactive functionalities, like data processing tasks, distributed voting and auctions, and many others.

Clearly, reactive secure multiparty computation is a more general setting than the non-reactive case. However, it is generally the case that one can obtain a secure multiparty computation for reactive functionalities from a protocol that only supports non-reactive ones by making use of a technique called *verifiable secret-sharing*. In short, this technique enables the parties to obtain a "shared state" of each checkpoint in a reactive computation using the non-reactive protocol, which can be reconstructed to obtain partial outputs. Although there are certain scenarios in which non-reactive computation is possible while reactive computation is not, it is generally the case that the two notions are back-to-back, so the difference between the two is typically irrelevant for the discussion of different secure multiparty computation protocols.

1.1.4.3 General vs Special-Purpose MPC

Another relevant distinction for secure multiparty computation protocols lies in whether they are designed to support *any* arbitrary function, or if they are tailored to specific functions. The former family is typically referred to as *general-purpose* MPC protocols, while the latter, being more targeted to particular computations, is called *special-purpose* MPC.

It is fair to say that most of the proposed secure multiparty computation protocols in the literature correspond to general-purpose constructions. However, at first glance, this may sound as an extremely difficult task: how can a single protocol support *any* arbitrary computation? The catch is in the way that computations are represented, which involves the concept of *arithmetic circuits*, discussed below.

Arithmetic Circuits and General-Purpose MPC. As we have mentioned above, researchers in the field of secure multiparty computation have focused mostly in designing general-purpose MPC protocols, which are intended to securely compute *any* functionality that the parties wish to compute. This is possible thanks to an abstraction known as *arithmetic circuits*, which concisely captures any possible function in terms of rather simple operations over certain algebraic structure.

Consider a finite field $\mathbb{F}$, and let $f : \mathbb{F}^n \mapsto \mathbb{F}$ be any function defined over this field. A well known fact in field theory is that *every* such function can be written as an arithmetic circuits, which in formal terms is simply a directed acyclic graph with labeled nodes. Denoting by (i, o) a node that has fan-in i and fan-out o, every node is either of the type $(0, 1)$, $(2, 1)$, or $(1, 0)$. $(0, 1)$ nodes are called the *input gates*, and they model the inputs to the computation. $(2, 1)$ nodes are call the *operation gates*, and they represent field operations. A field has two main operations, addition $(+)$ and multiplication $(\cdot)$, and this is reflected in the fact that there are two types of gates *addition* and *multiplication gates*, each corresponding to a different operation. Finally, $(1, 0)$ nodes are the *output gates*.

Edges are also called *wires*, and in an actual execution of the function f, each wire has associated a value to it corresponding to an intermediate result of the computation. Wires outgoing from an input gate are associated to the actual value provided as input, corresponding to the given input gate. Wires leaving an operation gate are matched with the values corresponding to the result of applying the corresponding operation (addition or multiplication) to the (associated values to the) incoming wires to the gate at hand. Finally, the wire that goes into the output gate is precisely the result of the computation.

With this tool at hand, designing a general-purpose secure multiparty computation protocol reduces to constructing a protocol for securely computing arithmetic circuits exclusively, which is what most of the research in the field of MPC is concerned with. Furthermore, among all different techniques to design secure multiparty computation protocols, there is a promising general template known as secret-sharing-based MPC that, in a nutshell, works by letting the parties have a “hidden” representation of the inputs to the computation, together with some methods to obtain a hidden version of the output of each operation gate, assuming the inputs are already hidden. Eventually, the output is reached in hidden form, point in which the parties can “reveal” it so that the result of the computation is learned. From this template, the task of secure multiparty computation reduces to designing a method to (1) keep “hidden” versions of different values, (2) obtain a hidden version of the result of an addition or a multiplication, assuming the inputs are already hidden, and (3) reveal a hidden value. This, at a first glance, seems more feasible than the daunting task of securely computing *any* conceivable function.

General-purpose MPC protocols are particularly useful in theory as they show what type of computations are possible, which, accompanied with impossibility results, provide us with a complete landscape of the sorts of computations we can hope for. However, their reachability in practice can be, in principle, more questioned. For instance, recall that these protocols work by first representing the desired computation as an arithmetic circuit over a finite field, which is simply a combination of additions and multiplications over this structure, but it is not at all clear what type of *practical* applications lend themselves to be *efficiently* expressed as an arithmetic circuit. As an starting example, applications that involve real-valued arithmetic, such as these in the domain of machine learning, for illustration, are more naturally written in terms of operations over the real numbers, including possibly additional processes that go beyond basic additions and multiplications (e.g. taking square roots, applying sine or cosine, or even non-mathematical operations such as flipping the bits of the given value in the bit-representation).

In spite of the above, many general-purpose MPC protocols are not a mere theoretical tool, as they tend to be the basis, or the starting point, of a wide range of more specialized protocols. This is achieved by adding certain subprotocols for specific operations that appear repeatedly across multiple applications, such as the case of real-valued arithmetic illustrated above. Additionally, the existence of highly-efficient general-purpose MPC protocols has allowed the creation of several *MPC frameworks* that enable a set of parties interested in securely computing a given function to achieve this task with little-to-none knowledge of secure multiparty computation. This is achieved by enabling computation over arbitrary computer programs that, in essence, resemble an arithmetic circuit with several available sub-operations added on top. Popular frameworks of this kind include MP-SPDZ [31], SCALE-MAMBA [1], EzPC [12], among others, and it is fair to say that these implementations play a pivotal role into taking secure multiparty computation techniques from theory to practice.

Special-Purpose MPC. A special-purpose secure multiparty computation protocol exploits particular properties of the given function in order to optimize the construction to the case at hand. This has major relevance in practice, but quite surprisingly, it also plays an important role in the theory of MPC.

First, in terms of practice, the benefits of considering special-purpose protocols are generally obvious: by exploiting the structure of the function to be computed it is typically the case that multiple savings in efficiency can be achieved with respect to the use of a more generic protocol for arbitrary computation. We remark, however, that many special-purpose MPC deployments have as a starting point more generic techniques to compute basic additions and multiplications, which come from the general-purpose MPC domain.

On the other hand, special-purpose secure multiparty computation constructions also have a tremendous impact in theory, by considering certain concrete functionalities. In general, the theory of MPC, as far as feasibility or impossibility results is concerned, is not interested in secure multiparty computation protocols for very specific tasks such as image analysis or distributed voting. This is because, at the end of the day, one of the major results in the field of MPC is that secure multiparty computation of *any* function is generally possible, so designing special-purpose for these practically-oriented tasks is not that relevant in this direction.

Instead of considering special-purpose MPC protocols as a way of improving efficiency of more generic constructions, the main contribution of these type of protocols in terms of theory lies in simplifying the construction of other, possibly more general-purpose, protocols. More precisely, different useful functions that turn out to be crucial for the development of other protocols are identified, and as a result, advances and developments in regards to MPC constructions for these primitives, specifically, lead to general improvements across all other constructions that make use of these concrete functionalities. A good example of a particular function that is not only particularly useful on its own, but also serves as a major building block in many other secure multiparty computation protocols, is Oblivious Transfer. In short, this is a two-party functionality that receives two inputs, (m_0, m_1) and a bit b from two parties P_1 and P_2 respectively, and returns m_b to P_2, effectively allowing this party to learn only the chosen value m_b among m_0 and m_1, while P_1 does not learn which value was chosen. For a more detailed definition on this primitive, constructions and applications, we refer the reader to e.g. [34].

1.1.5 Efficiency Metrics

Finally, we discuss some of the efficiency metrics that we are typically interested in when designing secure multiparty computation protocols.

Computation complexity. A first measure is the amount of computation that each party has to carry out locally. Fully homomorphic encryption techniques are typically very costly in terms of computation, although they tend to have minimal overhead in terms of communication.

Communication complexity. Since MPC is a distributed application, it is important to measure how many bits need to be transmitted overall by all the parties during a protocol execution. Protocols based on garbled circuits tend to have a large communication complexity, although they round count is generally small.

Round count. Finally, orthogonal to communication complexity, which is affected by bandwidth resources, is the concept of round count, which is more relevant in terms of the latency between the parties. A communication round consists of one execution of the parties sending one message to each other. If a protocol involves many rounds, and there are parties having high-latency links between them, then the overall efficiency of the given MPC protocol might be poor. Secret-sharing-based MPC protocols, although they tend to be very reasonable in terms of communication complexity, suffer from a round count that is proportional to the number of layers present in the arithmetic circuit at hand.

1.2 Simulation-Based Security

One of the major achievements of modern cryptography lies in properly formalizing several constructions like encryption or digital signatures, so that rigorous mathematical reasoning could be applied on these. This way, concrete guarantees and relationship across different notions could be achieved, as exemplified by the idea of “provable security”.

Secure multiparty computation appeared in the late 80’s as an interesting task to study, having many potential applications and involving very interesting techniques. However, it took almost a decade until more formal approaches to secure multiparty computation appeared, which enabled a more rigorous treatment of the area. The “mathematical framework” under which the task of secure multiparty computation can be phrased is highly non-trivial, and is undoubtedly considered a contribution on its own.

1.2.1 High-Level Idea

There exist several different frameworks for formalizing secure multiparty computation protocols, like the stand-alone model [9], the UC framework [10] and the SUC framework [11], among others. However, although there are minor differences from one model to the other, what is common across all these approaches is that security is defined via *simulation*, an idea that is already common in the area of zero-knowledge proofs, for example, and serves the purpose of properly defining the notion of "not learning anything beyond `X`". To provide a high level idea of what this technique is about, recall from Section 1.1 that the idea of a given secure multiparty computation protocol being secure is related to ensuring the adversary, who corrupts a subset of the parties, does not learn anything about the inputs from the honest parties, except perhaps from what is leaked by the output of the computation itself.

Formalizing the idea of an adversary not learning some data is not new in cryptography, as it has appears already, for example, in constructions such as encryption schemes, which are formalized using *game-based security*.[4] The main challenge in the MPC setting, however, is that the adversary does learn something about the data that is intended to be hidden, namely, the output of the computation. Furthermore, another major complication lies in the fact that secure multiparty computation is a distributed application, involving communication among the parties according to some specified pattern. The adversary gets to see all the messages exchanged with the corrupt parties during the protocol execution, and when the adversary is active, it even gets the power to modify the behavior of the corrupt parties. These complications put a barrier in the use of simpler game-based definitions widely used across cryptography.

The key idea to tackle the complication above, as we already hinted at in Section 1.1, is to consider an *ideal world* that captures the desired properties of the interaction, and somehow requiring that the real world, where the actual protocol execution takes place, is indistinguishable from the ideal world. Typically, the ideal world consists of the parties sending their inputs to a third trusted party who computes the function and returns the result, and only the result, to the parties. Now, to claim that the two worlds are indistinguishable, a natural approach is to claim that the adversary cannot distinguish the real from the ideal world. However, this approached is doomed as these two worlds are trivially distinguishable from the point of view of the adversary: in the real world there are messages going to and from all the parties, there are several rounds, and there is no trusted parties, while in the ideal world there is a third trusted party that simply receives inputs and sends output. These two patterns look entirely different, so the adversary can clearly distinguish between the two.

This is where the idea of simulation kicks in. In the real world, the adversary, corrupting a subset of the parties, interacts with the honest parties and learns a result at the end of the execution. In the ideal world, the adversary will not directly interact with the trusted party. Instead, while the honest parties do interact with the third trusted party, the adversary interacts with some "virtual" honest parties that, unlike the actual honest parties, do not have access to the inputs intended to be kept hidden towards the adversary. These virtual honest parties are coordinated by a *simulator*, who also controls the corrupt parties in the ideal world, sending input and receiving output from the third trusted party. This way, the simulator effectively serves as an interface that enables the adversary to interact with the third trusted party in the ideal world, while having an interaction that equals the one from the real world.

[4]In a nutshell, game-based security considers a scenario in which the adversary interacts with the system trying to distinguish certain data that is intended to be hidden, and security is formalized by requiring that no adversary can win at this "game" with high probability.

To prove that a given secure multiparty computation protocol is secure, it is then necessary to define a simulator that acts as the interface sketched above, in such a way that the real world, where the adversary interacts with the actual honest parties holding the real inputs, is indistinguishable from the ideal world, where the adversary interacts with the virtual honest parties controlled by the simulator. Notice that the only power the simulator counts on in order to "fool" the adversary is the access to the third trusted party, which receives inputs and reveals solely the output. As a result, intuitively, this means that the adversary's experience in the real world can alternatively be "recreated" by having access only to the third trusted party, which, from a philosophical standpoint, instantiates the core idea of the adversary *only learning the output of the computation*, after the interaction with the honest parties in the real world.

With the intuitive approach outlined above, we now proceed to provide slightly more formal details on how simulation-based security works. We remark that, in this document, we focus only on the UC framework, leaving other simulation-based security notions such as stand-alone security aside. Furthermore, the description here is not intended to be fully self-contained, and the approach to the UC framework used in this section is taken from [16]. For a more complete treatment of the UC framework we refer the reader to this reference.

1.2.2 Interactive Agents

We begin by considering all the different entities involved in the formalization of a secure multiparty computation protocol and their security. Our starting point is the concept of an *interactive agent*, which, at an intuitive level, is a computational device that receives and sends messages, holds internal state and carries out computations. For example, the parties in a secure multiparty computation protocol are interactive agents, but in the framework under which these ideas are formalized several other interactive agents appear. Interactive agents can be formalized by the means of interactive Turing machines, which are simply traditional Turing machines (or algorithms) that, additionally to carrying out computations, can send and receive data to and from certain *communication ports*, which can be thought of as computer buses or channels.

1.2.2.1 Relevant Interactive Agents in the UC Framework

Now we describe the different interactive agents that appear in the UC framework. As we have mentioned already, the first natural interactive agents are the parties, which are the actual devices carrying out the computation, but several other interactive agents such as the simulator or the "trusted third party" appear. We discuss these below.

Parties. The parties, which we denote by $P_1, \dots, P_n$, constitute the first natural example of interactive agents. Each party P_i, having certain input to the computation, proceeds according to the instructions of the protocol, performing local computation and sending/receiving data to/from the other parties, as required. At the end of this execution, each party P_i obtains the result of the computation.

Functionalities (in the real world). A functionality is simply an interactive agent that connects to the parties. It receives messages from them, performs local computation, maintains internal state, and sends messages back to the parties.

Although there is only one "type" of functionalities, these are used in two different contexts, with the first being in the real world, where the actual execution of the protocol takes place. In an execution of a secure multiparty computation protocol, the parties may be able to use certain "external" resources that may aid them during the computation. As an example, the parties may count on a third trusted party that, although it may not compute the whole desired function for the parties, may provide certain help like distributing some secret keys, or sending certain certificates. This can be formalized as a functionality that the parties talk to during the execution of the protocol at hand. Furthermore, what is crucial is that, as we will see in Section 1.2.5, if later on another protocol is developed that imitates the behavior of this functionality, then the parties can use this construction as a subroutine and the overall construction remains secure, without the need of a third trusted party to assumed to provide the secret-key or certificate service from the example above.

Furthermore, so far we have been implicitly assuming that the parties communicate among each other by means of special ports set between every pair of parties. However, now that we have introduced the concept of a functionality, it is convenient to consider communication instead as a functionality which receives messages from and sends messages to the parties. For example, a simple peer-to-peer network may be modeled as a functionality that acts as follows: party P_i sends a message of the type "send message m to party P_j", and the functionality sends to P_j the message "party P_i sends the message m".[5] Although this approach may seem as an unnecessary complication, it actually plays the important role of enabling flexibility in the way the parties talk to each other.[6] For example, in this document we consider as a basis a functionality that, in addition to modeling point-to-point encrypted and authenticated channels, supports an additional broadcast channel. More details are provided in Section 1.2.6.1.

Functionalities (in the ideal world). The second setting in which functionalities are used is in the ideal world. Recall that, in that world, there is a trusted third party that receives the inputs from the different parties and returns the result of the computation. This is precisely a functionality, which, as defined above, is an interactive agent that receives inputs from the parties, performs certain internal computation and sends a result to the parties.

In the most simple case, a functionality receives the inputs to a given function, evaluates this function internally, and returns the result to the parties. However, the concept of a functionality allows us to model much more complex interactions. For example, in Section 1.1.4.2 we discussed non-reactive computation, which enables the parties to obtain partial results and continue the computation afterwards. This can be captured by a functionality that receives inputs from the parties, stores some internal state, sends

[5]Functionalities of this type are referred to as *communication resources* in [16], but we avoid this terminology in order to make it more clear that these functionalities are no different than the ones considered in the ideal world.

[6]Furthermore, an important low level detail of functionalities is that they leak certain information to the environment, which can be used to model the fact that, in practice, the adversary might be able to see certain metadata such as when an honest party sent a message to another honest party, its size, etc.

partial results, and continues in this fashion as indicated by the parties. We will discuss in Section 1.2.6 some basic functionalities we will use throughout this document.

Adversary. The adversary, denoted by $\mathcal{A}$, is modeled as another interactive agent, and it has ports communicating it to the each of the corrupt parties. If the corruption is passive, these ports are used to inform the adversary about the internal state of the corrupt parties, including the messages they have received. On the other hand, if the corruption is active, these ports are used to "fully control" the corrupt parties.

Environment. This entity plays a crucial role within the notion of simulation-based security. Intuitively, the environment is in charge of *distinguishing* the real world from the ideal world. We have mentioned in Section 1.2.1 that it is the adversary who cannot distinguish between the real and ideal worlds, but this is unfortunately insufficient. The main reason for this is that, if we simply require that the adversary cannot distinguish between the two worlds, then, even though this would imply that the inputs from the honest parties are protected, it might be the case that the honest parties do not receive the correct output of the computation, which is also an important concern. This could occur since, to "fool" the adversary, the simulator only needs to create a similarly-looking interaction towards the adversary, but it could be that the honest parties in the real world end up computing a completely different result than in the real world, while in the ideal world they obtain the correct result. Since the adversary does not see these outputs as they belong to honest parties only, the two worlds would still be indistinguishable.

In order to address this issue, indistinguishability is defined in such a way that the inputs and the outputs of the computation are also taken into account. This is formalized by considering another agent, the *environment*, typically denoted by $\mathcal{Z}$, that indicates the parties which inputs to use, and receives from each party the result they obtained in the world under consideration (in the ideal world this corresponds to the correct result of the computation, while in the real world this is the result the party computes in the execution of the given protocol). Under this new consideration, a protocol is said to be secure if no environment can distinguish between the real and ideal worlds. Notice that this in particular means that the adversary cannot distinguish between the two worlds as otherwise the adversary could inform the environment which world is currently being executed, but, even if the two executions are indistinguishable to the adversary, the environment can still make use of the inputs it provided to the computation together with the outputs received to attempt to distinguish. If, even after this, the two executions are still indistinguishable, then it is because not only the distributions look similar to the adversary but also the outputs in the real world follow the same distribution as in the ideal world with respect to the inputs provided, which corresponds precisely to the correct results of the computation.

In the UC model, the environment and the adversary are essentially "one and the same", which is modeled by the fact that these two agents have a shared port that enables the environment to fully control the adversary, in essentially the same way as the adversary can fully control the honest parties in the case of an active corruption. Given that these two entities, including also the corrupt parties, are so entangled, in this work *we merge the environment, the adversary and the corrupt parties*, using the term environment/adversary indistinctively to refer to the resulting interactive agent. This entity is in charge of (1) playing the role of the corrupt parties and (2) sending inputs to the honest parties and receiving output from these.[7]

Simulator. Finally, as we have already discussed in Section 1.2.1, the simulator is in charge of acting as an "interface" between the adversary and the desired functionality in the ideal world. This is formalized by means of an interactive agent that connects to the adversary/environment in the ideal world through the same ports as the corrupt parties do in the real world, and also connects to the functionality under consideration, "on behalf" of the corrupt parties. This way, the simulator can send inputs to and receive outputs from the functionality, which constitutes the simulator's main tool to create an indistinguishable scenario towards the environment with respect to the real world.

1.2.3 Interactive Systems

An *interactive system* is simply a collection of interactive agents. As an example, a collection of parties is an interactive system, which we refer to as a *protocol*. We first review some notions that will be important for our discussions.

Open/closed ports. Recall that a communication port is simply a "channel" that different interactive agents have access to in order to send a receive messages. For example, each party has shared ports with the functionalities used in the protocol execution, which enables them to send and receive messages to/from it. An interactive system, being a collection of interactive agents, contains several ports. Many of them will involve at least two interactive agents, like the ports used between each party and a functionality. However, in an interactive system some ports may only involve one interactive agent. For example, the environment is in charge of sending inputs to and receiving output from the honest parties, plus it can send instructions to and receive information from the corrupt parties, which means that the parties have ports to communicate with the environment. Given this, in an interactive system such as a protocol, which does not contain the environment, these ports only involve one interactive agent (or, in other word, these ports are "open-ended"). Other open ports in a protocol are these that the parties use to communicate with the different functionalities.

Ports involving at least two interactive agents are known as *closed ports*, while ports involving only one interactive agent are called *open ports*.

Open/closed interactive systems. An interactive system with open ports is referred to as an *open interactive system*, and an interactive system that only has closed ports is known as a *closed interactive system*. For example, a protocol is an open interactive system, given

[7]More generally, the environment, as the name implies, gets to see all the "execution setting", which, on top of inputs and outputs, also involves other "metadata" such as information about when a party sends a message to another, the sizes of these, etc. This is formalized in [16] by means of *leakage ports*, which provide the environment with this type of information. The environment is also in charge of "scheduling" the execution of the protocol. We refer the reader to [16] for details.

that it has the open ports corresponding to the interaction between the environment and the parties, as well as between the parties and the different functionalities used in the real world.

Open interactive systems cannot in principle be run, as they may miss some data that should be written into the open ports. For example, a protocol cannot be run, given that it misses at least one functionality the parties can use for communication, and it also misses the inputs to be used have to be provided (by the environment) into the open ports (plus, the environment is also in charge of scheduling the execution itself). On the other hand, if we consider a larger interactive system consisting of the protocol (which is the set of parties), the different functionalities to be used, and the environment, now we obtain a closed interactive system. This system can be run, as the environment can now provide inputs to the parties, execute the protocol, obtain results (and in fact, it can do this multiple times).

Composition of interactive systems. Given two interactive systems $\mathcal{I}_1$ and $\mathcal{I}_2$, it is possible to obtain a bigger system from these two by considering the collection of all the interactive agents involved in these two sets, or, in other words, the union of the collections $\mathcal{I}_1$ and $\mathcal{I}_2$. This is denoted by $\mathcal{I} = \mathcal{I}_1 \diamondsuit \mathcal{I}_2$. For example, we considered above a closed system given by $\mathcal{Z} \diamondsuit \Pi \diamondsuit (\mathcal{F}_1 \diamondsuit \cdots \diamondsuit \mathcal{F}_\ell)$, where Π was the protocol under consideration and $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$ the various functionalities used in the protocol execution.

1.2.3.1 Relevant Interactive Systems in the UC Framework

We already discussed an important interactive system, a protocol, which is simply a collection of parties. Now we consider the two main interactive systems in the UC framework: the real and ideal worlds. Recall that, in the real world, is where the actual execution of the given protocol takes place, while in the ideal world the parties make use of a trusted third party, modeled as a functionality, to compute the function securely. These ideas are easily formalized via the notion of an interactive system.

Real world. Intuitively, the real world is where the actual execution of the secure multiparty computation protocol at hand takes place. We formalize this via the following interactive system. Let $\Pi = \{P_1, \ldots, P_n\}$ be the protocol and let $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$ be the functionalities to be used in the execution of the protocol. The *real world* is defined as the interactive system given by $\mathsf{Real} := \Pi \diamondsuit (\mathcal{F}_1 \diamondsuit \cdots \diamondsuit \mathcal{F}_\ell)$. Notice that this is an open system, as it requires the environment to provide inputs and schedule the protocol execution.

Ideal world. At a high level we have considered the ideal world as where the parties send their inputs to a third trusted party, and receive outputs afterwards. This had to be refined to include a simulator $\mathcal{S}$, that acts as the interface between the adversary/environment, and the third trusted party.

Let $\mathcal{F}$ be the functionality that models the desired computation to be carried out securely (i.e. the third trusted party), and let $\mathcal{S}$ be a simulator. The *ideal world* is defined as the interactive system given by $\mathsf{Ideal} := \mathcal{S}\diamond\mathcal{F}$. Once again, this is an open system, and in fact it has the same open ports as the interactive system Real: the simulator contains open ports for the environment to connect, as if it were connecting to the corrupt parties in the real world, and the functionality $\mathcal{F}$ has open ports for the environment to provide input to and receive output from the honest parties. In particular, the interactive systems $\mathcal{Z}\diamond\mathsf{Real}$ and $\mathcal{Z}\diamond\mathsf{Ideal}$ are both closed.

1.2.3.2 Parameterized Interactive Systems

Finally, before we dive into the actual security definitions we will consider in this work, we remark that some interactive agents (and hence, interactive systems) can contain several external tweakable parameters. For example, a protocol typically allows for computation over different algebraic structure (e.g. say fields, but of different sizes), or a functionality may be parameterizable according to the length of the messages it accepts, to cite some examples. These are all *external parameters*, meaning that they have to be set before considering an execution of these interactive agents. For illustration, they can be thought to be analogous to compile-time parameters in compiled programming languages.

A very important external parameter is the *security parameter*. Intuitively, this is a natural number that, as it grows larger, the protocol becomes "more secure". This will become clearer in Section 1.2.4 where we consider different notions of security. For now, it suffices to recall that, among all the different external parameters that the various interactive agents under consideration have, one we make explicit mention to is the security parameter, denoted by κ. To make this explicit we may sometimes write $\mathcal{I}(\kappa)$, where $\mathcal{I}$ is an interactive system/agent that is parameterized by κ.

1.2.4 Security Definition

Having defined the different interactive agents involved in our framework, we now turn out attention to defining security. As we have already mentioned, this will be achieved by requiring that no environment can distinguish between the real and ideal executions. In this section we approach in more detail the task of properly defining "indistinguishability".

We begin by introducing some minor preliminaries. First, we present the definition of a negligible function.

Definition 1.1 (Negligible functions). *A function $\mu : \mathbb{N} \mapsto [0,\infty)$ is negligible if, for every $c \in \mathbb{N}$, there exists $\kappa_c \in \mathbb{N}$ such that, for every $\kappa \geq \kappa_c$, it holds that $\mu(\kappa) \leq \kappa^{-c}$. Alternatively, μ is negligible if, for every polynomial $p(\mathtt{X})$, there exists $\kappa_{p(\mathtt{X})} \in \mathbb{N}$ such that, for every $\kappa \geq \kappa_{p(\mathtt{X})}$, it holds that $\mu(\kappa) \leq p(\kappa)$.*

An example of a negligible function is $\mu(\kappa) = 2^{-\kappa}$. Intuitively, a negligible function is a function whose inverse, asymptotically, grows faster than any possible polynomial. These functions are widely used throughout cryptography to represent very small quantities.

The second consideration we must take care of before approaching our formal definitions, is that we include additional semantic notion to the environment. This interactive agent is in charge of distinguishing the real from the ideal execution, and it does so by interacting with either of these worlds, and outputting a bit,[8] that is, either 0 or 1, that represents which world the environment considers it is interacting with. As we will see, the assignments between these bits and the two worlds is irrelevant. Whenever the environment $\mathcal{Z}$ interacts with an interactive system $\mathcal{I}$, and produces output b, we denoted this by $b \leftarrow \mathcal{Z}\diamondsuit\mathcal{I}$. Notice that this is a random variable, given that the whole computation carried out by $\mathcal{Z}$ is potentially randomized.

Below we consider a setting in which a protocol Π is used to securely compute a functionality $\mathcal{F}$, while making use of the functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$. We remark that all the notions below are set with respect to a given adversarial structure, which, as discussed in Section 1.1, dictates the possible sets that can be corrupted.

1.2.4.1 Perfect Security

First we define the idea of perfect security, which reflects a protocol whose security cannot be broken even by unboundedly powerful environments/adversaries.

Definition 1.2 (Perfect security.). *We say that a protocol Π securely instantiates a functionality $\mathcal{F}$ in the $(\mathcal{F}_1, \ldots, \mathcal{F}_\ell)$-hybrid model with **perfect security** if there exists a simulator $\mathcal{S}$ such that, for any environment $\mathcal{Z}$ and for every $\kappa \in \mathbb{N}$,*

$$\Pr[1 \leftarrow (\mathcal{Z}\diamondsuit\mathsf{Real})(\kappa)] = \Pr[1 \leftarrow (\mathcal{Z}\diamondsuit\mathsf{Ideal})(\kappa)],$$

where $\mathsf{Real} = \Pi\diamondsuit\mathcal{F}_1\diamondsuit\cdots\diamondsuit\mathcal{F}_\ell$ and $\mathsf{Ideal} = \mathcal{S}\diamondsuit\mathcal{F}$.

Let us analyze the definition above in detail. First, perfectly secure protocols typically do not rely on the parameter κ, so we can remove it from the definition (it is included for the sake of maintaining certain "uniformity" in the notation with respect to the other notions of security described below). Now, the security definition above states that, there must exist a simulator $\mathcal{S}$ such that $\mathcal{Z}$ outputs 1 when interacting with the system Real with exactly the same probability that $\mathcal{Z}$ would output 1 when interacting with the system Ideal. This means precisely that $\mathcal{Z}$ cannot distinguish between the two worlds since, if it could, it could choose for example to output 1 only in the real world, while outputting 0 in the ideal world (so $\Pr[1 \leftarrow \mathcal{Z}\diamondsuit\mathsf{Real}] = 1$ and $\Pr[1 \leftarrow \mathcal{Z}\diamondsuit\mathsf{Ideal}] = 0$).

Notice that there is nothing special about the output 1. The same definition could have been considered with the output 0, given that $\Pr[0 \leftarrow \mathcal{Z}\diamondsuit\mathsf{Real}] = 1 - \Pr[1 \leftarrow \mathcal{Z}\diamondsuit\mathsf{Real}]$ and $\Pr[0 \leftarrow \mathcal{Z}\diamondsuit\mathsf{Ideal}] = 1 - \Pr[1 \leftarrow \mathcal{Z}\diamondsuit\mathsf{Ideal}]$. This remark also holds for the other security notions below.

[8]An interactive agent, being an enhance Turing machine, can produce output simply by writing it to a special tape and halting.

Finally, the quantity $|\Pr[1 \leftarrow (\mathcal{Z}\Diamond\mathsf{Real})(\kappa)] - \Pr[1 \leftarrow (\mathcal{Z}\Diamond\mathsf{Ideal})(\kappa)]|$ is typically referred to as the *statistical advantage* of $\mathcal{Z}$, and it is essentially a measure of how well $\mathcal{Z}$ can distinguish between the real and ideal worlds. We see that, in the setting of perfect security, the advantage of any environment is 0.

1.2.4.2 Statistical Security

Now we consider a more flexible definition that allows certain small distinguishing advantage.

Definition 1.3 (Statistical security.). *We say that a protocol Π securely instantiates a functionality $\mathcal{F}$ in the $(\mathcal{F}_1, \dots, \mathcal{F}_\ell)$-hybrid model with **statistical security** if there exists a negligible function $\mu(\kappa)$ such that, for any environment $\mathcal{Z}$,*[9]

$$|\Pr[1 \leftarrow (\mathcal{Z}\Diamond\mathsf{Real})(\kappa)] - \Pr[1 \leftarrow (\mathcal{Z}\Diamond\mathsf{Ideal})(\kappa)]| \leq \mu(\kappa),$$

where $\mathsf{Real} = \Pi\Diamond\mathcal{F}_1\Diamond\cdots\Diamond\mathcal{F}_\ell$ *and* $\mathsf{Ideal} = \mathcal{S}\Diamond\mathcal{F}$.

In this case, $\mathcal{Z}$ might be able to distinguish the two worlds “a little”, which is reflected in the case that $\Pr[1 \leftarrow (\mathcal{Z}\Diamond\mathsf{Real})(\kappa)]$ and $\Pr[1 \leftarrow (\mathcal{Z}\Diamond\mathsf{Ideal})(\kappa)]$ may not be equal. In fact, it could be the case that, for some values of κ, the environment might distinguish the two worlds very well (for instance it can happen that $\Pr[1 \leftarrow (\mathcal{Z}\Diamond\mathsf{Real})(\kappa)] = 1$ and $\Pr[1 \leftarrow (\mathcal{Z}\Diamond\mathsf{Ideal})(\kappa)] = 0$ for some values of κ). However, the definition requires that, as κ grows, this distinguishing advantage shrinks at a good rate. For example, if $\mu(\kappa) = 2^{-\kappa}$, then choosing $\kappa = 1$ may be too bad since this means that the advantage that the environment has to distinguish between the two worlds is only $1/2$, but if $\kappa = 40$, then this is reduced to 2^{-40}, which is much more acceptable (in fact, 2^{-40} is a very common value to aim for when designing statistically secure protocols).

1.2.4.3 Computational Security

Finally, we consider the “weakest” of the security notions regarding secure multiparty computation protocols. In this case, the environment has a small distinguishing advantage, but this only holds if the environment is computationally bounded, meaning that it runs in polynomial time. In terms of practical meaning, this notion is good enough given that in an actual MPC deployment all parties involved will use a bounded amount of computational resources. Furthermore, as we will see in Section 1.3, some secure multiparty computation scenarios do not allow for any of the previous notions, and require computational security instead.

[9]Here we must slightly limit the environment with respect to these from Definition 1.2, which did not have any limitation. In this case, we must assume that, although $\mathcal{Z}$ might be computationally unbounded, it only makes a polynomial (in κ) number of “calls” to either Real or Ideal. Otherwise, the notion cannot be achieved, since by interacting with one of the two worlds a super-polynomial number of times the distinguishing probability can be arbitrarily improved.

Definition 1.4 (Computational security.). *We say that a protocol Π securely instantiates a functionality $\mathcal{F}$ in the $(\mathcal{F}_1, \ldots, \mathcal{F}_\ell)$-hybrid model with* ***computational security*** *if, for any* efficient *environment $\mathcal{Z}$,*[10] *there exists a negligible function $\mu_{\mathcal{Z}}(\kappa)$ such that*

$$|\Pr[1 \leftarrow (\mathcal{Z}\Diamond\mathsf{Real})(\kappa)] - \Pr[1 \leftarrow (\mathcal{Z}\Diamond\mathsf{Ideal})(\kappa)]| \leq \mu_{\mathcal{Z}}(\kappa),$$

where $\mathsf{Real} = \Pi\Diamond\mathcal{F}_1\Diamond\cdots\Diamond\mathcal{F}_\ell$ *and* $\mathsf{Ideal} = \mathcal{S}\Diamond\mathcal{F}$.

The first thing to notice with the definition above is that, unlike Definition 1.3, there is not a single negligible function $\mu(\kappa)$ that bounds the advantage of every possible environment $\mathcal{Z}$ when attempting to distinguish the real and the ideal worlds. This is not possible to achieve in general since there can be a series of environments $\mathcal{Z}_1, \mathcal{Z}_2, \ldots$ with each $\mathcal{Z}_c$ running in time κ^c (which is polynomial), so for a fixed κ_0 these environments have running times $\kappa_0^1, \kappa_0^2, \ldots$, which is unbounded. Eventually, with enough running time it would be possible to break the fixed bound on the advantage of $\mu(\kappa_0)$.

As a result, the best that can be hoped for is that, for every single environment $\mathcal{Z}$, its distinguishing advantage can be upper bounded by a negligible function $\mu_{\mathcal{Z}}(\kappa)$ that depends on this environment. A practical interpretation of this can be the following. After a careful analysis of the construction at hand, we consider the most efficient known attack on the protocol, derive an environment $\mathcal{Z}$ from this, determine the associated negligible function $\mu_{\mathcal{Z}}(\kappa)$ and choose κ so that the advantage of this environment in particular is below certain threshold (e.g. 2^{-80}). Given our observations above, it could be the case that this choice of κ is not sufficient to ensure a low distinguishing advantage for other environments, but at least it rules out the best one that is currently known.

We remark that, in this document, even though we consider settings in which only computational security is achievable, we only deal with perfect and statistical security in our actual security proofs. This is because, for the settings not admitting this type of security, we consider an offline/online paradigm that enables computation with perfect or statistical security with the help of certain functionalities.

1.2.5 The Composition Theorem

Consider a protocol $\Pi_{\mathcal{F}}$ that securely instantiates a functionality $\mathcal{F}$ with the help of some other functionality $\mathcal{R}$, that is, in the $\mathcal{R}$-hybrid model. In the real world, this functionality $\mathcal{R}$ acts as some kind of third trusted party that the parties can use to aid them in the task of securely computing $\mathcal{F}$; however, in practice, this functionality $\mathcal{R}$ must somehow be instantiated. For example, if $\mathcal{R}$ represents peer-to-peer encrypted and authenticated channels then a protocol like TLS must be executed to set these up. Formally, this would mean that a new protocol $\Pi_{\mathcal{R}}$ that instantiates $\mathcal{R}$, perhaps in some $\mathcal{T}$-hybrid model, is used, and a natural question is then the following: what types of formal security guarantees can the new protocol achieve? With the "new protocol", we mean protocol $\Pi_{\mathcal{F}}$ but replacing the

[10]An efficient environment is one that, at a high level, runs in polynomial time with respect to its parameters. However, there are several details that must be taken care of when properly defining this idea, given that the environment, and in general, any interactive agent, exchanges messages with other agents and can make "calls" to these. We refer the reader to [16] for details.

interaction with the functionality $\mathcal{F}_\mathsf{R}$ by executions of the protocol $\Pi_\mathcal{R}$, which instantiates this functionality.

The core result of the UC framework, or universal-composability framework, is that, precisely, the resulting *composed* protocol inherits the properties of the two protocols involved, $\Pi_\mathcal{F}$ and $\Pi_\mathcal{R}$. In particular, this protocol still instantiates $\mathcal{F}$, but instead of doing it in the $\mathcal{R}$-hybrid model, it does it in the $\mathcal{T}$-hybrid one, which is the functionality needed by the protocol $\Pi_\mathcal{R}$. It will typically be the case that $\mathcal{T}$ is much simpler than $\mathcal{R}$, which implies that progress has been achieved towards instantiating $\mathcal{F}$ securely.

Before we discuss the theorem in detail, we discuss some of the consequences of the above high-level description of the result. First, the composition theorem enables a modular description of highly complex protocols by breaking them into pieces and then proving the security of each fragment separately, an approach that we make extensive use of throughout this document. For example, in a large and complex protocol Π it might be the case that certain piece or pattern is repeated in several places of the protocol execution. This part can be isolated as a protocol Π' on its own, instantiating certain functionality $\mathcal{R}$, and the protocol Π could possibly be expressed in a much simpler way in terms of this functionality. Now, to prove security, we do not need to provide a proof of the big "monolithic" protocol Π, but rather, we can prove that its simpler variant, which is set in the $\mathcal{R}$-hybrid model, instantiates the desired functionality, and then we can focus only in proving that the protocol Π' indeed instantiates $\mathcal{R}$. For illustration purposes, it is useful to think of the approach above as splitting complex functions in a programming language into simpler constructions that make calls to other functions. This approach enables clear and modular proofs and protocol descriptions, and it is arguably one of the key factors that has enabled such a rich and fruitful body of research in the area of secure multiparty computation.

The composition theorem is not only useful as a pedagogical tool. In practice, secure multiparty computation protocols are deployed in large and complex distributed systems that are possibly running, concurrently, many other protocols to achieve other tasks. For example, keys must be negotiated, random values must be sampled, inputs must be provided, etc. The composition theorem ensures that, even if several protocols are executed simultaneously, as long as each of them can be proven secure, then the resulting group of protocols is also secure. This is a crucial observation that favors the UC framework with respect to other formal models, such as the stand-alone one, that does not accept such flexible concurrent composition.

Composing protocols. In order to properly state the composition theorem, it is important to clearly and explicit define the different interactive agents and systems involved.

Consider a protocol $\Pi_\mathcal{F} = \{P_1, \dots, P_n\}$ that instantiates a functionality $\mathcal{F}$ in the $\mathcal{R}$-hybrid model, and consider another protocol $\Pi_\mathcal{R}$ which has different parties $\{Q_1, \dots, Q_n\}$[11] and instantiates the functionality $\mathcal{R}$ in the $\mathcal{T}$-hybrid model. Composing the protocols $\Pi_\mathcal{F}$ and $\Pi_\mathcal{R}$ amounts to simply composing them as interactive systems, which is denoted by $\Pi_\mathcal{F} \Diamond \Pi_\mathcal{R}$.

[11]Recall that, formally, a party carries the "code" of the protocol that is executed, so different protocols involve different parties.

To make more sense of this notion, recall that the parties $Q_1, \dots, Q_n$ have ports to communicate with the environment, while the parties $P_1, \dots, P_n$ have ports to communicate with the functionality $\mathcal{R}$. These ports are one and the same: messages sent from P_i to $\mathcal{R}$ are received by Q_i as coming from the environment, and similarly in the opposite direction. This way, the interactive system $\{P_i, Q_i\}$ acts as one single party, interacting with the environment (through P_i's ports) and also with the functionality $\mathcal{T}$ (through Q_i's ports). With this new interpretation, we see that the composition of two "compatible" protocols is again a protocol, where the new parties might be interactive systems that behave just like an interactive agent. For more details we refer the reader to Section 4.2.7 in [16].

The main theorem. With the notation above at hand, we are ready to properly state the composition theorem, which is fully proven in [16].

Theorem 1.1 (Composition Theorem, Thm 4.20 in [16]). *Let $\Pi_{\mathcal{F}}$ be a protocol instantiating a functionality $\mathcal{F}$ in the $\mathcal{R}$-hybrid model with perfect/statistical/computational security, and let $\Pi_{\mathcal{R}}$ be a protocol instantiating $\mathcal{R}$ in the $\mathcal{T}$-hybrid model with the same type of security. Then, the composed protocol $\Pi_{\mathcal{F}} \diamondsuit \Pi_{\mathcal{R}}$ securely instantiates the functionality $\mathcal{F}$ in the $\mathcal{T}$-hybrid model, with the same type of security.*

1.2.6 Some Basic Functionalities

We end this chapter with a description of some functionalities we use throughout this document. This includes the basic communication resource that the parties use to interact with each other, and the functionality used to model the task of general purpose secure computation, with a variant to account for security with abort.

1.2.6.1 Underlying Communication Resource

As a starting point, we assume that the parties communicate through the following functionality.

Functionality $\mathcal{F}_{\mathsf{P2P+BC}}$

The functionality proceeds as follows:

- On input $(\mathsf{message}, j, m)$ from party P_i, send $(\mathsf{message}, i, m)$ to P_j.
- On input $(\mathsf{broadcast}, m)$ from party P_i, send $(\mathsf{broadcast}, i, m)$ to all parties.

The functionality above models a *peer-to-peer encrypted and authenticated network* in which the parties can send messages to each other confidentially, and the adversary cannot modified their contents when the sender is not an actively corrupt party. In addition to this, in includes a *broadcast channel*, in which a sender with a given message can distribute this data to the other parties in such a way that all parties are guaranteed to receive the exact same value.

All of our protocols assume $\mathcal{F}_{\mathsf{P2P+BC}}$ as a basis, so we do not write that a given instantiation is "in the $\mathcal{F}_{\mathsf{P2P+BC}}$-hybrid model". Only in Section 1.3, where we present several fundamental results, we sometimes consider a functionality $\mathcal{F}_{\mathsf{P2P}}$ that does not include the broadcast channel.

1.2.6.2 Arithmetic Black Box Model

Recall that, in general-purpose secure multiparty computation, our aim is to securely compute any possible function, written as an arithmetic circuit over certain algebraic structure. In this work, we model such computations in two different ways. First, we consider standard arithmetic circuits as defined in Section 1.1.4.3. These a directed acyclic graphs with input, operation (addition and multiplication) and output gates, and they are better suited for modeling non-reactive computation. Naturally, such type of computation can be easily described as a functionality that receives inputs from the parties, computes the given arithmetic circuit, and returns the output.

In some other places, however, we consider a more flexible functionality that is better suited for reactive computation, which, as defined in Section 1.1.4.2, enables the parties to obtain partial results, learn them, and continue the computation possibly depending on these values. The model we make use of to represent such behavior is the *arithmetic black box* model. At a high level, this allows the parties to access a "storage box" that can keep values sent by the parties, but it also allows for additions and multiplications to be carried out on stored values, saving the results. Finally, it enables the parties to read any stored value at any time, which effectively models the setting of reactive computation. The formal functionality is described below.

Functionality $\mathcal{F}_{\mathsf{ABB}}$: Arithmetic Black Box

The functionality proceeds as follows.

- On input $(\mathsf{input}, \mathsf{id}, i)$ from the honest parties, send $(\mathsf{input}, \mathsf{id}, i)$ to the adversary, wait for input $(\mathsf{value}, \mathsf{id}, x)$ from party P_i, where $x \in \mathbb{Z}/2^k\mathbb{Z}$, and then store (id, x) in memory.
- On input $(\mathsf{comb}, \{c_i\}_{i=0}^{\ell}, \{\mathsf{id}_i\}_{i\in[\ell]}, \mathsf{id}_{\ell+1})$ from the honest parties, retrieve (id_i, x_i) for $i \in [\ell]$ from memory and store $(\mathsf{id}_{\ell+1}, z)$, where $z = (c_0 + \sum_{i=1}^{\ell} c_i x_i) \bmod 2^k$. Then send $(\mathsf{comb}, \{c_i\}_{i=0}^{\ell}, \{\mathsf{id}_i\}_{i\in[\ell]}, \mathsf{id}_{\ell+1})$ to the adversary.
- On input $(\mathsf{mult}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from the honest parties, retrieve (id_1, x) and (id_2, y) from memory and store (id_3, z), where $z = x \cdot y$. Then send $(\mathsf{mult}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ to the adversary.
- On input $(\mathsf{open}, \mathsf{id})$ from the honest parties, retrieve (id, x) from memory and send x to all the parties. Then send $(\mathsf{open}, \mathsf{id})$ to the adversary.

Formalizing security with abort. We have discussed in Section 1.1 three different notions regarding the guarantees the honest parties have with respect to the output of the computation: guaranteed output delivery, where all parties will receive output, fairness, where honest parties receive output if the corrupt parties do so as well, and security with abort, where the adversary may cause the honest parties to abort, perhaps not obtaining any output, while the adversary may be able to learn the result nevertheless.

We capture security with abort in the UC framework by endowing all functionalities with the following behavior: at any point of the execution, the adversary can input a special signal abort to the given functionality, which sends abort to all honest parties. Upon receiving such message, each honest party immediately produces abort as output, and halts.

In the real world, whenever we say that "the parties abort", it means that they produce abort as output, and halt. In some cases, the event triggering an abort is only seen by one party (e.g. some party receives incorrect data). When we say that "party P_i aborts", we implicitly assume that this party sends abort through the broadcast channel, so all parties abort too.[12]

1.3 Fundamental Results

We now proceed to presenting some of the most fundamental results in the theory of secure multiparty computation, regarding the feasibility or impossibility of MPC in certain contexts. Below, we consider different results in the context in which the adversarial structure is a threshold structure with threshold t, categorized by whether $t < n/3$, $t < n/2$ or $t < n$. As we will comment in each relevant section, most of the results for $t < n/2$ and $t < n/3$ carry over to the case of $Q2$ and $Q3$ general adversarial structures, respectively.

1.3.1 Results for $t < n/3$

The context in which the adversary corrupts at most one third of the parties is particularly relevant as it allows for the strongest level of simulation-based security, namely, perfect security.

Positive results. We begin with the following crucial result, which shows that the most desired notions of perfect security and guaranteed output delivery can be achieved if $t < n/3$, even if the adversary is active.

Theorem 1.2. *There exists a protocol instantiating $\mathcal{F}_{\mathsf{ABB}}$ with perfect security and guaranteed output delivery in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model,*[13] *secure against an active adversary corrupting at most $t < n/3$ of the parties.*

[12] This is the crucial difference between *selective abort* and *unanimous abort*. In the former, it can happen that only some honest parties abort while the others remain in the computation. Through a broadcast channel, as shown in [27] and as used here, we can ensure unanimous abort by asking aborting parties to announce their status through the broadcast channel.

[13] As mentioned in Section 1.2.6.1, our security statements later in the document are all set in the $\mathcal{F}_{\mathsf{P2P+BC}}$-hybrid model and we do not write this explicitly.

The proof of this result can be found for example in [5, 13].

Remark 1.2. *Notice that, in Theorem 1.2, the basic communication resource is $\mathcal{F}_{\mathsf{P2P}}$, which represents encrypted and authenticated peer-to-peer communication,* without a broadcast channel. *Most constructions will still make use of a broadcast channel, but this is possible to construct from plain peer-to-peer channels with perfect security if $t < n/3$, as shown in [39].*

Negative results. An interesting fact is that, if the adversary breaks the $t < n/3$ condition, that is, if the adversary corrupts more than one third of the parties, then Theorem 1.2 does not hold anymore. More precisely, the "perfect security" part of the theorem cannot be fulfilled, which is summarized in the following theorem.

Theorem 1.3. *No protocol can instantiate $\mathcal{F}_{\mathsf{ABB}}$ in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model against an active adversary corrupting at least $n/3$ parties with perfect security.*

To show that this theorem holds, it suffices to exhibit a particular function that cannot be instantiated with perfect security in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model, if an active adversary corrupts at least $n/3$ parties. This invalidates the possibility of $\mathcal{F}_{\mathsf{ABB}}$ being instantiable, given that $\mathcal{F}_{\mathsf{ABB}}$ can be used to trivially instantiate any other functionality. There are several functions that, if $t \geq n/3$, cannot be securely computed in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model. A typical example being the broadcast functionality. This result can be found in [39].

Finally, although there are some functions such as broadcast that cannot be instantiated with perfect security in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model if $t \geq n/3$, it is natural to ask whether, even if we add broadcast as a basis, that is, if we work in the $\mathcal{F}_{\mathsf{P2P+BC}}$-hybrid model, there are still some functions that cannot be instantiated with perfect security if $t \geq n/3$. This turns out to be the case, as shown for example in [16] (see Theorem 5.12 in the reference).

Theorem 1.4. *No protocol can instantiate $\mathcal{F}_{\mathsf{ABB}}$ in the $\mathcal{F}_{\mathsf{P2P+BC}}$-hybrid model against an active adversary corrupting at least $n/3$ parties with perfect security.*

1.3.2 Results for $t < n/2$

Now we turn our attention to the setting in which the adversary corrupts at most one half of the parties. In this setting, although general-purpose secure computation with perfect security against an active adversary is not possible (as illustrated in Theorem 1.4), several other properties are still attainable.

1.3.2.1 The Case of a Passive Adversary

Positive results. First we discuss what happens when the adversary is passive. In this case, it turns out that a protocol with perfect security can be designed, as expressed by the following theorem.

Theorem 1.5. *There exists a protocol instantiating $\mathcal{F}_{\mathsf{ABB}}$ with perfect security in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model, secure against a passive adversary corrupting at most $t < n/2$ of the parties.*

Notice that this theorem is similar to Theorem 1.2, except that this time the adversary is passive and the corruption threshold is at most $n/2$, instead of being upper bounded by $n/3$. Protocols that illustrate the validity of Theorem 1.5 are presented in [5, 13].

Negative results. A protocol with perfect security to compute arbitrary functionalities cannot exist if the condition $t < n/2$ is broken. In fact, such protocol cannot exist even if we loosen the security notion to statistical security. This is shown in the following theorem.

Theorem 1.6. *No protocol can instantiate $\mathcal{F}_{\mathsf{ABB}}$ in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model against a passive adversary corrupting at least $n/2$ parties with statistical security.*

A proof of this result can be found, for example, in [37].

1.3.2.2 The Case of an Active Adversary

Positive results. Now we focus on the case in which the adversary corrupts a subset of the parties actively. As shown in Theorem 1.3, in the case in which $t < n/2$, given that in principle it could hold that $t \geq n/3$, it is not possible to instantiate $\mathcal{F}_{\mathsf{ABB}}$ with perfect security in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model (or even in the $\mathcal{F}_{\mathsf{P2P+BC}}$-hybrid model from Theorem 1.4). However, it turns out that, if we relax the security requirement to statistical security rather than perfect security, an instantiation can be realized. Furthermore, the strongest output notion of guaranteed output delivery can be attained. This is summarized below.

Theorem 1.7. *There exists a protocol instantiating $\mathcal{F}_{\mathsf{ABB}}$ with statistical security and guaranteed output delivery in the $\mathcal{F}_{\mathsf{P2P+BC}}$-hybrid model, secure against an active adversary corrupting at most $t < n/2$ of the parties.*

Protocols proving this theorem include [37], or the more recent results of [28] which improve over the communication complexity of the previous ones.

Negative results. Theorem 1.6 shows that, if the bound $t < n/2$ is violated, then no protocol can instantiate $\mathcal{F}_{\mathsf{ABB}}$ with statistical security in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model, even if the adversary is assumed to be passive. In particular, this impossibility extends ("with even more reason") if the adversary is active.

On the other hand, another natural question is whether the strong notion of guaranteed output delivery, achievable if $t < n/2$, is still attainable if $t \geq n/2$. There is a negative answer to this question, and in fact, not even the weaker notion of fairness can be realized if $t \geq n/2$. This is captured in the following theorem.

Theorem 1.8. *No protocol can achieve fairness when instantiating $\mathcal{F}_{\mathsf{ABB}}$ in the $\mathcal{F}_{\mathsf{P2P+BC}}$-hybrid model against an active adversary corrupting at least $n/2$ parties.*

A proof of this result can be found in [14].

1.3.3 Positive Results for $t < n$

Finally, we discuss what results are possible in the most general case in which the adversary can, in principle, corrupt all but one party, in contrast to the previous settings, in which the adversary was assumed to corrupt at most a $1/2$ or even a $1/3$ proportion of the parties. In terms of negative results, we can infer from Theorem 1.6 that information-theoretic security (that is, either perfect or statistical security) is out of the picture in this case, and from Theorem 1.8 we rule out the possibility of achieving fairness. As a result, protocols in this setting must make use of computational assumptions (even if the adversary is passive), or, in other words, they must involve cryptographic constructions whose security depends on the hardness of certain underlying problem, and they have to settle for security with abort.

Fortunately, in terms of positive results, it can be shown that protocols with the properties described above indeed exist. In this case the instantiation can be done over $\mathcal{F}_{\mathsf{P2P}}$ rather than $\mathcal{F}_{\mathsf{P2P+BC}}$ since it is possible to obtain broadcast by making use of computational assumptions.

Theorem 1.9. *There exists a protocol instantiating $\mathcal{F}_{\mathsf{ABB}}$ with computational security in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model, secure against a passive adversary corrupting possibly all but one of the parties.*

Several protocols of this type have been achieved in the literature, such as [6, 19, 21, 32, 33]. In Chapters 7 and 8 we include constructions in this setting with passive and active security, respectively. These constructions satisfy perfect and statistical security, which contradicts the impossibility results discussed above. This is because, as we will see in the relevant sections, these protocols are not set in the $\mathcal{F}_{\mathsf{P2P}}$-hybrid model, but instead, they are built making use of a stronger functionality that distributes certain preprocessed data among the parties.

1.3.4 Summary of Main Results

The following table summarizes the results we have seen so far in the section. A check mark (✓) represents that a construction in the given setting can be obtained, while an X mark (✗) indicates that it is not possible in general to instantiate $\mathcal{F}_{\mathsf{ABB}}$ in the scenario under consideration. Additionally, marks in black indicate results that can be trivially derived from the marks in red, and the numbers in parentheses after the latter type of marks represent the number of the theorem in previous sections associated to that result. Finally, this table omits certain details regarding the possibility/impossibility of broadcast, or more precisely, when $\mathcal{F}_{\mathsf{P2P}}$ or $\mathcal{F}_{\mathsf{P2P+BC}}$ is needed, in the $t < n/2$ row containing the asterisk ($\star$), so we refer the reader to the relevant section above for details.

		Privacy guarantees			Output guarantees		
		perf.	stat.	comp.	GOD	fair	abort
Active	$t < n$	✗	✗	✓ (1.9)	✗	✗ (1.8)	✓ (1.9)
	$t < n/2^\star$	✗ (1.6)	✓ (1.7)	✓	✓ (1.7)	✓	✓
	$t < n/3$	✓ (1.2)	✓	✓	✓	✓	✓
Passive	$t < n$	✗	✗ (1.6)	✓	✓	✓	✓
	$t < n/2$	✓ (1.5)	✓	✓	✓	✓	✓
	$t < n/3$	✓	✓	✓	✓	✓	✓

Chapter 2

Secret-Sharing-Based MPC

The goal of this chapter is to introduce the reader to several essential techniques used in secure multiparty computation *over fields*. Given that the focus of this section is simply to provide an overview of major existing techniques for secure multiparty computation over fields, we omit formal proofs in the simulation based model from Section 1.2, and content ourselves with including more intuitive and simple arguments about why the different techniques and protocols satisfy the different properties they intend to.

Essentially all existing approaches to secure multiparty computation in the literature begin by representing the function to be computed as an arithmetic circuit, which were described in Section 1.1.4.3. However, once this circuit can be established, the specific method used to securely evaluate such circuit tends to vary from work to work. In spite of this, it is still possible to identify some general patterns, and, although a few works may not properly fall within any of the categories below, these are inclusive enough to fit a large portion of the literature in the field of (general-purpose) secure multiparty computation.

First, some constructions make use of *homomorphic encryption* techniques to homomorphically evaluate the given circuit (or at least certain portions of it), typically without involving a lot of interaction. These techniques are mostly theoretical (at least these involving large encrypted computations) as the overhead in terms of computational complexity is typically too large. However, as the field of fully homomorphic encryption progresses, this approach becomes more and more practical and, as a result, it may eventually turn into a more practical solution for large and complex computation, at least if used in a partial and clever way (that is, instead of simply evaluating the entire function in one go using homomorphic encryption).

A different approach consists of somehow obtaining a "hidden" version of the circuit that then can be evaluated only on the set of inputs provided by the parties. This turns out to be the approach initially proposed by Yao [41] when the concept of secure multiparty computation was itself born, and a rich and extensive body of works has taken care of enhancing and improving this method, which is known as *garbled circuits*. This technique has several benefits in terms of efficiency as it is typically the case that, after the circuit has been "hidden" (or *garbled*), which requires interaction in a constant number of rounds, the evaluation of the circuit itself can happen with little to none communication. This makes this technique ideal for settings in which the parties are widely distributed and latency is high, so minimizing round trips becomes relevant. Unfortunately, a big downside of the garbled circuits approach is that the process of garbling the circuit, even though happens in a constant number of rounds, tends to involve a large amount of data, which ends up in consuming a lot of bandwidth, up to the extent that for certain applications this becomes a serious bottleneck. Furthermore, the garbled circuit technique is generally better suited for binary circuits (that is, circuits defined over $\mathbb{F}_2 = \{0, 1\}$), with a handful of (mostly theoretical) constructions considering more general arithmetic circuits (e.g. [2,3]).

The alternative approach to securely evaluate an arithmetic circuit is based on a tool called *secret-sharing*. In a secret-sharing scheme (a concept that we define more precisely in Section 2.1 below), a given value can be distributed among several parties so that each of these participants now holds a "share", which satisfy the following: certain sets of shares do not leak anything about the underlying value, while some other sets of shares completely determine it. In secret-sharing-based MPC the goal is then to obtain a secret-shared representation of the inputs to the computation, using a secret-sharing scheme that ensures that any possible set in the adversarial structure (that is, any set of parties that could be corrupted) cannot learn anything about the underlying secret. This is followed by methods to obtain a secret-shared representation of all intermediate values in the computation, until the output is reached.

We discuss the secret-sharing-based MPC approach in more detail in the sections below. Before we dive into it we remark, however, that there are several works in the literature that, although they make use of secret-sharing techniques, they do not adhere exactly to the template provided here. For example, it is very common to mix secret-sharing with garbled circuits (e.g. [22]), or even with homomorphic encryption techniques.

2.1 Linear Secret-Sharing Schemes

We begin our discussion on secret-sharing-based secure multiparty computation protocols, which is the general template to which all of the constructions considered in this work adhere, by first discussing the concept of a secret-sharing scheme itself.

For the purpose of this section we consider a finite field $\mathbb{F}$. Furthermore, we restrict to *threshold* secret-sharing schemes, which only protect the secret if less than certain amount of shares is known, and completely leak the value otherwise. Finally, we also remark that our description here is entirely informal (plus it makes several simplifications) and does not constitute in any way a formal nor precise treatment of linear secret-sharing schemes. For a more concrete mathematical presentation on these tools we refer the reader to [16].

Let $s \in \mathbb{F}$. At an intuitive level, a secret-sharing scheme for n parties with threshold t provides methods for, on input s, computing a set of values $(s_1, \ldots, s_n) \in \mathbb{F}^n$ such that

1. For any set $A \subseteq [n]$ with $|A| \leq t$, the set of shares $\{s_i\}_{i \in A}$ does not leak anything about the value s;

2. For any set $B \subseteq [n]$ with $|B| \geq t + 1$, the value s can be completely reconstructed from the set of shares $\{s_i\}_{i \in B}$.

When a value $s \in \mathbb{F}$ is secret-shared as above, it is common to denote this by $[\![s]\!] :=$

$(s_1, \dots, s_n)$. In a distributed setting with n parties it is typically assumed implicitly in the notation above that each party P_i has the share i, for $i \in [n]$.

A secret-sharing scheme as above is *linear* if, in words, each party can locally add/subtract their shares of different values to obtain shares of the corresponding operation on the secrets. A bit more precisely, it must hold that, if $[\![x]\!] = (x_1, \dots, x_n)$ and $[\![y]\!] = (y_1, \dots, y_n)$, then $[\![x \pm y]\!] = (x_1 \pm y_1, \dots, x_n \pm y_n)$. Now we discuss some simple examples of linear secret-sharing schemes

Example 2.1 (Additive secret-sharing). *The following is a construction of a linear secret-sharing scheme for n parties with threshold $n-1$, which means that every set of at least $(n-1)+1 = n$ shares can reconstruct the secret while any smaller set remains oblivious to this value. Notice that there is only one possible set of n shares, which is the set of all the shares.*

To secret-share a value $s \in \mathbb{F}$, a tuple $(s_1, \dots, s_n) \in \mathbb{F}^n$ is sampled uniformly at random constrained to $s_1 + \cdots + s_n = s$. This can be done, for example, by sampling $n-1$ random values $s_1, \dots, s_{n-1}$ and defining $s_n = s - (s_1 + \cdots + s_{n-1})$. More generally, any set of $n-1$ shares can be sampled uniformly at random while the last one is defined as the secret subtracted with the sum of the other shares. The set of shares is then $(s_1, \dots, s_n)$.

To analyze the required properties by a linear secret-sharing scheme, we observe the following:

- *Any set of at most $n-1$ shares follows the uniform distribution, so in particular it does not reveal anything about the secret s.*
- *Given all the shares $s_1, \dots, s_n$, the secret s can be fully determined as $s_1 + \cdots + s_n = s$. This, together with the point above, shows that this construction is a secret-sharing scheme.*
- *Given two shared values $[\![x]\!] = (x_1, \dots, x_n)$ and $[\![y]\!] = (y_1, \dots, y_n)$, that is, $x = x_1 + \cdots + x_n$ and $y = y_1 + \cdots + y_n$, since $x \pm y = (x_1 \pm y_1) + \cdots + (x_n \pm y_n)$, it holds that $[\![x \pm y]\!] = (x_1 \pm y_1, \dots, x_n \pm y_n)$. This shows that the proposed method constitutes a linear secret-sharing scheme.*

Example 2.2 (Replicated secret-sharing [30]). *The following is a construction of a linear secret-sharing scheme for n parties with a more general threshold $t < n$. To secret-share a value $s \in \mathbb{F}$, first a set of values $\{s_A\}_{A \subseteq [n], |A| = t} \subseteq \mathbb{F}$ is sampled uniformly at random, constrained to $\sum_{A \subseteq [n], |A| = t} s_A = s$. Each share $\mathbf{s}_i$ for $i \in [n]$ is defined to be a vector itself, which is given by $\mathbf{s}_i = (s_A)_{A \subseteq [n], i \notin A}$.*

Now we analyze the required properties by a linear secret-sharing scheme.

- *Given any set $B \subseteq [n]$ with $|B| \leq t$, the collection of shares $\{\mathbf{s}_i\}_{i \in B}$ miss the value $s_{B'}$ for any $B \subseteq B' \subseteq [n]$. As a result, these shares together do not have enough information to reconstruct s, since all of the "additive values" $\{s_A\}_{A \subseteq [n], |A| = t}$ are needed to do so.*
- *Given any set $B \subseteq [n]$ with $|B| \geq t+1$, the collection of shares $\{\mathbf{s}_i\}_{i \in B}$ contains all the summands $\{s_A\}_{A \subseteq [n], |A| = t}$, which enables the reconstruction of the secret s. To see this,*

let $A \subseteq [n]$ with $|A| = t$. The summand s_A is included in all $\mathtt{s}_i$ for $i \notin A$, and since $|A| = t < t + 1 \leq |B|$, there is at least one such indexes i in the set B.

- *The fact that the construction above constitutes a* linear *secret-sharing scheme is straightforward to see.*

2.2 MPC based on Linear Secret-Sharing Schemes

Consider a linear secret-sharing scheme $[\![\cdot]\!]$. As we know, thanks to the linearity properties of $[\![\cdot]\!]$, it is possible for the parties to obtain $[\![x+y]\!]$ given two shared values $[\![x]\!]$ and $[\![y]\!]$. Now, suppose that the parties count on a method to obtain, not only the addition (and subtraction), but also the product of two shared values. More precisely, suppose the parties can obtain $[\![x \cdot y]\!]$ from two shared values $[\![x]\!]$ and $[\![y]\!]$, possibly by performing some interaction. With this at hand, the parties can easily compute the given arithmetic circuit by following this procedure:

1. Each party P_i, having input x_i, distributes shares of this value to the other parties, so the parties obtain $[\![x_i]\!]$.

2. For each operation gate in the circuit where the inputs x and y are secret-shared as $[\![x]\!]$ and $[\![y]\!]$, the parties proceed as follows:

    - If the gate is an *addition gate*, then the parties use the linear property of the secret-sharing scheme to obtain $[\![x+y]\!]$ without any interaction.

    - If the gate is an *multiplication gate*, then the parties use the assumed method to obtain $[\![x \cdot y]\!]$, potentially with interaction.

    - Eventually, the parties get shares of the result of the computation $[\![z]\!]$. At this point, $t + 1$ of the parties announce their shares to all the others so that the parties, having at least $t + 1$ shares, can reconstruct the output z.

From the template above, we see that, to design a secure multiparty computation protocol, it suffices to consider a linear secret-sharing scheme with the same threshold as the upper bound on corrupted parties, together with a method to obtain $[\![x \cdot y]\!]$ from $[\![x]\!]$ and $[\![y]\!]$. As we will see throughout this text, this general template is highly effective for building efficient protocols in a wide variety of settings, and, most of the time, the major complications appear not in the secret-sharing scheme itself, but in the procedure to multiply secret-shared values.

2.2.1 The Case of an Active Adversary

The general template above works well if the adversary is passive, but, when the corruptions are active, care must be taken in some parts of the protocol. First, naturally, the assumed method to securely compute multiplications must be actively secure, since otherwise an active adversary can attack the whole protocol by simply attacking multiplication gates. However, the phase where shares of the inputs are distributed must also be revisited, since, as we will see in subsequent sections, there are several secret-sharing schemes where, if the party distributing shares behaves maliciously, sending perhaps "incorrect" shares, the protocol can be rendered insecure due to "inconsistencies" created among the parties.

To prevent actively corrupt parties from secret-sharing their inputs incorrectly, a typical approach consists of somehow reducing this to the broadcast channel by using a random shared value $[\![r]\!]$, where the secret t is known by the party providing input. If the input is x, this party simply needs to use the broadcast channel to announce $e = x - r$, which leaks nothing about x since r is uniformly random and only known to the party sending the message, and then the other parties can locally compute $[\![x]\!] = [\![r]\!] + e$, which leads to shares of the input x since $r + e = x$. This method has the advantage that, assuming that $[\![r]\!]$ was distributed "correctly", the resulting shares of x will also be correct.

Finally, another place where the adversary can attack the protocol is in the last step, where the shares of certain parties are announced in order to reconstruct the result of the computation. In this case, an actively corrupt party can simply lie about his own share, and it is not clear what would happen in such scenario. Indeed, this is a problem we will need to deal with throughout this work, and such attack, unless countered, tends to lead to the parties reconstructing a wrong result.

This type of behavior is addressed by enhancing the secret-sharing scheme with some method to check that the announced shares are somehow "correct", and have not been modified. In some settings, especially the $t < n/2$ and $t < n/3$ scenarios, this can be achieved without modifying the sharing procedure itself, while in some other cases, like in dishonest majority, some additional information that aids at ensuring integrity must be added. Details on all these problems and different approaches to solve them in various settings are given throughout this text, although the first relevant section where such techniques are encountered is Section 3.2.

2.2.2 Offline-Online Paradigm

In several cases, part of the interaction involved during the execution of a given secure multiparty computation protocols is independent of the inputs from the parties. For example, when we discussed in the previous section the general idea to obtain secret-shared inputs correctly, we made use of a secret-shared value $[\![r]\!]$ where r is known by the input provider. This type of data has to be produced by certain interaction in the MPC protocol, but it is independent of any of the inputs to the computation.

It is common in the field to refer to the steps in a given protocol that *do not* depend on the inputs to the computation as the *preprocessing* or *offline* phase, while the part of the protocol that requires the parties to know their inputs is typically called the *online* phase. The main motivation behind this terminology is that the preprocessing phase, being independent of the inputs, can be executed at the very start of the protocol execution, and after it is over the parties become ready to provide inputs and "actually compute" the function.

The distinction between an offline and an online phase is not only relevant at the language level. Consider two protocols having roughly the same performance overall, except that one has a very efficient online phase when compared to the other. Even though both protocols perform similarly, the total latency since the moment the inputs are provided until the output is obtained is smaller in the protocol with a fast online phase. Imagine a setting where the parties are idle before running the computation, which is scheduled in advance. This time can be then used to execute the offline phase, so that the parties are ready to run the efficient online phase when the inputs are known.

As we will see in Chapters 7 and 8, this offline/online paradigm takes even more relevance in the dishonest majority, since in this case, as we saw in Section 1.3, protocols must make use of heavy cryptographic tools to operate. Modern constructions, such as the ones we consider here, push all the complexities and inefficiencies of these mechanisms to the offline phase, while leaving a relatively simpler and much more efficient online phase (which, in addition, typically enjoys information-theoretic security).

Part II

Honest Majority

Chapter 3

Shamir Secret-Sharing

We begin by presenting the construction and properties of a very popular and widely used secret-sharing scheme, namely Shamir secret-sharing scheme. This was proposed by Adi Shamir in [38], and it is one of the most widely known and used examples of a linear secret-sharing scheme over a field.

Let $\mathbb{F}$ be a field of size q, where q is a power of a prime. Assume that $q > n$, and let $\alpha_0, \alpha_1, \ldots, \alpha_n \in \mathbb{F}$ be different points in $\mathbb{F}$. We denote by $\mathbb{F}_{\leq d}[\mathtt{X}]$ the $\mathbb{F}$-module of univariate polynomials over $\mathbb{F}$ of degree at most d in the variable $\mathtt{X}$. Let $\mathbb{F}^{u\times v}$ denote the set of matrices with dimensions $u \times v$.

Given $\beta_1, \ldots, \beta_u \in \mathbb{F}$, let $\mathsf{Van}^{u\times v}(\beta_1, \ldots, \beta_u) \in \mathbb{F}^{u\times v}$ be the matrix given by

$$\mathsf{Van}^{u\times v}(\beta_1, \ldots, \beta_u) \in \mathbb{F}^{u\times v} := \begin{pmatrix} 1 & \beta_1^1 & \beta_1^2 & \cdots & \beta_1^{v-1} \\ 1 & \beta_2^1 & \beta_2^2 & \cdots & \beta_2^{v-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \beta_u^1 & \beta_u^2 & \cdots & \beta_u^{v-1} \end{pmatrix}.$$

When the β's are clear from context we denote this matrix simply by $\mathsf{Van}^{u\times v}$. This is called a *Vandermonde matrix*, and it is well known that if $u = v$ (so the matrix is square) its determinant is equal to $\prod_{i<j}(\beta_i - \beta_j)$. In particular, this determinant is non-zero (and hence the matrix is invertible) if and only if all β's are different. Let $d \geq 0$ and let $\beta_0, \ldots, \beta_d \in \mathbb{F}$ be all different. Since given a polynomial $f(\mathtt{X}) = \sum_{i=0}^{d} c_i \mathtt{X}^i \in \mathbb{F}_{\leq d}[\mathtt{X}]$ it holds that

$$(f(\beta_0), \ldots, f(\beta_d))^\intercal = \mathsf{Van}^{(d+1)\times(d+1)}(\beta_0, \ldots, \beta_d) \cdot (c_0, \ldots, c_d)^\intercal,$$

this shows that every polynomial of degree at most d is determined by its evaluation at any $d + 1$ distinct points.

To secret-share a value $s \in \mathbb{F}$ using Shamir secret-sharing, the dealer samples a polynomial $f(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$ at random, restricted only to $f(\alpha_0) = s$. The share corresponding to party P_i is then $f(\alpha_i)$. As an example, if $\alpha_0 = 0$, it is easier to see more explicitly how such sampling could be done: the dealer samples $c_1, \ldots, c_d \in_R \mathbb{F}$ and sets $f(\mathtt{X}) = s + \sum_{i=1}^{d} c_i \mathtt{X}^i$, but for a general α_0 the process is slightly more complex, as we describe below.

Shamir Secret-Sharing

The dealer secret-shares a value $s \in \mathbb{F}$ among n parties $P_1, \ldots, P_n$ as follows.

1. Sample $s_1, \ldots, s_d \in_R \mathbb{F}$ and define

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_d \end{pmatrix} = \begin{pmatrix} 1 & \alpha_0^1 & \alpha_0^2 & \cdots & \alpha_0^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_d^1 & \alpha_d^2 & \cdots & \alpha_d^d \end{pmatrix}^{-1} \cdot \begin{pmatrix} s \\ s_1 \\ \vdots \\ s_d \end{pmatrix}.$$

   Let $f(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$ be given by $f(\mathtt{X}) = \sum_{i=0}^{d} c_i \mathtt{X}^i$.

2. For each $i = 1, \ldots, n$, the dealer distributes the share $f(\alpha_i)$ to party P_i

Reconstruction from any $d+1$ shares

Given shares $\{f(\alpha_i)\}_{i\in\mathcal{A}}$ for some subset $\mathcal{A} \subseteq [n]$ with $|\mathcal{A}| = d+1$, the secret is reconstructed as follows.

1. Fix some ordering in $\mathcal{A}$ and let us denote $(\alpha_i)_{i\in\mathcal{A}} = (\alpha_{a_1}, \ldots, \alpha_{a_{d+1}})$. Let

$$(\lambda_{a_1}^{\mathcal{A}}, \ldots, \lambda_{a_{d+1}}^{\mathcal{A}}) = (1, \alpha_0, \ldots, \alpha_0^d) \cdot \mathsf{Van}^{(d+1)\times(d+1)}(\alpha_{a_1}, \ldots, \alpha_{a_{d+1}})^{-1}.$$

2. The secret is computed as $s = \sum_{i=1}^{d+1} \lambda_{a_i}^{\mathcal{A}} \cdot f(\alpha_{a_i})$.

Definition 3.1. *Given $\mathcal{A} \subseteq \{1, \ldots, n\}$ with $|\mathcal{A}| = d+1$, we call the values above $\{\lambda_i^{\mathcal{A}}\}_{i\in\mathcal{A}}$* Lagrange coefficients. *These can be alternatively computed as*

$$\lambda_i^{\mathcal{A}} = \prod_{j\in\mathcal{A},\ j\neq i} \frac{\alpha_0 - \alpha_j}{\alpha_i - \alpha_j}.$$

The reconstruction of a secret distributed with Shamir secret-sharing from any $d+1$ shares is done by using these shares, which are evaluations of a polynomial of degree at most t, to recover such polynomial, followed by its evaluation at α_0. This is written much more explicitly in the description of the protocol above, which in particular shows that the secret is computed as a linear combination of the shares involved, a fact that will be used later in one of our described protocols.

Privacy of this secret-sharing scheme, that is, the fact that any set of d shares does not leak anything about the secret s, is also easy to see, as these are in a 1-1 correspondence with the randomness used by the dealer. To see this, consider for example the case in which the given set of shares is $\{f(\alpha_i)\}_{i=1}^{d}$. In the way that the secret-sharing scheme is defined, these d values constitute the seed used by the dealer, which are taken completely at random and independently from the secret s. In the general case it can be shown that there is a bijective affine transformation between the randomness used by the dealer and any set of d shares, which shows that these shares are uniformly random and independent of the secret. Thus, if we want to use Shamr secret-sharing to protect a secret against an adversary that may potentially obtain t shares, it suffices to take $d \geq t$. We omit the proof of this result since it follows in a straightforward manner from the properties introduced so far, and it is only heavy in notation.

3.1 Secret-Sharing and d-Consistency.

In our protocols, the parties will hold shares of different values, and the various parties will play the role of the dealer in Shamir secret-sharing to distribute certain secrets. We begin with the following definition.

Definition 3.2. *Let $\beta_1, \dots, \beta_\ell \in \mathbb{F}$ be all different. Given $\boldsymbol{s} = (s_1, \dots, s_\ell) \in \mathbb{F}^\ell$, we say that $\boldsymbol{s}$ is d-*consistent *if there exists $f(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$ such that $s_i = f(\beta_i)$ for $i = 1, \dots, \ell$. Observe that if $\ell \leq d+1$, then every vector $\boldsymbol{s} \in \mathbb{F}^\ell$ is d-consistent, but if $d + 1 < \ell$, then the set of d-consistent vectors constitutes a strict $\mathbb{F}$-vector subspace of $\mathbb{F}^\ell$.*

If the parties hold d-consistent Shamir shares $(s_1, \dots, s_n)$ of a value s, we denote this by $[\![s]\!]_d = (s_1, \dots, s_n)$. This definition makes sense in the context of a passive adversary: dealers will always follow the protocol so they will distribute valid shares, and the (passively) corrupt parties will always use the correct shares they hold when required. However, in the context of an active adversary, the following can happen:

- An actively corrupt dealer distributes shares $(s_1, \dots, s_n)$ that are not d-consistent.
- Even if a dealer distributes d-consistent shares $(s_1, \dots, s_n)$, an actively corrupt party P_i can modify its own share from s_i to any value s_i' at any given point.

Given these issues, it does not make too much sense to say that the parties *hold* a vector of shares $(s_1, \dots, s_n)$ (again, since the actively corrupt parties can change their shares). To address this, we expand the definition of $[\![s]\!]_d$ and d-consistency to the setting of an active adversary, as follows.

Definition 3.3. *Consider the setting of an active adversary. Let $\mathcal{H}, \mathcal{C} \subseteq [n]$ be the sets of indexes corresponding to honest and corrupt parties, respectively. We say that the parties hold d-*consistent *shares of a secret $s \in \mathbb{F}$ if there exists a polynomial $f(\mathtt{X})$ of degree at most d such that:*

1. *$s = f(\alpha_0)$;*
2. *Each honest party P_i for $i \in \mathcal{H}$ has $s_i = f(\alpha_i)$ (i.e. the honest parties' shares are d-consistent);*
3. *The adversary knows $s_j = f(\alpha_j)$ for $j \in \mathcal{C}$.*

Furthermore, when this holds, we write $[\![s]\!]_d = (s_1, \dots, s_n)$.[1]

One aspect that is not very formal from the definition above is what it means for the adversary to "know" a given value. This is formalized in the context of simulation-based

[1]We clarify that sometimes the notation $[\![s]\!]$ instead of $[\![s]\!]_d$ will be used when the degree d is clear from context.

proofs by requiring that the simulator is able to "extract" the given value from the adversary, after interacting with it on the emulated protocol execution. Details are left to the relevant sections that make use of this concept, such as Chapter 6. However, some intuition on this notion can be provided. Consider for example a setting in which an honest dealer sends d-consistent shares $(s_1, \dots, s_n)$ of a secret s to the parties, with $s_i = f(\alpha_i)$ for some $f(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$, but the corrupt parties P_i change their shares from s_i to s_i'. Even though the resulting vector may not be d-consistent anymore, the parties still have d-consistent shares $[\![s]\!]_d$: (1) the honest parties' shares are d-consistent, and (2) the adversary *knows* the "real" values s_i corresponding the corrupt parties P_i.

Remark 3.1. *Let $\mathcal{H} \subseteq [n]$ be the set of indexes corresponding to honest parties. Let $[\![s]\!]_d = (s_1, \dots, s_n)$ be a d-consistent sharing, which means that there exists a polynomial $f(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$ such that $s = f(\alpha_0)$, $s_i = f(\alpha_i)$ for $i \in \mathcal{H}$, and the adversary knows $s_j = f(\alpha_j)$ for $j \in [n] \setminus \mathcal{H}$. If $|\mathcal{H}| = n - t \geq d + 1$, then the polynomial $f(\mathtt{X})$ is unique, and in particular, so is the secret s. However, if $n - t \leq d$ then, for any secret $s' \in \mathbb{F}$, there exists a polynomial $f_{s'}(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$ such that $f_{s'}(\alpha_0) = s$ and $f_{s'}(\alpha_i) = s_i$ for $i \in \mathcal{H}$, so if the adversary knows $s_j' = f_{s'}(\alpha_j)$ for $j \in [n] \setminus \mathcal{H}$, the parties would simultaneously "hold" shares $[\![s']\!]_d$ of* any *secret s', or, in other words, there is not a well-defined secret from the honest shares alone.*

Finally, notice that if $t \geq n/2$ and $d = t$, then $n - t \leq d$. This is the reason why Shamir secret-sharing is typically only used if $t < n/2$: in this setting there are at least $t + 1$ honest parties, so their honest shares, if t-valid, define a unique secret.

Homomorphisms. Consider two shared values $[\![x]\!]_d$ and $[\![y]\!]_d$, using polynomials $f(\mathtt{X}), g(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$, respectively (that is, P_i's share of x is $f(\alpha_i)$ and the one of y is $g(\alpha_i)$). If each party adds their shares together, that is, each P_i computes $f(\alpha_i) + g(\alpha_i)$, they obtain shares of $x + y$ under the polynomial $f(\mathtt{X}) + g(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$. This is denoted by $[\![x + y]\!]_d \leftarrow [\![x]\!]_d + [\![y]\!]_d$. On the other hand, if the parties locally multiply their shares, they obtain shares of $x \cdot y$ under the polynomial $f(\mathtt{X}) \cdot g(\mathtt{X}) \in \mathbb{F}_{\leq 2d}[\mathtt{X}]$. This is denoted by $[\![x \cdot y]\!]_{2d} \leftarrow [\![x]\!]_d \cdot [\![y]\!]_d$. Note that the degree of the polynomial increases from d to $2d$.

Finally, observe that subtraction can be performed locally in a similar way as addition, as well as multiplying by any constant (that is, a value known by all parties). Furthermore, the parties can also locally add a constant, that is, obtain $[\![x + c]\!]$ from $[\![x]\!]$ and c, by each party adding this constant to their share.

3.2 Error Detection/Correction

As it has been already mentioned, Shamir secret-sharing is used, in the context of MPC, in order to distribute the inputs of the computation, as well as the intermediate values and the output, among the different parties, in such a way that the adversary does not learn anything about the underlying secrets. In the process of securely computing the given function, it will be necessary for the parties to *reconstruct*, or *open*, some secret-shared values. This is the case, for example, for obtaining the output of the computation, which is in secret-shared form and must be learned by all the parties. Additionally, processing certain operations like multiplications during the course of the computation requires the parties to learn certain secret-shared data.

Reconstructing a secret-shared value $[\![s]\!]$ can be achieved, for example, by asking $t+1$ parties to send their shares to all the other parties.[2] After a party receives at least $t+1$ shares (including possibly its own), this party can reconstruct the polynomial of degree at most t that interpolate these shares, hence obtaining the secret.

Unfortunately, when the parties are actively corrupt, they can misbehave in the reconstruction of a secret-shared value by sending incorrect shares. For example, a secret reconstructed from the shares $(s_1, \dots, s_{t+1})$ would be computed as $s = \sum_{i=1}^{t+1} \lambda_i^{[t+1]} \cdot s_i$, but if party P_1 announces a different share $s_1 + \delta_1$, then the reconstructed secret would be $s + \delta$, with $\delta = \lambda_1^{[t+1]}\delta_1$. Furthermore, the parties do not have a way of detecting that a change has been introduced, given that $(s_1 + \delta_1, s_2, \dots, s_{t+1})$ look like legitimate shares of $s + \delta$.

The reason why the parties cannot detect that an error has been introduced is because any set of $t+1$ values $(s'_1, \dots, s'_{t+1})$ is consistent with a polynomial of degree at most t. However, if the parties use, say, $t+2$ shares $(s_1, \dots, s_{t+1}, s_{t+2})$ to first check the existence of a polynomial $f(\mathtt{X})$ such that $f(\alpha_i) = s_i$ for $i = 1, \dots, t+2$, then it is "less likely" that the error δ_1 that P_1 introduced preserves the existence of this polynomial. This is because a vector of $t+2$ shares is not necessarily consistent with a polynomial of degree at most t, and in fact, we can show that if $\delta_1 \neq 0$ then there is no way in which $(s_1 + \delta_1, \dots, s_{t+1}, s_{t+2})$ is consistent with a polynomial of degree at most t.

Unfortunately, the method above of checking consistency using $t+2$ shares is insufficient to prevent an attack that modifies the reconstructed secret, given that the adversary has the power to modify not only P_1's share, but the shares of at most t parties. The following example shows that, if the parties only check that $2t$ shares are consistent with a polynomial of degree at most t, then the adversary, who controls t parties actively, can cause the reconstructed secret to be incorrect, without being detected. This is related to Remark 3.1, where it is mentioned that if $n' - t \leq d$, where n' is the total number of shares (so $2t$ for the purpose of this example, where also $d = t$), then the secret is not well-defined by the honest shares alone, and the corrupt parties can modify theirs to result in t-valid sharings of any secret.

Example 3.1. *Let $[\![s]\!]_t = (s_1, \dots, s_n)$, where the underlying polynomial is $f(\mathtt{X})$, that is, $f(\alpha_0) = s$ and $f(\alpha_i) = s_i$ for $i \in [n]$. Let $\mathcal{A} \subseteq [n]$ be the set of the t indexes corresponding to corrupt parties, and let $\mathcal{B} \supseteq \mathcal{A}$ such that $|\mathcal{B}| \leq 2t$. Suppose that at reconstruction time the parties check that the announced shares corresponding to indexes in $\mathcal{B}$ are consistent with a polynomial $h(\mathtt{X})$ of degree at most t, and output $h(0)$ if this is the case. Then the adversary can cause the parties to reconstruct a wrong secret as follows.*

1. *The adversary samples a polynomial $g(\mathtt{X})$ of degree at most t such that $g(\alpha_i) = 0$ for $i \in \mathcal{B} \setminus \mathcal{A}$ and $g(\alpha_0) = \delta$ for some $\delta \neq 0$ of the adversary's choice. Observe this is possible since $|(\mathcal{B} \setminus \mathcal{A}) \cup \{\alpha_0\}| \leq t+1$.*

[2]This incurs in a total communication complexity of $\approx t \cdot n$. This can be improved to $O(n)$, as described in Section 3.4.

2. *The corrupt parties P_i for $i \in \mathcal{A}$ modify their share s_i as $s_i' = s_i + \delta_i$, with $\delta_i = g(\alpha_i)$.*

At reconstruction time the parties check if there is a polynomial $h(\mathtt{X})$ of degree at most t such that $h(\alpha_i) = s_i$ for $i \in \mathcal{B} \setminus \mathcal{A}$, and $g(\alpha_i) = s_i'$ for $i \in \mathcal{A}$, and if this is the case, they output $h(0)$ as the reconstructed secret. Such polynomial indeed exists, namely $h(\mathtt{X}) = f(\mathtt{X}) + g(\mathtt{X})$. Indeed, if $i \in \mathcal{B} \setminus \mathcal{A}$, then $h(\alpha_i) = f(\alpha_i) + g(\alpha_i) = s_i + 0$, and if $i \in \mathcal{A}$ then $h(\alpha_i) = f(\alpha_i) + g(\alpha_i) = s_i + \delta_i = s_i'$. However, the reconstructed secret is equal to $h(\alpha_0) = f(\alpha_0) + g(\alpha_0) = s + \delta$.

In what follows we will see that the parties can check that the announced shares are correct if they use at least $2t + 1$ shares. In fact, we will see that they can identify the incorrect shares, remove them, and therefore reconstruct the right secret, if they use at least $3t+1$ shares. These results will be presented in a more general way, using terminology that resembles that in the field of error correcting codes, and later in Section 3.3 we will interpret what these results imply in our setting.

The results in this section are more general, since they are phrased in the context of error-correcting codes. Let ℓ and d be non-negative integers, and let $\beta_1, \ldots, \beta_\ell \in \mathbb{F}$ be all different. Let $\boldsymbol{s} = (s_1, \ldots, s_\ell) \in \mathbb{F}^\ell$ be a d-consistent vector, which, recalling from Definition 3.2, means that there exists $f(\mathtt{X}) \in \mathbb{F}_{\leq d}[\mathtt{X}]$ such that $s_i = f(\beta_i)$ for $i = 1, \ldots, \ell$. Suppose that this vector is modified as $\boldsymbol{s} + \boldsymbol{\delta}$, for some error vector $\boldsymbol{\delta} \in \mathbb{F}^\ell$. In the context of *error-correction*, the goal is to recover the polynomial f such that $f(\beta_i) = s_i$ for $i = 1, \ldots, \ell$ from the corrupted vector $\boldsymbol{s} + \boldsymbol{\delta}$. In *error-detection* we are only interested in determining whether $\boldsymbol{\delta}$ is non-zero.

3.2.1 Error Detection

First, notice that being d-consistent is an $\mathbb{F}$-linear property, so $\boldsymbol{\delta}$ is d-consistent if and only if $\boldsymbol{s} + \boldsymbol{\delta}$ is d-consistent as well. If these conditions hold, then any hope of error-correction or detection is lost given that $\boldsymbol{s} + \boldsymbol{\delta}$ will look as “legitimate” as $\boldsymbol{s}$. Given the above, we begin by looking at some conditions under which $\boldsymbol{\delta}$ can be d-consistent. Let e be an upper bound on the number of non-zero entries of δ. Observe that if $e < \ell - d$, then $\boldsymbol{\delta}$ cannot be d-consistent, unless it is the zero vector. This is because, if $\boldsymbol{\delta}$ was d-consistent, its $\ell - e > d$ zero entries would be enough to determine the underlying polynomial, which has to be the zero polynomial. On the other hand, if $e \geq \ell - d$, then it is easy to check that $\boldsymbol{\delta}$ could be d-consistent.[3]

We see then that, if $e < \ell - d$, the only way in which $\boldsymbol{s} + \boldsymbol{\delta}$ can be d-consistent is if $\boldsymbol{\delta} = \mathbf{0}$, so if no error was introduced. As a result, we can *detect* whether $\boldsymbol{\delta}$ is zero by checking if $\boldsymbol{s} + \boldsymbol{\delta}$ is d-consistent.

[3]In fact, this is the reason why the attack in Example 3.1 works. There we have $\ell \leq 2t$, $d = t$ and $e = t$, so $e \geq \ell - d$. This is also the reason why, if $n - t \leq d$, a degree-d secret-shared value is not well defined if the adversary is active, as pointed out in Remark 3.1.

3.2.1.1 A more intuitive view.

Another way to see the result displayed above is the following. Suppose that $e < \ell - d$, and that $\boldsymbol{s} + \boldsymbol{\delta}$ happens to be d-consistent. Then, the underlying polynomial could be recovered from any set of $d + 1$ entries, and in particular, it could be recovered from the $\ell - e \geq d + 1$ entries that were not affected by $\boldsymbol{\delta}$, which shows that the underlying polynomial of $\boldsymbol{s} + \boldsymbol{\delta}$ has to be the same as that of $\boldsymbol{s}$.

3.2.2 Error Correction

Unfortunately, even if $e < \ell - d$, knowing that $\boldsymbol{s} + \boldsymbol{\delta}$ is not enough to find the polynomial $f(\mathtt{X})$ underlying $\boldsymbol{s}$. For example, if $e = \ell - d - 1$, a pair of non-zero error vectors $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2 \in \mathbb{F}^\ell$ having at most e non-zero entries can be found such that $\boldsymbol{s}_2 = \boldsymbol{\delta}_1 - \boldsymbol{\delta}_2$ is d-consistent, but this is a problem since the modified vectors $\boldsymbol{s}_1 + \boldsymbol{\delta}_1$ and $\boldsymbol{s}_2 + \boldsymbol{\delta}_2$, where $\boldsymbol{s}_1 = \mathbf{0}$ (which is d-consistent), are the same, so given only this vector it is not possible to know if it is a modified version of $\boldsymbol{s}_1$ or of $\boldsymbol{s}_2$.

If the bound e is zero, then obviously we can always find f from $\boldsymbol{s} + \boldsymbol{\delta}$, so there must be a point in which the amount of errors in $\boldsymbol{\delta}$ is so small, that error-correction is possible. This point is reached when $e < (\ell - d)/2$, and moreover, this is optimal in the sense that, for $(\ell - d)/2 \leq e < \ell - d$, we can always build examples as the one suggested above that show that finding the original polynomial for $\boldsymbol{s}$ is not possible.

We claim that if $e < (\ell - d)/2$, if $\boldsymbol{s}_1, \boldsymbol{s}_2 \in \mathbb{F}^\ell$ are two different d-consistent vectors, and if $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2 \in \mathbb{F}^\ell$ are error vectors with at most e non-zero entries each, then $\boldsymbol{s}_1 + \boldsymbol{\delta}_1 = \boldsymbol{s}_2 + \boldsymbol{\delta}_2$ cannot hold. This would enable error correction of $\boldsymbol{s} + \boldsymbol{\delta}$ by looking through all possible error vectors with at most e non-zero entries, subtracting it from $\boldsymbol{s} + \boldsymbol{\delta}$, and checking if the result is d-consistent vector. To show the claim above simply notice that $\boldsymbol{s}_1 + \boldsymbol{\delta}_1 = \boldsymbol{s}_2 + \boldsymbol{\delta}_2$ implies that $\boldsymbol{\delta}_2 - \boldsymbol{\delta}_1 = \boldsymbol{s}_1 - \boldsymbol{s}_2$ would be a d-consistent vector, but this cannot happen since $\boldsymbol{\delta}_2 - \boldsymbol{\delta}_1$ has at most $2e < \ell - d$ non-zero entries, but we just showed above that a vector with strictly less than $\ell - d$ non-zero entries cannot be d-consistent unless it is the zero vector, which would imply that $\boldsymbol{s}_1 = \boldsymbol{s}_2$.

3.2.2.1 Efficient error-correction.

Above, we said that to error correct $\boldsymbol{s} + \boldsymbol{\delta}$, one would have to go over all possible vectors $\boldsymbol{\delta}$ could be equal to, which is of course very inefficient. This could be optimized slightly by looping over all possible subsets of e coordinates, and checking if the remaining coordinates of $\boldsymbol{s} + \boldsymbol{\delta}$ form a d-consistent vector (of dimension $\ell - e$). Because $(\ell - e) - d > e$, the results from before show that this can only happen if this “sub”-vector does not have any errors in it, which means that the guessed coordinates contain all the possible errors. This, unfortunately, is still too inefficient.

Instead, in practice we would recur to *error-correction algorithms*, also known as *decoders*, which achieve the task of identifying the error locations very efficiently. For the case at hand, we could use for example the Berlekamp-Welch algorithm [40], which is an efficient algorithm to solve the decoding problem.

3.2.2.2 A more intuitive view.

The process of error-correction can be also thought of as follows. Suppose that after looping through all subsets of coordinates of $\boldsymbol{s}+\boldsymbol{\delta}$ of size $\ell-e$, we find one that is d-consistent. Then, any set of $d+1$ coordinates among these determine all the others. Furthermore, we know that the chosen sub-vector has at most e errors, so it has at least $(\ell-e)-e=\ell-2e\geq d+1$ non-modified entries, which determine the polynomial completely. Since these entries are the same as in $\boldsymbol{s}$, we see that the polynomial underlying the chosen sub-vector is the same as the one for the original vector $\boldsymbol{s}$.

3.3 Error Correction/Detection in the Context of MPC

As motivated at the beginning of Section 3.2, the purpose of the theory of error detection/correction in the context of MPC is to allow parties to reconstruct secret-shared values correctly, in spite of the actively corrupt parties announcing incorrect shares. More precisely, the parties have a secret-shared value $[\![s]\!]_d=\boldsymbol{s}=(s_1,\dots,s_n)$, and in order to learn s, each party P_i announces its share s_i to the other parties. Actively corrupt parties may announce an incorrect $s_i+\delta_i$ for some error δ_i, which means the secret must be reconstructed from the shares $\boldsymbol{s}+\boldsymbol{\delta}$, where $\boldsymbol{\delta}\in\mathbb{F}^n$ has as its i-th entry 0 if i is an index corresponding to an honest party, and δ_i otherwise. Since there are t corrupt parties, there are at most t non-zero entries in $\boldsymbol{\delta}$. This puts us in the context of error correction/detection studied before with $\ell=n$ and $e=t$.

The results from the previous sections can be summarized as follows:

No error detection. If $e\geq\ell-d$, then the adversary can choose $\boldsymbol{\delta}$ so that $\boldsymbol{s}+\boldsymbol{\delta}$ is d-consistent, and the reconstructed secret will be $s+\delta$ for some δ of the adversary's choice.

Error detection. If $e<\ell-d$, then $\boldsymbol{s}+\boldsymbol{\delta}$ is d-consistent if and only if $\boldsymbol{\delta}=0$.

Error correction. If $2e<\ell-d$, then there exist efficient algorithms to recover $\boldsymbol{s}$ from $\boldsymbol{s}+\boldsymbol{\delta}$.

As we will see in subsequent sections, the values of d that we will need to make use of in our protocols are $d=t$ and $d=2t$. Furthermore, the two main settings in which we will make use of these techniques are the honest majority setting, in which $t<n/2$, and two-thirds honest majority where $t<n/3$. The following table summarizes, for the cases of $t<n/2$ and $t<n/3$, and for the different degrees $d=t$ and $d=2t$, when error detection/correction is possible. These results will be used later in Chapters 5 and 6 when we construct actively secure protocols with $t<n/3$ and $t<n/2$ respectively.

		error corr.	**error det.**
$t < n/3$	degree t	✓	✓
	degree $2t$	✗	✓
$t < n/2$	degree t	✗	✓
	degree $2t$	✗	✗

3.4 Reconstructing Secret-Shared Values Efficiently

The previous section addressed the question of ensuring the adversary does not fool the parties into reconstructing a secret-shared value $[\![s]\!]_d$ incorrectly as $s + \delta$. However, doing this naively would require each party to send its share to all other parties, which incurs in a total communication complexity of at least $n \cdot t$ field elements being transmitted. To alleviate this issue, an alternative is to let the parties send their shares to a single "intermediate" receiver, who reconstruct the secret and then informs all the parties of this result. This is in fact the approach taken in the protocol from Section 4.2.1 to reconstruct $[\![a]\!]_{2t}$: the parties send their shares to P_1, who sends the result back to all the other parties.

Unfortunately, the issue that arises with this approach is that an actively corrupt intermediate receiver may choose to lie about the reconstructed value, which would ultimately lead to the parties learning an incorrect secret. Designing a solution to this problem while still achieving linear communication complexity is far from trivial, and the one we will present below was introduced in [20].

Suppose that the parties are not reconstructing one value $[\![s]\!]_d$, but many of these $[\![s_0]\!]_d, [\![s_1]\!]_d, \dots, [\![s_d]\!]_d$. Consider the polynomial $f(\mathtt{X}) = \sum_{j=0}^{d} s_j \mathtt{X}^j$. The parties can compute $[\![f(\alpha_i)]\!]_d = \sum_{j=0}^{t} [\![s_j]\!]_d \alpha_j^i$ for $i \in [n]$ (observe that each $f(\alpha_i)$ can seen as a degree-d share of $f(\alpha_0)$). Then, the parties can reconstruct each of these shares towards the corresponding parties, which leads the parties to obtain shares $[\![f(\alpha_0)]\!]_d = (f(\alpha_1), \dots, f(\alpha_n))$. At this point they can reconstruct this new secret using the naive approach from above: each party P_i sends its share $f(\alpha_i)$ to *all* other parties. This enables each party to error correct/detect in order to recover the secret $f(\alpha_0)$, but, what is more important, is that each party will not only recover the secret, but the polynomial $f(\mathtt{X})$ itself, whose coefficients are $s_0, s_1, \dots, s_d$.

Regarding communication complexity, the solution above involves $\Theta(d \cdot n)$ field elements. However, since $d + 1$ secret-shared values are reconstructed, the amortized complexity per secret is $\Theta(n)$, as required. The protocol is summarized below.

$\Pi_{\mathsf{PublicRec}}$: Efficient Public Reconstruction

Input: Secret-shared values $[\![s_0]\!]_d, \dots, [\![s_d]\!]_d$
Output: All the parties learn $s_0, \dots, s_d$.
Protocol: The parties proceed as follows

1. Let $f(\mathtt{X}) = \sum_{j=0}^{d} s_j \mathtt{X}^j$. The parties locally compute $[\![f(\alpha_i)]\!]_d = \sum_{j=0}^{d} [\![s_j]\!]_d \alpha_i^j$ for $i \in [n]$.

2. Each party P_k for $k \in [n]$ sends its share of $[\![f(\alpha_i)]\!]$ to P_i, for $i \in [n]$.
3. Upon receiving the shares of $[\![f(\alpha_i)]\!]$, each P_i for $i \in [n]$ does the following:
   - If $n > d + 2t$, then perform error correction to recover $f(\alpha_i)$.
   - If $n > d + t$, then perform error detection to either recover $f(\alpha_i)$, or fail at reconstruction. If the latter happens then the party aborts.
4. If no abort was produced in the previous step, each P_i for $i \in [n]$ sends the reconstructed $f(\alpha_i)$ to each other party P_j.
5. Upon receiving the shares, each party P_j proceeds as follows:
   - If $n > d + 2t$, then perform error correction to recover the polynomial $f(\mathtt{X})$, and output its coefficients $s_0, \ldots, s_t$.
   - If $n > d + t$, then perform error detection to either recover the polynomial $f(\mathtt{X})$, outputting its coefficients, or fail at reconstruction. If the latter happens then abort.

To see why the protocol works as intended we proceed as follows. First, by the results from Section 3.3, each party P_i for $k \in [n]$, upon receiving the shares of $[\![f(\alpha_i)]\!]_d$ from the other parties, is able to perform error correction if $n > d + 2t$ to recover $f(\alpha_i)$, or alternatively it can perform error detection if $n > d + t$.

Now notice the polynomial $f(\mathtt{X})$ has degree at most d, and at this point if no abort has happened the parties hold the evaluation points $(f(\alpha_1), \ldots, f(\alpha_n))$. Again by the results from Section 3.3 we have that, when these shares are announced, the parties can perform error correction if $n > d + 2t$ to recover not only the "secret" $f(\alpha_0)$, but the polynomial $f(\mathtt{X}) = \sum_{j=0}^{d} s_j \mathtt{X}^j$, and if $n > d + t$, error detection can be performed. As a result, if no party aborts, the parties finish the protocol reconstructing the correct original secret-shared values.

Remark 3.2. *The protocol $\Pi_{\mathsf{PublicRec}}$ requires the parties to reconstruct several secret-shared values $[\![s_0]\!]_d, \ldots, [\![s_d]\!]_d$ in order to benefit from the improved efficiency. However, it can be the case during certain steps of a protocol execution only a few secret-shared values must be reconstructed. For example, this is the case if the function being computed only has one output, which is obtained in secret-shared form and must be reconstructed. In this case, instead of using the protocol $\Pi_{\mathsf{PublicRec}}$, the parties can simply use the more naive approach of sending the shares to each other in one single round, with each party error correcting on its own to get the output. This involves quadratic communication complexity, but this is acceptable since this is only called once at the end of the protocol execution.*

Chapter 4

Passive and Perfect Security for Honest Majority

With the tools that have been described so far we are now ready to describe a perfectly secure protocol that can withstand a passive adversary corrupting t parties where $t < n/2$. Recall from Section 1.3.2 that the combination of passive and perfect security with honest majority cannot be enhanced in any way, in the sense that improving any of the three properties degrades the others. For example, shifting from passive to active security requires us to either switch from perfect to statistical simulation, or from honest majority to two-thirds honest majority. Also, for instance, if the threshold $t < n/2$ does not hold, that is $t \geq n/2$, then even if we settle with passive security, the simulation would have to be computational.

Notice that, since in this first protocol the adversary is assumed to be passive, the theory on error correction/detection developed in Section 3.2 does not play a role here given that the corrupt parties do not modify their shares when reconstructing a secret-shared value. This theory will be used in Chapters 5 and 6 when we consider active adversaries.

4.1 A First Protocol

Let us begin with a protocol that is conceptually very simple, which is taken from [5, 25].

Recall from Chapter 3 that, for any subset of indexes $\mathcal{A} \subseteq [n]$ of size $d+1$, there exists coefficients $\lambda_1^{\mathcal{A}}, \dots, \lambda_{d+1}^{\mathcal{A}}$ such that, whenever the parties have a shared secret $[\![s]\!]_d = (s_1, \dots, s_n)$, this value can be computed as $s = \sum_{i \in \mathcal{A}} \lambda_i^{\mathcal{A}} \cdot s_i$. This will be used in the protocol below.

Passively secure protocol with perfect security

Setting: Each party P_i has input $x_i \in \mathbb{F}$.

Input phase: Each party P_i acts as the dealer in Shamir secret-sharing to distribute shares of its input x_i using polynomials of degree t. The parties obtain $[\![x_i]\!]_t$.

Addition gates: For each addition gate with secret-shared inputs $[\![x]\!]_t$ and $[\![y]\!]_t$, the parties locally compute $[\![x+y]\!]_t \leftarrow [\![x]\!]_t + [\![y]\!]_t$.

Multiplication gates: For each multiplication gate with secret-shared inputs $[\![x]\!]_t$ and $[\![y]\!]_t$, the parties proceed as follows:

1. The parties compute locally $[\![x \cdot y]\!]_{2t} \leftarrow [\![x]\!]_t \cdot [\![y]\!]_t$.
2. Let us write $[\![x \cdot y]\!]_{2t} = (z_1, \dots, z_n)$. Let $\lambda_1, \dots, \lambda_{2t+1} \in \mathbb{F}$ be the (publicly known) coefficients such that $x \cdot y = \sum_{i=1}^{2t+1} \lambda_i \cdot z_i$. Each party P_i for $i = 1, \dots, 2t+1$ acts as the dealer in Shamir secret-sharing to distribute shares $[\![z_i]\!]_t$.
3. The parties compute locally $[\![x \cdot y]\!]_t = \sum_{i=1}^{2t+1} \lambda_i \cdot [\![z_i]\!]_t$

Output phase: Let $[\![z]\!]_t$ be the output of the computation.

1. Each P_i for $i = 1, \dots, t+1$ sends its own share of z to all the other parties.
2. After each party P_i receives $t+1$ shares (possibly counting its own), it reconstruct the output z.

Correctness and privacy of the protocol should be clear given that the adversary is passive: after the input phase the parties have shares of each input to the protocol, which does not leak information to the adversary due to the privacy properties of Shamir secret-sharing scheme since the adversary knows only t shares coming from the corrupt parties, which are not enough to determine a $[\![\cdot]\!]_t$-shared secret. After the input phase, all parties proceed in a "gate-by-gate" fashion, computing shares of each intermediate value, or wire, in the computation. Addition gates are clearly correct since these make use of the homomorphic properties of Shamir secret-sharing. For multiplication gates, correctness follows from inspection, and privacy follows again from the fact that each party P_i is distributing information only in secret-shared form, which does not leak information towards the adversary.

Intuition for the security proof

As explained at the beginning of the chapter, we will not include formal security proofs for now, as these are delayed until Part II of this work, where the actual contributions are presented. However, it is still a fruitful exercise to provide some intuition as to how such proof would work in the protocol we have at hand here.

In order to obtain a proof of the security of the protocol described above, we would have to define a simulator $\mathcal{S}$ that interacts with the adversary corrupting t parties, in such a way that the adversary does not know if he is interacting in a real-world execution of the protocol, where the actual honest parties are running the protocol at hand, or in a ideal-world execution, where the honest parties only provide their inputs to an ideal functionality that is in charge of computing the function on the given inputs and returning only the output.

For simplicity in the notation, let us assume that the corrupt parties are $P_1, \dots, P_t$. The simulator $\mathcal{S}$ can interact with the t corrupt parties, and also with the functionality that computes the given function ideally by providing inputs to it on behalf of the corrupt parties. $\mathcal{S}$ needs to "fool" the adversary into believing that he is interacting with real honest parties, so to do this $\mathcal{S}$ emulates a set of honest parties $\overline{P}_{t+1}, \dots, \overline{P}_n$ that will interact with the corrupt parties in what should look like a genuine execution of the protocol. Notice that the trick here is that the simulator does not know what the actual inputs from the honest parties are!

Let us begin by analyzing how the input phase could be simulated. In the protocol description, all parties need to secret-share their input. $\mathcal{S}$ does not know the inputs from the honest parties, so it cannot distribute towards the corrupt parties shares of these inputs. However, this does not matter: the adversary corrupts only t parties, and recall from the properties of Shamir secret-sharing that any set of t shares looks completely random. Hence, the emulated honest parties can simply send random values to the t corrupt parties as the shares of their inputs, without actually knowing what their inputs are. Therefore, so far, the adversary cannot tell whether he is interacting with the actual honest parties in the real world, or with the simulator in the ideal world.

On the other hand, as part of the input phase, the emulated honest parties $\overline{P}_{t+1}, \ldots, \overline{P}_n$ will receive shares of the inputs $x_1, \ldots, x_t$ from the corrupt parties $P_1, \ldots, P_t$, and since there are at least $t + 1$ parties among the parties emulated by $\mathcal{S}$ (given that $t < n/2$), this enables him to reconstruct these inputs $x_1, \ldots, x_t$. Furthermore, this also enables $\mathcal{S}$ to reconstruct not only the inputs $x_1, \ldots, x_t$, but also the shares that the corrupt parties have of these inputs (since any $t + 1$ shares not only determine the secret, but the polynomial that was used to distribute it, so in particular they also determine all of the other shares).

At this point the corrupt parties have “shares” of all the inputs, with the quotes serving the purpose of emphasizing that the shares corresponding to the inputs of honest parties are just random values. On the other hand, the emulated honest parties do not really have any share. In a sense, the only parties that hold shares are the corrupt ones, but there are only t of them so these shares do not actually determine any secret, which enables the computation to “take place”, even though in the ideal world only the inputs and the outputs exist. This will be made clearer in subsequent paragraphs.

Observe that $\mathcal{S}$ knows all the shares that the corrupt parties have, which is crucial as we will show towards the end. The next steps of the computation involve proceeding gate-by-gate, obtaining shares of their outputs from shares of the inputs. Addition gates are handled in a simple way as they require no interaction: the corrupt parties simply add their shares together (notice in particular that the simulator still knows the shares held by the corrupt parties if it knew the ones for the input summands).

Multiplication gates require a bit more care. Suppose that the shares held by the corrupt parties $P_1, \ldots, P_t$ corresponding to the inputs of the multiplication are $(x_1, \ldots, x_t)$ and $(y_1, \ldots, y_t)$, which are known to $\mathcal{S}$. According to the description of the protocol, each corrupt party P_i will send to the other parties degree-t shares of $x_i \cdot y_i$. The simulator receives through the emulated honest parties at least $t + 1$ shares of each $x_i \cdot y_i$, which enables him to determine the shares that the other corrupt parties received. Also, the protocol requires parties $\overline{P}_{t+1}, \ldots, \overline{P}_{2t+1}$ to similarly secret-share the product of their shares of the inputs, but this is not possible since, again, the simulator does not know this information. This is again not a problem since these parties can simply send random shares to the t corrupt parties without actually knowing what value is being secret-shared. The adversary cannot distinguish this from what happens in the real execution.

Finally, the protocol reaches the output phase. The corrupt parties have shares of this output, and the simulator knows what these shares are. Recall that $\mathcal{S}$ learned the inputs $x_1, \ldots, x_t$ from the corrupt parties in the input phase. The simulator can interact with the functionality to obtain the output z of the computation, using the *real* inputs from the actual honest parties. This is the exact same output that would have been computed if the adversary was involved instead in a real-world execution, where the actual honest parties participate in the protocol. So far the adversary cannot tell the difference.

Since $\mathcal{S}$ knows the output z, and he also knows the t shares of this value that the corrupt parties have, these $t+1$ points enables $\mathcal{S}$ to compute what the share corresponding to $\overline{P}_{t+1}$ should be so that it look consistent with the corrupt parties' shares and the given output z. Once this is done, $\overline{P}_{t+1}$ can easily play the output phase of the protocol by sending to the corrupt parties the computed share. The adversary ends up reconstructing z, since this is how the share of $\overline{P}_{t+1}$ was computed, and this is exactly what would have happened in a real execution.

4.2 A More Efficient Protocol

The protocol described above is conceptually very simple. However, its main disadvantage is the amount of data the parties have to communicate measured in terms of n, the number of parties. Let M denote the number of multiplication gates in the circuit. For each of these gates, the protocol requires the parties $P_1, \ldots, P_{2t+1}$ to secret-share a value towards the other parties, which in turn requires each of these parties to send a share to each other party. This amounts to $n-1$ field elements sent by each of the parties in $\{P_1, \ldots, P_{2t+1}\}$, which gives a total of $\Theta(t \cdot n)$ field elements transmitted over the network, per multiplication gate. Alternatively, this can be written as $\Theta(t \cdot n \cdot M)$ for the whole computation.[1]

As a practical observation, it is natural to increase t as n increases, given that this value determines the amount of parties that the adversary needs to corrupt in order to break the privacy of the protocol, and it can be argued that the more parties that participate in the protocol, the "easier" it becomes for the adversary to corrupt a large portion of these. Furthermore, it is typical to consider the maximal case for which $t < n/2$, namely $t = \lfloor (n-1)/2 \rfloor$, an in this case the total communication complexity of the protocol above becomes $\Theta(n^2 M)$, which is also referred to as *quadratic communication complexity*. It would be much more ideal if we had a protocol with *linear communication complexity*, that is, $\Theta(nM)$. A protocol with such communication complexity has the property that, in average, the communication required by each party, which is $\frac{1}{n}\Theta(nM) = \Theta(M)$, is not affected by how many parties participate in the protocol. This can be phrased as follows: even if more parties join the computation, the amount of data each party has to send remains, in average, constant.

The goal of this section is to present a perfectly secure MPC protocol in the honest majority setting against a passive adversary that achieves linear communication complexity. The protocol is taken from [20], and it follows a similar template to the one described in

[1]This ignores the communication involved in other steps of the protocol such as the input and output phases. This is, however, reasonable, as in typical applications the "inner" complexity of the function (measured by the amount of multiplications in our case) is much bigger than the amount of inputs and outputs.

Section 4.1: each party first secret-shares its input, and then the parties proceed in a gate-by-gate fashion, obtaining shares of the output of each gate, until they reach the final output of the computation, whose shares are exchanged so that the parties can reconstruct the result in the clear. The main difference lies in the way multiplication gates are handled, which is the main source of inefficiency in the previous protocol. The multiplication protocol from the previous section can be seen as having the following structure: The parties locally multiply the shares of the inputs, obtaining degree-$2t$ shares of the product of the underlying secrets, and then they perform an action that converts these shares from degree-$2t$ to degree-t. In the previous protocol this conversion was achieved via *resharing*: each party secret-shares its own degree-$2t$ share using degree-t sharings, and since reconstruction is linear, these shares can be combined in an appropriate manner to obtain degree-t sharings of the original secret. Instead of using resharing, the protocol we will discuss next uses the so-called *double-sharings* to reduce the $2t \to t$ conversion to the task of simply reconstructing certain shared value, which is much more efficient to achieve as we will see.

4.2.1 Using Double-Sharings for Secure Multiplication

Our protocol relies heavily on the concept of double-sharings, which constitute a special type of preprocessing material that enables the parties to handle multiplications securely. A double-sharing is a pair of the form $(\llbracket r \rrbracket_t, \llbracket r \rrbracket_{2t})$, where $r \in \mathbb{F}$ is uniformly random and unknown to the adversary.[2] These double-sharings constitute *preprocessing material*, since they do not depend in any way on the inputs to the multiplication protocol, or in general, the inputs to the function under consideration.

Secure multiplication protocol via double-sharings

Preprocessing: A double-sharing $(\llbracket r \rrbracket_t, \llbracket r \rrbracket_{2t})$.
Input: $\llbracket x \rrbracket_t$ and $\llbracket y \rrbracket_t$ two secret-shared values.
Output: $\llbracket z \rrbracket_t$, where $z = x \cdot y$
Protocol: The parties execute the following

1. The parties compute locally $\llbracket x \cdot y \rrbracket_{2t} \leftarrow \llbracket x \rrbracket_t \cdot \llbracket y \rrbracket_t$ and $\llbracket a \rrbracket_{2t} \leftarrow \llbracket x \cdot y \rrbracket_{2t} - \llbracket r \rrbracket_{2t}$
2. The parties $P_2, \dots, P_{2t+1}$ send their shares of $\llbracket a \rrbracket_{2t}$ to P_1 who, together with his own share, reconstructs a.
3. P_1 sends a to all the other parties, so this value becomes publicly known.
4. The parties compute locally and output $\llbracket z \rrbracket_t \leftarrow \llbracket r \rrbracket_t + a$.

First, notice that the protocol achieves linear communication complexity: in step 2, $2t + 1$ parties send a single field element to only one party, P_1, who sends in step 3 one field element to all other parties. This yields a total communication complexity of $\Theta(n)$. Naturally, this only holds assuming that the parties can get the double sharing with linear communication complexity too, which is discussed in Section 4.2.2 below.

[2]This is formalized as a *functionality* that samples r internally and acts as the dealer in Shamir secret-sharing, distributing the appropriate shares to the parties. However, we stress that this chapter is not concerned with the formalisms of the protocols, so we omit this.

4.2.1.1 A small optimization.

Instead of sending a to all parties, P_1 can secret-share this value, so that the parties get $[\![a]\!]_t$. The rest of the protocol remains the same, changing $[\![z]\!]_t \leftarrow [\![r]\!]_t + a$ with $[\![z]\!]_t \leftarrow [\![r]\!]_t + [\![a]\!]_t$. The advantage of doing this is that, since a does not need to be kept private, t of the shares can be fixed to be 0, and the remaining shares can be computed from these together with the "secret" a. This means that P_1 only needs to communicate the shares to $n - t$ parties, since t of the parties know already that their share of a will be 0. This optimization was introduced in [28].

4.2.2 Producing Double-Sharings Efficiently

The task for this section is to describe a protocol in which the parties can compute double-sharings. This protocol could be used in a preprocessing stage, before the inputs of the parties are known.

To get started, let us consider the following simple protocol:

1. Each P_i for $i \in [t+1]$ samples $r_i \in_R \mathbb{F}$ and secret-shares this value towards the parties twice: using degree-t and degree-$2t$ polynomials. The parties obtain $[\![r_i]\!]_t$ and $[\![r_i]\!]_{2t}$.
2. The parties produce the double-sharing $([\![r]\!]_t, [\![r]\!]_{2t})$, where $[\![r]\!]_t = \sum_{i=1}^{t+1} [\![r_i]\!]_t$ and $[\![r]\!]_{2t} = \sum_{i=1}^{t+1} [\![r_i]\!]_{2t}$.

Since the adversary corrupts t parties, there is at least one honest party among $P_1, \ldots, P_{t+1}$, which implies that the value of r looks uniformly random to the adversary who knows all but one of the random summands. Unfortunately, this approach, although simple, does not suffice for our purposes since it has quadratic communication complexity (each party P_i for $i \in [t+1]$ needs to send shares to all other parties).

The following approach, proposed in [20], enables the parties to produce, using quadratic communication, a total of $\Theta(n)$ double-sharings. As a result, the amortized communication cost *per* double-sharing is linear. The protocol works as follows. Let $\boldsymbol{M} = \mathsf{Van}^{n\times(n-t)}(\beta_1, \ldots, \beta_n)$, where $\beta_1, \ldots, \beta_n$ are mutually-different elements of $\mathbb{F}$.

Preprocessing double-sharings

Output: A set of double sharings $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=1}^{n-t}$
Protocol: The parties proceed as follows

1. Each party P_i samples $s_i \in_R \mathbb{F}$ and secret-shares it using degree-t and degree-$2t$ polynomials. The parties obtain $[\![s_i]\!]_t$ and $[\![s_i]\!]_{2t}$.

2. The parties compute locally the following shares:

$$\begin{pmatrix} [\![r_1]\!]_t \\ [\![r_2]\!]_t \\ \vdots \\ [\![r_{n-t}]\!]_t \end{pmatrix} = \boldsymbol{M}^\intercal \cdot \begin{pmatrix} [\![s_1]\!]_t \\ [\![s_2]\!]_t \\ \vdots \\ [\![s_{n-1}]\!]_t \\ [\![s_n]\!]_t \end{pmatrix}, \qquad \begin{pmatrix} [\![r_1]\!]_{2t} \\ [\![r_2]\!]_{2t} \\ \vdots \\ [\![r_{n-t}]\!]_{2t} \end{pmatrix} = \boldsymbol{M}^\intercal \cdot \begin{pmatrix} [\![s_1]\!]_{2t} \\ [\![s_2]\!]_{2t} \\ \vdots \\ [\![s_{n-1}]\!]_{2t} \\ [\![s_n]\!]_{2t} \end{pmatrix}.$$

3. The parties output the double sharings $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=1}^{n-t}$.

Let us analyze that the protocol produces correct double-sharings, for which it suffices to show that the values $r_1, \ldots, r_{n-t}$ produced by the protocol look uniformly random to the adversary. We claim that these values are in a 1-1 correspondence with the values s_i sampled by the $n-t$ honest parties, which is enough to reach the desired conclusion as these are uniformly random and unknown to the adversary. To see that the claim holds, assume for simplicity that the first $n-t$ parties, $P_1, \ldots, P_{n-t}$, are honest. Let $\boldsymbol{M}' = \mathsf{Van}^{(n-t)\times(n-t)}(\beta_1, \ldots, \beta_{n-t})$ and $\boldsymbol{M}'' = \mathsf{Van}^{(n-t)\times t}(\beta_{n-t+1}, \ldots, \beta_n)$, then $\boldsymbol{M}^\intercal$ can be written in block form as $\boldsymbol{M}^\intercal = [\boldsymbol{M}'^\intercal | \boldsymbol{M}''^\intercal]$, so

$$\begin{pmatrix} r_1 \\ \vdots \\ r_{n-t} \end{pmatrix} = \boldsymbol{M}'^\intercal \cdot \begin{pmatrix} s_1 \\ \vdots \\ s_{n-t} \end{pmatrix} + \boldsymbol{M}''^\intercal \cdot \begin{pmatrix} s_{n-t+1} \\ \vdots \\ s_n \end{pmatrix}.$$

Since $\boldsymbol{M}'$ is invertible, we obtain the desired result. The general case in which the honest parties may not be $P_1, \ldots, P_{n-t}$ is handled in a similar way by taking the appropriate $(n-t)\times(n-t)$ submatrix of $\boldsymbol{M}$.

Regarding communication complexity, observe that in the first step, which is the only step involving interaction, each party sends a share to each other party, which leads to a communication complexity of $\Theta(n^2)$. Since $n-t$ double-sharings are produced, we conclude that the amortized communication complexity of generating each double-share is $\Theta(n^2/(n-t))$, which is linear in n since $n-t > n/2$.

Chapter 5

Active and Perfect Security for Two-Thirds Honest Majority

In the previous section we studied a perfectly secure protocol that is secure against a passive adversary corrupting t parties where $t < n/2$. The goal now is to extend this to active security. As mentioned before, this requires us to either lower the threshold from $t < n/2$ to $t < n/3$, or consider statistical instead of perfect security. In this section we take the first route, that is, we consider two-thirds honest majority and maintain the requirement on perfect security. The second approach, active and statistical security in the honest majority setting, is discussed in Chapter 6.

Before we get into the description of our protocol, recall that, as shown in Section 3.3, in the setting under consideration, $t < n/3$, the parties can reconstruct sharings $[\![s]\!]_d$ with error-correction (i.e. the parties are guaranteed to learn the correct secret) if $d = t$, and with error-detection (i.e. either the parties reconstruct the right secret, or the presence of errors is detected and the parties abort) if $d = 2t$. Furthermore, it is described in Section 3.4 how to do this efficiently via the protocol $\Pi_{\mathsf{PublicRec}}$.

5.1 Actively Secure Multiplication for $t < n/3$

The tools developed in the previous sections are essential for the construction of an actively secure version of the multiplication protocol described in Section 4.2.1. The main issue with that protocol when ported to the actively secure scenario lies in opening, or reconstructing, the secret-shared value $[\![a]\!]_{2t}$. To this end we can use the reconstruction techniques from Section 3.4. In this case, $d = 2t$, and since we assume that $t < n/3$, we can take $\ell = n$ and ensure that the error detection bound $\ell > d + t$ holds (if $d = t$, then the error correction bound can be achieved, a fact that will be useful later on). The resulting protocol is described below. It assumes several simultaneous multiplications are to be processed, given that the reconstruction protocol from the previous section requires $t+1$ values to be opened to operate efficiently. Furthermore, as the preprocessing protocol from Section 4.2.2, the actively secure method to compute double-sharings we will discuss in Section 5.2 also operates in batches.

Actively secure multiplication protocol via double-sharings

Preprocessing: A double-sharing $(\llbracket r \rrbracket_t, \llbracket r \rrbracket_{2t})$.
Input: Secret-shared values $\llbracket x \rrbracket_t$ and $\llbracket y \rrbracket_t$.
Output: $\llbracket z = x \cdot y \rrbracket_t$.
Protocol: The parties execute the following

1. The parties compute locally $\llbracket x \cdot y \rrbracket_{2t} \leftarrow \llbracket x \rrbracket_t \cdot \llbracket y \rrbracket_t$ and $\llbracket a \rrbracket_{2t} \leftarrow \llbracket x \cdot y \rrbracket_{2t} - \llbracket r \rrbracket_{2t}$
2. The parties call the protocol $\Pi_{\mathsf{PublicRec}}$ from Section 3.4 to learn a.[a]
3. The parties compute locally and output $\llbracket z \rrbracket_t \leftarrow \llbracket r \rrbracket_t + a$.

[a]Recall that this protocol operates in batches of secret-shared data so this would be called once for many simultaneous secure multiplications.

5.2 Instantiating the Offline Phase

The protocol from above required as preprocessing material double-sharings $(\llbracket r \rrbracket_t, \llbracket r \rrbracket_{2t})$. Unfortunately, the protocol from Section 4.2.2 to achieve such task cannot be used directly in the actively secure setting, with the main reason being the fact that, when a corrupt party P_i is asked with distributing shares $\llbracket s_i \rrbracket_t$ and $\llbracket s_i \rrbracket_{2t}$, it may not do this *consistently*. More precisely, the underlying secrets in the degree-t and degree-$2t$ must be equal according to the protocol specification, but P_i may choose them to be different. Furthermore, what is worse is that P_i can send shares that are not $t/2t$-consistent, that is, they may not be the result of evaluating a polynomial of the appropriate degree on the points $\alpha_1, \dots, \alpha_n$. This is very sensitive, since the theory of error detection and correction that we developed in Section 3.2 relies heavily on the fact that the shares that the parties had were consistent.

The protocol we will consider to deal with this situation is taken from [4], and it makes use of the so-called hyper-invertible matrices in order to guarantee that the sharings distributed by each party satisfy the necessary consistency requirements. These are defined below, and used to generate double-sharings in Section 5.2.2.

5.2.1 Hyper-Invertible Matrices

A matrix $\boldsymbol{M} \in \mathbb{F}^{k \times \ell}$ is said to be *hyper-invertible* if every square sub-matrix obtained by taking subsets of the rows and columns of $\boldsymbol{M}$ is invertible.

An example of a hyper-invertible matrix is the following. Let $\alpha_1, \dots, \alpha_k, \beta_1, \dots, \beta_\ell \in \mathbb{F}$ be all different field elements, and let $M_{uv} = \prod_{i \in [\ell] \setminus \{v\}} \frac{\beta_u - \alpha_i}{\alpha_v - \alpha_i}$ for $u \in [k], v \in [\ell]$. As shown in [4], the matrix $\boldsymbol{M} \in \mathbb{F}^{k \times \ell}$ whose (u, v) entry is given by M_{uv} is hyper-invertible.

5.2.2 Generating Double-Sharings

The protocol to generate the necessary double-sharings using hyper-invertible matrices is presented below. We let $\boldsymbol{M} \in \mathbb{F}^{n\times n}$ be a hyper-invertible matrix.

Preprocessing double-sharings with active security

Output: A set of double sharings $\{(\llbracket r_i \rrbracket_t, \llbracket r_i \rrbracket_{2t})\}_{i=2t+1}^n$

Protocol: The parties proceed as follows

1. Each party P_i samples $s_i \in_R \mathbb{F}$ and secret-shares it using degree-t and degree-$2t$ polynomials. The parties obtain $\llbracket s_i \rrbracket_t$ and $\llbracket s_i \rrbracket_{2t}$, but observe that corrupt parties may distribute shares *inconsistently*.
2. The parties compute locally the following shares:
$$\begin{pmatrix} \llbracket r_1 \rrbracket_t \\ \llbracket r_2 \rrbracket_t \\ \vdots \\ \llbracket r_n \rrbracket_t \end{pmatrix} = \boldsymbol{M} \cdot \begin{pmatrix} \llbracket s_1 \rrbracket_t \\ \llbracket s_2 \rrbracket_t \\ \vdots \\ \llbracket s_n \rrbracket_t \end{pmatrix}, \qquad \begin{pmatrix} \llbracket r_1 \rrbracket_{2t} \\ \llbracket r_2 \rrbracket_{2t} \\ \vdots \\ \llbracket r_n \rrbracket_{2t} \end{pmatrix} = \boldsymbol{M} \cdot \begin{pmatrix} \llbracket s_1 \rrbracket_{2t} \\ \llbracket s_2 \rrbracket_{2t} \\ \vdots \\ \llbracket s_n \rrbracket_{2t} \end{pmatrix}.$$
3. For each $i \in [2t]$, all the parties send their shares of $\llbracket r_i \rrbracket_t$ and $\llbracket r_i \rrbracket_{2t}$ to P_i.
4. Upon receiving these shares, each P_i for $i \in [2t]$ checks that the received sharings of $\llbracket r_i \rrbracket_t$ and $\llbracket r_i \rrbracket_{2t}$ are t and $2t$-consistent, respectively. If any of the sharings is not consistent, or if both are but the reconstructed value is not equal in both cases, P_i sends abort to all parties and halts.
5. If no party sends an abort message in the previous step, then the parties output the double-sharings $(\llbracket r_i \rrbracket_t, \llbracket r_i \rrbracket_{2t})$ for $i \in \{2t+1, \dots, n\}$.

To analyze the protocol let us assume without loss of generality that the corrupt parties are $P_1, \dots, P_t$. We claim that if there are no abort messages, then the following holds:

1. For each $i \in \{2t+1, \dots, n\}$ the sharings $\llbracket r_i \rrbracket_t$ and $\llbracket r_i \rrbracket_{2t}$ held by the honest parties are t and $2t$-consistent, respectively, and their underlying secrets match.
2. For each $i \in \{2t+1, \dots, n\}$, the secret r_i looks uniformly random and unknown to the adversary.

For the first claim we use the fact that no party among $P_1, \dots, P_{2t}$ sent an abort message in step 4. Since $P_1, \dots, P_t$ are actively corrupt, they may refrain from sending such message when they were actually supposed to. However, $P_{t+1}, \dots, P_{2t}$ are all honest, so if none of these parties sent an abort message it is because the sharings they received, $(\llbracket r_i \rrbracket_t, \llbracket r_i \rrbracket_{2t})$ for $i \in \{t+1, \dots, 2t\}$, pass the check these parties perform. This means these sharings are consistent and their underlying secrets match.

Now, let us partition $\boldsymbol{M}$ in block form as follows:
$$\begin{pmatrix} \boldsymbol{A} & \boldsymbol{B} & \boldsymbol{C} \\ \boldsymbol{D} & \boldsymbol{E} & \boldsymbol{F} \\ \boldsymbol{G} & \boldsymbol{H} & \boldsymbol{I} \end{pmatrix},$$

where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}, \boldsymbol{E} \in \mathbb{F}^{t\times t}$, $\boldsymbol{C}, \boldsymbol{F} \in \mathbb{F}^{t\times(n-2t)}$, $\boldsymbol{G}, \boldsymbol{H} \in \mathbb{F}^{(n-2t)\times t}$ and $\boldsymbol{I} \in \mathbb{F}^{(n-2t)\times(n-2t)}$. Given this partition, we see that, for $d = t, 2t$, it holds that

$$\begin{pmatrix} [\![r_{t+1}]\!]_d \\ [\![r_{t+2}]\!]_d \\ \vdots \\ [\![r_{2t}]\!]_d \end{pmatrix} = \boldsymbol{D} \cdot \begin{pmatrix} [\![s_1]\!]_d \\ [\![s_2]\!]_d \\ \vdots \\ [\![s_t]\!]_d \end{pmatrix} + \boldsymbol{E} \cdot \begin{pmatrix} [\![s_{t+1}]\!]_d \\ [\![s_{t+2}]\!]_d \\ \vdots \\ [\![s_{2t}]\!]_d \end{pmatrix} + \boldsymbol{F} \cdot \begin{pmatrix} [\![s_{2t+1}]\!]_d \\ [\![s_{2t+2}]\!]_d \\ \vdots \\ [\![s_n]\!]_d \end{pmatrix}.$$

Since $\boldsymbol{M}$ is hyper-invertible, the square submatrix $\boldsymbol{D}$ is invertible, which means that we can rewrite the equation above as

$$\begin{pmatrix} [\![s_1]\!]_d \\ [\![s_2]\!]_d \\ \vdots \\ [\![s_t]\!]_d \end{pmatrix} = \boldsymbol{D}^{-1} \cdot \begin{pmatrix} [\![r_{t+1}]\!]_d \\ [\![r_{t+2}]\!]_d \\ \vdots \\ [\![r_{2t}]\!]_d \end{pmatrix} - \boldsymbol{D}^{-1}\boldsymbol{E} \cdot \begin{pmatrix} [\![s_{t+1}]\!]_d \\ [\![s_{t+2}]\!]_d \\ \vdots \\ [\![s_{2t}]\!]_d \end{pmatrix} - \boldsymbol{D}^{-1}\boldsymbol{F} \cdot \begin{pmatrix} [\![s_{2t+1}]\!]_d \\ [\![s_{2t+2}]\!]_d \\ \vdots \\ [\![s_n]\!]_d \end{pmatrix}.$$

Observe that all the sharings that appear in the right-hand side of the equation above are d-consistent for $d = t, 2t$, and their underlying secrets are the same: we already argued this for $([\![r_{t+1}]\!]_d, \dots, [\![r_{2t}]\!]_d)$, and for the remaining shares this holds since these were distributed by honest parties. As a result, since these properties are preserved under linear combinations, we see that the shares on the left-hand side also satisfy said properties. This shows that *all* sharings provided by the parties, $\{([\![s_i]\!]_t, [\![s_i]\!]_{2t})\}_{i=1}^n$, are t and $2t$-degree consistent and the underlying secrets match, which implies that the same holds for the final double-sharings produced by the protocol, $\{([\![r_i]\!]_t, [\![r_i]\!]_{2t})\}_{i=2t+1}^n$, since these are obtained as linear combinations of the ones above. This proves the first claim.

To prove the second claim we observe that we can write

$$\begin{pmatrix} r_{2t+1} \\ r_{2t+2} \\ \vdots \\ r_n \end{pmatrix} = \boldsymbol{G} \cdot \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_t \end{pmatrix} + \boldsymbol{H} \cdot \begin{pmatrix} s_{t+1} \\ s_{t+2} \\ \vdots \\ s_{2t} \end{pmatrix} + \boldsymbol{I} \cdot \begin{pmatrix} s_{2t+1} \\ s_{2t+2} \\ \vdots \\ s_n \end{pmatrix}.$$

Since $\boldsymbol{I}$ is invertible, we see that $(r_{2t+1}, \dots, r_n)$ is in a 1-1 correspondence with the vector $(s_{2t+1}, \dots, s_n)$, but the latter is chosen at random by the honest parties and is unknown to the adversary, so $(r_{2t+1}, \dots, r_n)$ will inherit such properties as well.

Finally, it is easy to see that the communication complexity of the protocol is $\Theta(n^2)$. However, since $n - 2t > n/3$ double-sharings are produced per execution, the amortized complexity per double-sharing is $\Theta(n)$, as required.

5.3 Actively Secure Input Phase

So far we have discussed how to deal with multiplications and output reconstruction when the adversary is behaving actively. The only missing step to completely port the protocol from Section 4.2 to the actively secure setting is the input phase in which each party distributes shares of its own input. The problem here when the adversary is active is the same problem we have already encountered before: corrupt parties may distribute inconsistent shares.

Consistency is enforced by means of the following protocol. It requires as preprocessing material a secret-shared value $[\![r]\!]_t$, where $r \in \mathbb{F}$ is random only known by the party P_i who will provide input. This can be generated by taking a double-sharing $([\![r]\!]_t\, [\![r]\!]_{2t})$, discarding the degree-$2t$ part (or alternatively ignoring it from the start when generating the double-sharing), and letting the parties send their shares of $[\![r]\!]_t$ to P_i, which enables P_i to error correct to learn r. The protocol also requires the broadcast primitive that is part of the communication channel we assume, as discussed in Section 1.2.6.1. However, as discussed in Section 1.3.1, in the case in which $t < n/3$ and the corruption is active, which is the setting in this chapter, a protocol with perfect security instantiating this broadcast primitive exists.

Distributing shares of a given input

Input: Party P_i has an input x.
Preprocessing: A secret-shared value $[\![r]\!]_t$ where $r \in \mathbb{F}$ is random and only known by P_i.
Output: Parties get consistent shares $[\![x]\!]_t$. If P_i is corrupt then the underlying secret may not be equal to x.

Protocol: The parties proceed as follows:

1. *P_i broadcasts $e = x - r$.*
2. The parties locally compute $[\![x]\!]_t = [\![r]\!]_t + e$ as the final shares of the input x.

If the sender is honest then its input is kept private since the only information revealed is $e = x - r$, and since r is uniformly random and unknown to the adversary, this does not leak anything about x. Furthermore, in case P_i is corrupt, the resulting shares are still consistent since the they are obtained by adding a publicly known value e to an already consistently-shared value $[\![r]\!]$. Observe that this assumes that e is known by everyone, which implicitly means that all parties know the *same* value. This may not be the case if the rogue P_i sends different values for e to different parties. However, this is easy to enforce by means of a *broadcast protocol*, as described below.

Chapter 6

Active and Statistical Security for Honest Majority

In this section we study active security in the honest majority setting, that is, where the number of corrupted parties t is strictly less than $n/2$. As discussed in Section 1.3.2, the best security notion achievable with this threshold is statistical security, which is the type of security we aim at in this section.

The protocol we will consider here follows a similar approach as the protocol from the previous section: in a preprocessing phase the parties generate double sharings which are then used in an online phase to compute multiplications securely. However, the main issue that will appear in the $t < n/2$ case is, as we will see, that the adversary can inject certain errors in the online phase that may cause the computation to be incorrect and, moreover, this may lead to leakage of sensitive information about the honest parties' inputs. This is dealt with by executing a check before the output phase that is intended to verify that no errors were introduced during the computation.

Throughout this section we will assume that $n = 2t + 1$. This is not only for simplicity: our protocol is designed to tolerate *exactly* t corruptions while assuming that there are $t + 1$ honest parties. As mentioned in Remark 1.1 in page 19, contrary to intuition, it is not generally true that a protocol that has been designed to withstand t corruptions is also secure against less than t corruptions, and the one presented in this section is an example of that. We will discuss this issue in detail towards the end of this section.

6.1 Reconstructing Secret-Shared Values

As in the previous protocols, an operation that the parties will need to execute several times lies in the reconstruction of a secret-shared value $[\![s]\!]_d$, where the degree d is either equal to t or $2t$. From Section 3.2, we see that in our current setting a party receiving n shares can error-detect if $d = t$, and moreover, reconstruction of secret-shared values in this case can be done with a communication complexity of $O(n)$ field elements with the help of the protocol $\Pi_{\mathsf{PublicRec}}$ from Section 3.4. However, as discussed in Section 3.3, if $d = 2t$, then the adversary can cause the parties to reconstruct an incorrect secret $s + \delta$ for some (potentially non-zero) chosen δ. Fortunately, sharings of degree $2t$ are only used to compute multiplications securely, and, as we will soon see, cheating in this opening leads to incorrect multiplications which can be verified using different techniques. As a result, in spite of the adversary being able to cheat in the multiplications, leading to incorrect results, the validity of these can be checked by the parties.

We will make use of the protocol $\Pi_{\mathsf{PublicRec}}$ from Section 3.4. However, this protocol does not consider the case $d = 2t$ in which the adversary can cause reconstruction to result in incorrect values. A secret-shared value $[\![s]\!]_{2t}$ can be reconstructed with the same communication complexity of $O(n)$ field elements as follows:

1. The parties send their shares of $[\![s]\!]_{2t}$ to P_1;
2. P_1 uses the first $2t+1$ shares $s_1, \dots, s_{2t+1}$[1] to interpolate the unique polynomial $f(\mathtt{X})$ of degree at most $2t$ such that $f(\alpha_i) = s_i$ for $i \in [2t+1]$, and sets $s = f(\alpha_0)$;
3. P_1 sends s to all the parties.

As expected, after the execution of this protocol the adversary can cause the parties to reconstruct $s + \delta$ for some chosen error δ. This can happen if P_1 is corrupt and adds this error when sending the result to the parties[2], or it can also occur even if P_1 is honest if the actively corrupt parties send wrong shares to P_1.

6.2 Preprocessing Phase

As we have already mentioned, the protocol we will use for the setting $t < n/2$ resembles a lot the protocol from Chapter 5 in which parties produce double-sharings $([\![r]\!]_t, [\![r]\!]_{2t})$ in the preprocessing phase, which are then used to obtain shares of a product $[\![xy]\!]_t$ from two shared values $[\![x]\!]_t$ and $[\![y]\!]_t$. This is done by letting the parties locally obtain $[\![xy]\!]_{2t}$, then open $a \leftarrow [\![xy]\!]_{2t} - [\![r]\!]_{2t}$ and later compute $[\![xy]\!]_t \leftarrow [\![r]\!]_t + a$ non-interactively.

We first discuss how the parties can obtain the necessary preprocessing material $([\![r]\!]_t, [\![r]\!]_{2t})$. In Section 5.2 we presented a protocol based on the so-called hyper-invertible matrices (HIM) to obtain this type of correlation for the case in which $t < n/3$. Unfortunately, this protocol is not suitable for our setting in which $t < n/2$, which can be seen by thoroughly inspecting the construction. However, in Section 4.2.2 we presented a *passively secure* protocol for generating double-sharing in the honest majority setting, and this method will serve as the basis for the actively secure mechanism to produce double-sharings we need in this section.

Let us begin by recalling briefly how the passively secure protocol from Section 4.2.2 works. It begins by asking each party P_i to sample $s_i \in_R \mathbb{F}$, and then secret-share this value twice

[1] Since we assume that $n = 2t+1$, these $2t+1$ shares constitute *all* the shares, but this method also works for $2t+1 < n$.

[2] If P_1 is actively corrupt then he can even perhaps add different errors to the value sent to different parties, which results in the parties learning different values. For simplicity in the presentation we assume this is not the case, that is, the honest parties obtain the *same* value $s+\delta$. This can be achieved by asking P_1 to use a broadcast channel to send this value. However, this is not necessary as the protocol still works even if P_1 distributes different values. However, for the sake of clarity and simplicity we assume this does not happen.

using thresholds t and $2t$ as $(\llbracket s_i \rrbracket_t, \llbracket s_i \rrbracket_{2t})$. Then the parties locally apply to these sharings a matrix that acts as a randomness extractor in order to obtain the final double-shares.

As mentioned in Section 5.2, this protocol is not actively secure, mainly because an actively corrupt party P_i may cheat when asked to secret-share the value s_i. This cheating may take place in different ways:

- P_i does not distribute $\llbracket s_i \rrbracket_t$ consistently, that is, the shares $(s_{i1}, \dots, s_{in})$ of $\llbracket s_i \rrbracket_t$ sent by P_i to the other parties are not t-consistent.
- P_i does not distribute $\llbracket s_i \rrbracket_{2t}$ consistently, that is, the shares $(s'_{i1}, \dots, s'_{in})$ of $\llbracket s_i \rrbracket_{2t}$ sent by P_i to the other parties are not $2t$-consistent.
- The shares $(s_{i1}, \dots, s_{in})$ and $(s'_{i1}, \dots, s'_{in})$ are t and $2t$-consistent, respectively, but the underlying secrets are not the same.

Although these issues, at a high level, seem harmful for the protocol, we can show that the ultimate effect they have on the execution is that a multiplication of two shared values $\llbracket x \rrbracket_t$ and $\llbracket y \rrbracket_t$ may result in $\llbracket x \cdot y + \delta \rrbracket_t$ for some adversarially-chosen error $\delta \in \mathbb{F}$. This is of course a problem for the correctness of the protocol, since an adversary can cause intermediate values to be computed incorrectly. Fortunately, it is possible to check, quite efficiently, that the multiplications have been computed correctly, which is explained in Section 6.4.

From the observation above, the protocol the parties use in order to generate double-sharings efficiently is essentially the same as the protocol from Section 4.2.2. The protocol is described explicitly below for the sake of completeness. Let $\boldsymbol{M} = \mathsf{Van}^{n \times (n-t)}(\beta_1, \dots, \beta_n)$, where $\beta_1, \dots, \beta_n$ are different elements of $\mathbb{F}$.

Preprocessing double-sharings

Output: A set of double sharings $\{(\llbracket r_i \rrbracket_t, \llbracket r_i \rrbracket_{2t})\}_{i=1}^{n-t}$
Protocol: The parties proceed as follows

1. Each party P_i samples $s_i \in_R \mathbb{F}$ and secret-shares it using degree-t and degree-$2t$ polynomials. The parties obtain $\llbracket s_i \rrbracket_t$ and $\llbracket s_i \rrbracket_{2t}$.
2. The parties compute locally the following shares:

$$\begin{pmatrix} \llbracket r_1 \rrbracket_t \\ \llbracket r_2 \rrbracket_t \\ \vdots \\ \llbracket r_{n-t} \rrbracket_t \end{pmatrix} = \boldsymbol{M}^\intercal \cdot \begin{pmatrix} \llbracket s_1 \rrbracket_t \\ \llbracket s_2 \rrbracket_t \\ \vdots \\ \llbracket s_{n-1} \rrbracket_t \\ \llbracket s_n \rrbracket_t \end{pmatrix}, \qquad \begin{pmatrix} \llbracket r_1 \rrbracket_{2t} \\ \llbracket r_2 \rrbracket_{2t} \\ \vdots \\ \llbracket r_{n-t} \rrbracket_{2t} \end{pmatrix} = \boldsymbol{M}^\intercal \cdot \begin{pmatrix} \llbracket s_1 \rrbracket_{2t} \\ \llbracket s_2 \rrbracket_{2t} \\ \vdots \\ \llbracket s_{n-1} \rrbracket_{2t} \\ \llbracket s_n \rrbracket_{2t} \end{pmatrix}.$$

3. The parties output the double sharings $\{(\llbracket r_i \rrbracket_t, \llbracket r_i \rrbracket_{2t})\}_{i=1}^{n-t}$.

As mentioned before, even though this protocol is in principle not actively secure (in the sense that there are seemingly a lot of places where the adversary can cheat to cause a potentially harmful outcome), we will be able to show that, when this protocol is used in conjunction with the protocol for multiplying two shared values from Section 6.3 below, the end result is that the adversary is able to inject additive errors to the result of a multiplication. Fortunately, this type of attack can be prevented as described in Section 6.4.

Instead of analyzing in detail the security guarantees of this protocol on their own, we postpone the analysis to Section 6.3 below where we analyze the properties of the multiplication protocol that aims to produce $[\![xy]\!]_t$ from $[\![x]\!]_t$ and $[\![y]\!]_t$. However, before we move into that, we provide in this section a bit of intuition about why is it the case that none of the attacks proposed above is relevant. In a nutshell, the reason lies in the fact that, the notion of d-consistency from Definition 3.3 in Section 3.1, is only concerned with the consistency of the shares held by honest parties.

Distributing shares of degree t inconsistently. An actively corrupt party can misbehave when acting as a dealer in Shamir secret-sharing, and can choose to send arbitrary values $(s_1, \dots, s_n)$ that are not t-consistent to the parties. This is not a problem nonetheless, or rather, an adversary can cause the exact same effect even if the dealer is honest, as we now show.

For simplicity in the notation, assume that the corrupt parties are $P_1, \dots, P_t$. Since $n = 2t+1$, there are exactly $t+1$ honest parties $P_{t+1}, \dots, P_n$. Let $f(\mathtt{X}) \in \mathbb{F}_{\leq t}[\mathtt{X}]$ be the unique polynomial of degree at most t such that $f(\alpha_i) = s_i$ for $i = t+1, \dots, n$. Since the dealer is actively corrupt, the adversary knows $f(\mathtt{X})$ and therefore it knows $s'_i = f(\alpha_i)$ for $i = 1, \dots, t$. In particular, from Definition 3.3, the parties hold t-consistent shares $[\![s]\!]_t = (s'_1, \dots, s'_t, s_{t+1}, \dots, s_n)$, where $s = f(\alpha_0)$.

In conclusion, even if the dealer is actively corrupt, any set of shares it sends will be by definition t-consistent since there are exactly $t+1$ parties and the shares these parties receive uniquely define a polynomial $f(\mathtt{X}) \in \mathbb{F}_{\leq t}[\mathtt{X}]$.

Distributing shares of degree $2t$ inconsistently. Assume for simplicity in the notation that the corrupt parties are $P_1, \dots, P_t$. As before, an actively corrupt dealer can misbehave and choose to send arbitrary values $(s_{t+1}, \dots, s_n)$ to the honest parties. Since $n = 2t+1$ (this also works for $n \geq 2t+1$), for any secret s there exists a polynomial $f(\mathtt{X}) \in \mathbb{F}_{\leq 2t}[\mathtt{X}]$ such that $f(\alpha_0) = s$ and $f(\alpha_i) = s_i$ for $i = t+1, \dots, n$. Since the adversary knows $f(\mathtt{X})$, it knows in particular $f(\alpha_j)$ for $j = 1, \dots, t$. This means, according to Definition 3.3, that the parties *trivially* hold $2t$-consistent shares of *any* secret.

As in Remark 3.1, this is not a good thing, since it means the adversary can change the corrupt parties' shares in order to obtain sharings of different values. However, as pointed out before, this will be acceptable in our protocol: the concrete effect that this type of attack will have in the overall protocol is that the adversary will be able to add errors to the output of secure multiplications, but the correctness of these will be verified with a simple protocol as described in Section 6.4.

Shares of degree t and $2t$ having different secrets. From the two notes above we see that the adversary cannot, by definition, distribute shares t or $2t$-inconsistently. The last attack it could carry out then in the protocol for preprocessing double-sharings is that the secret s in the shares of degree t is not the same as the secret s' in the shares of degree $2t$. Once again, although the shares of degree t uniquely define a secret s, the shares of degree $2t$ are consistent with any possible secret, so there is not even an s' defined. As mentioned before, this gap will result in a concrete attack in the multiplication protocol from Section 6.3 below, which can be prevented using certain checks after the multiplication has been performed as explained in Section 6.4.

6.3 Online Phase

We now move to the description of the online phase of our protocol. Recall that the function to be evaluated, $F : \mathbb{F}^n \to \mathbb{F}$, is given by an arithmetic circuit over $\mathbb{F}$. Let $x_i \in \mathbb{F}$ be the input of party P_i. As in previous sections, the protocol consists of having the parties obtain shares $[\![x_1]\!]_t, \dots, [\![x_n]\!]_t$ of their inputs, followed by methods to obtain from two given shared values $[\![x]\!]_t$ and $[\![y]\!]_t$, shares of $[\![x+y]\!]_t$ and $[\![x \cdot y]\!]_t$. This allows the parties to obtain shares of all intermediate values of the computation, until shares of the output $[\![z]\!]_t$, with $z = F(x_1, \dots, x_n)$, are produced. At this point the parties can simply reconstruct this result to learn the output of the computation.

The input phase in which the parties obtain shares of their inputs, is handled in exactly the same way as in Section 5.3, that is, for a party P_i to provide input x_i, we assume the parties have a random shared value $[\![r]\!]_t$ where $r \in_R \mathbb{F}$ is only known by P_i. This could be easily adapted from the protocol to obtain double-shares. Then P_i uses the broadcast channel to send $e = x_i - r$ to all the parties, who define $[\![x_i]\!] = [\![r]\!] + e$ as their shares of x_i.

Given $[\![x]\!]_t$ and $[\![y]\!]_t$, it is straightforward for the parties to obtain $[\![x+y]\!]_t$ given the linearity properties of Shamir secret-sharing. On the other hand, to obtain $[\![x \cdot y]\!]$, the parties first execute the following protocol, which is exactly the same as the one presented in Section 4.2.1.

Actively secure multiplication protocol via double-sharings

Preprocessing: Double-sharings $([\![r]\!]_t, [\![r]\!]_{2t})$.
Input: Secret-shared values $[\![x]\!]_t$ and $[\![y]\!]_t$.
Output: $[\![z = x \cdot y]\!]_t$.

Protocol: The parties execute the following

1. The parties compute locally $[\![x \cdot y]\!]_{2t} \leftarrow [\![x]\!]_t \cdot [\![y]\!]_t$ and $[\![a]\!]_{2t} \leftarrow [\![x \cdot y]\!]_{2t} - [\![r]\!]_{2t}$
2. The parties call the protocol $\Pi_{\mathsf{PublicRec}}$ from Section 6.1 to learn a.
3. The parties compute locally and output $[\![z]\!]_t \leftarrow [\![r]\!]_t + a$.

The main difference of this protocol with respect to the one from Section 5.1 is the set of guarantees this one provides. In the protocol from Section 5.1, we could prove that the parties obtain the correct $[\![x \cdot y]\!]_t$ at the end of the protocol execution. In our case here, we will not be able to prove this. This is because, as has been mentioned before, in our case where $t < n/2$, opening degree-$2t$ shares cannot be done while ensuring the integrity of the underlying secret, which is not the case when $t < n/3$. Here, we show the following:

Proposition 6.1. *Let $[\![x]\!]_t$ and $[\![y]\!]_t$ be inputs to the multiplication protocol above. Then, at the end of the protocol execution, the parties get shares $[\![xy+\delta]\!]_t$, where $\delta \in \mathbb{F}$ is a value known to the adversary.*

Proof. The result of the call to protocol $\Pi_{\mathsf{PublicRec}}$ is $a+\delta = (xy-r)+\delta$, so the parties compute $[\![zy+\delta]\!] \leftarrow [\![r]\!] + (xy-r)+\delta$ in the last step of the protocol. □

6.4 Verification Phase

As we showed in Proposition 6.1, an active adversary can inject errors to the result of secure multiplications. This is of course a problem since correctness of the computation is not guaranteed anymore. Furthermore, it can lead to concrete privacy leakage attacks. For example, if $n=3$ and $F(x_1,x_2,x_3) = (x_1 \cdot x_2)\cdot x_3$, a corrupt party could add a non-zero error in the first multiplication so that the output of the computation is $(x_1\cdot x_2+\delta)\cdot x_3 = F(x_1,x_2,x_3)+\delta\cdot x_3$. If the adversary corrupts P_1 and sets $x_1=0$, then the correct output is always 0 regardless of the inputs of the other parties, so the adversary should not be able to learn anything about these according to the security definition of MPC. However, with the attack above, the result becomes $0+\delta\cdot x_3$, which in particular means that the adversary can learn x_3 by multiplying the result with δ^{-1}.

Given this, it is imperative that, before the parties reconstruct the final result, they check that no errors have been introduced in any of the multiplications involved in the computation. The more concrete setting is the following. The parties have (t-consistent) shares $([\![x]\!]_t, [\![y]\!]_t, [\![z]\!]_t)$, where z is supposed to be equal to $x\cdot y$. However, due to adversarial behavior, it is actually the case that $z = x\cdot y+\delta$ for some adversarially chosen value $\delta\in\mathbb{F}$, and the parties want to check that $\delta=0$. At a high level, the method we will present to address this issue consists of the following. First, the parties generate a triple such as the one above $([\![a]\!]_t, [\![b]\!]_t, [\![c]\!]_t)$, where $c = a\cdot b+\epsilon$ for some adversarially chosen value ϵ, and in addition, $a,b\in\mathbb{F}$ are uniformly random and unknown to the adversary. Then, the parties will make use of this triple of shared values to check the correctness of z.

Generating the tuple $([\![a]\!]_t, [\![b]\!]_t, [\![c]\!]_t)$ is straightforward given the tools we have presented thus far: getting $[\![a]\!]_t$ and $[\![b]\!]_t$ can be done by a simple modification of the protocol from Section 6.2 to obtain double-sharings, without considering the degree-$2t$ part, and $[\![c]\!]_t$ can be obtained from $[\![a]\!]_t$ and $[\![b]\!]_t$ by applying the multiplication protocol from Section 6.3. Now, using such tuple to check the correctness of z is done with the following protocol. Below, we let $\mathcal{F}_{\mathsf{Coin}}$ denote a functionality that returns public random values to all the parties.

Verifying secure multiplications

Preprocessing: A tuple $([\![a]\!]_t, [\![b]\!]_t, [\![c]\!]_t)$, where $c = a\cdot b+\epsilon$ for some value $\epsilon\in\mathbb{F}$ known by the adversary, and $a,b\in\mathbb{F}$ are uniformly random and unknown to the adversary.
Input: Secret-shared values $([\![x]\!]_t, [\![y]\!]_t, [\![z]\!]_t)$, where $z = x\cdot y+\delta$ for some value $\delta\in\mathbb{F}$ known by the adversary.
Output: A signal `pass`/`fail`.

Protocol: The parties execute the following

1. The parties call $s \leftarrow \mathcal{F}_{\mathsf{Coin}}$;
2. The parties compute locally $[\![d]\!]_t \leftarrow [\![x]\!]_t - s \cdot [\![a]\!]_t$ and $[\![e]\!]_t \leftarrow [\![y]\!]_t - [\![b]\!]_t$
3. The parties call the protocol $\Pi_{\mathsf{PublicRec}}$ from Section 6.1 to reconstruct d and e.
4. The parties compute locally $[\![w]\!] \leftarrow s \cdot e \cdot [\![a]\!] + d \cdot [\![b]\!] + s \cdot [\![c]\!] + d \cdot e - [\![z]\!]$.
5. The parties call the protocol $\Pi_{\mathsf{PublicRec}}$ to reconstruct w, and check that $w = 0$. If this is the case, output `pass`. Else, output `fail`.

Proposition 6.2. *Let* $([\![x]\!]_t, [\![y]\!]_t, [\![z]\!]_t)$ *with* $z = xy + \delta$ *be an input to the protocol above. Then, if* $\delta \neq 0$*, the probability that the protocol results in the parties outputting* `pass` *is at most* $1/|\mathbb{F}|$*. Furthermore, nothing about* x *or* y *is learned by the adversary after the execution of the protocol.*

Proof. Since $d = x - s \cdot a$, $e = y - b$, $z = x \cdot y + \delta$ and $c = a \cdot b + \epsilon$, we have that

$$\begin{aligned} w =& s \cdot e \cdot a + d \cdot b + s \cdot c + d \cdot e - z \\ =& s \cdot (y - b) \cdot a + (x - s \cdot a) \cdot b + s \cdot (a \cdot b + \epsilon) + (x - s \cdot a)(y - b) - (x \cdot y + \delta) \\ =& sya - sba + xb - sab + sab + s\epsilon + xy - xb - say + sab - xy - \delta \\ =& s \cdot \epsilon - \delta. \end{aligned}$$

From this, we see that $w = 0$ if and only if $s \cdot \epsilon - \delta = 0$, or $s \cdot \epsilon = \delta$. Assume that $\delta \neq 0$ and nevertheless $w = 0$. Then $\epsilon \neq 0$ since otherwise $\delta = s \cdot 0 = 0$, but this implies that $s = \delta/\epsilon$, which happens with probability $1/|\mathbb{F}|$ since s is uniformly random and sampled independently of δ and ϵ. □

Remark 6.1. *The verification step above can be improved so that, when many checks are performed simultaneously (as expected in an actual secure computation scenario), the overhead in communication by performing this check is very small. More concretely, this overhead can be made* sub-linear *in the number of multiplications being checked thanks to the novel techniques presented in [28].*

Part III

Dishonest Majority

Chapter 7

Passive Security for Dishonest Majority

All the protocols we have seen so far assume that the adversary corrupts strictly less than $n/2$ or $n/3$ parties. However, if such assumption is violated, privacy would break, which can be seen from the fact that if the adversary corrupts more parties than the threshold used for Shamir secret sharing then the underlying secret is revealed.

It would be ideal if we could design protocols where, from the view of *each single party*, their input is kept private even if *all* of the other parties collude against the single party. In other words, we would like to guarantee security under an adversary corrupting t parties, even if t grows as large as $n-1$, leaving only one honest party. This setting, where the only bound on t is $t < n$, is called *dishonest majority* since in principle a majority of the parties could be corrupt; this is in contrast to the case in which $t < n/2$ where the majority of parties are guaranteed to be honest.

In this chapter we explore an MPC protocol in the dishonest majority setting with passive security, which means that each party's input is secure even if all the other parties collude, as long as these parties follow the protocol specification. In Chapter 8 we explore the case of active security, which ensures privacy even if the other parties misbehave. This is the strongest possible setting, but it is also, naturally, the most expensive.

Another important aspect of the dishonest majority setting is that it includes the relevant case in which $n = 2$ and $t = 1$, since in this case none of the bounds $t < n/2$ nor $t < n/3$ hold. This particular scenario appears in many different applications, so it must be considered as well.

7.1 Additive Secret-Sharing

We assume that $t = n - 1$. Unlike the results from Chapter 6, assuming t reaches the maximum possible bound is only done for the sake of clarity in the notation, instead of being a security feature. All of our results carry over if $t < n - 1$.

Additive secret-sharing

The dealer secret-shares a value $s \in \mathbb{F}$ among n parties $P_1, \dots, P_n$ as follows.

1. Sample $s_1, \dots, s_{n-1} \in_R \mathbb{F}$ and define $s_n = s - (s_1 + \cdots + s_{n-1})$.

2. Distribute the value s_i to party P_i, for $i \in [n]$.

In other words, the tuple $(s_1, \dots, s_n)$ is uniformly random in $(\mathbb{Z}/2^k\mathbb{Z})^n$, constrained to $s = s_1 + \cdots + s_n$. As a result, for every set $A \subseteq [n]$ with $|A| \leq n-1$, the distribution of the shares $\{s_i\}_{i \in A}$ is uniformly random, and in particular, it is independent of the secret s.

When the parties have shares as above, we denote $[\![s]\!] = (s_1, \dots, s_n)$. Notice that, given two shared values $[\![x]\!]$ and $[\![y]\!]$, the parties can locally add/subtract their shares of these values to obtain $[\![x \pm y]\!]$. Furthermore, given a value $c \in \mathbb{F}$ known to all the parties, the parties can locally obtain $c\,[\![x]\!]$ by multiplying c to every share, and they can obtain $[\![x \pm c]\!]$ by asking only one party, say P_1, to add/subtract the value of c to its share.

7.2 Protocols for Secure Multiplication

As in previous sections, we obtain a secure computation protocol by asking the parties to distribute shares of their inputs (which is trivial since, if x is known to party P_i, the parties can non-interactively obtain $[\![x]\!]$ by writing $x = x_1 + \cdots + x_n$ where $x_j = 0$ if $j \neq i$, and $x_i = x$), followed by the parties executing different subprotocols to obtain $[\![x+y]\!]$ and $[\![xy]\!]$ from shared values $[\![x]\!]$ and $[\![y]\!]$. As we mentioned before, the case of addition can be easily handled by the parties adding their shares locally. However, obtaining $[\![xy]\!]$ is, as usual, a much harder task. Furthermore, what complicates matters in the dishonest majority scenario is that, as we have mentioned already in Section 1.3.3, this setting requires the use of tools from the public-key cryptography domain, which are, in their nature, much more expensive than the information-theoretic techniques we have been making use of so far.

7.2.1 Product-to-Sum Conversion

We begin by reducing the problem of obtaining $[\![xy]\!]$ from $[\![x]\!]$ and $[\![y]\!]$ to a simpler problem. Write $[\![x]\!] = (x_1, \dots, x_n)$ and $[\![y]\!] = (y_1, \dots, y_n)$, so

$$xy = (x_1 + \cdots + x_n)(y_1 + \cdots + y_n) = \sum_{i=1}^{n} x_i y_i + \sum_{i,j \in [n], i \neq j} x_i y_j.$$

The goal is to obtain shares of each of these summands, which can in turn be added locally to obtain shares of $x \cdot y$. Since each term of the form $x_i y_i$ is known by party P_i, the parties can obtain $[\![x_i y_i]\!]$ trivially by setting P_i's share to be $x_i y_i$, and the setting the other shares to be 0. The main challenge lies on the terms of the form $x_i y_j$ for $i \neq j$, since one factor is known by one party P_i, and the other factor is known by a different party P_j.

Assume the existence of a protocol for *product-to-sum conversion*, in which P_i inputs x_i, P_j inputs y_j, and P_i and P_j receive z_i and z_j respectively, with these values being uniformly random constrained to $x_i y_j = z_i + z_j$. With such a tool, the parties can obtain $[\![x_i y_j]\!]$ by letting P_i and P_j execute the product-to-sum protocol, obtaining z_i and z_j, and defining the other parties' shares to be zero.

From our observations above, the parties can locally compute $[\![x_i y_i]\!]$ for $i \in [n]$, and, with the help of a product-to-sum conversion protocol, they can also compute $[\![x_i y_j]\!]$ for $i, j \in [n]$ and $i \neq j$. As a result, they can compute shares of x and y as follows.

$$[\![xy]\!] = \sum_{i=1}^{n} [\![x_i y_i]\!] + \sum_{i,j \in [n], i \neq j} [\![x_i y_j]\!] \, .$$

7.2.2 Product-to-Sum Conversion Based on Homomorphic Encryption

From the previous section, we see that, in order for the parties to compute $[\![xy]\!]$ from $[\![x]\!]$ and $[\![y]\!]$, it suffices to design a two-party protocol for product-to-sum conversion. Recall that, in such protocol two parties, which we denote by P_1 and P_2, each input a value x_1 and x_2, and they receive z_1 and z_2, which are uniformly random values constrained to $x_1 x_2 = z_1 + z_2$. In this section we show how such primitive can be instantiated making use of Additively Homomorphic Encryption, or AHE for short. We remark that our aim is simply to provide intuition on how this can be done, so we do not provide a lot of details nor present a lot of formalism.

An encryption scheme consists of an encryption and decryption algorithms $\mathsf{Enc}_{\mathsf{pk}}(\cdot)$ and $\mathsf{Dec}_{\mathsf{sk}}(\cdot)$ such that, intuitively:

1. $\mathsf{Enc}_{\mathsf{pk}}(m)$ does not leak anything about the message m if the key pair $(\mathsf{sk}, \mathsf{pk})$ is sampled by a sampling algorithm.

2. If $c = \mathsf{Enc}_{\mathsf{pk}}(m)$, then $m = \mathsf{Dec}_{\mathsf{sk}}(c)$.

In an *additively homomorphic encryption scheme* (AHE), in addition, there is a way to "add/subtract" the ciphertexts to obtain encryptions of the respective operations on the plaintexts, that is, given $c = \mathsf{Enc}_k(x)$ and $d = \mathsf{Enc}_k(y)$, it is possible to compute $c \pm d = \mathsf{Enc}_k(x \pm y)$. An example of an additively homomorphic encryption scheme is Paillier's [36]. Such an encryption scheme can be used to instantiate the product-to-sum conversion primitive as follows.

Secure multiplication

Input: Party P_i has input x_i, for $i \in \{1, 2\}$
Setup: Key pair $(\mathsf{sk}, \mathsf{pk})$ with pk known by P_1 and P_2, and sk known by P_1.
Output: P_i gets z_i for $i \in \{1, 2\}$, where (z_1, z_2) is uniformly random constrained to $x_1 x_2 = z_1 + z_2$.
Protocol:

1. P_1 sends $c = \mathsf{Enc}_{\mathsf{pk}}(x_1)$ to P_2.

2. P_2 samples z_2 and sends $d = x_2 \cdot c - \mathsf{Enc}_{\mathsf{pk}}(z_2)$ to P_1

3. P_1 computes $z_1 = \mathsf{Dec}_{\mathsf{sk}}(d)$

We see that P_2 does not learn anything about x_1 since it only receives $c = \mathsf{Enc}_{\mathsf{pk}}(x_1)$, which by the properties of the encryption scheme, completely hides x_1. On the other hand, P_1 decrypts $z_1 = x_1 \cdot x_2 - z_2$, as desired.

7.3 Preprocessing Model

The tools we have described so far enable the parties to securely compute any arithmetic circuit comprised of additions and multiplications over the finite field $\mathbb{F}$: the parties hold additive shares of the inputs to the computation, addition gates can be processed non-interactively, and multiplication gates make use of the method from Section 7.2, which relies on an AHE scheme. Unfortunately, this approach would provide poor efficiency when compared to the other MPC protocols we have explored in precious sections. These protocols, to process multiplication gates, only required simple arithmetic over $\mathbb{F}$, while the use of AHE techniques, and in general, the different tools used in practice to perform secure multiplication in the dishonest majority settings, is considerably more expensive.

Unfortunately, the use of these techniques is unavoidable when the adversary corrupts more than a majority of the parties, even if the corruption is passive. Given this limitation, an standard approach to limit its effect in practice is to split the computation in two phases: an offline phase, also called preprocessing phase, which is independent of the inputs of the computation, and an online phase, which now makes use of the inputs. The online phase is designed to be much more efficient than the protocol we have sketched so far. In fact, this phase typically achieves information-theoretic security and only makes use of simple arithmetic operations, so it achieves high efficiency. This way, by pushing the offline phase to a much earlier time before the parties set their inputs, say, when the parties are idle, the execution of the MPC protocol is much more efficient from a practical perspective, counting the latency from the time the parties provide input to the time they produce the output.

7.4 Offline Phase

To accelerate the computation of secure multiplications in the online phase, the parties will need to produce a set of multiplication triples. A multiplication triple, also called Beaver triple, is a tuple of the form $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$, where a, b are uniformly random in $\mathbb{F}$ and unknown to any party, and $c = a \cdot b$. In the offline phase, the parties generate one such tuple for every multiplication gate expected in the arithmetic circuit under consideration. Notice that these tuples only contain random data and do not make use of the inputs of the computation, which are used later in the online phase. As we have mentioned, this is crucial for the preprocessing paradigm to provide any benefit since, as we will see, it is in the production of these tuples where the parties spent most of their computational resources, in order to be able to compute the online phase in a much more efficient way.

To generate a multiplication triple, the parties can proceed as follows.

Generating multiplication triples with passive security

Output: A multiplication triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$, where a, b are uniformly random in $\mathbb{F}$ and unknown to the adversary, and $c = a \cdot b$.
Protocol:

1. Each party P_i samples $a_i, b_i \in_R \mathbb{F}$. This leads to sharings $\llbracket a \rrbracket = (a_1, \dots, a_n)$ and $\llbracket b \rrbracket = (b_1, \dots, b_n)$.
2. The parties execute a multiplication protocol to obtain $\llbracket c \rrbracket$, where $c = ab$.
3. The parties output the shares $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$.

Notice that $a = a_1 + \cdots + a_n$ and $b = b_1 + \cdots + b_n$ look uniformly random and unknown to the adversary since there is at least one honest party contributing with a uniformly random summand in each of these expressions.

7.5 Online Phase

Now we show how the parties can make use of a multiplication triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ to securely obtain in the online phase $\llbracket xy \rrbracket$ from $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$, in a much more efficiently way than simply using a multiplication protocol like the one from Section 7.2.

Multiplication based on multiplication triples

Input: Shared values $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$.
Output: Shared value $\llbracket z \rrbracket$, where $z = xy$.
Preprocessing: Multiplication triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$
Protocol:

1. The parties compute locally $\llbracket d \rrbracket \leftarrow \llbracket x \rrbracket - \llbracket a \rrbracket$ and $\llbracket e \rrbracket \leftarrow \llbracket y \rrbracket - \llbracket b \rrbracket$
2. The parties send their shares of $\llbracket d \rrbracket$ and $\llbracket e \rrbracket$ to each other to learn d and e.[a]
3. The parties compute locally $\llbracket z \rrbracket \leftarrow d \llbracket b \rrbracket + e \llbracket a \rrbracket + \llbracket c \rrbracket + de$.

[a]This can be optimized by asking parties to send their shares to one single party, say P_1, who reconstructs d and e and announces these values to the parties.

To see that the protocol works as intended, observe first that, given that $d = x - a$, $e = y - b$, and $c = ab$, it holds that $db + ea + c + de = xy$, so the protocol indeed produces shares of $x \cdot y$. On the other hand, nothing is leaked about x or y since the only values that are opened during the protocol execution are d and e, which look uniformly random to the adversary given that a and b are random and unknown to the adversary. In particular, notice that the protocol is perfectly secure. Furthermore, the protocol shines from its simplicity, involving only the reconstruction of two secret-shared values and simple local arithmetic.

Chapter 8

Active Security for Dishonest Majority

In Chapter 7 we described a protocol for secure computation in the dishonest majority setting, assuming the corruption is passive. However, that protocol is not secure if the corrupted parties behave maliciously. As an example of what goes wrong when the corruption is active, consider a secret-shared value $[\![x]\!] = (x_1, \ldots, x_n)$. Suppose that, at reconstruction time, the corrupt parties P_i for $i \in \mathcal{C}$ change their share from x_i to $x_i' = x_i + \delta_i$ for some $\delta_i \in \mathbb{F}$. Since there is no method for the honest parties to detect this change, the reconstructed value would be $\sum_{i\in\mathcal{C}} x_i' + \sum_{i\in\mathcal{H}} x_i = \sum_{i=1}^{n} x_i + \sum_{i\in\mathcal{C}} \delta_i = x + \delta$, where $\delta = \sum_{i\in\mathcal{C}} \delta_i$. In particular, the adversary can cause the reconstruction to lead to an incorrect value, similar to what happened in Section 6.1 when shares of degree $2t$ had to be reconstructed. This was fixed in the protocol for $t < n/2$ by noticing that the ultimate effect that this attack has is that the adversary can affect the result of secure multiplications, which can be checked with a verification protocol. In our current dishonest majority setting this "share-modification" attack is much more devastating, since it does not only affect multiplications but every single step that requires an opening.

To fix the issue of the adversary modifying the corrupt parties' shares, we need to add certain "redundancy" to the sharings, comparable to the redundancy present in Shamir shares for $t < n/2$. This comes in the form of a tool called *Message Authentication Codes*, or MACs for short. This term is taken from the symmetric key cryptography literature, and in general, we use it to represent a primitive that guarantees *data integrity*, that is, that an adversary cannot modify certain piece of data without being detected. This is precisely the type of tool we need in our current context to disallow the adversary from modifying the shares of the corrupt parties.

MACs are used in order to authenticate the parties' shares so that they cannot change them at a later point, and the way these are used is divided into two. In the first approach, described in Section 8.1, each single party has an extra piece of information to check the integrity of every other party's share at reconstruction time. The second approach, described in Section 8.2, consists of all the parties jointly having a way of checking not the integrity of each individual share, but rather the integrity of the reconstructed *secret*. This works better for a large number of parties since it is not necessary for every party to hold authentication information of the share held by each other party.

Both of these methods are based on the following basic idea for ensuring integrity. Given a piece of data $m \in \mathbb{F}$, compute a random value $\alpha \in_R \mathbb{F}$, and let $\tau = \alpha \cdot m$. Integrity is checked by verifying that m, multiplied by the value α, results in τ. If m is modified as $m' = m + \delta$ and τ is modified as $\tau' = \tau + \epsilon$, then the only way in which $\alpha m'$ can equal τ' is if $\alpha\delta = \epsilon$. If $\delta \neq 0$, that is, if the data m was indeed modified to a different m', then this equation translates to $\alpha = \epsilon/\delta$. If we somehow guarantee that ϵ and δ are independent of α, then, given that α is uniformly random in $\mathbb{F}$, this equation can only be satisfied with probability $1/|\mathbb{F}|$.

8.1 Integrity via Pairwise MACs

We begin by presenting the construction in which each party has a way to check the share announced by each other party. This construction was proposed initially in [6].

Additive secret-sharing with pairwise MACs

The dealer secret-shares a value $s \in \mathbb{F}$ among n parties $P_1, \dots, P_n$ with pairwise MACs as follows.

1. Sample $s_1, \dots, s_n \in \mathbb{F}$ uniformly at random constrained to $s = s_1 + \cdots + s_n$.
2. For each $i, j \in [n]$, the dealer does the following:
   - Sample $(\alpha_{ij}^{(s)}, \beta_{ij}^{(s)}) \in_R \mathbb{F}$.
   - Compute $\tau_{ji}^{(s)} = \alpha_{ij}^{(s)} s_j + \beta_{ij}^{(s)}$.
3. Distribute the tuple $\mathsf{s}_i = (s_i, \{(\alpha_{ij}^{(s)}, \beta_{ij}^{(s)})\}_{j\in[n]}, \{\tau_{ij}^{(s)}\}_{j\in[n]})$ to party P_i, for $i \in [n]$.

Throughout this subsection we will denote by $\langle s \rangle$ the situation in which the parties have shares $(\mathsf{s}_1, \dots, \mathsf{s}_n)$ of s, with the additional redundancy, as above. Notice that this type of sharing does not leak anything about s to the adversary.

8.1.1 Reconstructing secret-shared values

Now, assume the parties have shares $(\mathsf{s}_1, \dots, \mathsf{s}_n)$ of some value s, with $\mathsf{s}_i = (s_i, \{(\alpha_{ij}^{(s)}, \beta_{ij}^{(s)})\}_{j\in[n]}, \{\tau_{ij}^{(s)}\}_{j\in[n]})$. At reconstruction time, each party P_i announces $(s_i', \{\tau_{ij}'^{(s)}\}_{j\in[n]})$, where $(s_i', \{\tau_{ij}'^{(s)}\}_{j\in[n]}) = (s_i, \{\tau_{ij}^{(s)}\}_{j\in[n]})$ for at least one $i \in [n]$, which corresponds to the indexes of the honest parties who announce their shares correctly. To check the validity of these values, each party P_i executes the following.

Reconstructing shared values with pairwise MACs

Given $\langle s \rangle = (\mathsf{s}_1, \dots, \mathsf{s}_n)$, with $\mathsf{s}_i = (s_i, \{(\alpha_{ij}^{(s)}, \beta_{ij}^{(s)})\}_{j\in[n]}, \{\tau_{ij}^{(s)}\}_{j\in[n]})$, the parties reconstruct s as follows:

1. At reconstruction time, each party P_i sends s_i to all the other parties and $\tau_{ij}^{(s)}$ to party P_j.
2. Each party P_i checks if for all $j \in [n]$ it holds that $\tau_{ji}^{(s)} = \alpha_{ij}^{(s)} s_j + \beta_{ij}^{(s)}$. If so then P_i reconstructs the value $s = s_1 + \cdots + s_n$. Else, the parties abort.

Proposition 8.1. *If the protocol above does not result in abort, then its output is the correct s with probability at least $1 - 1/|\mathbb{F}|$.*

Proof. Assume the parties did not abort, and let $i_0 \in \mathcal{H}$. Let $(s'_j, \tau'^{(s)}_{ji_0})$ for $j \in \mathcal{C}$ be the actual values sent by the corrupt parties to P_{i_0}. Since P_{i_0} did not abort, it holds that, for all $j \in \mathcal{C}$, $\tau'^{(s)}_{ji_0} = \alpha^{(s)}_{i_0j} s'_j + \beta^{(s)}_{i_0j}$. Let us write $s'_j = s_j + \delta_j$ and $\tau'^{(s)}_{ji_0} = \tau^{(s)}_{ji_0} + \epsilon_j$ for $j \in \mathcal{C}$. Recalling that $\tau^{(s)}_{ji_0} = \alpha^{(s)}_{i_0j} s_j + \beta^{(s)}_{i_0j}$, the equations above are equivalent to $\epsilon_j = \alpha^{(s)}_{i_0j}\delta_j$ for $j \in \mathcal{C}$.

Now, suppose that $\delta_{j_0} \neq 0$ for some $j_0 \in \mathcal{C}$. From the above we have that $\alpha^{(s)}_{i_0j_0} = \frac{\epsilon_{j_0}}{\delta_{j_0}}$. It is easy to see that the view of the adversary before the execution of the reconstruction protocol is independent of α_{i_0j} for $j \in \mathcal{C}$ since the adversary only sees $\tau_{ji_0} = \alpha^{(s)}_{i_0j} \cdot s_j + \beta_{i_0j}$, but the uniformly random value β_{i_0j} is unknown to the adversary, which perfectly hides the term $\alpha^{(s)}_{i_0j} \cdot s_j$. From this, we see that the errors δ_{j_0} and ϵ_{j_0} added by the adversary are independent of the uniformly random value $\alpha^{(s)}_{i_0j_0}$, so the equation $\alpha^{(s)}_{i_0j_0} = \frac{\epsilon_{j_0}}{\delta_{j_0}}$ from above can only be satisfied with probability $1/|\mathbb{F}|$.

We obtain that, except with probability $1/|\mathbb{F}|$, it holds that $\delta_j = 0$ for all $j \in \mathcal{C}$, so the announced shares s_i for $i \in [n]$ are all correct and therefore the reconstructed value is correct as well. ☐

8.1.2 Local Operations

Finally, to show that our secret-sharing scheme is suitable for secure computation, we need to show that basic operations can be handled locally by the parties. This is shown below.

8.1.2.1 Addition/Subtraction.

Assume the parties have two shares values $\langle x \rangle = (\mathrm{x}_1, \ldots, \mathrm{x}_n)$ and $\langle y \rangle = (\mathrm{y}_1, \ldots, \mathrm{y}_n)$, with $\mathrm{x}_i = (x_i, \{(\alpha^{(x)}_{ij}, \beta^{(x)}_{ij})\}_{j\in[n]}, \{\tau^{(x)}_{ij}\}_{j\in[n]})$ and $\mathrm{y}_i = (y_i, \{(\alpha^{(y)}_{ij}, \beta^{(y)}_{ij})\}_{j\in[n]}, \{\tau^{(y)}_{ij}\}_{j\in[n]})$. According to our description of the sharing procedure, the values $\alpha^{(x)}_{ij}$ and $\alpha^{(y)}_{ij}$ are sampled separately when sharing the value x and y. However, to make this scheme compatible with local addition and subtraction, we need to assume that $\alpha^{(x)}_{ij}, \alpha^{(y)}_{ij}$ for $i, j \in [n]$ are sampled uniformly at random with $\alpha_{ij} := \alpha^{(x)}_{ij} = \alpha^{(y)}_{ij}$, or, more precisely, that the dealer samples and distributes $\alpha_{ij} \in_R \mathbb{F}$ once, and uses these to compute the values $\{\tau^{(z)}_{ij}\}_{j\in[n]}$ for every new secret-shared value z.

To obtain $\langle x + y \rangle$, each party P_i defines z_i as $(z_i, \{(\alpha_{ij}, \beta^{(z)}_{ij})\}_{j\in[n]}, \{\tau^{(z)}_{ij}\}_{j\in[n]})$, where:

$$\begin{cases} z_i = x_i + y_i \\ \beta^{(z)}_{ij} = \beta^{(x)}_{ij} + \beta^{(y)}_{ij}, \ j \in [n] \\ \tau^{(z)}_{ij} = \tau^{(x)}_{ij} + \tau^{(y)}_{ij}, \ j \in [n]. \end{cases}$$

Subtraction works in a similar fashion.

8.1.2.2 Canonical shares of public values.

Let c be a publicly known value, that is, a value known by all the parties. The parties can obtain shares $\langle c\rangle$ by defining $\mathsf{c}_i = (c_i, \{(\alpha_{ij}, \beta_{ij}^{(c)})\}_{j\in[n]}, \{\tau_{ij}^{(c)}\}_{j\in[n]})$ as follows.

$$c_i = \begin{cases} 0 & \text{for } i \in [n]\setminus\{1\} \\ c & \text{for } i \in \{1\} \end{cases}$$
$$\beta_{ij}^{(c)} = \begin{cases} 0 & \text{for } i \in [n], j \in [n]\setminus\{1\} \\ -\alpha_{ij}\cdot c & \text{for } i \in [n], j \in \{1\} \end{cases}$$
$$\tau_{ij}^{(c)} = \begin{cases} 0 & \text{for } i,j \in [n]\ . \end{cases}$$

Each party P_i can compute its share c_i locally, and, moreover, it can be easily checked that $\tau_{ij}^{(c)} = \alpha_{ji}c_i + \beta_{ij}^{(c)}$ for $i,j\in[n]$, as required by the syntax of the secret-sharing scheme.

8.2 Integrity via Global MACs

In the previous method from Section 8.1 to add integrity to the basic additive secret-sharing scheme $[\![s]\!] = (s_1,\dots,s_n)$, for each (ordered) pair of parties (P_i,P_j), P_i could verify the correctness of P_j's share s_j by means of a key $(\alpha_{ij}, \beta_{ij}^{(s)})$ and a *tag* $\tau_{ji}^{(s)}$ held by s. This way, if any of the announced shares is incorrect, the check the honest party performs would fail, which results in the parties aborting. However, the main drawback with this approach is that each party P_i must hold a key and tag with respect to every other single party P_j, which ultimately means that the size of each party's share grows linearly with n (more concretely, each party's share consists of $1+3n$ elements in $\mathbb{F}$.) This may matter little if n is relatively small, which is the case in the relevant setting of two-party computation, for example. However, for a large number of parties, a share size that grows as $\Omega(n)$ may be just too large.

Given the above, we present in this section a different method to ensure integrity that only adds a small overhead to the size of each share with respect to the basic additive secret-sharing scheme. This was first proposed in the work of [21]. To provide intuition on how this method works, recall that the main goal is to add some redundant information to a given additively-shared value $[\![s]\!] = (s_1,\dots,s_n)$ so that, when the parties announce their shares $(s_1',\dots,s_n')$, the parties can verify that the reconstructed value $s' = s_1' + \cdots + s_n'$ is indeed correct. The method from Section 8.1 ensures this by providing the parties with a method for checking that each party's share s_i' is announced correctly, that is, $s_i = s_i'$, which implies that $s' = s$. However, a crucial observation is that, ultimately, what is desired is that $s' = s$, which can happen even if $s_i' \neq s_i$ for some values of i. Hence, the core idea is to add integrity not to each individual share, but rather to the shared value s itself. This is described in detail below. Notice that each party's share is now made of only 3 elements in $\mathbb{F}$.

Additive secret-sharing with global MACs

The dealer secret-shares a value $s \in \mathbb{F}$ among n parties $P_1, \dots, P_n$ with a global MAC as follows.

1. Sample $s_1, \dots, s_n \in \mathbb{F}$ uniformly at random constrained to $s = s_1 + \cdots + s_n$.
2. Sample $\alpha_1^{(s)}, \dots, \alpha_n^{(s)} \in_R \mathbb{F}$, and let $\alpha^{(s)} = \sum_{i=1}^n \alpha_i^{(s)}$.
3. Sample $\gamma_1^{(s)}, \dots, \gamma_n^{(s)} \in \mathbb{F}$ uniformly at random constrained to $\alpha^{(s)} \cdot s = \sum_{i=1}^n \gamma_i^{(s)}$.
4. Distribute the tuple $\mathtt{s}_i = (s_i, \alpha_i^{(s)}, \gamma_i^{(s)})$ to party P_i, for $i \in [n]$.

For the sake of this subsection, we denote by $\langle s \rangle$ the situation in which the parties have shares $(\mathtt{s}_1, \dots, \mathtt{s}_n)$ of s, with the additional redundancy, as above. Intuitively, we may write $\langle s \rangle = (\llbracket s \rrbracket, \llbracket \alpha \rrbracket, \llbracket \alpha \cdot s \rrbracket)$.

8.2.1 Reconstructing Secret-Shared Values

8.2.1.1 Partial openings.

Assume now that the parties have additive shares $\langle s \rangle = (\mathtt{s}_1, \dots, \mathtt{s}_n)$ of some value s. By *partially opening* $\langle s \rangle$, we mean the following:

1. Each party P_i sends their additive share s_i of $\llbracket s \rrbracket$ to P_1.

2. P_1 computes $s = s_1 + \cdots + s_n$ and then he broadcasts P_1 to all parties.

This basic opening does not ensure the correct value is reconstructed, hence the name partial. In this case the adversary can cause the reconstruction to be $s + \delta$, where, furthermore, δ can depend on s if P_1 is corrupted.

8.2.1.2 Commit-and-open.

Before we describe the mechanism for the parties to reconstruct values correctly, we describe another type of opening that does not necessarily ensure that the reconstructed value is correct, but at least guarantees that the added adversarial error is independent of the secret. For this construction we will need to make use of a cryptographic tool known as a *commitment*. For a formal treatment on these see for example [17]. Intuitively, a commitment scheme is a pair $(\mathsf{Commit}, \mathsf{Open})$ where $\mathsf{Commit}(m, r)$ allows a participant to "commit" to a value m using a uniformly random "key" r, and $\mathsf{Open}(m, r, c)$ checks whether the commitment c corresponds to m and r. The basic properties of such a scheme are (1) $\mathsf{Commit}(m, r)$, for a uniformly random r, does not reveal anything about m and (2) given $c = \mathsf{Commit}(m, r)$, it is not possible to find m' and r' with $m \neq m'$ such that $\mathsf{Open}(m', r', c)$ reports that the commitment c corresponds to m', r'. An effective construction of such a scheme consists of $\mathsf{Commit}(m, r) = \mathsf{H}(m \| r)$, where H is a cryptographic hash function.

With this tool at hand, the parties commit-and-open to a shared value $[\![z]\!] = (z_1, \dots, z_n)$ as follows.

1. Each party P_i samples r_i and computes the commitment $c_i = \mathsf{Commit}(z_i, r_i)$. Then P_i broadcasts c_i.

2. After all these values are broadcast, each party P_i broadcasts (z_i, r_i).

3. The parties check that, for all $i \in [n]$, $\mathsf{Open}(z_i', r_i', c_i')$ accepts, where c_i' and (z_i', r_i') are the values broadcast by P_i in the previous two steps. If $\mathsf{Open}(z_i', r_i', c_i')$ rejects, then the parties abort.

A corrupt party P_i may still lie about its own share by broadcasting $z_i + \delta_i$, so the resulting reconstructed value may still be incorrect. However, P_i only broadcasts $z_i + \delta_i$ after he has broadcasted the commitment c_i, and by the properties of the commitment scheme sketched above, this party cannot announce a different share than the one it has committed to, which means that P_i has chosen δ_i based on the information sent in the first part of the protocol. At this stage only commitments to the shares have been sent, which leak nothing about the shares themselves, so the possible errors δ_i are independent of the other shares and hence independent of the secret, as desired.

8.2.1.3 Reconstruction.

Let $\langle s \rangle = (\mathtt{s}_1, \dots, \mathtt{s}_n)$ be a value shared by the parties. Now we show how to put together the different reconstruction methods from above so that the parties learn s correctly.

Reconstructing shared values with global MACs

Given a shared value $\langle s \rangle = ([\![s]\!], [\![\alpha]\!], [\![\alpha \cdot s]\!])$, the parties reconstruct s as follows:

1. The parties partially open $s' \leftarrow [\![s]\!] = (s_1, \dots, s_n)$.
2. The parties compute locally $[\![\mu]\!] \leftarrow [\![\alpha \cdot s]\!] - s' [\![\alpha]\!]$.
3. The parties commit-and-open $\mu' \leftarrow [\![\mu]\!]$. If $\mu' = 0$, then the parties accept s' as the opened value. Else, the parties abort.

Proposition 8.2. *If the protocol above does not result in abort, then each party outputs s with probability at least $1 - 1/|\mathbb{F}|$.*

Proof. Let us write $s' = s + \delta$, where δ is an additive error introduced by the adversary that might depend on s. Also, let us write $\mu' = \mu + \epsilon$, where ϵ is also an additive error introduced by the adversary, but this, unlike δ, does not depend on the value of the secret μ. We have that

$$\mu' = \mu + \epsilon = (\alpha s - s'\alpha) + \epsilon = \alpha s - (s + \delta)\alpha + \epsilon = \epsilon - \alpha \cdot \delta,$$

so $\mu' = 0$ if and only if $\epsilon - \alpha \cdot \delta = 0$.

Now, assume that $\delta \neq 0$, we would have that $\alpha = \epsilon/\delta$, and since ϵ is chosen independently of α, which is uniformly random, this equation can only be satisfied with probability $1/|\mathbb{F}|$. From this we see that, if the protocol does not result in abort, $\delta = 0$ except with probability $1/|\mathbb{F}|$, which means that the reconstructed value is $s' = s$. □

8.2.2 Local Operations

8.2.2.1 Addition/Subtraction.

Given two shared values $\langle x \rangle = (\mathtt{x}_1, \ldots, \mathtt{x}_n)$ and $\langle y \rangle = (\mathtt{y}_1, \ldots, \mathtt{y}_n)$, with $\mathtt{x}_i = (x_i, \alpha_i^{(x)}, \gamma_i^{(x)})$ and $\mathtt{y}_i = (y_i, \alpha_i^{(y)}, \gamma_i^{(y)})$ for $i \in [n]$, it is possible for the parties to locally obtain shares $\langle x + y \rangle$. This requires that $\alpha_i := \alpha_i^{(x)} = \alpha_i^{(y)}$, that is, the dealer samples $\alpha_1, \ldots, \alpha_n \in_R \mathbb{F}$ once, and uses $\alpha = \sum_{i=1}^{n} \alpha_i$ to define the shares of all subsequent values. This way, if $\langle x \rangle = ([\![x]\!], [\![\alpha]\!], [\![\alpha \cdot x]\!])$ and $\langle y \rangle = ([\![y]\!], [\![\alpha]\!], [\![\alpha \cdot y]\!])$, $\langle x + y \rangle$ may be computed locally exploiting the homomorphic properties of basic additive secret-sharing as $([\![x]\!] + [\![y]\!], [\![\alpha]\!], [\![\alpha \cdot x]\!] + [\![\alpha \cdot y]\!])$. More precisely, the shares $(\mathtt{z}_1, \ldots, \mathtt{z}_n)$ are defined as $\mathtt{z}_i = (z_i, \alpha_i, \gamma_i^{(z)})$, where $z_i = x_i + y_i$ and $\gamma_i^{(z)} = \gamma_i^{(x)} + \gamma_i^{(y)}$ for $i \in [n]$.

8.2.2.2 Canonical shares of public values.

Given a value $c \in \mathbb{F}$ known by all the parties, the parties can obtain shares $\langle c \rangle = (\mathtt{c}_1, \ldots, \mathtt{c}_n)$ by defining $\mathtt{c}_i = (c_i, \alpha_i, \gamma_i^{(c)})$, with $c_i = 0$ for $i \in [n] \setminus \{1\}$, $c_i = c$ for $i = 1$, and $\gamma_i^{(c)} = \alpha_i \cdot c$ for $i \in [n]$.

8.3 Online Phase

Let $\langle \cdot \rangle$ denote "authenticated" sharings as the ones from either Section 8.1 or Section 8.2. These have in common that local addition/subtraction of shared values, together with local multiplication and addition/subtraction of publicly known values, is possible. Furthermore, when reconstructing secret-shared values, although the adversary may initially cause the parties to open a shared-value incorrectly, the parties can execute a verification step that ensures that, with high probability, the opened value under consideration is reconstructed correctly.

For our protocol we assume that the extra data required for the authenticated secret-sharing scheme $\langle \cdot \rangle$, like the necessary keys and tags, or their shares, depending on whether the authentication mechanism chosen is pairwise or global MACs, is computed by the parties in a preprocessing phase. During this phase the parties also obtain a multiplication triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ with $a, b \in_R \mathbb{F}$ uniformly random and unknown to the adversary and $c = a \cdot b$, for every multiplication gate in the circuit under consideration. Furthermore, for every input gate corresponding to party P_i, the parties have $\langle r \rangle$, where $r \in \mathbb{F}$ is uniformly random and known only to P_i.

With these tools at hand, the parties can securely compute the given arithmetic circuit in a similar way as in Section 7.5. Multiplication gates are processed in essentially the same way: the parties reconstruct a masked version of the inputs to the multiplication gate, making use of a multiplication triple. However, since the secret-sharing scheme $\langle\cdot\rangle$ is more complex than simple additive secret-sharing $[\![\cdot]\!]$, it is not possible for the parties to obtain shares of the inputs to the computation non-interactively. This is achieved in a similar way as the protocol from Section 5.3, by letting each party broadcast a masked version of their inputs using random values that are secret-shared, and then the parties add this publicly known value to these shares. This is detailed below.

Secure computation based on authenticated secret-sharing

Offline phase: The parties obtain from the preprocessing phase:

- The necessary keys/shares for the secret-sharing scheme $\langle\cdot\rangle$
- A multiplication triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ for every multiplication gate
- For every input gate with owner P_i, a uniformly random shared value $\langle r\rangle$ only known to P_i.

Online phase: The parties execute the following.

Input gates. For every party P_i holding input x, the parties execute the following. Let $\langle r\rangle$ be a random shared value, where P_i knows r.

1. P_i broadcasts $e = x - r$.
2. The parties compute the sharings $\langle x\rangle \leftarrow \langle r\rangle + e$.

Addition gates. These are handled locally by using the properties of the secret-sharing scheme $\langle\cdot\rangle$.

Multiplication gates. Given two shared values $\langle x\rangle$ and $\langle y\rangle$, the parties obtain $\langle xy\rangle$ as follows. Let $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ be a multiplication triple.

1. The parties compute locally $\langle d\rangle \leftarrow \langle x\rangle - \langle a\rangle$ and $\langle e\rangle \leftarrow \langle y\rangle - \langle b\rangle$
2. The parties reconstruct $d \leftarrow \langle d\rangle$ and $e \leftarrow \langle e\rangle$.
3. The parties compute locally $\langle z\rangle = d\langle b\rangle + e\langle a\rangle + \langle c\rangle + de$.

Output gates. The parties reconstruct $\langle z\rangle$ for every output shared value.

As with the protocol from Chapter 7, privacy is guaranteed in the input phase since the input x is masked with the random value r when P_i broadcasts $e = x - r$, and similarly for the reconstructed values $d = x - a$ and $e = y - b$ in the multiplication. Correctness follows from the fact that, if $d = x - a$, $e = y - b$ and $c = ab$, it holds that $db + ea + c + de$ is equal to xy.

Bibliography

[1] A. Aly, K. Cong, D. Cozzo, M. Keller, E. Orsini, D. Rotaru, O. Scherer, P. Scholl, N. Smart, T. Tanguy, et al. Scale–mamba v1. 14: Documentation, 2021.

[2] B. Applebaum, Y. Ishai, and E. Kushilevitz. How to garble arithmetic circuits. *SIAM Journal on Computing*, 43(2):905–929, 2014.

[3] M. Ball, T. Malkin, and M. Rosulek. Garbling gadgets for boolean and arithmetic circuits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 565–577, 2016.

[4] Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure mpc with linear communication complexity. In *Theory of Cryptography Conference*, pages 213–230. Springer, 2008.

[5] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 351–371. 2019.

[6] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 169–188. Springer, 2011.

[7] D. Bogdanov, M. Jõemets, S. Siim, and M. Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *International conference on financial cryptography and data security*, pages 227–234. Springer, 2015.

[8] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, et al. Secure multiparty computation goes live. In *International Conference on Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[9] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202, 2000.

[10] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[11] R. Canetti, A. Cohen, and Y. Lindell. A simpler variant of universally composable

security for standard multiparty computation. In *Annual Cryptology Conference*, pages 3–22. Springer, 2015.

[12] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. Ezpc: programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511. IEEE, 2019.

[13] D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[14] R. Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 364–369, 1986.

[15] R. Cramer, I. Damgård, and U. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 316–334. Springer, 2000.

[16] R. Cramer, I. Damgard, and J. B. Nielsen. Secure multiparty computation and secret sharing-an information theoretic approach, 2012.

[17] I. Damgård. Commitment schemes and zero-knowledge protocols. In *School organized by the European Educational Forum*, pages 63–86. Springer, 1998.

[18] I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 445–465. Springer, 2010.

[19] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

[20] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer, 2007.

[21] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[22] D. Demmler, T. Schneider, and M. Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[23] C. Dwork, A. Roth, et al. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

[24] M. Franklin and M. Yung. Communication complexity of secure computation. In

Proceedings of the twenty-fourth annual ACM symposium on Theory of computing, pages 699–710, 1992.

[25] R. Gennaro, M. O. Rabin, and T. Rabin. 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*, pages 101–111, 1998.

[26] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[27] S. Goldwasser and Y. Lindell. Secure multi-party computation without agreement. *Journal of Cryptology*, 18(3):247–287, 2005.

[28] V. Goyal, Y. Song, and C. Zhu. Guaranteed output delivery comes free in honest majority mpc. In *Annual International Cryptology Conference*, pages 618–646. Springer, 2020.

[29] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, 2016.

[30] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

[31] M. Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1575–1590, 2020.

[32] M. Keller, E. Orsini, and P. Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016.

[33] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 158–189. Springer, 2018.

[34] J. Kilian. Founding cryptography on oblivious transfer. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 20–31, 1988.

[35] A. Lapets, F. Jansen, K. D. Albab, R. Issa, L. Qin, M. Varia, and A. Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, pages 1–5, 2018.

[36] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.

[37] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85, 1989.

[38] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[39] R. Shostak, M. Pease, and L. Lamport. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[40] L. R. Welch and E. R. Berlekamp. Error correction for algebraic block codes, Dec. 30 1986. US Patent 4,633,470.

[41] A. C. Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.