Universidade do Minho
Escola de Engenharia

Daniel Casanova Faria Torres

Secure Multiparty Computation Protocols

Dezembro de 2014

Universidade do Minho
Escola de Engenharia

Daniel Casanova Faria Torres

Secure Multiparty Computation Protocols

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Professor José Carlos Bacelar Almeida

Dezembro de 2014

Acknowledgments

I would like to express my gratitude to my supervisor, José Carlos Bacelar Almeida, for giving me guidance and for his availability throughout my studies. His suggestions were of high value, as they made me understand the subject at an increasingly deeper lever.

I am thankful to the Alexandra Institute for hosting me as an intern. It was a pleasure to work with such inspiring people.

To Jakob Illeborg Pagter, for accepting me as an intern in the Security Lab, and providing me the best working conditions.

To Janus Dam Nielsen, for the tremendous help with my integration. I am very thankful to him for reviewing my texts and for his suggestions on how to organize my writing.

My thanks also go to Manuel Barbosa for helping me with finding a place to have the internship.

Finally, I would like to thank my family for supporting my studies. None of this could have been made without their unconditional help.

Abstract

Secure Multiparty Computation (abrv. MPC) is a group of techniques that enable multiple entities to compute some function on their private inputs. More formally, it enables a set of players $\{P_1, \ldots, P_n\}$, each of them holding private inputs x_i, where i is the index of the player, to evaluate a function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ so that every player P_i learns y_i.

Multiparty Computation has many application scenarios such as private auctioning, computation outsourcing, or private information retrieval. These and other applications make MPC a very appealing object of study, since it may be a solution to problems in different fields.

The objective of this thesis is twofold. First, we aim to provide a comprehensive guide to the current state of multiparty computation protocols. We do so by inspecting the two categories where most general functionality secure function evaluation protocols are inserted - boolean or arithmetic circuit evaluation. Moreover, we show how to implement MPC protocols with current constructions, and what problems are inherently connected to the chosen representations.

Second, we document some of the design choices behind a partial implementation of a concrete secure arithmetic circuit evaluation protocol - the *SPDZ* protocol by Damgård et al. [10]. We explain the protocol in general terms, and then go into the details of some subprotocols, namely those that were implemented.

Resumo

Secure Multiparty Computation (abrv. MPC) é uma área que engloba um conjunto de técnicas que permitem que diferentes entidades avaliem uma função sem revelar os seus valores privados.

Formalmente, esta técnica permite a um conjunto de participantes $\{P_1, \ldots, P_n\}$, que possuem inputs privados x_i, onde i é o índice de cada participante, avaliarem a função $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ de forma a que cada participante P_i obtenha unica e exclusivamente y_i.

Multiparty Computation destaca-se pelo facto de ser aplicável em diversas áreas como private auctioning, computation outsourcing ou private information retrieval. Estas aplicações tornam MPC um tópico de bastante interesse, pois engloba a solução para problemas de diferentes ramos.

Esta dissertação é composta por duas partes.

Primeiro, disponibilizamos um manual de introdução ao tópico, com o objectivo de descrever o estado actual da tecnologia. Com esse fim, introduzimos os dois grandes ramos onde todos os protocolos de avaliação genérica se enquadram - avaliação de circuitos booleanos ou avaliação de circuitos aritméticos. A nossa descrição engloba também aspectos práticos, como os recursos existentes em termos de frameworks para desenvolvimento de protocolos multiparty. Por fim, descrevemos os problemas que estão associados à escolha de uma representação em vez de outra.

Na segunda parte, descrevemos algumas das decisões que foram tomadas aquando da implementação parcial do protocolo SPDZ de Damgård et al. [10]. Com esse objetivo, começamos por descrever o protocolo em termos gerais, e posteriormente explicamos o funcionamento em detalhe de alguns subprotocolos, nomeadamente os que foram implementados.

Contents

1 Introduction 1
1.1 Secure Function Evaluation 3
1.2 Applications 4
1.3 Goals and structure of this dissertation 6

2 Multiparty Computation Protocols 8
2.1 Settings 9
2.2 Passive, Active and Covert Security 10
2.3 Information-theoretic Security and Cryptographic Security . 11
2.4 UC Framework 12
2.5 Communication Models 13
2.6 Initial Setup 14

3 Circuit Evaluation 15
3.1 Boolean Circuit Evaluation 16
3.1.1 Garbled Circuits Overview 16
3.1.2 Garbled Circuits in Detail 17
3.1.3 Yao's Garbled Circuits Protocol 21
3.1.4 Execution Example 21
3.1.5 Implementations 23
3.2 Arithmetic Circuit Evaluation 25
3.2.1 Secret Sharing 25
3.2.2 Secret Sharing as a tool for MPC 26
3.3 Comparison 28

4 SPDZ 30
4.1 Description 30
4.1.1 Secret Sharing 31
4.1.2 Online phase 31
4.1.3 The Preprocessing Model 34
4.1.4 Commitment Scheme 34

4.1.5 MAC Algorithm 35
4.2 Protocols 36
4.2.1 EncCommit 36
4.2.2 Reshare 38
4.2.3 MacCheck 39

5 Implementations **42**
5.1 Class Hierarchy 42
5.2 Network 43
5.3 Protocol Classes 44
5.4 AES-PRNG 45
5.5 Commitment Classes 46
5.6 Details 46

6 Conclusion and Results **49**
6.0.1 Multiparty Computation 49
6.0.2 Boolean and Arithmetic Circuits 50
6.0.3 The SPDZ Protocol 51
6.1 Future Works 52

Listings **56**

List of Figures

1.1 Setting with physical TP / Same setting where TP was replaced by a MPC protocol 3

2.1 A visual representation of a player in the passive security (left) and active security (right) settings. In the passive setting the player has a leakage port (*Leak*) whereas in the active setting the player also has an influence (*Inf*) port 10
2.2 The real world and ideal world scenarios, from left to right. Extracted from [12] 13

3.1 Boolean circuit that represents the `AND` operator for 4-bit numbers 15
3.2 Simple OR gate 21

4.1 Arithmetic Circuit representing $f(x, y, z) = (x + y) * z$ 32
4.2 Commitments Protocol[21] 35
4.3 EncCommit Protocol[21] 37
4.4 Reshare Protocol[21] 39
4.5 MACCheck Protocol[21] 40

5.1 Interaction between two Protocol instances 43
5.2 Use of Templates in Network Class 44
5.3 PRNG used in SPDZ 45
5.4 Interaction between CommitObject and CommitVerifier Classes, where Alice is the prover and Bob the verifier. 46
5.5 Boost ASIO framework 48

List of Tables

2.1 Necessary conditions for MPC to be possible, depending on the number t of corrupted players. 12

3.1 Truth table value *boxes* 22

Chapter 1

Introduction

A combination of factors has led organisations to store enormous amounts of data. Having access to the right information has become an aspect of vital importance, as it can enable a sales department, marketing team or even an executive board of a company to take crucial decisions. In addition, the costs of storing and collecting information keep decreasing, thanks to technological advances such as cloud-based infrastructures.

Despite having these massive databases, organisations still struggle on how to handle sensitive data in a safe way. Cryptography has helped in this field. Cryptographic techniques such as symmetric/public-key encryption have been available for years, and are used extensively when it comes to securing communications or storage.

Even though there are tools to develop these secure, robust and privacy-preserving systems, there is another set of problems that can be of much interest to companies, for which the community has been working on. One of them is the secure multiparty computation (abrv. SMPC or just MPC) paradigm. Secure multiparty computation is a set of techniques that enable multiple parties to jointly compute some function on their private inputs.

In this chapter we provide a concise explanation of what MPC is and what set of problems it solves. We do so by describing some problems that organisations face, and what are the available solutions to those problems. Later, we propose MPC alternatives and explain their benefits.

The problem and the usual approach There are many situations where multiple parties want to obtain some result, but do not want to reveal their contributions.

As an example, one can imagine an art auction. The investors may want the auction to be closed (no bids displayed), so that no investor knows the intentions of the other.

From the point of view of the investors, they expect the auction to be secure. In essence, the investors only want two security requirements to be fulfilled:

- Privacy - No bid will be disclosed to anyone.
- Correctness - The result of the auction corresponds to the actual highest bid.

Now that we described the problem, we will focus on the solution. The typical way to solve this problem is to have some kind of *referee* that receives the votes and announces the result. To reduce the abstraction level, we can think of the referee as a company that specialises on this kind of auctions. Trivially, it follows that the investors have to trust the referee. They trust that he will process the auction in such a way that the aforementioned security requirements are respected. In summary they trust the referee to pick the best bid, without revealing to anyone the other bids.

This classical solution suffers from this trust problem. The way to make this work is to pay the referee enough money so that he doesn't have an appealing reason to fail. If there are many investors and the exposed art items are expensive, an investor may try to bribe the referee so that he selects his bid, instead of the highest. In the same fashion, an investor may force the referee to disclose the other investors bids, thus learning how much money each of them is willing to pay for a certain item.
The need to trust some entity is common among this set of problems. As such, from now on, we will use the term *trusted party* instead of referee. Generally speaking, the role of the trusted party is to receive the private information from each of the participants and send the correct result of the computation to every participant.

The Multiparty Computation approach The goal of secure multiparty computation techniques is to achieve the same results by replacing the previously mentioned trusted party by a secure protocol (see figure 1.1).

Based on the security requirements of the auction example , we generalise our definition of security for any multiparty computation protocol . We state that a multiparty computation protocol is secure if it accomplishes these two requirements:

- Privacy - The private input of the players is never revealed.
- Correctness - The output of the computation is correct.

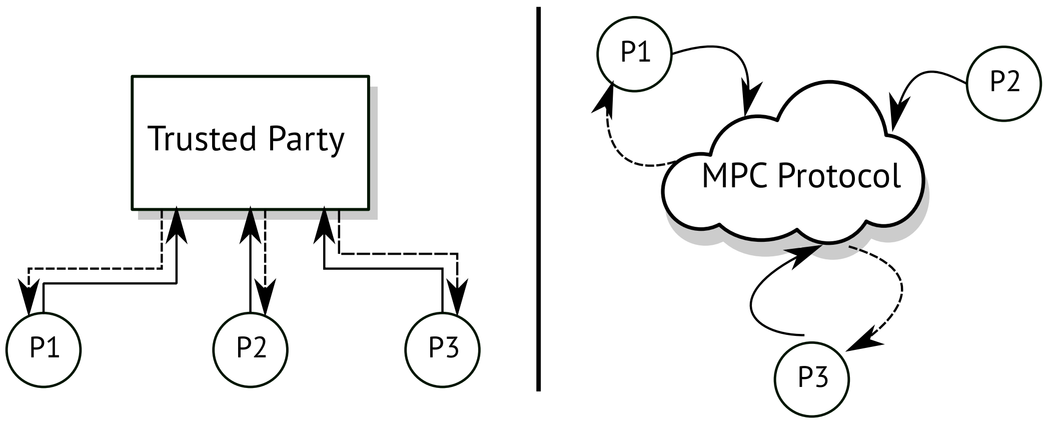


Figure 1.1: Setting with physical TP / Same setting where TP was replaced by a MPC protocol

Multiparty Computation and Applications In the following sections we provide a more detailed explaination of what MPC is. We describe the particular problem of Secure Function Evaluation (see section 1.1). Examples of practical applications that could be achieved with MPC are listed in section 1.2.

1.1 Secure Function Evaluation

Along with MPC comes the concept of Secure Function Evaluation (abbreviated SFE) which is a more concrete example of a Secure Multiparty Computation . Secure Function Evaluation enables any function to be evaluated by various parties, in such a way that no party learns nothing more than:

- The party's own private contribution to the function
- The output of the function.
- Information that can be deducted from the output of the function

In a formal notation, SFE is a cryptographic protocol that allows multiple parties $P_1 \dots P_N$ with respective private inputs $X_1, X_2, \dots X_N$ to evaluate function $f(X_1, X_2, \dots X_N)$.

Permitted Leaks The last item refers to the obvious case were the party can learn the other parties' input from the result. As an example, we can image two parties which want to securely evaluate a *sum* function. A party can always learn the other's input by subtracting the result with it's contribution. We do not consider this to be a leak.

The millionaires problem An easy example of a Secure Function Evaluation is the Millionaire's problem, which was introduced by Andrew Yao [27]. The problem was formulated as:

- Two millionaires (M_1, M_2) have their respective fortunes (F_{M_1}, F_{M_2}).
- They want to know which one of them is richer.
- Both millionaires do not want to reveal their fortune. That means:
  - M_1 should not learn F_{M_2};
  - M_2 should not learn F_{M_1};

This classical example clearly exposes the purpose of SFE. Yao published a scheme that solves this problem without the need of an external trusted party. Moreover, Yao designed a general protocol for two-party computation, that was latter named "Garbled Circuits". We provide a full description of Garbled Circuits in section 3.1

1.2 Applications

There are multiple scenarios where a party may want to perform computations over sensitive data.

Private Information Retrieval

An airline company wants to check wether there are terrorists in the list of passengers of a certain flight. The company has to query the government server to know whether a passenger is a suspected terrorist or not. The problem is that nor the company wants to reveal the passenger details nor the Government wants to reveal the full list of suspected terrorists. The airline company and the Government server could engage in a MPC and check if any of the passengers is a terrorist. Note that at the end of this process, the government does not know which passengers where queried, and the company does not have the list of suspected terrorists. This way both the passengers and government privacy are preserved.

Distributed Certificate Authority

A Certificate authority (abbreviated CA) sells certificates to clients, by signing them with it's own private key. In order to protect the key, the CA could have it split in multiple shares, in different locations. The problem is that for the signing operation to work, the shares have to be joined by some entity. This entity would be a point of failure since it would have all the shares and hence the private key of the CA. One can think of a Distributed Certificate Authority to avoid this point of failure. The signing operation could be done in a MPC fashion, with each CA server to contribute to the signing operation with it's private share. This way there is no entity with access to the CA private key.

Electronic Bidding

Suppose that the government wants to make a contract with the company that offers the best price. Each of the companies could contribute with their bid without revealing it. The result would be the best price along with the company that made that bid.

Private Satellite Collision Analysis

There are many (< 7000) spacecrafts orbiting the Earth. Recently, two satellites belonging to the US and Russia collided in orbit. This problem could be solved if countries revealed their satellites orbit information. But in reality, countries do not want to reveal the orbit of their most strategic satellites. Recent works by [17] propose a MPC solution to the Satellite Collision problem. In this work, the authors propose a way for operators to discover satellites with high probability of collision, by doing MPC calculations, thus not revealing the private orbit details. Floating point operations in multiparty computation are also described in the same article.

Location-Sharing Applications

Location-aware smartphone applications (dating, networking) typically work by sending the user location to a central server. As an example, in the case of location sharing based on close by contacts, the central server processes the query and sends the list of contacts that are physically close to the client. With MPC , each user can run a protocol with the server and obtain the list of contacts that are close to him, while the server remains oblivious as to which location was queried or which of the contacts are near that exact location.

Analysis between State Institutions

Different institutions cannot merge their data into a super-database, as this is illegal in most countries. With MPC, they can obtain mutual information without disclosing private information or breaking the law.

MPC techniques have evolved to the point where some schemes can actually be used in practice. In Denmark [5], there was a project involving sugar beet farmers and the Aarhus University. The project was the first to put MPC in practice in a large-scale scenario. The farmers were able to find the *market trading price* which is a price per unit of the commodity that is traded. This was all done without a physical trusted party and in such a way that the bids remained private.

Even though this topic has seen some growth in terms of number of publications, there aren't many documented cases of large-scale use of MPC. Nevertheless, there are important initiatives like the EU PRACTICE project that aims to build a secure cloud framework, based on techniques such as MPC.

1.3 Goals and structure of this dissertation

Despite the growing number of publications, we still consider MPC to be a hard topic to understand. A consequence of being an area of great expansion is that many definitions are being added, and even core concepts are still under study. For this reason, it is hard to have a big picture of the current status.

This logic brings us to our first goal, which is to provide a useful learning resource on MPC. We want to present MPC in understandable terms, and give a summary of the current knowledge, so that even an unfamiliar reader may enter the topic. By showing concrete protocols that work under different models, we expect to provide a good overview of the current schemes.

The second goal is to document the partial implementation of the SPDZ [10] protocol. The work behind this implementation was carried out while the author was an intern at the Alexandra Institute (Denmark). By explaining the design choices and the protocol itself, we expect to provide a useful description for further implementation efforts.

This dissertation is structured in two parts, each of them comprising two chapters.

Chapters 2 and 3, *Multiparty Computation Protocols*
This part is focused on describing the current state of MPC protocols.

We do so by first showing the different security parameters that differentiate current MPC schemes. Here we include notions like different adversarial capabilities, unconditional/computational security or the UC framework.

Every general purpose multiparty computation protocol belongs to one of two sets. Those that evaluate boolean circuits and those that evaluate arithmetic circuits. We describe the two approaches with details and examples.

Finally we compare the two approaches (boolean, arithmetic) and discuss what are the outcomes of choosing one representation over the other.

Chapters 4 and 5, ***The SPDZ protocol***

Here we focused on describing the SPDZ [10] protocol, which enables any arithmetic circuit to be securely evaluated. We explain the protocol and exemplify with the evaluation of a concrete arithmetic circuit. As the author had an internship where he worked on the implementation of some parts of the protocol, we describe some of the subprotocols that were implemented as well.

Later, we present some of the implementation details such as how the classes were modeled, or how we implemented constructions that were used inside the protocols, like the commitment scheme and the pseudo random number generator.

The final chapter summarizes our results and leaves a suggestion for further works.

Chapter 2

Multiparty Computation Protocols

Multiparty Computation is a problem that was first described and studied by Yao [27] and Goldreich et al. [14]. In these classical results, it was proved that secure multiparty computation was achievable. The results revealed that n parties are able to compute the value of a function with n inputs, in a way that every party learns the result, but nothing more. Since then, MPC evolved to become a research field.
It is of major importance to begin this section with security definitions, so that different protocols can be compared. As such, the security aspects are provided in section 2.1. In the same section there is a characterisation of the possible adversaries and the communication models.

Finally the difference between information-theoretic security and cryptographic security is given in section 2.3.

Most of the MPC protocols fall in two main categories. There are protocols that allow the evaluation of *Boolean Circuits* and others that allow the evaluation of *Arithmetic Circuits*.

Boolean Circuit Evaluation A Boolean circuit evaluation protocol, which is found in literature by the name of *Garbled Circuits* is discussed in section 3.1. As *Garbled Circuits* is a protocol (or family of protocols) with various steps, we provide a simpler explanation (see section 3.1.1) followed by a detailed description (see 3.1.3).
Garbled Circuits are also shown "in action". A transcript of an execution can be followed in section 3.1.4. Finally, we describe a framework, called Fairplay, which uses the same protocol . This is available in section 3.1.5, where we also provide an explanation of how to implement MPC programs with it.

Arithmetic Circuit Evaluation Schemes for secure evaluation of Arithmetic circuits are mostly based on Secret Sharing. In section 3.2.1 we describe what secret sharing is, and exemplify with a concrete secret sharing scheme. Further, we describe an arithmetic circuit scheme by Bogdanov [4] that uses secret sharing.

2.1 Settings

Security is usually defined by what properties are wanted from the cryptographic technique. For instance, when we talk about Encryption schemes we may say that we require ciphertext *indistinguishability* i.e., that an adversary cannot distinguish two ciphertexts based on the message they encrypt.

In the case of multiparty computation, the aspects of security are also usually defined in terms of adversarial capabilities, but there are other variables that add to the equation. Since multiparty computation is interactive by nature, it's security notions should be based on those of protocols. Many authors worked on the construction of protocols with provable security.

Protocols for securely evaluating a function fall in this set of works. Canetti et al. [6] define the types of adversaries, security requirements and how to define a SFE protocol. That terminology will be used in the upcoming descriptions.

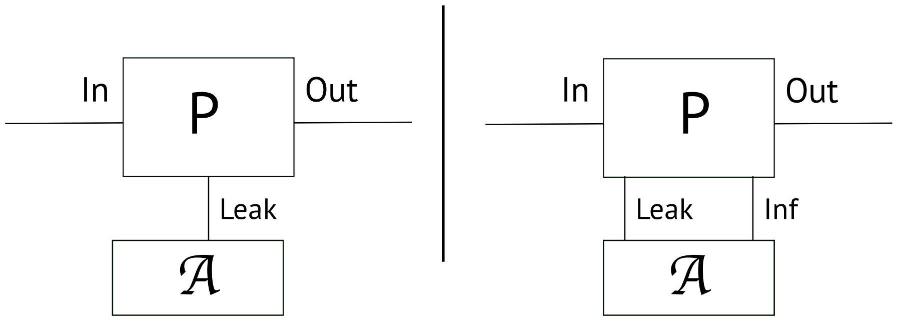


Figure 2.1: A visual representation of a player in the passive security (left) and active security (right) settings. In the passive setting the player has a leakage port (*Leak*) whereas in the active setting the player also has an influence (*Inf*) port

.

A major contribution by Canetti et al. [6] was the introduction of the concept of *Universally Composable* protocols. In this section we will give an explanation of the *UC* framework and how it can help build secure protocols.

2.2 Passive, Active and Covert Security

We consider an adversary as en entity which can potentially corrupt honest parties, access their private values and take control of them. This entity might have different capabilities. A way to model adversary behaviour can be to define how the participants P of the protocol interact with the adversary $\mathcal{A}$ (see figure 2.1). Each participant of the protocol has always an Input and an Output port. If the adversary is capable of using data from the participants, then each participant also has a leakage (*Leak*) port. This port may leak additional information from the participants to the adversary.

The participant might as well have an influence port (*Inf*) . This port is used by the adversary to send instructions to the participant. With this abstraction of participant in mind, we now describe the different types of adversaries found in literature.

- A *passive* or *semi-honest* adversary. In this setup, the player only has a leakage port. The adversary can learn the private inputs of a group of parties but can not control those parties. That means the parties still follows the protocol.

- An *active* adversary. In this scenario each party has a leakage and an influence port. This means that the adversary is able to control a party or a group of parties. The parties follow the adversary orders instead of the protocol. Nowadays it is considered to be more realistic than the passive model. Usually solutions in this model are less efficient than those in the passive model.

- The less often referred *covert* adversary. In this setup, an adversary who deviates from the protocol is caught with high probability. This model usually leads to more efficient constructions than those in the active security setting.

  Despite it's security relaxation (the fact that it allows an attacker to cheat), this model might still be realistic for many application scenarios. Consider two business partners that will do a joint computation on their private values. It seems reasonable to admit that no party will cheat, knowing in advance that it may get caught with 99/100 probability, thus ruining the partnership. This concept is relatively recent and does not appear on older publications.

2.3 Information-theoretic Security and Cryptographic Security

When we say that a certain protocol is information-theoretic secure, it means that the security is based on information theory. A technique that has information-theoretic security is secure against an adversary with unlimited computing power. It is not an unachievable goal. In fact, some techniques such as Shamir secret sharing (that will be later discussed in section 3.2.1) have this characteristic. Many MPC protocols also have information-theoretic security. Notice we are assuming the channels to be secure (i.e. they provide authentication and confidentiality).
If a protocol has Cryptographic assumption, it's security is based on some computational assumption. That means that an attacker with sufficient computing power can break the security. It is also assumed that the channels provide authentication.

Usually this schemes rely on some kind of hardness assumption. A typical example of cryptographic security is the set of schemes that base their security on the RSA problem. The RSA problem says that given a public key (N, e) and a ciphertext $c = p^e \pmod N$ it is hard to find p, where hard means that there is no PPT (Probabilistic Polynomial Time) algorithm that solves the problem.

Bounds on the number of corrupted players Fundamental results by Goldreich et al. [13] define the bounds on the number of corrupted players t for every MPC protocols with Cryptographic security. They show that any function can be securely evaluated with cryptographic security against t corrupted players where $t < n$ for a passive adversary and $t < n/2$ in the case of an active adversary.

Following this line of works, Chaum et al. [7] and Ben-Or et al. [3] proved the bounds in the Information Theoretic setting. They show that any function can be securely evaluated with unconditional security, as long as $t < n/2$ in the case of a passive adversary and $t < n/3$ in the case of an active adversary. These results are displayed in table 2.1. The table was extracted from Maurer [19] and summarizes the results in Goldreich et al. [13], Ben-Or et al. [3] and Chaum et al. [7].

setting	Cryptographic		Information-theoretic	
adversary type	passive	active	passive	active
condition	$t < n$	$t < \frac{n}{2}$	$t < \frac{n}{2}$	$t < \frac{n}{3}$

Table 2.1: Necessary conditions for MPC to be possible, depending on the number t of corrupted players.

Corruption Strategy - adaptive vs static A non-adaptive or static adversary has to decide, before the protocol execution, which set of players he will corrupt, whereas an adaptive adversary may corrupt new players during the protocol execution. The adaptive setting is usually the most preferred when modelling the adversary corruption strategy.

2.4 UC Framework

The UC framework was introduced by Canetti et al. [6] and since then has been used by many authors to prove the security of multiparty computation protocols. For a clearer understanding of the UC framework we must first consider an *ideal* world. This ideal world is secure by definition. In the ideal world, all the parties give their private inputs to a *functionality* $\mathcal{F}$. An ideal functionality is a sequence of actions that represent what the protocol should do. An environment , $\mathcal{Z}$ is an agent that "sees" the messages between the functionality and the players.

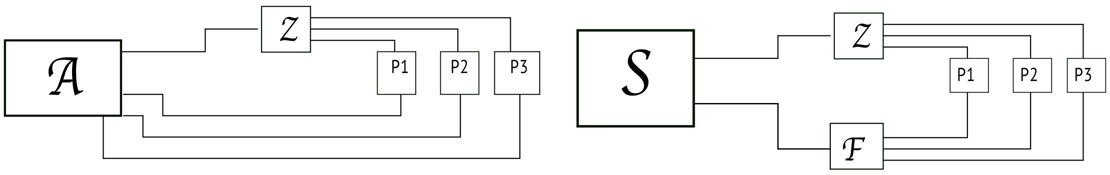


Figure 2.2: The real world and ideal world scenarios, from left to right. Extracted from [12]

A protocol execution in the real world π has an adversary $\mathcal{A}$ instead of a functionality. The way to prove that the protocol is secure is to "fool" the environment $\mathcal{Z}$, so that he cannot distinguish whether he is in the real world or the ideal world.

To do so, one has to build a simulator (see Figure 2.2, right drawing) $\mathcal{S}$ to work on top of the ideal functionality $\mathcal{F}$, such that the environment cannot distinguish whether it is running the protocol π or the simulator $\mathcal{S}$. If we achieve this, we can prove that the protocol is UC-secure.

Composability UC-secure protocols enjoy the composability property. This means that if a protocol that is proven UC-secure uses as resource another UC secure protocol, then the composition of the two protocols results in a UC-secure protocol which implements the functionality of the first protocol with access to the functionality of the second protocol.

2.5 Communication Models

The way we model the network can have dramatic consequences for the practical efficiency of a protocol. As such we make a brief distinction between the two main communication models.

Synchronous By synchronous it is meant that we make assumptions about the time. For instance, we can define t as the time between a message being sent and received. Therefore, we can say that a message is lost if it takes more than t to deliver it.

Asynchronous In an asynchronous model there is no time assumption. That means that the protocol cannot take actions based on what time took a certain event.

2.6 Initial Setup

Every *UC-secure* protocol assumes some kind of initial *trusted setup*. The most common models are the *CRS* (Common Reference String) and the *PKI* (Public Key Infrastructure). CRS assumes that the players have access to a common random string taken for instance from some physical event. A PKI is a infrastructure that is held accountable for binding public keys with the respective players.

Chapter 3

Circuit Evaluation

Before starting any MPC protocol, the parties must agree on what function to compute. The available protocols differ on what type of representation they use for this function.

In general terms, there are two different ways to represent the function. The first set of protocols expresses the function by means of *boolean circuits*, while the second uses *arithmetic circuits*. In this chapter, we describe protocols for boolean circuit evaluation and arithmetic circuit evaluation. Moreover, we discuss the advantages and drawbacks of using one representation over the other.

Boolean Circuits A way to express the wanted functionality, is by defining a boolean function, i.e. a function $f : \{0,1\}^n \rightarrow \{0,1\}^n$. Every boolean function can be converted to a boolean circuit representation. This boolean circuit is a composition of wires and logical gates that entirely captures the behavior of the function. For a better understanding, consider figure 3.1, where we provide an example of a boolean circuit that does the `AND` operation between two 4-bit sized values.

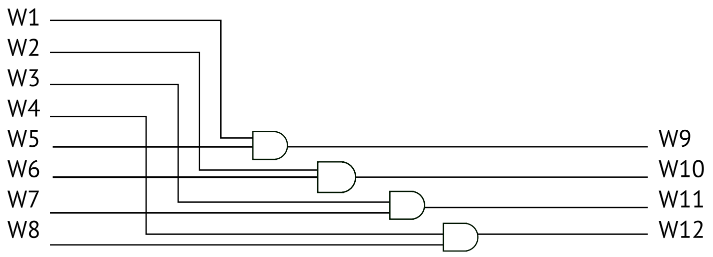


Figure 3.1: Boolean circuit that represents the `AND` operator for 4-bit numbers

The wires $w_1, \ldots, w_8$ are considered input wires. The first value corresponds to the input wires $w_1, \ldots, w_4$, while the second value corresponds to input wires $w_5, \ldots, w_8$. The wires $w_9, \ldots, w_{12}$ are the output wires of the gates they are connected to, and at the same time output wires of the circuit.

Now that we gave an introduction on boolean circuits, we proceed to explain Yao's scheme, known in literature by the name of *Garbled Circuits*.

3.1 Boolean Circuit Evaluation

The first general-purpose MPC protocol was presented by Andrew Yao in 1982. The protocol was designed so that 2 parties could securely evaluate any deterministic function, as long as it was represented in the form of a *boolean circuit*. In this chapter we explain the secure function evaluation protocol by Lindell and Pinkas [18] which is based on Yao's work, and also uses the boolean circuit representation.

Yao's scheme is secure in the passive/semi-honest model. Revisiting section 2.2,we recall that a semi-honest adversary , by definition, is not able to induce malicious behavior on other parties. This means that despite being able to gain access to the private values of the corrupted parties, the adversary is not able to instruct them to follow his orders.

Lindell and Pinkas [18] developed a formal proof of security for the protocol. Without covering the proof, we proceed to describe Yao's protocol, based on the description and notation found in the same article.

3.1.1 Garbled Circuits Overview

Yao's technique was first designed as a 2-party computation protocol, yet other works by Ben-David et al. [2] extended it to a multi-party protocol. For the sake of simplicity we will explain the original 2-party protocol.

In Yao's protocol there are two parties P_1 and P_2 who wish to compute the function $f(x_1, x_2) = (y_1, y_2)$ so that P_i holds the input x_i and receives the output y_i. Function f can be viewed as a boolean function, that has a circuit representation C and where the inputs x_1 and x_2 are also the inputs of the circuit (as bit strings). The circuit C will be the one used in the start of the protocol.

P_1 starts by encrypting (garbling) the circuit C in order to make the input values private. For each wire of the circuit, two random values are chosen. One represents 0 while the other represents 1. Then, for each gate g, with inputs $b_1 \in \{0, 1\}$ and $b_2 \in \{0, 1\}$, the random values corresponding to the input values b_1 and b_2 are used as keys to encrypt the value corresponding to the output wire of g (the result of the computation of $g(b_1, b_2)$). Each gate's computation table is also randomly permuted so that one cannot guess the entries by their order.

Notice that we are hiding the direct result of the gate evaluation. This is not safe because the output of the intermediate gates are revealed after decryption. To ease our explanation, we will not consider this a leak, and will come to it later in our description.

After constructing the garbled circuit, P_1 sends it to P_2, together with his encrypted inputs. P_2 evaluates the circuit. While evaluating, he uses a 1-out-of-2 Oblivious Transfer for each of his input bits to obtain the corresponding encrypted value from P_1. We will later describe in detail the Oblivious Transfer, but for now it is sufficient to say that it is a protocol where P_2 obtains from P_1 the key corresponding to his input bit while P_1 remains oblivious as to which key he sent.

Note that P_2, in no case gets access to P_1's private input bits. During the Oblivious Transfer protocol, P_1 sends the keys used to encrypt the output of the gate, so that P_2 only learns the keys, not the values.

P_2 uses this technique to evaluate all the gates, and finishes the circuit evaluation, obtaining the output of the function. He then sends the resulting output to P_1.

3.1.2 Garbled Circuits in Detail

To explain Yao's protocol we have to understand the first phase of the protocol which is the construction of the garbled circuit. There are four types of wires in the circuit. Circuit-input wires (wires that are input to the circuit), circuit-output wires (output of the circuit), gate-input wires(intermediate input wires) and gate-output wires (intermediate output wires).

Our previous construction "leaked" the intermediate values of the circuit. Now, we define more explicit rules.

We state that no intermediate value should be revealed. In other words, only the final output must be revealed. With this security requirements in mind, now it is not the result of each intermediate gate that is encrypted, but the *key* which leads to the result. The final wires (circuit-output wires) do not need to be hidden. They can simply return the plain values, as it is not necessary to encrypt something that is supposed to be public.

Before the presentation of the secure function evaluation protocol, we describe two important components, *garbled circuits* and *oblivious transfer*.

Garbled Circuit Construction Every deterministic function can be viewed as a boolean circuit, where there are input wires followed by multiple gates that lead to the final output wire(s). Let C be a boolean circuit that receives two inputs $x, y \in \{0,1\}^n$ and outputs $C(x,y) \in \{0,1\}$ (for simplicity, we assume the output length is 1, and leave the security parameter implicit). The circuit C is boolean, so any gate is represented by a function $g : \{0,1\} \times \{0,1\} \to \{0,1\}$. Function g has two input wires , named w_1 and w_2 . Let $\{k_1^0, k_1^1, k_2^0, k_2^1, k_3^0, k_3^1\}$ be six keys of size n generated by a key generation algorithm $G(1^n)$. To garble the gate we use the keys associated with each input. The garbled version of g would be:

$$
\begin{aligned}
c_{0,0} &= E_{k_1^0}(E_{k_2^0}(k_3^{g(0,0)})) \\
c_{0,1} &= E_{k_1^0}(E_{k_2^1}(k_3^{g(0,1)})) \\
c_{1,0} &= E_{k_1^1}(E_{k_2^0}(k_3^{g(1,0)})) \\
c_{1,1} &= E_{k_1^1}(E_{k_2^1}(k_3^{g(1,1)}))
\end{aligned}
$$

where E is a private key encryption scheme (G, E, D) that has undistinguished encryptions under chosen plaintext attacks. Another property of the encryption scheme is that it allows verifiable range, i.e. one can verify if the decryption was successful. In order to mask the identity of each of these values, we make a random permutation of them. The result of this random permutation is stored in c_0, c_1, c_2, c_3. A possible permutation could be

$$
\begin{aligned}
c_0 &= C_{0,1} = E_{k_1^0}(E_{k_2^1}(k_3^{g(0,1)})) \\
c_1 &= C_{1,0} = E_{k_1^1}(E_{k_2^0}(k_3^{g(1,0)})) \\
c_2 &= C_{0,0} = E_{k_1^0}(E_{k_2^0}(k_3^{g(0,0)}) \\
c_3 &= C_{1,1} = E_{k_1^1}(E_{k_2^1}(k_3^{g(1,1)})
\end{aligned}
$$

Decryption Let's assume the input values $\{\alpha, \beta\}$, and the output value γ. With access to the keys k_1^α, k_2^β we are able to decrypt all the entries and verify which one was correct. With this in mind, for every $i \in \{0, \ldots, 3\}$ we compute $D_{k_2^\beta}(D_{k_1^\alpha}(c_i))$. The only successful decryption (the one which does not return a $\perp$ value) will be k_3^γ. This results from the fact that the entry was encrypted with k_1^α, k_2^β.

Entire Circuit Garbling For each wire w_i, we choose two keys $k_i^0, k_i^1 \leftarrow G(1^n)$. These keys must be independent. With these keys, we can now compute the four garbled values of each gate, using the steps described above. Then, we make a random permutation of the values.

At last, the decryption tables of the circuit are computed. These tables have the values $(0, k_i^0), (1, k_i^1)$, where w_i is the gate-output wire. For the last gates (circuit-output gates) one can skip the encryption. This way we can simply obtain the correct output bits directly. In other terms, for the circuit-output wires, $c_{\alpha,\beta} = E_{k_1^\alpha}(E_{k_2^\beta}(g(\alpha, \beta)))$ for every $\alpha, \beta \in \{0, 1\}$.)

Oblivious Transfer Oblivious Transfer is a protocol where there is a sender S, who wishes to send one of multiple pieces of data to a receiver R. The main goal is to do this transfer in such a way that S remains oblivious as to which the pieces were transferred.

The first form of Oblivious Transfer was described by Rabin [23]. Rabin's scheme had some unwanted effects. One of them was that the sender could only send a piece with probability $1/2$. A more practical solution was found by Even et al. [11], called *1 out of 2 oblivious transfer*. The protocol is general, yet a version using RSA by Rivest et al. [24] will be described.

Player P_1 has two messages x, y. P_2 wants to obtain one of them, without P_1 knowing which of them was chosen.

1. P_1 generates an *RSA* key pair (N, e, d) where N is the modulus, e the public exponent and d the private exponent.

2. P_1 picks two random numbers a_0, a_1, and sends them to P_2, along with his public key.

3. P_2 selects $a_b \in \{a_0, a_1\}$ where b is P_2's choice.

4. P_2 generates a random value k and masks a_b by computing $v = (a_b + k^e) \mod N$. Then, sends it to P_1.

5. P_1 Computes the following two possible values for k.

$$k_0 = (v - a_0)^d \mod N$$
$$k_1 = (v - a_1)^d \mod N$$

6. P_1 sets

$$x'_0 = x_0 + k_0$$
$$x'_1 = x_1 + k_1$$

7. P_1 sends x'_0, x'_1 to P_2

8. P_2 computes the correct message $x_b = x'_b - k$

It must be remembered that the original protocol is a generalization. Therefore, cryptosystems other than RSA could be employed.

Correctness The protocol is correct if the final computed value corresponds to the chosen message x_b. Proof:

$$\begin{aligned} x'_b - k &= (x_b + k_b) - k = \\ &= x_b + (v - a_b)^d - k = \\ &= x_b + ((a_b + k) - a_b)^d - k \\ &= x_b + (k^e)^d - k \\ &= x_b + k - k = \\ &= x_b \end{aligned}$$

Security Generally speaking, an OT protocol is considered secure if it obeys the two following security requirements:

1. The receiver only obtains one of the messages.

2. The sender does not learn the receiver's choice

Now we provide a simplified explanation of why the previous protocol respects each of these requirements.

1. **The receiver only obtains one of the messages.** From the correction of the scheme, the receiver is able to successfully recover one message. To learn the other message, he has to guess k_α, where α is the index of the message that was not chosen. But k_α is an RSA encryption using d, which is private to the sender. This means that the receiver has to break RSA to learn the other message.

2. **The sender does not learn the receiver's choice.** One of the messages (k_0, k_1), will be equal to k, the random value generated by the receiver. But the sender does not know which, since v is the sum of x_b with the random number k. Since k can be any number, he cannot guess the receivers choice, so this security property is achieved with information theoretic security.

3.1.3 Yao's Garbled Circuits Protocol

After explaining the protocol at a high level of abstraction, we went into each of it's building blocks. We described how the circuit garbling is performed, and gave an intuition on oblivious transfer. Additionally, we provided an example of a *1-out-of-2* oblivious transfer scheme, based on the RSA construction.

In order to summarise our description of the protocol, we finally enumerate each of the involved steps:

1. P_1 constructs the garbled circuit using the garbling scheme described in the section "Garbled Circuit Construction".

2. P_1 sends the garbled circuit to P_2.

3. Let $w_1, \dots, w_m$ be the circuit-input wires of x and $w_{m+1}, \dots, w_{2m}$ be the circuit-input wires of y. P_1 sends to P_2 the keys $k_1^x, \dots, k_m^x$, where $x \in \{0, 1\}$ is P_1's input bit.

   Then, for every of P_2's input bit i, P_1 and P_2 run a 1-out-of-two oblivious transfer protocol in which P_1 contributes with (k_{m+i}^0, k_{m+i}^1) while P_2 contributes with y_i. At the end of the oblivious transfer P_2 obtains the keys associated with his inputs, while P_1 learns nothing.

4. Now that P_2 has the garbled circuit and $2m$ keys corresponding to the $2m$ input wires, he evaluates the circuit, obtaining $f(x, y)$. Afterwards, he sends $f(x, y)$ to P_1 and both parties output this value.

3.1.4 Execution Example

For a better understanding of the protocol, we will describe how to evaluate a simple circuit that is formed by a single gate. In our case we are going to use an OR gate (see figure 3.2). As this is a one gate circuit, it does not make sense to encrypt intermediate values, since there are no intermediate values at all. So we made the simplification of applying the encryption function directly to the output of the gate.

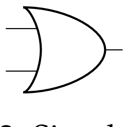


Figure 3.2: Simple OR gate

The protocol starts with two parties, P_1 and P_2. P_1 holds his private input x while P_2 holds his private input y. They want to compute $f(x, y) =$

$B_{00} = 0$
$B_{10} = 1$
$B_{01} = 1$
$B_{11} = 1$

Table 3.1: Truth table value *boxes*

z, where $f : \{0,1\} \times \{0,1\} \to \{0,1\}$ is a boolean function that behaves like an OR logical gate, and x,y are single bit values. The scheme works as follows.

- P_1 picks two random keys for each input wire. An OR gate has 2 input wires, so P_1 picks random keys $\{k_1^0, k_1^1, k_2^0, k_2^1\}$, using the key generation algorithm G.
- P_1 builds 4 *boxes* containing the result of the corresponding truth table value. See table 3.1.
- P_1 builds an encrypted truth table. For each row, he uses the generated keys.

$$E_{k_1^0}(E_{K_2^0(B_{00})})$$
$$E_{k_1^0}(E_{K_2^1(B_{01})})$$
$$E_{k_1^1}(E_{K_2^0(B_{10})})$$
$$E_{k_1^1}(E_{K_2^1(B_{11})})$$

- P_1 makes a random permutation of the encrypted truth tables.

  Which ends up being the unordered table

$$E_{k_1^1}(E_{K_2^1(B_{11})})$$
$$E_{k_1^1}(E_{K_2^0(B_{10})})$$
$$E_{k_1^0}(E_{K_2^0(B_{00})})$$
$$E_{k_1^0}(E_{K_2^1(B_{01})})$$

- P_1 sends the permuted boxes to P_2.
- P_1 sends the key k_1^x corresponding to his input bit $x \in \{0,1\}$. Notice that P_2 learns nothing about x.
- P_1 and P_2 run an Oblivious Transfer protocol, where P_2 learns k_2^y where y is P_2's input bit, while P_1 learns nothing.

- P_2 uses the keys he received to decrypt the row corresponding to the input bits. As an example, if the bits were $x = 0$ and $y = 1$ P_2 would compute $E_{K_1^0}(E_{K_2^1}(B_{01}))$. The outcome is that he learns B_{01}, since this value will be the only successful decryption.

- P_2 sends the result to P_1.

With this we end our example of the secure evaluation of an OR gate. In the next section we deal with how to implement secure evaluation protocols using existing tools.

3.1.5 Implementations

There are some implementations of SFE toolkits based on Yao's protocol. For instance, JustGarble by Bellare et al. [1] is focused on the garbling phase of Yao's protocol. It enables very efficient garbling and evaluation of boolean circuits, but it does not help in constructing the circuits, making it less easy to use from a programmers perspective.

TASTY, by Henecka et al. [16] is a compiler that enables the creation of MPC programs. With TASTY, the programmer can explicitly say which parts of the program should be computed using boolean circuits or arithmetic circuits.

Fairplay by Ben-David et al. [2] is a framework for developing applications with multiparty computation functionality. It enables programmers to easily implement programs to evaluate any type of function, using Yao's Garbled circuits.

FairPlay is a multilevel framework. It has:

- A high level procedural definition language called *SFDL* tailored to the SFE paradigm;

- A compiler of SFDL into a one-pass Boolean circuit presented in a language called *SHDL*

- Bob/Alice programs that evaluate the SHDL circuit using Yao's garbled circuits technique.

Since we are only describing boolean circuit evaluation protocols, we will use FairPlay to demonstrate how MPC protocols can be easily implemented. We chose FairPlay because it goes beyond the garbling phase of the Yao technique. It is a full-stack framework, so it enables a programmer to write programs without using any additional external tool, like a functionality to circuit translator.

Fairplay Application As an example, we built a simple AND program. We specified the program in Fairplay's SFDL as seen on the appendix 6.1 The FairPlay SFDL language has a syntax that resembles common languages like C or Java. The program must be described as a class that implements an Output method. As class variables we must declare the types, in this case we always use a 4-bit size integer. The Output method in our example is self explanatory; We pick Alice's input and *bitwise AND* it with Bob's input. Finally, we send this result to Alice's and Bob's output.

SFDL to SHDL Conversion FairPlay was able to convert our program to a boolean circuit representation. The resulting file is available in appendix 6.2. The lines 0-7 represent both Bob's and Alice's input gates. Lines 8-11 represent the logical AND operator gates, with the output going to alice's output wires. As can be observed, each of these gates has two input wires. For instance, in line 8, the AND gate is supplied with input wires 4 and 0. Wire 0 is one of Alice's and wire 4 is one of Bob's wires. Finally, lines 12-15 represent Bob's output wires. They come directly from Alice's output wires.

Multiplication The previous example had a very simple circuit, which may trick the reader to think that the conversion is always efficient. To provide counter evidence, we implemented a simple 4-bit integer multiplication program, available in Appendix 6.3. Since multiplication is not implemented in FairPlay, we did our own. We have encoded the shift and add multiplication algorithm.

The resulting circuit is found on 6.4. An interesting observation is that the circuit grew in complexity when compared to the AND circuit. For a single 4-bit multiplication we needed a circuit with 92 wires. This problem is inherent to boolean circuits. While comparisons are very efficient, multiplications are a problem, since they make the circuit grow larger, increasing the overall complexity of the protocol execution.

Of course we could have used other procedures for building our circuit. We could have used a hardware description language like Verilog to reduce the number of wires. Nevertheless, these would be only optimisations, and while they might alleviate the problem, they are not able to solve it.

In the following section we will describe the other set of protocols, the ones based on arithmetic circuit evaluation. Furthermore, we will distinguish which protocols might be better suitable for certain types of computations.

3.2 Arithmetic Circuit Evaluation

An Arithmetic Circuit is an abstract structure that takes as inputs either variables or numbers, and is allowed to either add or multiply two expressions it already computed. More formally, an arithmetic circuit is an acyclic graph where each node is a gate, variable or constant in the field. Gates can be either additive (+) or multiplicative (x).

Arithmetic circuits are specially efficient for representing and computing polynomials. But since it is possible to encode boolean operations by representing the bits as the values 0 and 1 in the finite field, this means that all boolean circuits can also be embedded in arithmetic circuits. Consequently, arithmetic circuits are able to represent any functionality.

Arithmetic circuit evaluation protocols are usually based on Secret Sharing. As such, we proceed to explain how Secret Sharing works. Furthermore, we show how Secret Sharing is used as a tool for building MPC protocols that evaluate Arithmetic Circuits.

3.2.1 Secret Sharing

Secret Sharing is a cryptographic scheme that is the building block of many Arithmetic Circuit MPC protocols. A Secret Sharing scheme allows a secret S to be divided in pieces $S_1, \ldots, S_n$ in such a way that:

1. Knowledge of a number of S_i pieces, namely k, makes S easily computable
2. Knowledge of any $k-1$ or fewer S_i pieces makes S completely undetermined.

This definition of Secret Sharing was presented by Shamir [26]. Using the above nomenclature, we consider a (k, n) threshold scheme, where k is the minimum number of pieces needed to reveal the secret, while n is the total number of pieces (i.e. shares).

Shamir Secret Sharing Shamir Secret Sharing , first described by Shamir [26] is a (k, n) secret sharing scheme based on polynomial interpolation.

The way the scheme works is that we generate a polynomial

$$q(x) = a_0 + a_1^x + \ldots a_{k-1}x^{k-1}$$

Where secret value S is the first term (a_0). Valid points of the polynomial (i.e: tuples in the form $(x, q(x))$) can now be used as a shares.

In order to rebuild the secret we use polynomial interpolation. Using an interpolation technique such as the Lagrange interpolation, we are able to recover $q(x)$, and consequently learn $a_0 = S$. Note: Due to the thresholding property of the scheme it would only reveal S if k shares were known.

3.2.2 Secret Sharing as a tool for MPC

Secret Sharing can be used as a building block for Arithmetic MPC protocols. Secret Sharing provides MPC functionality like privacy-preserving and controlled exposure of information.
MPC based on Secret Sharing usually follows a pattern [25]:

- *Input Sharing*: Each party shares its secret
- *Computing*: Parties engage in MPC protocols which make operations on shared vales.
- *Output*: Each party collects the shares from other parties and reconstructs the output.

In the computation step, parties are able to make operations on shared values, for instance using the homomorphic properties of the Secret Sharing Scheme, as explained bellow. Computations can be viewed as multiple tasks. In each task, a minimal protocol transforms shared inputs on shared outputs. This construction is useful so that we can build secure protocols on top of each other.
Secret Sharing plays a big role in the process, as intermediate computations may (or may not) be reconstructed (public vs private values). In order to reconstruct the shares, the set of parties with access to the secret , must be in agreement, so no information is leaked in the process.

Homomorphic properties in Shamir Secret Sharing It is possible to do basic arithmetic computation with shamir shares. As presented in [4], the basic operations are defined as follows:

Addition Assuming shared values $[u] = [u_1, \ldots, u_n]$ and $[v] = [v_1, \ldots, v_n]$, we say that each party may run the protocol given in algorithm 1 to add two shared values.

Share multiplication with a public value Having a shared value $[u]$ and a public value t, the product $[w] = t[n]$ can be obtained by using the algorithm 2

Algorithm 1 Addition between two Shamir shares

```
Require: Shares $u_k$ and $v_k$
Ensure: Share $w_k$, that is the sum of $[u]$ and $[v]$
  Round 1:
  $w_k = u_k + v_k$
```

Algorithm 2 Multiplication between Shamir share and public value

```
Require: Shares $u_k$ and public value $t$
Ensure: Share $w_k$, that represents the value $t[u]$
  Round 1:
  $w_k = tu_k$
```

Share Multiplication Having $[u] = [u_1, \ldots, u_n]$ and $[v] = [v_1, \ldots, v_n]$, one could try to multiply the shares, but it would be of no use, for two reasons. First, the result would raise the threshold, because the resulting polynomial would be of degree $2t$, requiring $2t + 1$ shares for reconstruction. Second, the new polynomial is not random (since it is the product of two polynomials). Reconstructing the result from the shares reveals the product of the polynomial, which is not wanted. Algorithm 3 is based on the fact that the product can be computed as a linear combination of secret values with public coefficients.

Algorithm 3 Multiplication of shares

```
Require: Shares $u_i$ and $v_i$, precomputed value $\beta_i$
Ensure: Share $w_i$, that represents the value of $[u][v]$
  Round 1:
  $z_i = u_i v_i \beta_i$
  Share $z_i$ to $z_{i1}, \ldots, z_{in}$
  Send to every other node $P_l, i \neq l$ the share $z_{il}$
  Round 2:
  Receive shares $z_{ji}, j \neq i$ from other nodes $w_i = z_{ii} + \sum_{j=1, j \neq i}^{n} z_{ij}$
```

Full circuit evaluation Using the aforementioned protocol we are able to evaluate a full arithmetic circuit. Though this construction "works", it has many assumptions. We are abstracting from details like how to share a value across a set of participants in a secure way or how to open the shared values so that no participant cheats (malicious behaviour). Besides, we do not explain how the players generate and agree on the random values used for the multiplication protocol.

Modern protocols try to address those questions, and others, without incurring in overhead. The overhead can either be in round complexity, communication or computational.

In the next chapter we will talk about a family of modern arithmetic protocols, named SPDZ, that address these and other problems.

3.3 Comparison

As there are protocols for evaluating both boolean and arithmetic circuits, we can make comparisons and try to understand which representation is the best.

It turns out that some types of computations are better performed on certain representations over others. For instance, polynomials can be expressed very efficiently in an arithmetic circuit. On top of that, evaluating an arithmetic circuit is just doing basic arithmetic operations in the field like adding and multiplying.

On the other hand there are operations that also happen to become very inefficient with arithmetic circuits. Comparison, equality or interval tests are examples. In contrast, these operations are very efficient in boolean circuits. Take the example of comparison. Comparing two values in a boolean circuit is "cheap", since we can do bit comparison.

Summarising, on the one hand we have the generality of arithmetic circuits, because they allow operations on a finite field, and on the other hand there are these operations like equality test or comparison that are very problematic using arithmetic operations, but that can be done efficiently in a boolean representation.

Bit Decomposition A solution to join the two "worlds" could be by using bit-decomposition protocols such as the one by Damgård et al. [8]. A bit decomposition protocol converts elements in $\mathcal{Z}_p$ into sharing of bits. More specifically, given a shared value $a \in \mathcal{Z}_p$ it returns a shared value $b \in \{0,1\}^l$, where b is a binary representation of the value a, of length l. This is, the protocol handles a share b_i to each player P_i , such that$\sum_{i=0}^{l-1} 2^i b_i = a$, where $b_i \in \{0,1\}$.

Having the values as bitwise sharings makes it easy to do the bit operations and convert back the result to a number in $\mathcal{Z}_p$. The problem is that the known bit-decomposition protocols are expensive.

Another alternative to this problem is to use specific protocols for these operations. Nishide and Ohta [20] proposed protocols for interval test, comparison and equality without using bit-decomposition. All of these protocols are more efficient than using the bit-decomposition alternative suggested by Damgård et al. [8].

Conversions Another interesting aspect is that any boolean circuit can be converted to an Arithmetic circuit and also the other way around. A boolean circuit can be converted to arithmetic by constructing a circuit that has only the possible values $\{1, 0\}$ and then in the end converts the final bit values to a value in the field. The binary gates can be converted to arithmetic operations as follows:

$$\begin{aligned} AND(x, y) &= x * y \\ NOT(x) &= 1 - x \\ OR(x, y) &= 1 - ((1 - x) * (1 - y)) \end{aligned}$$

To convert an Arithmetic circuit in $\mathcal{F}_p$ to boolean, one can simply set all the input values to boolean strings of size $\log_2 p$ and convert the +, * operations to binary adders and multipliers.

A conclusion we can take from these observations is that there is no "best" representation. There are operations that are more efficient on boolean representations and others that are more efficient on arithmetic representations. Nonetheless we may exchange representations/protocols depending on the type of computation that we want to perform.

An alternative is to use mixed solutions. For instance in the case of arithmetic circuits, we might consider bit decomposition/specific protocols for handling the expensive operations such as interval test, equality and comparison.

Summarising, we consider that all these approaches have their benefits and problems, and is difficult to make a fair comparison because of the aforementioned reasons.

Chapter 4

SPDZ

During the development of this thesis, I had a 6 month internship at the Alexandra Institute in Aarhus, Denmark. There, I worked on implementing some parts of the SPDZ protocol by Damgård et al. [10].

In this chapter there is an overview description of the SPDZ protocol followed by a detailed description of some of the sub-protocols that were implemented. The implementation details were left for the next chapter.

4.1 Description

SPDZ, by Damgård et al. [10] is a protocol that enables n parties to securely evaluate any arithmetic circuit.

Unlike the previously referred protocol by Bogdanov, SPDZ is secure in the active setting. Being secure against active adversaries is a strong requirement, that usually forces designers to add complexity to the protocols. SPDZ is no exception, but it handles this additional complexity in a clever way, that we will detail further.

SPDZ uses the preprocessing model, meaning that there are two distinct phases of the protocol - the online and offline phase. The offline phase is a procedure for generating some random data. This random data will be consumed later in the online phase, which is where the actual circuit is evaluated.

A great feature of SPDZ is that it has an extremely fast online phase. In fact, the communication and computational complexity in the online phase are both linear in n, the number of players. This efficient evaluation comes at the cost of shifting heavy operations to the offline phase. The ideas that led to the approach of having a light online phase and heavy offline phase are explained in section 4.1.3.

Like most arithmetic circuit evaluation protocols, SPDZ uses Secret Sharing as a building block. In the next section we explain the Secret Sharing scheme behind SPDZ, so that other components of the protocol can also be understood.

4.1.1 Secret Sharing

The SPDZ protocol uses additive secret sharing instead of the Shamir-based secret sharing described in section 3.2.2.

To additively share a value a across a set of n participants , we simply generate a set $(a_1, \ldots, a_n)$ of random values such that

$$\sum_{i=1}^{n} a_i \pmod p = a$$

An easy way of generating values in this form is to generate $(a_1, \ldots, a_{n-1})$ values and then compute the last number,

$$a_n = a - \sum_{i=1}^{n-1} a_i \pmod p$$

Notice that this scheme is perfectly secure. To recover a one must have every share a_i where $i \in \{1, \ldots, n\}$. An attacker with access to the shares $\{a_1, \ldots, a_{n-1}\}$ still has no clue about a.

Before going into details about how to evaluate the circuit, we give a brief overview of the notation that will be used.

$\langle . \rangle$ Notation We will follow the $\langle . \rangle$ notation that is used in the original paper [10]. A value a is $\langle . \rangle$ shared if every party P_i holds a tuple (a_i, γ_{a_i}), such that a_i is a valid share of a and γ_{a_i} is a valid share of γ_a.

γ_a is the MAC (Message Authentication Code) that validates a's integrity. The MAC algorithm is described further in section 4.1.5.

4.1.2 Online phase

The online phase is where the actual circuit evaluation takes place. In order to explain how it works, we will show the secure evaluation of the circuit that represents the function $f(x, y, z) = (x + y) * z$. Figure 4.1 shows a visual representation of the circuit.

We assume that the values x, y, z have been $\langle . \rangle$ shared. This means that each player P_i has a set of tuples $\{(x_i, \gamma_{x_i}), (y_i, \gamma_{y_i}), (z_i, \gamma_{z_i})\}$, where $\sum_{i=1}^{n} x_i$

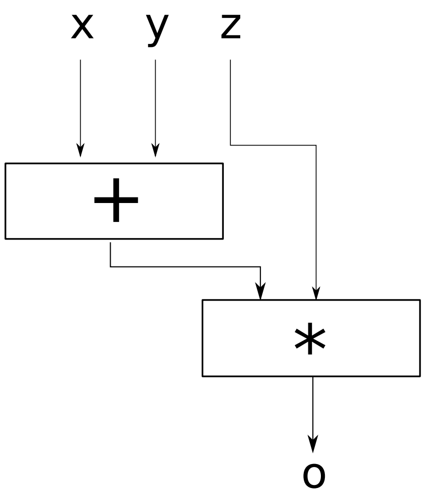


Figure 4.1: Arithmetic Circuit representing $f(x,y,z) = (x+y) * z$

$\pmod p = x, \sum_{i=1}^{n} \gamma_{x_i} \pmod p = \gamma_x, \sum_{i=1}^{n} y_i \pmod p = y, \sum_{i=1}^{n} \gamma_{y_i} \pmod p = \gamma_y, \sum_{i=1}^{n} z_i \pmod p = z, \sum_{i=1}^{n} \gamma_{z_i} \pmod p = \gamma_z$

It can be seen from the topology of the circuit (see figure 4.1) that the first gate to be computed is the addition gate. To evaluate the addition gate, the players simply sum their local shares

$$\langle x \rangle + \langle y \rangle = (x_i + y_i, \gamma_{x_i} + \gamma_{y_i}) = \langle x + y \rangle$$

As can be observed, the addition gates are basically for free. Players perform the addition on their own shares, so no communication is needed. The correctness of this operations comes from the fact that the shares are additively shared. Notice that adding the MACs will also result in a MAC that is still correct .This is a property of the MAC scheme, described in section 4.1.5.

Now, the players proceed the circuit evaluation and encounter a multiplication gate. Multiplication is trickier. Simply multiplying the shares is not a solution as this would yield undefined results. We proceed to explain how the multiplication algorithm works.

We now have to make the assumption that each player has access to a triplet of shared values $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ such that $c = a * b$. This values are assumed to have been generated in the offline phase, in such a way that no player learned a, b, c, and no player behaved maliciously.

We want to process the gate $o = t * z$, where t is the result of the previously evaluated addition gate , i.e. $t = x + y$. To do it, each player computes ϵ and δ, as follows:

$$\epsilon = \langle t \rangle - \langle a \rangle, \delta = \langle z \rangle - \langle b \rangle$$

As can be seen, the players make ϵ and δ public. Since each player has a triplet $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ and public values ϵ, δ, then he is able to compute the multiplication $o = t * z$ as follows:

$$\langle o\rangle = \langle c\rangle + \epsilon * \langle b\rangle + \delta * \langle a\rangle + \epsilon * \delta$$

Proof:

$$\begin{aligned}\langle o\rangle &= \langle c\rangle + \epsilon * \langle b\rangle + \delta * \langle a\rangle + \epsilon * \delta \\ &= \langle c\rangle + (\langle t\rangle - \langle a\rangle) * \langle b\rangle + (\langle z\rangle - \langle b\rangle) * \langle a\rangle + (\langle t\rangle - \langle a\rangle) * (\langle z\rangle - \langle b\rangle) \\ &= \langle c\rangle - \langle b\rangle * \langle a\rangle + \langle t\rangle * \langle z\rangle \\ &= \langle c\rangle - \langle c\rangle + \langle t\rangle * \langle z\rangle \\ &= \langle t\rangle * \langle z\rangle\end{aligned}$$

At this phase, every player has a share of the output o_i, and a MAC value associated with that share γ_{o_i}. Before opening the output value, the players verify if the MAC values are correct. This procedure is called MacCheck, and the full description of it is found in section 4.2.3. If the MACs are correct, then the players can open o, revealing the output of the circuit evaluation.

Randomness Generation What we provided here was a simplified explanation of the online phase of the protocol. Notice that we made many assumptions. One of them was that a triplet $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ had been previously generated, in a secure way. This work is done in the preprocessing phase. In fact, the preprocessing phase has to generate not only one, but a set of triplets, since every multiplication will consume a triplet, making it unusable for further use. We are also not considering details such as how the players agreed on a secret-shared key. We leave this technical details to the preprocessing phase .

Optimization Even though we can already do multiplications using triplets, there are some tricks to make some operations faster. For instance, if we give the players square pairs $\langle a\rangle, \langle b\rangle$ such that $b = a^2$, we can compute squares more efficiently. If every player computes $\epsilon = \langle x\rangle - \langle a\rangle$ then the players can compute $\langle z\rangle = \langle b\rangle + 2 * \epsilon * \langle x\rangle - \epsilon^2$. Proof :

$$\begin{aligned}\langle z\rangle &= \langle b\rangle + 2 * \epsilon * \langle x\rangle - \epsilon^2 \\ &= \langle a^2\rangle + 2 \cdot (\langle x\rangle - \langle a\rangle) * \langle x\rangle - (\langle x\rangle - \langle a\rangle)^2 \\ &= \langle a^2\rangle + 2\langle x^2\rangle - 2\langle a\rangle\langle x\rangle - \langle x^2\rangle + \langle x\rangle\langle a\rangle + \langle a\rangle\langle x\rangle + \langle a^2\rangle \\ &= \langle x^2\rangle\end{aligned}$$

Other optimisations can be done if we store shared bits (more efficient comparisons , for instance). So we end up with the need to generate :

- Triplets $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, such that $c = a * b$
- Square Pairs $(\langle a\rangle, \langle b\rangle)$, such that $b = a^2$
- Shared bits $\langle b\rangle$ such that $\sum_{i=1}^{n} b_i = 0 \pmod 2$ and $b \in \mathcal{F}_2$

The number of triplets, square pairs and shared bits will depend on the number of gates to evaluate.

4.1.3 The Preprocessing Model

In the preprocessing model of MPC there are two distinct phases - the online phase and the offline phase (also named preprocessing phase). The online phase corresponds to the actual secure computation (for instance, the evaluation of an arithmetic circuit). The offline phase is a procedure that is used to generate data that is needed when the protocol goes online.

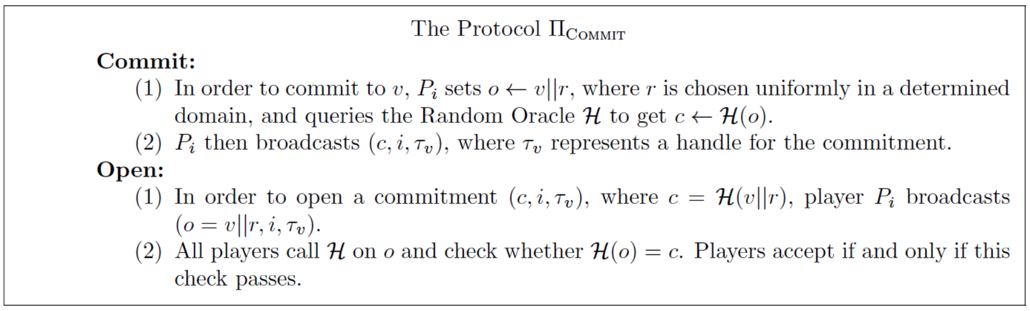
Online and offline are totally separated. The players running the offline phase do not need to know the circuit nor the inputs of the players. The benefit of this model comes from the fact that we can shift heavy operations of the protocol to the preprocessing phase, resulting in less overhead when the protocol goes live. This can be a push towards practicability. If some servers want to securely evaluate multiple circuits on demand, they can spend some time generating the needed data (offline) and then use it anytime they want to perform the secure evaluation (i.e. run the online phase).

Tools The subprotocols we are about to demonstrate assume some cryptographic tools such as a Commitment Scheme and a Mac Algorithm. In the following sections we explain how these tools work and what schemes are used in the concrete case of SPDZ.

4.1.4 Commitment Scheme

A commitment scheme is a cryptographic tool that enables a party to commit to a value and reveal it later. Commitment schemes have two distinct phases:

- Commitment phase where the commiter commits to a certain value.
- Opening phase where the commiter reveals the value.



The Protocol Π_{COMMIT}

Commit:

(1) In order to commit to v, P_i sets $o \leftarrow v||r$, where r is chosen uniformly in a determined domain, and queries the Random Oracle $\mathcal{H}$ to get $c \leftarrow \mathcal{H}(o)$.

(2) P_i then broadcasts (c, i, τ_v), where τ_v represents a handle for the commitment.

Open:

(1) In order to open a commitment (c, i, τ_v), where $c = \mathcal{H}(v||r)$, player P_i broadcasts $(o = v||r, i, \tau_v)$.

(2) All players call $\mathcal{H}$ on o and check whether $\mathcal{H}(o) = c$. Players accept if and only if this check passes.

Figure 4.2: Commitments Protocol[21]

After the commitment phase, the commiter cannot change the committed value. This is known as the *binding* property. Another important aspect is that the other party should not be able to learn the committed value before the opening phase. This is known as the *hiding* property.

Commitment schemes are particularly useful in secure multiparty computation protocols. In the case of the SPDZ protocol there is a sub-protocol for commitments (see figure 4.2). The protocol works as follows:

Lets suppose that a party (from now on we will call it commiter) wants to commit to value v. The commiter picks a random number r and computes $o = v||r$ and $c = H(o)$ where H is a hash function. The hash function should be pre-image resistant - it should be computationally unfeasible to find a value o' such that $H(o') = c$. The commiter then broadcasts c to the other players, finalising the commitment phase.

In the opening phase the commiter broadcasts (v, r) and the players verify whether $H(v||r) = c$. Players continue only if this verification passes. The commitment protocol is used throughout the preprocessing phase, most of the times as part of a *cut-and-choose* procedure.

4.1.5 MAC Algorithm

A MAC (Message Authentication Code) is a piece of information that guarantees the authenticity and integrity of a certain message. If Alice and Bob have a common secret key sk, Alice can send a message m to Bob, and a MAC of the message $\gamma_m = MAC_{sk}(m)$. This enables Bob to later verify the MAC, and see if the message is authentic and unchanged.

The MAC scheme used in SPDZ is very simple. Assuming a key α and a message m, then

$$\gamma_m = MAC_\alpha(m) = \alpha * m \pmod p$$

This MAC scheme enables operations on MAC values. For instance, we can add two MAC values and obtain the MAC of the sum of the values they are authenticating.

$$\begin{aligned}\gamma_x + \gamma_y &= (\alpha * x \pmod p) + (\alpha * y \pmod p) = \\ &= \alpha(x+y) \pmod p = \\ &= \gamma_{x+y}\end{aligned}$$

The same goes with multiplication by a public value t

$$\begin{aligned}\gamma_x * t &= (\alpha * x \pmod p) * (\alpha * t \pmod p) = \\ &= \alpha(x * t) \pmod p = \\ &= \gamma_{x*t}\end{aligned}$$

With both of these operations, it's also possible to multiply two MAC's, by using the same scheme that we used for multiplying shares, i.e. reducing the operations to public values.

4.2 Protocols

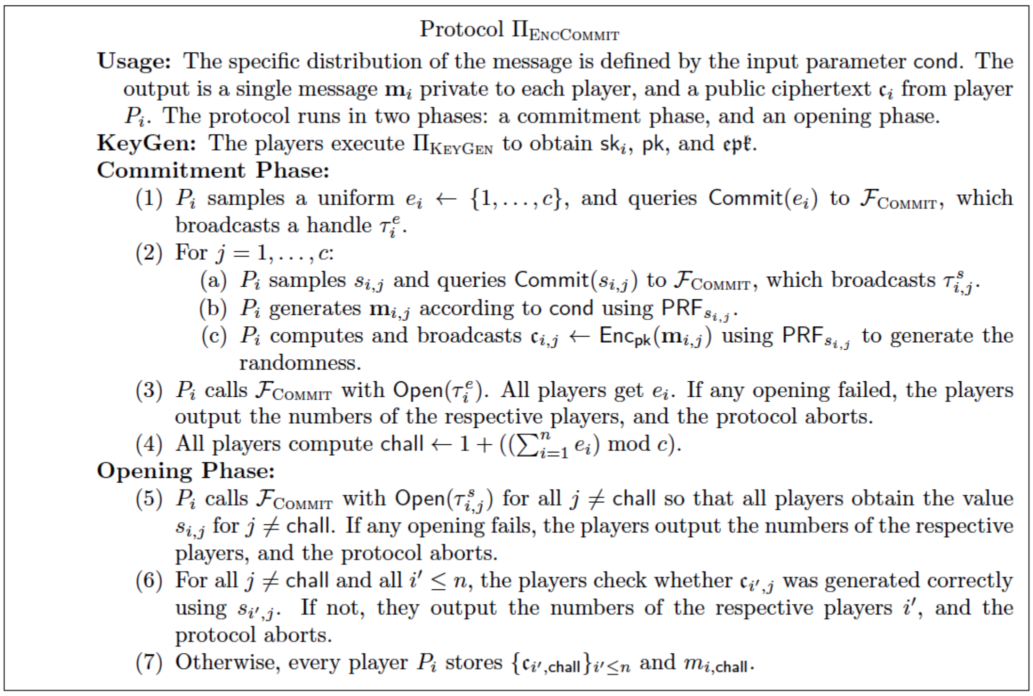
The SPDZ protocol is composed of many subprotocols, where each of them performs a different task. In this section we give a theoretical description of some of the subprotocols that compose the preprocessing phase, more specifically the ones that were implemented during the internship. We also provide some context for a better understanding of what these subprotocols are supposed to do.

4.2.1 EncCommit

At some stage of the preprocessing phase of the SPDZ protocol, each player needs to have a set of public ciphertexts $\{c_1, \ldots, c_n\}$ where each of these $c_i, i \in \{1, \ldots, n\}$ ciphertexts is an encryption of the private message m_i from the player P_i. Summarised, each player must have a public ciphertext (composed by ciphertexts from every player) and a private message.

The goal of the EncCommit protocol is that the players generate these values in a secure way. One must ensure, for instance, that each ciphertext is a genuine encryption of a valid message. A valid message is one that is picked from a specific distribution.

In the EncCommit protocol (see Figure 4.3), if a malicious player deviates from the protocol (i.e. picks the message from another distribution), then the honest players will detect this behaviour and abort the protocol.



Protocol $\Pi_{\text{EncCommit}}$

Usage: The specific distribution of the message is defined by the input parameter cond. The output is a single message $\mathbf{m}_i$ private to each player, and a public ciphertext $\mathfrak{c}_i$ from player P_i. The protocol runs in two phases: a commitment phase, and an opening phase.

KeyGen: The players execute Π_{KeyGen} to obtain sk_i, pk, and $\mathfrak{cpk}$.

Commitment Phase:

(1) P_i samples a uniform $e_i \leftarrow \{1, \ldots, c\}$, and queries $\mathsf{Commit}(e_i)$ to $\mathcal{F}_{\text{Commit}}$, which broadcasts a handle τ_i^e.

(2) For $j = 1, \ldots, c$:

  (a) P_i samples $s_{i,j}$ and queries $\mathsf{Commit}(s_{i,j})$ to $\mathcal{F}_{\text{Commit}}$, which broadcasts $\tau_{i,j}^s$.

  (b) P_i generates $\mathbf{m}_{i,j}$ according to cond using $\mathsf{PRF}_{s_{i,j}}$.

  (c) P_i computes and broadcasts $\mathfrak{c}_{i,j} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}_{i,j})$ using $\mathsf{PRF}_{s_{i,j}}$ to generate the randomness.

(3) P_i calls $\mathcal{F}_{\text{Commit}}$ with $\mathsf{Open}(\tau_i^e)$. All players get e_i. If any opening failed, the players output the numbers of the respective players, and the protocol aborts.

(4) All players compute $\mathsf{chall} \leftarrow 1 + ((\sum_{i=1}^{n} e_i) \bmod c)$.

Opening Phase:

(5) P_i calls $\mathcal{F}_{\text{Commit}}$ with $\mathsf{Open}(\tau_{i,j}^s)$ for all $j \neq \mathsf{chall}$ so that all players obtain the value $s_{i,j}$ for $j \neq \mathsf{chall}$. If any opening fails, the players output the numbers of the respective players, and the protocol aborts.

(6) For all $j \neq \mathsf{chall}$ and all $i' \leq n$, the players check whether $\mathfrak{c}_{i',j}$ was generated correctly using $s_{i',j}$. If not, they output the numbers of the respective players i', and the protocol aborts.

(7) Otherwise, every player P_i stores $\{\mathfrak{c}_{i',\mathsf{chall}}\}_{i' \leq n}$ and $m_{i,\mathsf{chall}}$.

Figure 4.3: EncCommit Protocol[21]

The EncCommit protocol is secure in the covert setting, which means that players controlled by a malicious adversary will succeed with a probability $1/c$ where c is the covert parameter.

Protocol description

(1) All players start by picking a random value e_i from the interval $\{1, \ldots, c\}$. Each player then commits to this value. This commit will be a proof that the value was chosen before the ciphertext generation phase. The commitment scheme is the one described in section 4.1.4.

(2) The players generate multiple ciphertexts that will be later opened (except for one ciphertext, that will remain closed). They proceed as follows: Every player generates c seeds, and commits to them. Using those seeds, the player does some deterministic steps to obtain the messages and ciphertexts. Finally, he broadcasts those ciphertexts to every other player. The intention of the deterministic steps b) and c) is to provide an easy way to verify that the messages and ciphertexts

were correctly generated. Notice that once given the seed, any party can reproduce the operations thus obtaining the ciphertexts.

(3) All the players open the previously committed e_i. If any opening fails, they abort.

(4) All the players compute the value $chall$, that is now a public value. $chall$ is a value in the range $\{1, \dots, c\}$

(5) Now that the players agree on this public value every player will open every seed, except for the one with the index corresponding to $chall$. If any opening fails the players abort the protocol.

(6) Each player will use the received seeds (except for the one with index $chall$) and compute the respective ciphertexts using the deterministic steps b) and c). If any of the recovered ciphertexts does not match with the previously received ciphertexts, the player aborts.

(7) All the players store their private message m_i and the unopened ciphertexts from the other players.

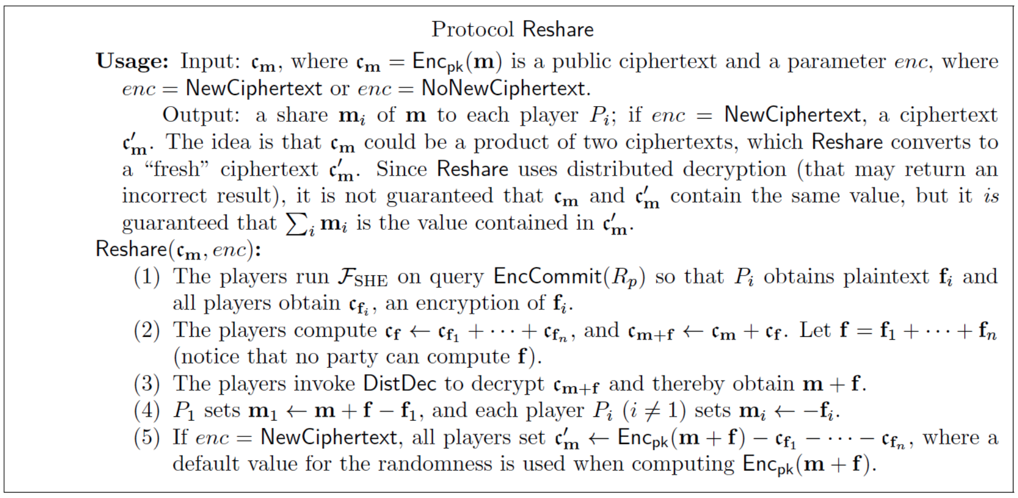
Security The EncCommit protocol uses a *cut-and-choose* technique. It works as follows: Each player commits to c seeds. Then, using those seeds, multiple ciphertexts are generated and broadcasted. Later, each player opens exactly $c - 1$ seeds. Every player can now use these seeds to recover the ciphertexts and verify if they match with the previously received ciphertexts.

The covert security feature comes from this cut-and-choose phase. If a player generates a bad ciphertext, he will get caught with probability $1/c$. If the player generates a bad ciphertext and is not caught, then he was lucky in guessing in which ciphertext should he cheat.

The commitments of the e_i value are important as well. When the players commit to e_i they become locked with that value. Every player sends/receives a handle that gives no information about the underlying value. Since they don't know others e_i values, they simply cannot learn $chall$ until the values are opened. Only after the ciphertexts are generated, the players open e_i and learn $chall$. EncCommit is UC-secure, and a proof is available in [21].

4.2.2 Reshare

In Reshare (see figure 4.4) the players start with an encryption $c_m = Enc_{pk}(m)$. The output of the protocol is a share (m_i), for every player



Protocol Reshare

Usage: Input: $\mathfrak{c}_{\mathbf{m}}$, where $\mathfrak{c}_{\mathbf{m}} = \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})$ is a public ciphertext and a parameter enc, where $enc = \mathsf{NewCiphertext}$ or $enc = \mathsf{NoNewCiphertext}$.

Output: a share $\mathbf{m}_i$ of $\mathbf{m}$ to each player P_i; if $enc = \mathsf{NewCiphertext}$, a ciphertext $\mathfrak{c}'_{\mathbf{m}}$. The idea is that $\mathfrak{c}_{\mathbf{m}}$ could be a product of two ciphertexts, which Reshare converts to a "fresh" ciphertext $\mathfrak{c}'_{\mathbf{m}}$. Since Reshare uses distributed decryption (that may return an incorrect result), it is not guaranteed that $\mathfrak{c}_{\mathbf{m}}$ and $\mathfrak{c}'_{\mathbf{m}}$ contain the same value, but it *is* guaranteed that $\sum_i \mathbf{m}_i$ is the value contained in $\mathfrak{c}'_{\mathbf{m}}$.

$\mathsf{Reshare}(\mathfrak{c}_{\mathbf{m}}, enc)$:

(1) The players run $\mathcal{F}_{\mathrm{SHE}}$ on query $\mathsf{EncCommit}(R_p)$ so that P_i obtains plaintext $\mathbf{f}_i$ and all players obtain $\mathfrak{c}_{\mathbf{f}_i}$, an encryption of $\mathbf{f}_i$.

(2) The players compute $\mathfrak{c}_{\mathbf{f}} \leftarrow \mathfrak{c}_{\mathbf{f}_1} + \cdots + \mathfrak{c}_{\mathbf{f}_n}$, and $\mathfrak{c}_{\mathbf{m}+\mathbf{f}} \leftarrow \mathfrak{c}_{\mathbf{m}} + \mathfrak{c}_{\mathbf{f}}$. Let $\mathbf{f} = \mathbf{f}_1 + \cdots + \mathbf{f}_n$ (notice that no party can compute $\mathbf{f}$).

(3) The players invoke DistDec to decrypt $\mathfrak{c}_{\mathbf{m}+\mathbf{f}}$ and thereby obtain $\mathbf{m} + \mathbf{f}$.

(4) P_1 sets $\mathbf{m}_1 \leftarrow \mathbf{m} + \mathbf{f} - \mathbf{f}_1$, and each player P_i $(i \neq 1)$ sets $\mathbf{m}_i \leftarrow -\mathbf{f}_i$.

(5) If $enc = \mathsf{NewCiphertext}$, all players set $\mathfrak{c}'_{\mathbf{m}} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m} + \mathbf{f}) - \mathfrak{c}_{\mathbf{f}_1} - \cdots - \mathfrak{c}_{\mathbf{f}_n}$, where a default value for the randomness is used when computing $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m} + \mathbf{f})$.

Figure 4.4: Reshare Protocol[21]

P_i, and possibly a new ciphertext. This is done in such a way that no player learns m.

Description

(1) All players run EncCommit, so that each of them obtains f_i and a public set of encryptions $c_{f_1}, \ldots, c_{f_n}$.

(2) The players obtain the sum of the encryptions , c_f, and then compute $c_{m+f} = c_m + c_f$.

(3) The players run the protocol DistDec (distributed decryption) on the value c_{m+f} and obtain $m + f$

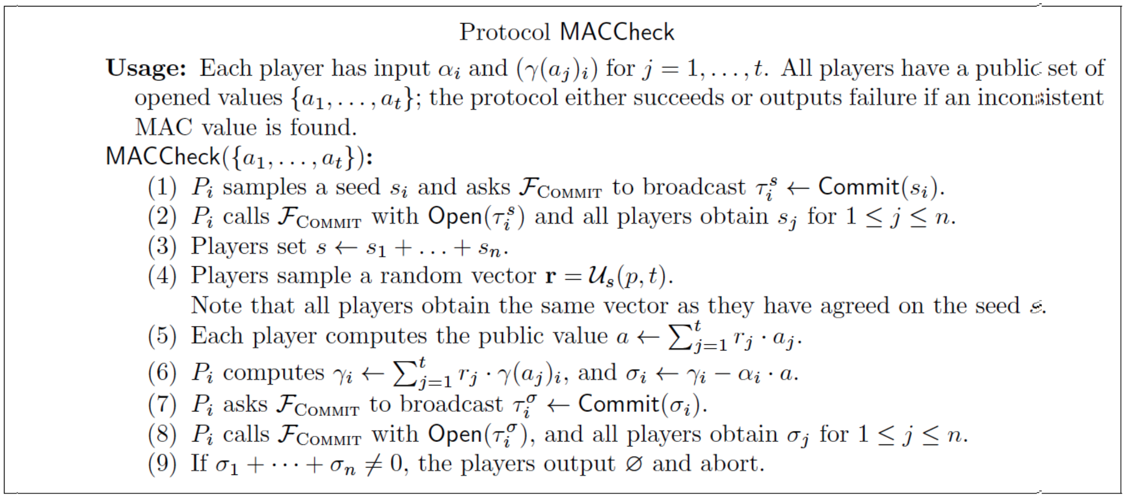
(4) Each player P_i sets $m_i = -f_i$, except for P1, that sets $m_i = m + f - f_1$

(5) If $enc = NewCiphertext$, the players compute the new ciphertext $c'_m = Enc_{pk}(m + f) - c_{f_1} - \cdots - c_{f_n}$

We will not describe the DistDec protocol, but rather use it a black box. DistDec is a protocol that enables the players to jointly decrypt a value, without revealing the secret-shared decryption key.

4.2.3 MacCheck

MACCheck is a procedure that is called both in the online phase and offline phase. This procedure enables the players to verify the correctness of the



Protocol MACCheck

Usage: Each player has input α_i and $(\gamma(a_j)_i)$ for $j = 1, \ldots, t$. All players have a public set of opened values $\{a_1, \ldots, a_t\}$; the protocol either succeeds or outputs failure if an inconsistent MAC value is found.

MACCheck($\{a_1, \ldots, a_t\}$):

(1) P_i samples a seed s_i and asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^s \leftarrow \mathsf{Commit}(s_i)$.

(2) P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^s)$ and all players obtain s_j for $1 \leq j \leq n$.

(3) Players set $s \leftarrow s_1 + \ldots + s_n$.

(4) Players sample a random vector $\mathbf{r} = \mathcal{U}_s(p, t)$.
Note that all players obtain the same vector as they have agreed on the seed s.

(5) Each player computes the public value $a \leftarrow \sum_{j=1}^{t} r_j \cdot a_j$.

(6) P_i computes $\gamma_i \leftarrow \sum_{j=1}^{t} r_j \cdot \gamma(a_j)_i$, and $\sigma_i \leftarrow \gamma_i - \alpha_i \cdot a$.

(7) P_i asks $\mathcal{F}_{\text{COMMIT}}$ to broadcast $\tau_i^\sigma \leftarrow \mathsf{Commit}(\sigma_i)$.

(8) P_i calls $\mathcal{F}_{\text{COMMIT}}$ with $\mathsf{Open}(\tau_i^\sigma)$, and all players obtain σ_j for $1 \leq j \leq n$.

(9) If $\sigma_1 + \cdots + \sigma_n \neq 0$, the players output $\varnothing$ and abort.

Figure 4.5: MACCheck Protocol[21]

MAC's of a public set of values $\{a_1, \ldots, a_t\}$.

Having a triple (α, a, γ_a) , where α is the key, a is the value to be checked and γ is the MAC, we need to check wether

$$MAC(\alpha, a) = \alpha.a \mod p = \gamma_a$$

MacCheck Each of the players has a share α_i of the key α. At some phase they want to run a protocol to verify if the already computed values are correct. A solution could be to open α and verify every single MAC value, but this would disclose α, making it unusable in further operations. The purpose of the MacCheck protocol is for this verification to take place without opening α. It works as follows:

(1) Each player starts with the public set of values to be checked $\{a_1, \ldots, a_t\}$, a set containing his shares of the MAC of each of the $a's$ $\{\gamma(a_j)_i, \ldots, \gamma(a_j)_i\}$ and a share of the key a_i, for $1 \leq i \leq n, 1 \leq j \leq t$.

The players generate a random value s_i and commit to that value.

(2) The players open the commitment, revealing s_i.

(3) Each player obtains $s = s_1 + \cdots + s_n$.

(4) The value s is then used as a seed to obtain a vector of uniformly generated values r. Because every player uses the same seed, value r is public.

(5) All players compute the public value a.

(6) Each player computes γ_i, σ_i

(7) Each player commits σ

(8) Each player opens σ.

(9) Players sum all σ values. The sum should be equal to 0. Otherwise, the players assume that some MAC value was forged.

Correctness For $1 \leq i \leq n$, σ_i can be computed as follows:

$$\begin{aligned}
\sigma_i &= \gamma_i - \alpha_i.a = \\
&= (\sum_{j=1}^{t} r_j.\gamma(a_j)_i) - \alpha_i(\sum_{j=1}^{t} r_j.a_j) = \\
&= \sum_{j=1}^{t} r_j.\gamma(a_j)_i - \alpha_i \sum_{j=1}^{t} r_j.a_j = \\
&= \sum_{j=1}^{t} r_j.(\alpha_i.a_j) - \alpha_i \sum_{j=1}^{t} r_j.a_j = \\
&= \sum_{j=1}^{t} \alpha_i.r_j.a_j - \alpha_i \sum_{j=1}^{t} r_j.a_j = \\
&= \alpha_i \sum_{j=1}^{t} r_j.a_j - \alpha_i \sum_{j=1}^{t} r_j.a_j = \\
&= 0
\end{aligned}$$

If every MAC is correct, then

$$\begin{aligned}
\sum_{i=1}^{n} \sigma_i &= \sigma_1 + \cdots + \sigma_n = \\
&= 0 + 0 + \cdots = \\
&= 0
\end{aligned}$$

These linear combinations enable us to check the MAC values of the set $\{a_0, \ldots, a_t\}$ without opening α.

Chapter 5

Implementations

We implemented some of the protocols that compose the preprocessing phase of the SPDZ protocol. Many of the details of the implementation will not be described due to legal concerns. Instead, we will show how the code was organised and how some of the different components interacted with each other. Moreover, we describe how we implemented constructions like the pseudorandom generator and the commitment scheme.

5.1 Class Hierarchy

The classes were organised so that the basic structure and API remained the same across different protocols (see figure 5.1). A `Protocol` is an object that is responsible for acting as an execution of a single party in the protocol. When we mention protocol we are referring to one of the subprotocols (MacChec,EncCommit or Reshare). The members of this class are objects or pointers to objects that store sent/received values and parameters of the protocol execution. The `Protocol` class is not entirely responsible for all the business logic of the protocol. Indeed, most of the state is held in another abstraction - the *protocol player*.

The protocol player is an object that is member of the `Protocol` and has all the values that matter to the specific protocol. For instance, an EncCommit protocol class has an EncCommitPlayer object inside (more specifically, a pointer to an object) that keeps track of the values and most of the business logic. A `Player` is modelled as a states machine. It's internal state changes as the protocol runs. The API of the player object differs from protocol to protocol, due to the nature of the values that need to be stored and methods that need to called. A player in the EncCommit protocol will have a different API than a player in the MacCheck protocol.

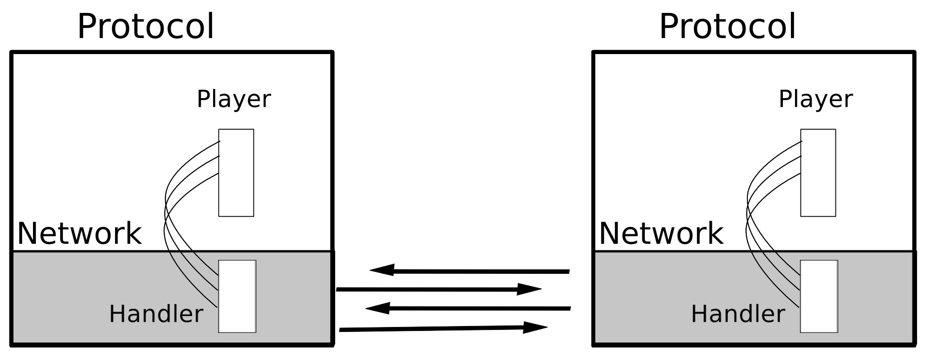


Figure 5.1: Interaction between two Protocol instances

This granularity is needed because each protocol has different behaviour, so having a distinct "player" abstraction per protocol is more useful than having a generic player bloated with methods and member variables.

5.2 Network

Our communication layer provides simple methods to connect/disconnect parties and send messages between them. We decided to use asynchronous I/O. Asynchronous communication requires the programmer to write handlers that deal with certain events. In the case of our protocols, we implemented Handler classes for this purpose (see figure 5.1) .

A Handler receives a message as input and decides on what should be done based on the message information. The handler class is designed to interact with the specific protocol that is being executed. This means that for every protocol we need to implement the respective Handler class.

A problem with this design is that the Network class must hold different Handler objects, depending on which protocol is running. Having a NetworkClass specifically designed for every protocol would increase the complexity and duplicate much of the code. The solution to this is to pass the type of handler as a template to the `Network` Class. This way, the compiler builds different versions of `Network`, while the code base remains the same (only one Network class).

The template hierarchy is shown in figure 5.2. As can be observed, it is possible to build "custom" network classes. As an example, we could instantiate a `Network<Reshare>`, and this class could be used as a member of `ReshareProtocol`.

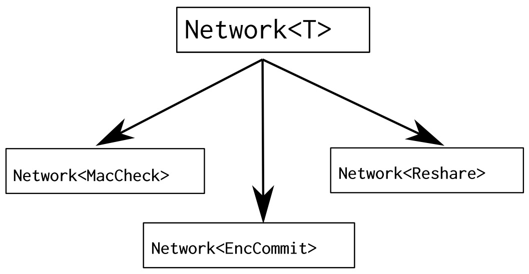


Figure 5.2: Use of Templates in Network Class

5.3 Protocol Classes

Each protocol class has a method `run()` that triggers the protocol execution. It basically loads the players hosts from a file, connects to those hosts and listens for incoming connections. Once he successfully connects to all the players, the protocol execution proceeds. The following paragraphs describe some of the functionality of the Protocol classes.

Protocol objects act as a middle tier between the network and the player state object. Protocols send/receive messages to/from the Network object. Also, they decide what to do based on the received messages. They can for instance abort the protocol or "feed" the player state object with data. Moreover, they can query the player state for important information, like asking whether some commitment opens to the right value.

Message Handling Every time a message is received, it is parsed and analysed. All the messages have a message identification tag. These messages tags are dependent on the protocol being executed, so we use templates again to define different `Message` classes. So a `EncCommitProtocol` class deals with `Message<EncCommit>` types of messages, and the same applies to the other protocol classes.

A message can be simply a control message without content like `ResendShare` or a tag like `HereGoesCommitment` accompanied by some raw data.

Verification Points In many instances we need the players to stop until everyone agrees on something. For instance, when parties need to verify if some commitments are correct before proceeding, they have to wait for everyone's confirmation. These verification points are solved by having methods that block for a certain time and then check if the number of received confirmations is correct.

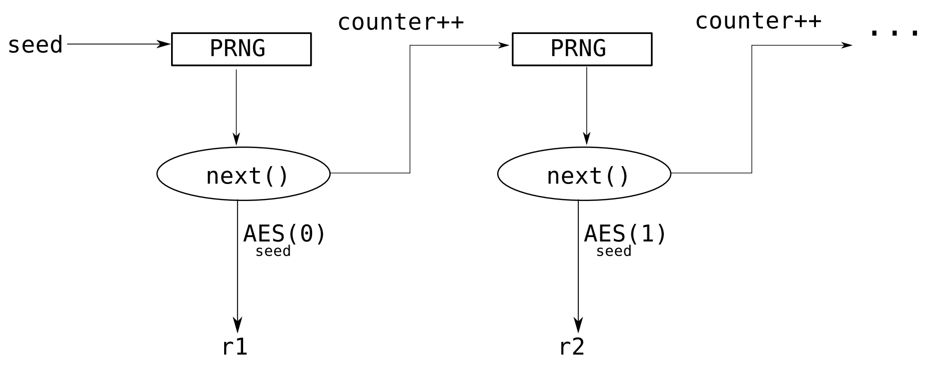


Figure 5.3: PRNG used in SPDZ

We proceed to explain how we implemented our constructions like the random number generator and the commitment scheme.

5.4 AES-PRNG

Many of the protocols make use of a pseudo random number generator. The authors in [10] suggest to use a RNG based on the AES encryption scheme. As such, we built our PRNG by using the AES encryption function. This resulted in a very efficient implementation, since modern CPU's support the AES-NI [15] instruction set, that has specific instructions for AES encryption/decryption.

As any other PRNG, our receives a seed as input, and based on that seed, generates pseudo random values. The seed is supposed to come from a source of "randomness". In our implementation we use `/dev/random/`. Once initialised, the PRNG can retrieve pseudo-random numbers with the `next()` method. Multiple calls to `next()` will retrieve different values.

We proceed to explain how the PRNG works internally. When initialised, it sets the AES encryption function with the seed as key, and sets an internal variable `counter` to zero. So, we obtain a seed the size of an aes-128 key, 16 bytes. The admitted seed values are basically all the values within the key space of AES-128, i.e. $\{0, \ldots, 2^{128} - 1\}$.

Each time a `next()` method is invoked, an encryption of the counter is retrieved, and the counter is incremented. The PRNG design can be seen in Figure 5.3. This design enables the PRNG to be very efficient, since every `next()` invocation incurs only in two costs: incrementing a value and encrypting it.

Most modern cpu's have embedded instructions for AES encryption/decryption. This makes our PRNG construction very efficient.

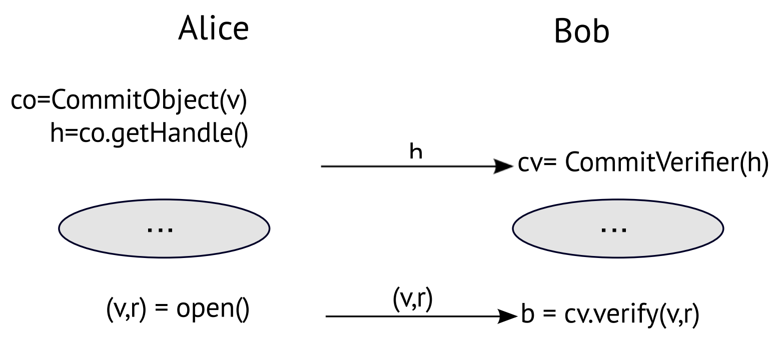


Figure 5.4: Interaction between CommitObject and CommitVerifier Classes, where Alice is the prover and Bob the verifier.

5.5 Commitment Classes

The commitment functionality of our implementation is based on two classes, the CommitObject and CommitVerifier. We provide a graphical representation of the interaction between prover and verifier players in Figure 5.4.

A CommitObject is simply an abstraction that captures the functionality of the prover. The constructor accepts the value to be committed as parameter. After the declaration we can query the object for the handle (in this context handle means the commitment value). This handle can then be sent to any of the other players. When the player is ready to open his value, he calls the method `open()`, that returns a pair `(v, r)`. This pair is then sent to the other players for them to open the commitment.

On the side of the verifier, there is a CommitVerifier object. It is initialised with the *handle (h)* that the prover sent him. Later, the player can query the object with the method `open(v, r)`. This method verifies if the handle was associated with v or not, and returns the result as a boolean value.

5.6 Details

The code was programmed using c++11 and the clang compiler. We chose GMP as our multi precision library. We also made intense use of the Boost library, specially for the network-related classes.

GMP C++ Interface GMP provides a interface that is more friendly for C++ programming. Instead of using `mpz_t` types, we chose to use `mpz_class`. The latter allows operator overloading, which means that instructions such as

```
mpz_add (c,a, b);
mpz_set (x,10);
mpz_mul (x,x,c);
```

are written instead in the form

```
c = a+b;
x = 10;
x = x*c;
```

The overloaded operators make the code easier to understand, specially when implementing complex algorithms.

Serialization Our network classes only work at the byte level, i.e. they only process bytes. There was a frequent need to send values represented as GMP's `mpz_class` from a player to another.

GMP provides I/O methods that convert these objects to a raw byte representation that can be later recognised and parsed. But despite working, it made things harder when we wanted to send a batch of values.

We needed a more abstract structure that would efficiently translate a memory represented object to a sequence of bytes that could be sent over the network and rebuilt on the other end.

The boost libraries provide a serialization framework. We used it in binary mode, which is less portable but more efficient. Another problem is that Boost doesn't support GMP's `mpz_t` types serialization. We solved that problem by extending Boosts serialisation support to GMP types.

In the end, serialising an object was as easy as

```
// v is a vector<mpz_class>  that we want to serialise
serialized = v.serialize();

//convert back to original representation
//in a realistic scenario this is performed on the other end
vector<mpz_class> deserialized;
deserialized = deserialize<vector<mpz_class>>(serialized);
```

This simplification is possible because Boost "knows" how to serialize a vector and how to serialize an object of the type `mpz_class`. The serialisation of containers such as `vector<>` is already implemented in Boost.

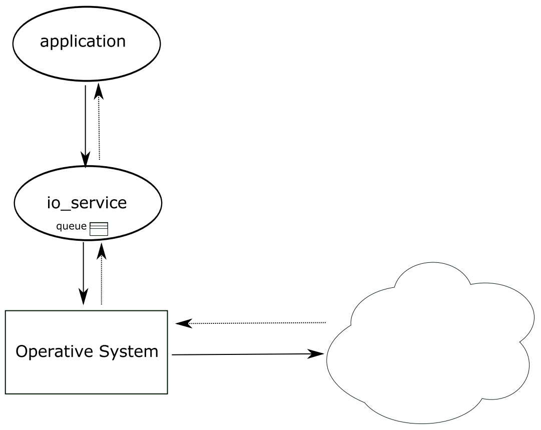


Figure 5.5: Boost ASIO framework

C++11 We made use of C++11 features such as `unique_ptr` and `shared_ptr`. These are classes that encapsulate *raw* pointers and are capable of calling the `free` function when the object runs out of scope. In the case of `shared_ptr` there is a counter that is incremented every time the object is called. Because of this overhead we chose to use `shared_ptr` with caution.

Cryptographic Operations For symmetric encryption and hashing we used CryptoPP. It supports the AES-NI set of instructions, which makes the AES encryption/decryption faster. This aspect is very useful because we make many AES calls in our PRNG. A test showed performance gains by a factor of 4 when using the special instructions.

Boost ASIO We used boost's ASIO library to handle the asynchronous I/O network operations. It works by having a `io_service` routine that from time to time checks if there are incoming messages in the queue. For instance, if a player sends a message to another player, the message is added to a queue, and I/O continues. Later, the `io_service` picks the message from the queue and sends it to the Operative System (see figure 5.5).

When the reply comes, the operative system puts the message in the queue. Once again, `io_service` will pick the message at some point, and finally dispatch it to the application.

By using asynchronous calls we can make better use of the resources, since there is no blocking. The downside is that it grows the programming complexity, since we have to design protocols in a less intuitive way.

Chapter 6

Conclusion and Results

The very first secure function evaluation (SFE) protocol was proposed by Yao, who demonstrated how two parties could evaluate a function represented as a boolean circuit. This result was of tremendous importance as it would later set the race for constructing more general, efficient *SFE* protocols.

The research in the field continued at a steady pace, but there has been an increase in the number of publications in the recent years. According to Google Scholar, the number of articles on multiparty computation until 1999 was approximately 130. The same value increases to approximately 5200 if we add all the articles until 2014.

This interest can be justified by the fact that MPC can be potentially used in a wide array of sectors. Some of the possible applications are electronic voting, anonymous auctioning, private information retrieval or computation outsourcing. Another important aspect is that the efficiency of current protocols is narrowing the gap between theory and practice. For instance, the results by Damgard et al. [9](2012) show that using the SPDZ protocol it takes 0.24s to encrypt an AES block in the active setting, whereas an implementation based on Yao by Pinkas et al. [22] (2009) in the same setting, took 19 minutes.

For the aforementioned reasons, we think MPC will become more practical within the next years, and as such it is an important object of study.

6.0.1 Multiparty Computation

In the first two chapters we gave an introduction on the topic of multiparty computation. We described the motivation for such studies, by giving concrete application scenarios where MPC techniques could be employed.

Later, we went into the problem of defining security in MPC. The community has put a lot of effort into modelling the security properties that should be sufficient for a multiparty computation protocol to be considered secure. In general terms, there are two major requirements that are commonly agreed as being the standard requirements, and those are *privacy* and *correctness*. We introduced those two requirements as our building block for subsequent reasoning about security in MPC.

Proving that a certain protocol is secure in these terms without any further assumptions can be very hard. For instance, one has to define very precisely what privacy and correctness actually mean. The UC framework by Canetti et al. [6] is a rich reference model for proving security, that mitigates this problem. It builds on the idea of defining an ideal world protocol and then proving that any instantiation of the concrete protocol in the real world will leak exactly the same amount of information as the ideal-world version. We describe the UC framework in section 2.4, and give an intuition on how to build proofs using this framework.

It is important to categorise protocols into different settings because of two reasons. First, because different settings may lead to better (or worse) solutions. As an example, the way we model the adversary can be of great importance, since it normally has direct influence on the overall efficiency and security of the scheme. Second, because it makes it easier to compare protocols in a fair way. For these reasons, we wrote sections on the different settings found in literature.

We listed the the different adversarial models (active, passive and covert) and described how they are modelled. Moreover, we made the distinction between cryptographic security and information-theoretic security.

Finally, we finished the chapter by discussing the different models of communication (synchronous, asynchronous) and initial setup assumptions (CRS, PKI).

6.0.2 Boolean and Arithmetic Circuits

The first form of secure multiparty computation protocol was presented by Andrew Yao [27]. The scheme is formerly known as "Garbled Circuits" and allows the evaluation of any deterministic function represented as a boolean circuit. In section 3.1.1 we describe Yao's scheme. We explain the protocol in general terms and then elaborate on a full and detailed description. Moreover, we explain what a boolean circuit is and show how to evaluate a simple one-gate circuit using Yao's Garbled circuits.

On the practical side, we used the FairPlay framework by Ben-David et al. [2] to demonstrate practical uses of Garbled circuits. We gave an intuition on the complexity issues that result from certain operations in boolean circuit representations. We did so by implementing a simple multiplier and extracting information from the circuit generated by FairPlay.

Since Yao many authors came with different protocols that differ in many details. For instance, there are protocols that allow more than two parties and there are protocols that work over arithmetic circuit representations.

We went into describing arithmetic circuit evaluation, and showed a simple protocol by Bogdanov [4] that shows how multiple parties can evaluate a full arithmetic circuit. As many other arithmetic circuit evaluation protocols, it is based on Secret sharing. We discussed Secret Sharing, and described how a concrete scheme, by Shamir [26] is defined.

Finally, we compared the two approaches (boolean, arithmetic) and discussed the benefits and problems that come from using one representation over the other.

6.0.3 The SPDZ Protocol

Recent works showed protocols that are closer to being practical. Damgård et al. [10] proposed a very efficient arithmetic circuit evaluation protocol in the so-called *preprocessing* model. The protocol, named *SPDZ*, has a computational heavy offline phase that can be run at any time, and a very fast online phase where the actual secure evaluation is performed.

The main author of the thesis had an internship at the Alexandra Institute, Denmark, where he worked on implementing some parts of the SPDZ preprocessing phase. This internship led to the writing of two chapters on the SPDZ protocol. The first chapter is mostly theoretical, and replicates the contents of the original article in a more compact and easy format, whilst the second is about the implementation that was performed while working as an intern at the Alexandra Institute.

The SPDZ protocol is composed of two phases, the preprocessing and online phase. In section 4.1 we described the protocol in general terms. We started by explaining the online phase, by assuming some random values were already generated . This randomness generation is basically what the preprocessing phase does, so we skipped it. We also gave an example of how the players proceed to evaluate a simple circuit composed of a sum and multiplication gate.

Later, we described the functionality of some of the subprotocols that compose the preprocessing phase. More concretely, we explained the subprotocols that were implemented (EncCommit, Reshare, MacCheck).

The second chapter is focused on describing how the subprotocols were implemented. It shows how we converted protocol functionalities to concrete classes. All the classes were implemented with code reuse in mind. Consequently, every protocol can be described by the same class structure, despite the inner behaviour being different. We explained the basic protocol class hierarchy and how the different components interact with each other. Additionally, we made a description of how the network layer is modelled.

Some of the protocols dependent on cryptographic constructions like commitments and pseudo random number generators. As such, we explain how we implemented our commitment scheme and PRNG. Finally, we made reference to the libraries that were used during the construction of the protocols.

6.1 Future Works

Despite the growing number of MPC protocols available, there is no large scale adoption taking place. This aspect leads us to conjecture that current MPC protocols may not be practical enough and that the designers may be missing some important properties.

One of the problems in this domain is to understand what protocols are more efficient, because there are many variables involved. Protocols can be either two-party or multi-party. They also differ in terms of adversarial behaviour, underlying hardness assumption, communication, round complexity and many other variables. All these variables make it difficult to judge which protocol is the best. Another problem is that authors claim their protocols to be the most efficient, but it also happens that those protocols may be efficient in models that probably do not express the requirements of the industry.

A positive contribution would be to study the available MPC protocols and understand what characteristics are relevant from a practical point of view. The study would consist of implementing MPC protocols and comparing them under different parameters so that useful information could be extracted. The result could be in the form of a more systemised way of comparing the available protocols, that catches the requirements of practical application scenarios.

This new knowledge could be potentially useful to protocol designers as it would provide them some insights on what aspects should be improved.

Bibliography

1: M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. Cryptology ePrint Archive, Report 2013/426, 2013.

2: A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM Conference on Computer and Communications Security*, pages 257–266. ACM, 2008. ISBN 978-1-59593-810-7.

3: M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

4: D. Bogdanov. Foundations and properties of shamir secret sharing scheme research seminar in cryptography. 2007.

5: P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In *Financial Cryptography*, pages 325–343, 2009.

6: R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. *IACR Cryptology ePrint Archive*, 2002:140, 2002.

7: D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19, 1988. doi: 10.1145/62212.62214. URL `http://doi.acm.org/10.1145/62212.62214`.

8: I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 285–304, 2006.

9: I. Damgard, M. Keller, E. Larraia, C. Miles, and N. Smart. Implementing aes via an actively/covertly secure dishonest-majority mpc protocol, 2012.

10: 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 *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pages 1–18, 2013.

11: S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

12: M. Geisler. Cryptographic protocols - theory and implementation, 2010.

13: O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

14: O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM. ISBN 0-89791-221-7. doi: 10.1145/28395.28420. URL `http://doi.acm.org/10.1145/28395.28420`.

15: S. Gueron. Intel advanced encryption standard(aes) new instructions set. 2012. URL `https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf`.

16: W. Henecka, S. KÃ¶gl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: Tool for automating secure two-party computations, 2010.

17: L. Kamm and J. Willemson. Secure floating-point arithmetic and private satellite collision analysis. Cryptology ePrint Archive, Report 2013/850, 2013. `http://eprint.iacr.org/`.

18: Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

19: U. M. Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

20: T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, Beijing, China, April 16-20, 2007, Proceedings*, pages 343–360, 2007.

21: V. Pastro. Zero knowledge protocols and multiparty computation. 2013.

22: B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure two-party computation is practical, 2009.

23: M. O. Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 1981.

24: R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2): 120–126, 1978.

25: SecureSCM. Cryptographic aspects - secure computation models and frameworks. Technical report, July 2008. secureSCM Deliverable D9.1.

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

27: A. C.-C. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.

Listings

Listing 6.1: Fairplay 4-Bit AND Operation

```
/*
 * Compute AND of two 4-bit int
 */
program And {
        const N=4;
        type Size = Int<N>;
        type AliceInput = Size;
        type BobInput = Size;
        type AliceOutput = Size;
        type BobOutput = Size;
        type Input = struct {AliceInput alice,  BobInput bob};
        type Output = struct {AliceOutput alice, BobOutput bob};

        function Output output(Input input) {
           output.alice = (input.bob & input.alice);
           output.bob = (input.bob & input.alice);
        }
}
```

Listing 6.2: Fairplay 4-Bit AND Operation in SHDL circuit format

```
0 input         //output$input.bob$0
1 input         //output$input.bob$1
2 input         //output$input.bob$2
3 input         //output$input.bob$3
4 input         //output$input.alice$0
5 input         //output$input.alice$1
6 input         //output$input.alice$2
7 input         //output$input.alice$3
8 output gate arity 2 table [ 0 0 0 1 ] inputs [ 4 0 ]//output$output.alice$0
9 output gate arity 2 table [ 0 0 0 1 ] inputs [ 5 1 ]//output$output.alice$1
10 output gate arity 2 table [ 0 0 0 1 ] inputs [ 6 2 ]//output$output.alice$2
11 output gate arity 2 table [ 0 0 0 1 ] inputs [ 7 3 ]//output$output.alice$3
12 output gate arity 1 table [ 0 1 ] inputs [ 8 ]//output$output.bob$0
13 output gate arity 1 table [ 0 1 ] inputs [ 9 ]//output$output.bob$1
14 output gate arity 1 table [ 0 1 ] inputs [ 10 ]//output$output.bob$2
```

```
15 output gate arity 1 table [ 0 1 ] inputs [ 11 ]//output$output.bob$3
```

Listing 6.3: Fairplay 4-bit Multiply Operation

```
/*
Multiplication of two 4-bit integers
 */
program Mult {

// Type Definitions

type AliceInput = Int<4>;
type AliceOutput = Int<8>;
type BobInput = Int<4>;
type BobOutput = Int<8>;
type Input = struct {AliceInput alice, BobInput bob};
type Output = struct {AliceOutput alice, BobOutput bob};


// This is the main function
function Output output(Input input) {

  var Int<8> r;
  r=0;
  if (input.bob & 8) // test bit 3
    {r = r + input.alice;}

  r = r + r;
  if (input.bob & 4) // test bit 2
    {r = r + input.alice;}

  r = r + r;
  if (input.bob & 2) // test bit 1
    {r = r + input.alice;}

  r = r + r;
  if (input.bob & 1) // test bit 0
    {r = r + input.alice;}

  output.bob = r;
  output.alice = r;
}

}
```

Listing 6.4: Fairplay 4-bit Multiply Operation Circuti

```
0 input         //output$input.bob$0
1 input         //output$input.bob$1
2 input         //output$input.bob$2
```

```
3 input          //output$input.bob$3
4 input          //output$input.alice$0
5 input          //output$input.alice$1
6 input          //output$input.alice$2
7 input          //output$input.alice$3
8 gate arity 2 table [ 0 0 0 1 ] inputs [ 4 3 ]
9 gate arity 2 table [ 0 0 0 1 ] inputs [ 7 3 ]
10 gate arity 2 table [ 0 0 0 1 ] inputs [ 6 3 ]
11 gate arity 2 table [ 0 0 0 1 ] inputs [ 5 3 ]
12 gate arity 2 table [ 0 0 0 1 ] inputs [ 5 8 ]
13 gate arity 2 table [ 0 1 1 0 ] inputs [ 6 11 ]
14 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 12 6 11 ]
15 gate arity 2 table [ 0 1 1 0 ] inputs [ 12 13 ]
16 gate arity 2 table [ 0 1 1 0 ] inputs [ 7 10 ]
17 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 14 7 10 ]
18 gate arity 2 table [ 0 1 1 0 ] inputs [ 14 16 ]
19 gate arity 2 table [ 0 1 0 0 ] inputs [ 7 3 ]
20 gate arity 3 table [ 0 0 0 1 0 0 1 1 ] inputs [ 17 7 3 ]
21 gate arity 2 table [ 0 1 1 0 ] inputs [ 17 19 ]
22 gate arity 2 table [ 0 1 1 0 ] inputs [ 20 19 ]
23 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 9 22 2 ]
24 gate arity 2 table [ 0 0 0 1 ] inputs [ 4 2 ]
25 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 9 21 2 ]
26 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 10 18 2 ]
27 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 11 15 2 ]
28 gate arity 3 table [ 0 1 0 1 0 1 1 0 ] inputs [ 8 5 2 ]
29 gate arity 2 table [ 0 0 0 1 ] inputs [ 5 24 ]
30 gate arity 2 table [ 0 1 1 0 ] inputs [ 6 28 ]
31 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 29 6 28 ]
32 gate arity 2 table [ 0 1 1 0 ] inputs [ 29 30 ]
33 gate arity 2 table [ 0 1 1 0 ] inputs [ 7 27 ]
34 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 31 7 27 ]
35 gate arity 2 table [ 0 1 1 0 ] inputs [ 31 33 ]
36 gate arity 2 table [ 0 1 1 0 ] inputs [ 7 26 ]
37 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 34 7 26 ]
38 gate arity 2 table [ 0 1 1 0 ] inputs [ 34 36 ]
39 gate arity 2 table [ 0 1 1 0 ] inputs [ 7 25 ]
40 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 37 7 25 ]
41 gate arity 2 table [ 0 1 1 0 ] inputs [ 37 39 ]
42 gate arity 2 table [ 0 1 1 0 ] inputs [ 7 23 ]
43 gate arity 2 table [ 0 1 1 0 ] inputs [ 40 42 ]
44 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 25 41 1 ]
45 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 23 43 1 ]
46 gate arity 2 table [ 0 0 0 1 ] inputs [ 4 1 ]
47 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 26 38 1 ]
48 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 27 35 1 ]
49 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 28 32 1 ]
50 gate arity 3 table [ 0 1 0 1 0 1 1 0 ] inputs [ 24 5 1 ]
51 gate arity 2 table [ 0 0 0 1 ] inputs [ 5 46 ]
```

```
52 gate arity 2 table [ 0 1 1 0 ] inputs [ 6 50 ]
53 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 51 6 50 ]
54 gate arity 2 table [ 0 1 1 0 ] inputs [ 51 52 ]
55 gate arity 2 table [ 0 1 1 0 ] inputs [ 7 49 ]
56 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 53 7 49 ]
57 gate arity 2 table [ 0 1 1 0 ] inputs [ 53 55 ]
58 gate arity 2 table [ 0 1 1 0 ] inputs [ 7 48 ]
59 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 56 7 48 ]
60 gate arity 2 table [ 0 1 1 0 ] inputs [ 56 58 ]
61 gate arity 2 table [ 0 1 1 0 ] inputs [ 7 47 ]
62 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 59 7 47 ]
63 gate arity 2 table [ 0 1 1 0 ] inputs [ 59 61 ]
64 gate arity 2 table [ 0 1 1 0 ] inputs [ 7 44 ]
65 gate arity 3 table [ 0 0 0 1 0 1 1 1 ] inputs [ 62 7 44 ]
66 gate arity 2 table [ 0 1 1 0 ] inputs [ 62 64 ]
67 gate arity 2 table [ 0 1 1 0 ] inputs [ 7 45 ]
68 gate arity 2 table [ 0 1 1 0 ] inputs [ 65 67 ]
69 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 47 63 0 ]
70 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 44 66 0 ]
71 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 45 68 0 ]
72 gate arity 2 table [ 0 0 0 1 ] inputs [ 4 0 ]
73 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 48 60 0 ]
74 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 49 57 0 ]
75 gate arity 3 table [ 0 1 0 1 0 0 1 1 ] inputs [ 50 54 0 ]
76 gate arity 3 table [ 0 1 0 1 0 1 1 0 ] inputs [ 46 5 0 ]
77 output gate arity 1 table [ 0 1 ] inputs [ 72 ]      //output$output.bob$0
78 output gate arity 1 table [ 0 1 ] inputs [ 76 ]      //output$output.bob$1
79 output gate arity 1 table [ 0 1 ] inputs [ 75 ]      //output$output.bob$2
80 output gate arity 1 table [ 0 1 ] inputs [ 74 ]      //output$output.bob$3
81 output gate arity 1 table [ 0 1 ] inputs [ 73 ]      //output$output.bob$4
82 output gate arity 1 table [ 0 1 ] inputs [ 69 ]      //output$output.bob$5
83 output gate arity 1 table [ 0 1 ] inputs [ 70 ]      //output$output.bob$6
84 output gate arity 1 table [ 0 1 ] inputs [ 71 ]      //output$output.bob$7
85 output gate arity 1 table [ 0 1 ] inputs [ 72 ]      //output$output.alice$0
86 output gate arity 1 table [ 0 1 ] inputs [ 76 ]      //output$output.alice$1
87 output gate arity 1 table [ 0 1 ] inputs [ 75 ]      //output$output.alice$2
88 output gate arity 1 table [ 0 1 ] inputs [ 74 ]      //output$output.alice$3
89 output gate arity 1 table [ 0 1 ] inputs [ 73 ]      //output$output.alice$4
90 output gate arity 1 table [ 0 1 ] inputs [ 69 ]      //output$output.alice$5
91 output gate arity 1 table [ 0 1 ] inputs [ 70 ]      //output$output.alice$6
92 output gate arity 1 table [ 0 1 ] inputs [ 71 ]      //output$output.alice$7
```