

UNIVERSIDAD DE SALAMANCA
Facultad de Ciencias

GRADO EN MATEMÁTICAS

OBLIVIOUS TRANSFER

TRABAJO DE FIN DE GRADO
realizado por

IRENE COTRINA GONZÁLEZ

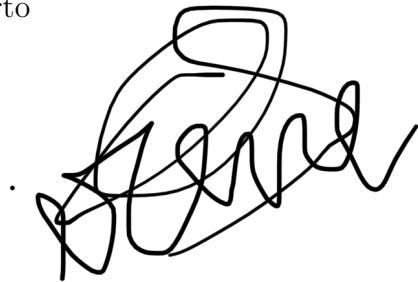
bajo la tutela de:
JOSÉ IGNACIO IGLESIAS CURTO (Departamento de Matemáticas)



Salamanca, July 6, 2022

OBLIVIOUS TRANSFER

Trabajo de fin de Grado presentado por
Irene Cotrina González
Bajo la tutela de
José Ignacio Iglesias Curto



AUTORA: Irene Cotrina González

TUTOR: José Ignacio Iglesias Curto

Salamanca, July 6, 2022.

Contents

1	Introduction	1
2	Preliminaries	3
2.1	Cryptography	3
2.1.1	ElGamal Protocol	4
2.2	Turing Machines	5
2.3	Complexity	7
2.4	Multiparty Computation	8
3	<i>1-out-of-2 OT</i>	11
3.1	General Definition	11
3.2	<i>1-out-of-2 OT</i>	13
3.3	Two-party Secure Computation of Functions	16
3.3.1	Simulators	17
3.3.2	Two-party Computation of <i>XOR</i>	19
4	<i>1-out-of-n OT</i>	21
4.1	General Definition	21
4.2	<i>1-out-of-n OT</i> Protocols	22
4.2.1	<i>1-out-of-n OT</i> Protocol as a Variant of ElGamal	22
4.2.2	<i>1-out-of-n OT</i> Protocol Using Pseudo-random Functions	24
5	Commitment Protocols and ZKP	29
5.1	Introduction	29
5.2	Width 5 Permutation Branching Programs	30
5.3	Committal Protocol Using Oblivious Transfer	31
5.3.1	Simple Committal Protocol	31
5.4	Permutation Randomizers, Condensers, and Tableaus	33
5.5	Introducing Zero-knowledge Proofs on Commitment Protocols	34
5.6	Oblivious Transfer	36
5.6.1	Oblivious Circuit Evaluation	37
5.6.2	Oblivious Decommittal	37
6	Dealing with Malicious Players	39
6.1	Two-party Secure Computation Against Malicious Players	39
6.2	GMW Compiler	41
7	Uses of Oblivious Transfer	45
7.1	PIR to SPIR Transformation	45
7.2	Polynomial Evaluation	46

8 Conclusions**47****Bibliography****49**

Chapter 1

Introduction

Oblivious transfer, or OT, is a communication scheme that dates back to 1982. It was Michael O. Rabin who presented the first OT protocol in [16]. He wanted to solve the *exchange of secrets* problem. That mentioned problem is as follows: in a communication scheme with a sender Alice and a receiver Bob, both parties wish to exchange some secrets they own, because each one's secret is owned by the other party. For example, let us suppose that Alice has a secret s_a , which is Bob's password to a file, and Bob has another secret s_b , which is Alice's password to a file. They want to open their files, but if they use a wrong password, the file is destroyed. They wish to send those secrets to the other party without no one else participating on the exchange, so there is no other party who is able to know the value of the secrets. Rabin's idea was to guarantee that the secrets are received by the other party correctly, so they can open and read their respective files. That is, Alice receives s_b , and Bob s_a . Therefore, he wanted to guarantee that Alice does not change the value of s_a , neither Bob changes the value of s_b . If any of them changed the value before sending it, and the other party tried to open its file with a wrong password, then the file would be destroyed and the other party would notice about it.

How to solve the problem looks simple, but just if Alice and Bob behave as it is supposed they should. Nevertheless, both parties can cheat the other one in some different ways. Let us suppose that Alice sends Bob a secret $s \neq s_a$, and Bob sends Alice his secret s_b . When Bob uses secret s for opening his file, it will be destroyed because the password is incorrect. However, Alice will be able to open her file with the correct password and read it. If this happened, Bob could open a case against Alice, and it seems like he would win. But Alice could say that Bob did not use the secret she had sent him, and she could also reveal the value of s_a so that anyone could then open Bob's file.

Also, any of the parties could decide to stop its participation on the protocol after receiving its password, and that party would have received its secret, but the other party would have not. We would be, indeed, in a similar situation as we were before.

Michael O. Rabin presented a protocol, that was called *Oblivious transfer*, for solving this problem, the *exchange of secrets* problem. The protocol is called *Oblivious transfer* because the involved parties do not know certainly if the other party opened the file or not, that is, Alice is oblivious about Bob reading file or not, and Bob is oblivious about Alice opening hers. Along the project, *Oblivious transfer* is studied as it evolved through the years.

Firstly, in Chapter 2 we expose a brief introduction to cryptography, and we present the main ideas that we are going to need and use in several occasions along this project. They were all read on [4, 7, 15, 12].

In Chapter 3, the first OT protocol presented by Michael O. Rabin in [16] is explained.

Then, we present a development of that protocol, [17], which was defined for the first time in 1985 in [5], by Shimon Even, Oded Goldreich, and Abraham Lempel, and uses cryptographic algorithms. This protocol has a difference with the one presented by Rabin. In this case, Alice has two secrets and sends them to Bob. Then, Bob can just look at one of those values. Alice will know nothing about the value Bob selected, and Bob will know nothing about the value of the other option he had. Moreover, we define the different types of players: semi-honest players (they are not going to try to cheat, but if they can, they will private information about other players) and malicious players (those who try to cheat). Completely honest players are not considered in real life, but they will be part on ideal models, where the theoretical ideas of the protocol are defined, and everything is perfect. We will be considering Alice and Bob as semi-honest players until we define all protocols secure against this type of player. Also, we implement OT as a two-party computation, which consists on evaluating a function by Alice and Bob, such that neither of them knows the other one's input, Alice gets no output, and Bob gets the output of the function. Later, an exclusive-or, *XOR*, function is evaluated, because it is going to be used in other chapters.

Chapter 4 presents another development of the *Oblivious transfer*, the *1-out-of- n OT* protocol, [13, 18], in which Alice sends Bob n messages and he can just select one of them. Since we are still defining an OT protocol, Alice knows nothing about the value that Bob selected, and Bob knows nothing about the values of the messages he did not select. This was presented for the first time in 1987 by G. Brassard, C. Crépeau, and J.-M. Robert in [3]. In this chapter we show two different ways of implementing the protocol, and conclude that one of them is computationally easier.

Chapter 5 introduces commitment protocols and zero-knowledge proofs, taken from [9]. With these procedures we explain how different *Oblivious transfer* protocols are defined using these aspects. Although the constructions here shown are really interesting, I was not able to find the extended version of the mentioned paper, [9], and the constructions of the latter protocols are briefly defined due to the lack of bibliography.

As we have mentioned, we were going to consider Alice and Bob as semi-honest players until we defined all protocols secure against semi-honest players. At this point we will have already explained protocols secure against semi-honest players. Therefore, in Chapter 6 we focus on malicious players and how to deal with them, exposing how we can implement any *Oblivious transfer* protocol that involves malicious players. We will show that, even if any of the players tries to cheat, the player will not be able to do it. The main idea is to force players to follow the steps of the protocol such that if they do not do it, they can not participate in the execution. Therefore, our protocols are secure against any type of players, which was read in [6].

Although all the theoretical ideas are interesting to study, *Oblivious transfer* protocols can be implemented into daily life. Therefore, Chapter 7 shows some uses of the described algorithms and how they are implemented. In short, *Oblivious transfer* has become really useful for creating databases, and for polynomial evaluation ([13]).

Chapter 2

Preliminaries

The notions used along this chapter have been selected from [4, 7, 15, 12].

2.1 Cryptography

When we talk about cryptography, we refer to a branch of *information theory*. It studies how to transmit information between a sender and a receiver through a channel. Cryptography transforms ordinary text into unintelligible text (encrypt) and vice-versa (decrypt). With this process, a message is transmitted such that just the legitimate receiver can receive and read it. If any third party wishes to intercept the messages, it will not be able to do it.

We can easily make a distinction: symmetric and asymmetric cryptography.

- Symmetric cryptography: this is the most ancient cryptography, and it is also called secret key cryptography. In this type of cryptography a sender and a receiver share a password which they use for encrypting and decrypting their messages. This password is unique for each pair of participants. That is, if we consider a protocol where there are more than one pair, each one will have a shared secret password. Furthermore, for establishing that password, a secure channel must previously be defined. Computationally, these protocols are usually easier to compute because the algebraic operations they need are simpler than the ones in asymmetric cryptography. Nevertheless, they use a huge number of repetitions and the keys are enormous.
- Asymmetric cryptography: it is also called public key cryptography and the protocols we are working with belong to this section. The sender and the receiver have got two keys each one. One of the keys is public, and everybody performing the communication knows it. The other key is secret, and just the owner of the key knows it. These protocols do not assume the existence of a secure channel for sending information. Computationally, these type of protocols are more complex because, although they use a lower number of operations, those operations are more complex than the ones in symmetric cryptography, and that is what makes them hard to compute.

Definition 2.1 (Public key cryptosystem scheme (PKCS)). A public key cryptosystem scheme is an asymmetric cryptosystem for sending messages between a sender and a receiver in which each participant has a private and a public key. The public key is known by all participants, and it is used for encrypting the messages they want to send to the other participant. The secret key is just known by the owner, and it is used for decrypting the messages it receives.

A public key cryptosystem scheme consists of three algorithms, Gen, Encrypt and Decrypt, that work as follows:

1. Gen is an algorithm that outputs (p, s) , the public and secret key, respectively.
2. $\text{Encrypt}(m, p)$ encrypts the message m using the public key p .
3. $\text{Decrypt}(c, s)$ decrypts the cyphertext c using the secret key s .

PKCS are usually used for transmitting a message. The original message, without the encryption, is sometimes called *plaintext*. A protocol is said to behave properly if it verifies *correctness* and *security*.

Definition 2.2 (Correctness). A public key cryptosystem is correct if the decrypted message is equal to the original one. That is, for every original message m , it verifies that, for every pair of public-private key (p, s) ,

$$\text{Decrypt}(\text{Encrypt}(m, p), s) = m \quad (2.1)$$

Definition 2.3 (Security). A public key cryptosystem is secure if just the receiver of the message can read it. Also, no secret key is revealed to any party.

2.1.1 ElGamal Protocol

ElGamal is a PKCS we are going to be using this protocol along the project. It is considered interesting in research and applications. The security of this method is based on the Decisional Diffie-Hellman assumption, which is considered a hard problem.

Let us consider a multiplicative cyclic group \mathbb{F}_p , where p is a prime number and the order of the group, and let us denote $g \in \mathbb{F}_p$ as a generator. This means that $\mathbb{F}_p = \{1, g^2, g^3, \dots, g^{p-1}\}$, and $g^p = 1$.

Definition 2.4 (Decisional Diffie-Hellman assumption). Suppose two numbers $a, b \in \mathbb{F}_p$ independently chosen. The decisional Diffie-Hellman (DDH) assumption states that, given g^a and g^b , we can not distinguish them from the value of g^{ab} , and we say it is indistinguishable from a random element in \mathbb{F}_p .

ElGamal is a PKCS which a sender and a receiver use to exchange messages. We will call the sender A(lice) and the receiver B(ob). Alice and Bob have, each one, a pair public-secret key, and they wish to exchange some messages. This algorithm works as follows:

Algorithm 2.5 (ElGamal cryptosystem).

1. A random prime number p and a generator $g \in \mathbb{F}_p$ are fixed.
2. Alice and Bob select their private key. They are random numbers s_A and s_B such that $1 < s_A, s_B < p - 1$.

3. The elements g^{s_A} and g^{s_B} are the public keys for Alice and Bob. Those values and g, p are published in a list.
4. Alice chooses randomly a value $k \in \mathbb{F}_p$ and sends a message to Bob, $n \in s_B$. Alice computes g^k and, using the public key's list, also computes $n(g^{s_B})^k$.
5. Alice sends $(g^k, n(g^{s_B})^k)$.
6. Bob receives two numbers, (x, y) . Using his private key, he calculates x^{s_B} . Then, he performs the operation $y(x^{s_B})^{-1}$. This value is the same as the original message.

This protocol guarantees *correctness*, because the decrypted message is the same as the initial one, and it also guarantees *security*, which holds under the DDH assumption.

2.2 Turing Machines

We need to briefly define Turing machines because we will need them for defining some other ideas in the following sections of this chapter. Moreover, Turing machines are sometimes used as an element of the communication scheme.

Definition 2.6 (Turing machine). A deterministic Turing Machine is a tuple $T = (Q, \Sigma, \delta, q_0, s)$ where:

- Q is a finite set of states, and $q_0 \in Q$ is a distinguished element called the initial state.
- $\Sigma = \{0, 1, \#\}$ is the alphabet we are using and its elements are written on a tape, where the symbol $\#$ means a blank space.
- δ is a function

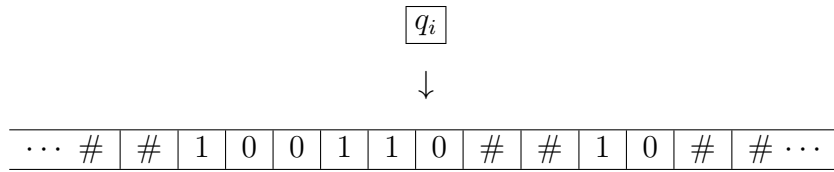
$$\begin{aligned} \delta : Q \times \Sigma &\longrightarrow Q \times \Sigma \times \{L, R, N\} \\ (q, i) &\longmapsto (q', i', m) \end{aligned}$$

that, for each pair of inputs (q, i) , associates a new state $q' \in Q$, an overwrite symbol $i' \in \Sigma$, and a movement $m \in \{L, R, N\}$ on the tape the machine is reading. L is a movement to the left, R is a movement to the right, and N is not to move.

- s is a stop criterion.

The domain of the function δ is not the whole $Q \times \Sigma$, therefore, there exist some elements $(q, i) \in Q \times \Sigma$ for which $\delta(q, i)$ does not exist. Accordingly, a Turing Machine can be defined as a finite set of elements (q, i, q', i', m) such that $\delta(q, i) = (q', i', m)$.

With the definition of a Turing machine, we have now to interpret what it means. The machines have a tapehead that moves along a tape. The tapehead reads and writes over that tape. Each tape has a finite number of elements from the alphabet $\Sigma = \{0, 1, \#\}$. Every tape can be filled with blank spaces on its sides, transforming it to an infinite tape.



In each step, the tapehead is placed over a position, and there it reads a symbol i from the tape. Suppose q is the state in which the tapehead is. Then, $\delta(q, i) = (q', i', m)$ means that the tapehead writes i' over i , makes a movement m (L , R or N), and changes its state to q' . The tapehead will always start reading (or writing) in the leftmost symbol different to the blank space from the tape. The final state of a Turing machine is an element $q \in Q$ such that $(q, i) \notin \text{Dom } \delta$, for any $i \in \Sigma$, and it verifies the stop criterion. Moreover, a Turing machine recognizes a tape if, starting from its initial state q_0 , the machine stops at some final state when reading (or overwriting) the tape.

Furthermore, there also exist nondeterministic Turing machines. The difference is that in nondeterministic Turing machines there are two tapes, one that the machine reads and it always moves to the right, and other one where it writes. The tape the machine reads is called random tape, because the elements from the alphabet are randomly written over it, but not the blank space, and it introduces a random decision for each step. Then, the machine writes over the other tape, which it is simply called tape. The probability for the machine to make a mistake, that is, it does not recognize the random tape, is called the error probability and it is bounded by a constant. In nondeterministic Turing machines there are a set of possible movements, because there is a probability of the random tape to be recognized. Therefore, there will be a set of possible final states, so the computation of the machine is a tree of states that can be reached from the initial one. An input is accepted if there is at least one node of the tree which is an accepted final state, otherwise it is not accepted.

Definition 2.7 (Nondeterministic Turing machine). A nondeterministic Turing machine is a tuple $T = (Q, \Sigma, \delta, q_0, A, S)$ where:

- Q is a finite set of states, and q_0 is the initial state.
- $\Sigma = \{0, 1, \#\}$ is the alphabet we are using and it is written on a tape, where the symbol $\#$ means a blank space.
- $A \subseteq Q$ is the set of accepted final states.
- δ is a function

$$\begin{array}{ccc}
 \delta : Q \times \Sigma & \longrightarrow & Q \times \Sigma \times \{R\} \\
 (q, i) & \longmapsto & (q', i', m)
 \end{array}$$

- s is a stop criterion.

Nondeterministic Turing machines are also called probabilistic Turing machines, due to the fact that it depends on the tape it reads, which is fulfilled with random symbols from the alphabet, and there is a probability of making a mistake.

2.3 Complexity

Another important aspect we are interested in is the complexity of the algorithm, which describes how the runtime of a protocol depends on the amount of input data. For complexity, we use *big O* notation.

Definition 2.8. (Order of a function) Let f be a real or complex valued function whose complexity we want to determine, and let g be another real valued function. Let both functions to be defined on some unbounded subset of the positive real numbers, and $g(x)$ be strictly positive for all large enough x . Then:

$$f(x) \approx O(g(x)) \Leftrightarrow |f(x)| \leq Cg(x) \text{ for some constant } C \in \mathbb{N}$$

and we will say that the function $f(x)$ is of order $O(g(x))$.

Definition 2.9 (Complexity). The complexity of an algorithm represents the amount of temporal resources that the algorithm needs in order to solve a problem.

The complexity of a problem can not be calculated with exactitude, because we can not calculate the number of operations done when a problem is solved. But that number of operations can be bounded by a function that depends on the time that the operations from the problem need to be solved.

There exist some different orders of complexity. Those can be observed in the following chart, where they are organized by complexity hierarchy:

Order	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Lineal
$O(n \log n)$	Almost lineal
$O(n^2)$	Cuadratic
$O(n^3)$	Cubic
$O(a^n)$	Exponential

These orders verify that

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(a^n) \quad (2.2)$$

We are going to focus on the computability of problems, that is, if they are solvable by an algorithm or a Turing machine (deterministic or nondeterministic). Since a Turing machine can perform a huge number of computations, the most basic means for analyzing this is to calculate the execution time of a model based on Turing machines. If a problem can be modeled by a Turing machine, then the problem is said to be in class \mathcal{P} , that is, it has polynomial complexity. If a problem can not be modeled by a Turing machine, then it does not have polynomial complexity, and it is said to be in class \mathcal{NP} .

Definition 2.10 (Probabilistic polynomial-time complexity (PPT)). A problem is in PPT if it can be solved by a probabilistic Turing machine, with an error probability of less than $1/2$ for all recognizable tapes.

Problems in PPT are going to be used along the project.

Moreover, we are using the so called Nick's class. It was called this way after Nick Pippenger, who had done several studies about this computational complexity. This class includes the problems solvable by a deterministic Turing machine in some time bounded by $(\log n)^i$, for a constant $i \in \mathbb{N}$, using a polynomial number of parallel processors. That is, the problem is separated into a polynomial number of smaller problems that are executed simultaneously for solving the initial problem. The problems in Nick's class are denoted as problems of order $O(\log^i n)$.

Definition 2.11 (Parallel computation). Parallel computation is a set of instructions that are executed simultaneously and independently. Those instructions solve a problem that can be separated into smaller problems such that the result is the same as the result of the problem. The smaller problems are said to be solvable by parallel processors.

Definition 2.12 (NC^i complexity). A problem is in NC^i if there exists a constant $k \in \mathbb{N}$, such the problem can be solved in time $O(\log^i n)$ using $O(n^k)$ parallel processors.

2.4 Multiparty Computation

Finally, we introduce the concept of multiparty computation (MPC). When we talk about multiparty computation, we are referring to a protocol in which there is a fixed number of participants, having some private data each one, who want to compute a function whose inputs are those data, but without revealing their private data to any of the parties.

Let us fix a number n of participants in the communication, p_1, p_2, \dots, p_n , and their private data, d_1, d_2, \dots, d_n , respectively. There is a public boolean function known by all of them, and they wish to compute $f(d_1, d_2, \dots, d_n) = x_{1, \dots, n}$, from which all participants receive $x_{1, \dots, n}$ and their private data remain hidden.

Definition 2.13 (Multiparty computation). Multiparty computation is a process in which a fixed number of participants possessing some private data evaluate a boolean function using their data as inputs of the function. They all receive the output, but their private data are not revealed.

We present the MPC because along the project we need to use a variant of this, the two-party secure computation. That is, we have two participants, Alice and Bob, and a function f that they want to compute with some private data they own. This process must be secure, with no external party guessing Alice and Bob's inputs neither the output of the function. Besides, it must verify privacy for Alice and Bob, which means that they can not know anything about each other's input.

MPC can be illustrated with Yao's Millionaires' problem, introduced in 1982 by Andrew Yao. In this problem, two parties, Alice and Bob, are millionaires. They want to know who is richer without revealing how much money they possess, so a two-party secure computation would reveal the information they want to know, but not the amount of money they own.

Chapter 3

1-out-of-2 OT

The references for this chapter have been [16, 17].

3.1 General Definition

Oblivious transfer (OT) is a recent cryptosystem scheme that uses asymmetric cryptography for the communication, that is, it uses PKCS. It was presented by Michael O. Rabin for the first time in [16], with the aim of solving the *Exchange of secrets* (*EOS*) problem. There are a sender and a receiver on the communication, whom we will call Alice and Bob.

In the *EOS* problem, Alice has a secret s_a that Bob needs to receive, and Bob has a secret s_b that Alice needs to receive. Then, when they receive the message, they will use it, and if it is wrong, they will know it. Therefore, they will engage on a communication scheme in which they send those messages to the other party such that Alice is oblivious about if Bob receives the message, and Bob is oblivious about if Alice receives the message. That is why the protocol we present was called *oblivious transfer*.

Moreover, Alice and Bob wish to exchange those secrets without involving any other party. What we want is to define a protocol between Alice and Bob that verifies the mentioned ideas, without any other party involved. Although those secrets can be anything, from now on we will consider that Alice and Bob exchange bits, for simplicity.

Accordingly, there are multiple scenarios in which Alice and Bob do not behave as they are supposed to. If this happened, the messages could not be received by the legitimate receiver, or they could be wrong. In order to avoid this, they will follow a procedure for signing the messages they send. They can use the traditional digital signature, that is presented in [15].

We want to define a protocol in which, if Alice receives her message, then Bob receives his. This is the *EOS* problem, and it was solved by Michael O. Rabin when he defined the first *Oblivious Transfer* protocol.

To present that protocol, we will assume that if Alice receives the message s_b , she will use it, and then Bob will know that she has used it. Also, if Bob receives the value of s_a , he will use it and Alice will know it too.

Firstly, we will also assume that Alice and Bob will have public keys, k_a and k_b , respectively. They are going to use these keys for signing the messages using digital signatures.

Algorithm 3.1 (Exchange of secrets).

1. Alice and Bob choose two large primes each one. Alice chooses p_a, q_a , and Bob p_b, q_b , and calculate single use keys

$$n_a = p_a \cdot q_a$$

$$n_b = p_b \cdot q_b$$

2. Alice sends Bob a signed message with the value of n_a , and Bob sends Alice a signed message with the value of n_b .
3. Bob chooses a random value $x < n_a$, computes $c = x^2 \bmod n_a$, and sends Alice a message: “ $E_{k_b}(x)$ is the encryption by my public key k_b of my secret number, and c is its square mod n_a . Signed, Bob”.
4. Alice knows the factorization of n_a . Therefore, she can calculate x_1 such that $x_1^2 = c \bmod n_a$. Then, she sends Bob a signed message in which she tells him that x_1 is a square-root mod n_a of c .
5. Bob calculates $(x - x_1, n_a) = d_b$. With probability $1/2$, $d_b = p_a$ or $d_b = q_a$. Thereupon, with probability $1/2$ Bob knows the factorization of n_a .
6. They define

$$\nu_b = \begin{cases} 0 & \text{if } (x - x_1, n_a) = p_a, q_a \\ 1 & \text{otherwise} \end{cases}$$

Thus, if $\nu_b = 0$, then Bob knows the factorization of n_a . Also, Alice’s bit ν_a is defined similarly.

7. Bob computes $\varepsilon_b = s_b \oplus \nu_b$, and sends it to Alice in a signed message.
8. Alice computes $\varepsilon_a = s_a \oplus \nu_a$. Then, she sends ε_a to Bob in a signed message.
9. Alice chooses a random message m_a and places her secret as the center bit of the random message. Then, she encrypts m_a using a PKCS that requires factor p_a and q_a for decrypting. The encrypted message is $d_a = E_{n_a}(m_a)$, and Alice sends it to Bob in a signed message.
10. Bob performs the same procedure using s_b , and sends the encrypted message to Alice in a signed message.

At the end of the step 5 from Algorithm 3.1, Alice does not know if Bob has the factorization of n_a or not. Also, we need to say that the operator \oplus denotes the exclusive or operation (XOR), that is zero if both values, s_b and ν_b , are equal, and one if s_b and ν_b are different (1 and 0).

The above protocol, Algorithm 3.1, was the first *oblivious transfer* protocol to be presented and defined, [16].

Theorem 3.2. ([16]) *Algorithm 3.1 gives a solution for the exchange of secrets problem, and the probability that neither party will obtain the other party's secret is $1/4$.*

Proof. This proof will be made following the assumption that, if Alice uses the message s_b , then Bob will know it, and vice-versa. Then, if any of them cheated by sending a different message, the other party would know it.

If Alice or Bob stop their participation in the protocol before the final phase, the other one will notice about it and will also stop. Therefore, none of them would know each other's secret, so none of them would receive the message.

Suppose they participate in the protocol until the end. Assume that Alice sends $d_a = E_{n_a}(m_a)$ to Bob. If Bob knows the factorization of $n_a = p_a \cdot q_a$, with probability $1/2$, then $\nu_b = 0$, and he can decrypt d_a . Thereupon, he will know m_a and s_a . If Bob knows s_a with certainty, he will use it, and Alice will know it. Hence, since Alice knows that Bob used the message she has sent him, then she knows that $\nu_b = 0$, and then $\varepsilon_b = s_b \oplus \nu_b = s_b$. Consequently, Alice also knows s_b , because Bob had sent her the value of ε_b .

An analogous process could be followed if Bob gave Alice $E_{n_b}(m_b)$ in the final phase. So if Alice guessed the value of s_b , then Bob would guess the value of s_a .

In conclusion, if Alice knows s_b , then Bob knows s_a , and vice-versa. Then, when the protocol is finished, the probability that neither of them knows each other's secret is

$$\left(\frac{1}{2}\right)^2 = \frac{1}{4}. \quad (3.1)$$

□

3.2 1-out-of-2 OT

OT developed in different ways from the initial Algorithm 3.1. Following the idea of exchanging secrets, it is used to select one secret from a set of options. That is, the sender has some options (secrets) and the receiver can just select one of them, providing that Alice does not know what value was selected by Bob, and Bob does not know anything about the values he did not select.

For the simplest case, we will consider that Alice keeps two inputs, which are the options, and Bob can just look at one of them. This is what we call a *1-out-of-2 OT* protocol, and it was first presented by Shimon Even, Oded Goldreich, and Abraham Lempel in 1985, in [5].

Hence, for this problem, Alice has two inputs, let's say $x_0, x_1 \in \{0, 1\}^k$, and Bob has one input, $b \in \{0, 1\}$. Therefore, in a *1-out-of-2 OT* protocol, Alice sends Bob her inputs, and Bob can just look at x_b , being b defined as above. Bob will know nothing about x_{1-b} , and Alice will know nothing about Bob's input neither output.

We will define this procedure step by step, such that it becomes easy to understand. We will first define it using a trusted third party. This party will be called Charlie, and Alice and Bob will send him their messages. Furthermore, Charlie will not reveal each other's inputs to none of the parties. We can consider this trusted party as a function, say f . This function will receive inputs from Alice and from Bob. When it is implemented,

the output will be sent to Bob, and Alice will receive nothing. We can define a protocol as follows:

Algorithm 3.3 (1-out-of-2 OT protocol with a trusted third party).

- **Inputs:** Alice's inputs are $x_0, x_1 \in \{0, 1\}^k$. Bob's input is $b \in \{0, 1\}$.

- **Outputs:** Alice gets no output. Bob's output is $x_b \in \{0, 1\}^k$

1. Alice sends x_0, x_1 to Charlie.
2. Bob sends b to Charlie.
3. Charlie computes the function f , that is, $f((x_0, x_1), b) = x_b$. Then, he sends to Bob the output x_b .

Algorithm 3.3 must verify *security* and *correctness*, as we explained in the previous chapter. It is clear that it guarantees *correctness*, because Bob receives the message x_b . Since no information is revealed to any party, it also guarantees *security*. This is because of the assumption of Charlie being a trusted party. Unfortunately, this trusted players do not really exist, so this algorithm can not be used in real life. We will consider two types of players: a semi-honest (also called honest but curious) player, and a malicious (also called Byzantine) player.

Definition 3.4 (Semi-honest player). A player is called semi-honest when it follows the protocol correctly, but is eager to learn private information of the other players.

Definition 3.5 (Malicious player). A player is called malicious when it deviates from the protocol in any way. That is, the player tries to cheat sending incorrect messages or giving unreal information.

Firstly, we will consider all participants as semi-honest, because the malicious players must be studied in a different way due to their different behaviour. When we define the protocols secure against semi-honest players, we will focus on the malicious ones (in Chapter 6).

Since the trusted party Charlie does not exist in real life, we want to focus on defining a protocol in which he does not participate. The idea is to create a protocol in which we can achieve similar *security* and *correctness* without using any other party. Before exposing that protocol, we will define one which is divided into two phases, and Charlie just participates in one of them. The first phase is the *Setup phase*, where Charlie participates and creates the keys. The second phase is the *Transfer phase*, and it is completed by Alice and Bob without Charlie's help. Therefore, Charlie just participates on the first phase, and the rest of the protocol is concluded by Alice and Bob. This is the first step on removing Charlie. Then, after the protocol is defined, we will prove the *correctness* and *security* and, if both are guaranteed, then a protocol completely without Charlie is defined.

Algorithm 3.6 (1-out-of-2 OT protocol with a trusted third party during *Setup phase*).

- **Inputs:** Alice's inputs are $x_0, x_1 \in \{0, 1\}^k$. Bob's input is $b \in \{0, 1\}$.
 - **Outputs:** Alice gets no output. Bob's output is $x_b \in \{0, 1\}^k$
1. Bob sends b to Charlie.
 2. Charlie generates two public-private key pairs, (p_0, s_0) and (p_1, s_1) , using ElGamal's PKCS. Then, Charlie sends the public keys, p_0 and p_1 , to Alice, and s_b to Bob (his secret key).
 3. Alice, using ElGammal's PKCS, computes

$$c_0 = \text{Encrypt}(p_0, x_0) \quad \text{and} \quad c_1 = \text{Encrypt}(p_1, x_1) \quad (3.2)$$

and sends (c_0, c_1) to Bob.

4. Bob receives (c_0, c_1) , and tries to decrypt both using the secret key that Charlie sent him, s_b . Bob keeps the plaintext he could correctly decrypt.

As we can see, Bob receives its output correctly, so *correctness* is guaranteed as before. Besides, since Charlie is a trusted party and ElGamal is a protocol whose security has already been proved, *security* is also guaranteed. Hence, we have defined a protocol where Charlie just participates on the initial phase.

Let \mathbb{F}_p^* be a multiplicative cyclic group of order p , $g \in \mathbb{F}_p^*$ a generator, and $x \in \mathbb{F}_p^*$ a random element. To define a protocol without Charlie, Bob can generate his public and secret keys as Charlie did in Algorithm 3.6, so the protocol without the third party can be defined as follows:

Algorithm 3.7 (1-out-of-2 OT protocol without a trusted third party).

- **Inputs:** Alice's inputs are $x_0, x_1 \in \{0, 1\}^k$. Bob's input is $b \in \{0, 1\}$.
 - **Outputs:** Alice gets no output. Bob's output is $x_b \in \{0, 1\}^k$
1. Bob generates his public and secret keys, (p_b, s_b) , using ElGamal's PKCS and his input $b \in \{0, 1\}$, where

$$p_b = g^x \quad \text{and} \quad s_b = x \quad (3.3)$$

Bob randomly samples an element $z_{1-b} \in \mathbb{F}_p^*$ and then sends $(z_b = p_b, z_{1-b})$ to Alice.

2. Alice receives (z_0, z_1) and computes, with $x_0, x_1 \in \{0, 1\}^k$,

$$c_0 = \text{Encrypt}(z_0, x_0) \quad \text{and} \quad c_1 = \text{Encrypt}(z_1, x_1) \quad (3.4)$$

Then, Alice sends (c_0, c_1) to Bob.

3. Bob receives (c_0, c_1) and decrypts c_b using his secret key s_b . Therefore, Bob obtains the value of x_b .

Again, Bob receives the value of x_b correctly, so *correctness* is verified. Furthermore, Alice knows nothing about that value, and Bob knows nothing about x_{1-b} because, since he is a semi-honest player, he just decrypted, therefore received, x_b . Thus, *security* is also guaranteed. Hence, we have defined a *1-out-of-2 OT* protocol without a trusted third party.

Although any other PKCS could have been used, I personally decided to use ElGamal for the *oblivious transfer* protocols shown in this project, because is the one that has been most used across the bibliography I read, [17]. The use of any other scheme would just modify the beginning of Algorithm 3.7, where the keys are generated, and the encryption and decryption of the messages, which would be done following that other scheme.

3.3 Two-party Secure Computation of Functions

Definition 3.8 (Boolean function). A boolean function is a function

$$f : \{0, 1\}^k \longrightarrow \{0, 1\} \quad (3.5)$$

where $k \in \mathbb{N}$.

Oblivious transfer can be used with the aim of evaluating functions. Suppose each party, Alice and Bob, has their private inputs, that is, Bob does not know Alice's inputs neither Alice knows Bob's. Suppose there is a boolean function f they want to compute.

We will present a *1-out-of-2 OT* protocol as a two-party secure computation. With this procedure, we can define *security* more precisely. Therefore, Alice has two inputs, $x_0, x_1 \in \{0, 1\}^k$, and Bob has an input $b \in \{0, 1\}$. Let f be the boolean function they want to compute. This function is defined as:

$$\begin{aligned} f : \{0, 1\}^{2k} \times \{0, 1\} &\longrightarrow \{0, 1\} \\ ((x_0, x_1), b) &\longmapsto f((x_0, x_1), b) = x_b \end{aligned} \quad (3.6)$$

Alice and Bob are able to compute f without revealing their inputs to the other party, and Bob is the one who receives the output of the function, as a *1-out-of-2 OT* protocol.

We want to define the *security* of a protocol that consists on the evaluation of the above function with the mentioned requirements. For that purpose, we have to make a comparison between the so-called real and ideal worlds.

The ideal world is the theoretical idea we have about how the protocol is going to be performed. In this ideal world everything works as it should do, as in the definitions made, and the run of the protocol has no mistakes. In an ideal world, the trusted third party, Charlie, evaluates the function using Alice and Bob's inputs without revealing any information about them to the other party. Charlie, can participate in the ideal model evaluation of the protocol because we can assume his existence in this ideal world.

Moreover, both of them receive their respective outputs, but they will not see each other's output. In the *1-out-of-2 OT protocol*, Bob will receive the output of the function and Alice will receive nothing.

By real world we refer to the implementation of the theoretical concepts from the ideal world. Obviously, when a protocol is performed in the real world it might run differently from what it was supposed. In the real world, Charlie does not participate in the execution of the protocol, so Alice and Bob interact with each other. Thereupon, it is Alice and Bob who perform the protocol in order to compute the desired function. Remember the fact that Alice and Bob are considered to be semi-honest. Therefore, they will follow the protocol, but if they could guess something about the other party's input, they would. The amount of data that a participant see from the protocol is called participant's view.

Definition 3.9 (View). Let P be a participant of a multiparty protocol. View of P is a tuple (T, x) , where x is P 's input and T is the set of actions that the participant computes and its output. T is going to be called transcript.

In case Alice or Bob knew extra information, like the other party's input, then the view of that party would be different in the real and ideal world, that is, it would see more in the real than in the ideal world. If this happened, the real world protocol would differ from the ideal one, and that means that the protocol would not be secure. We are trying to define a protocol in the real world which works similarly as it would work on the ideal world.

Since we want protocols to be secure, we will introduce a definition for secure two-party computation.

Definition 3.10 (Security in MPC). A multi-party protocol is secure if participant P 's view in the real world run of the system is computationally indistinguishable from its view in the ideal world run of the system.

Every OT protocol can be considered as a two-party secure computation, and its *security* follows the above definition.

3.3.1 Simulators

As we want the real model to be as similar as possible to the ideal one, we want to avoid what we mentioned earlier, that is, that participant P 's view of the real world is different as it is in an ideal world. To achieve that goal, we can use a simulator between the two involved parties in order to define a protocol in the ideal world and his security. Therefore, we can compare that protocol with the one we have in the real world.

Let P be a participant in the protocol whose input is x . If we consider the ideal world, Charlie participates and evaluates the function f .

Definition 3.11 (Simulator). A PPT machine M in the ideal world that behaves as participant P is called simulator when

$$Output_{P_{real}}(f(x)) \approx Output_M(f(x))$$

where $Output_M(f(x))$ denotes the output of the simulator M when it computes $f(x)$ in the ideal world, and $Output_{P_{real}}(f(x))$ the output of P 's computation of $f(x)$ in the real world. Besides, \approx means that those values are computationally indistinguishable.

Let us consider that we have a two-party protocol between Alice and Bob in the ideal world and, at first, suppose our simulator is an Alice simulator. This means, a PPT machine M (the simulator) will be an intermediary between Alice and Bob, and will simulate Alice's behavior. It is placed between Alice and Charlie, and interacts with Charlie as if it were Alice. The idea is that Charlie's interaction with the simulator is the same as if it interacted with Alice.

Thus, since the simulator is a machine, it needs to know Alice's inputs, so it receives Alice's inputs, encrypts them using a PKCS, and gives them to Charlie, who will compute the function. Charlie also obtains Bob's input b , and then computes $f((x_0, x_1), b) = x_b$. Then it sends this result to Bob. Alice receives no output, neither does her simulator.

Let the simulator be called S . We will define a 1-out-of-2 OT protocol using the simulator. The procedure is going to be similar as what we described in the previous paragraph, so the simulator is going to be placed between Alice and Charlie. This simulator S will use a PKCS and will interact with the ideal function. Since we have been using ElGamal's PKCS, suppose S uses that scheme. S behaves as follows:

Algorithm 3.12 (1-out-of-2 OT protocol using a simulator).

- **Inputs:** Alice's inputs are $x_0, x_1 \in \{0, 1\}^k$. Bob's input is $b \in \{0, 1\}$.

- **Outputs:** Alice gets no output. Bob's output is $x_b \in \{0, 1\}^k$

1. S obtains Alice's inputs, (x_0, x_1) .
2. S sends (x_0, x_1) to Charlie.
3. Charlie receives Bob's input b . Then, it selects x_b from the inputs it had received from the simulator.
4. Charlie sends x_b to Bob.

As we can observe, Algorithm 3.12 is a 1-out-of-2 OT protocol, but in the ideal world.

Security of the mentioned Algorithm is based on the idea of the indistinguishability of views. Then, we need Alice's views to be the same in both worlds, and Bob's selections to be also the same in the real and ideal worlds, because if he changed his selection from the real and ideal world, then the output would be different, as his view would also be different. Therefore, from Definition 3.9, his views must also be the same in both worlds.

Definition 3.13 (Security with simulators). An OT scheme involving participants Alice and Bob is secure if there exists a simulator S behaving as Alice (like in Algorithm 3.12) such that

$$(View_{A_{real}}, selection_{B_{real}}) \approx (View_{A_{ideal}}, selection_{B_{ideal}})$$

where $View_{A_{real}}, View_{A_{ideal}}$ denote to Alice's views in the real and ideal world, respectively, and $selection_{B_{real}}, selection_{B_{ideal}}$ to Bob's selection bit in those worlds.

With the definition of the real and ideal worlds, an OT protocol can be defined in both worlds, using the simulators in the ideal one, and observe if *security* is verified or not. If we consider Algorithm 3.7 and 3.12, it is easy to observe that Alice's views are equal (in the ideal world, Alice is her simulator), and Bob's selection is also equal, because he is a semi-honest player.

3.3.2 Two-party Computation of XOR

We will now consider a boolean function, and its implementation as a two-party secure computation.

In this section, we implement the computation of the boolean function XOR , because it is going to be used along the project. Therefore, we want to define a secure computation of XOR , in which the previous definitions about *security* is verified. This is the exclusive-or operation, which needs two inputs, and its computation is true if one of its inputs is true, but not the other. That is, it is true if the inputs are different, and false if they are equal. We can define the function as

$$XOR : \{0, 1\}^2 \longrightarrow \{0, 1\} \quad (3.7)$$

which verifies that

$$XOR(a, b) = a \oplus b = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases} \quad (3.8)$$

We want to define a two-party secure protocol that computes (3.8). As the XOR function needs two arguments, let us suppose that Alice has an input $a \in \{0, 1\}$, and Bob another input $b \in \{0, 1\}$. The simplest case would be that Alice sends her input to Bob, and Bob evaluates $a \oplus b$. This contradicts the idea we had, in which none of the parties knows the other party's input. However, from the definition of XOR , Alice and Bob can easily guess the other one's input. For example, if $a \oplus b$ is given to Bob, he can simply compute $(a \oplus b) \oplus b = a$, and Alice could do the same and she would guess Bob's input. Therefore, for this protocol we need to change this fact, and we are going to base the security of the protocol in a comparison between the real and ideal worlds.

Remember that all parties are considered to be semi-honest, so what we want is that the computation of XOR (which would be considered as an ideal function in an ideal world) remains similar to the protocol we are presenting (which would be implemented in the real world).

The implementation in the ideal world computation is simple. The ideal function is the defined XOR , 3.8. It receives an argument from Alice, $a \in \{0, 1\}$, and another one from Bob, $b \in \{0, 1\}$. The output $XOR(a, b) = a \oplus b$ is sent to both parties. This would be the ideal function in an ideal world. Also, as it is an ideal world, simulators can also participate and behave as one of the parties.

A protocol that computes the XOR function between Alice and Bob in real world can be described as follows:

Algorithm 3.14 (Two-party computation of XOR).

- **Inputs:** Alice's input is $a \in \{0, 1\}$. Bob's input is $b \in \{0, 1\}$.
 - **Outputs:** Alice's outputs are b and $a \oplus b$. Bob's outputs are a and $a \oplus b$.
1. Alice sends a to Bob.
 2. Bob sends b to Alice.
 3. Alice and Bob output the value of

$$XOR(a, b) = a \oplus b = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases} \quad (3.9)$$

Theorem 3.15. ([17]) *Algorithm 3.14 is indistinguishable from the computation of function XOR from the perspective of a semi-honest player. Specifically,*

$$\begin{aligned} \exists \text{ simulator } S_A / (\text{View}_{A_{real}}, \text{output}_{B_{real}}) &\approx (\text{View}_{A_{ideal}}, \text{output}_{B_{ideal}}) \text{ or} \\ \exists \text{ simulator } S_B / (\text{View}_{B_{real}}, \text{output}_{A_{real}}) &\approx (\text{View}_{B_{ideal}}, \text{output}_{A_{ideal}}) \end{aligned}$$

where S_A is a Bob simulator and S_B an Alice simulator.

Proof. Let us consider a PPT simulator in the ideal world S_A placed between Alice and the function XOR . This simulator will behave as Alice. Therefore, Alice sends her input a to S_A , and S_A sends it to the function, which receives Bob's input also. Then, the simulator receives the value of $a \oplus b$ computed by the function. S_A calculates $a \oplus (a \oplus b) = b$ and sends it to Alice. Therefore, Alice will know the value of b .

With this procedure, we achieve that $\text{View}_{A_{real}} \approx \text{View}_{A_{ideal}}$, because in the real world Alice knows her input value a and learns the value b (step 2 from Algorithm 3.14), as in the ideal world. Besides, $\text{output}_{B_{real}} \approx \text{output}_{B_{ideal}}$, because in the ideal world, Bob's output is the computation of XOR function. In the real world, since Bob is semi-honest he will follow the steps defined in Algorithm 3.14, and his output will be the same, $a \oplus b$.

Similarly, there exists a simulator S_B placed between Bob and the function XOR that performs a similar procedure, because Alice and Bob behave similarly in this protocol. Accordingly, Bob will always send the value of b , and will always learn the value of a . \square

With this, we can conclude that a XOR protocol has been correctly defined, and this XOR function will be used along the project.

Chapter 4

1-out-of- n OT

4.1 General Definition

We have been discussing the simplest OT case in the previous chapter, the *1-out-of-2 OT* protocol. After that protocol, a development of that work was presented in 1987 by G. Brassard, C. Crépeau, and J.-M. Robert in [3]. This method was called *1-out-of- n OT*, and following the notation we used in the previous chapter, now Alice has n inputs, $x_1, x_2, \dots, x_n \in \{0, 1\}^k$, and Bob has one, $b \in \{1, 2, \dots, n\}$. Alice sends Bob all her inputs, and Bob can just look at one, denoted x_b . With this procedure, Bob knows nothing about the rest of the elements that Alice sent him, and Alice knows nothing about Bob's output.

Any *1-out-of- n OT* protocol must verify the next properties:

1. *Correctness*: Both sender and receiver obtain what they are supposed to obtain at the end of the protocol. That is, Bob obtains the output x_b of the function, while Alice gets no output.
2. *Receiver's security*: We know that Alice does not get an output in an *oblivious transfer* protocol. Therefore, for any of Alice's inputs, $x_1, x_2, \dots, x_n \in \{0, 1\}^k$, and any $b, b' \in \{1, 2, \dots, n\}$, Alice's views in case Bob tries to obtain the value of x_b or $x_{b'}$ are computationally indistinguishable.
3. *Sender's security*: The sender's security is defined by making a comparison between the real and the ideal model exposed in the previous chapter. As Charlie is a trusted party, he can participate in the ideal model run of the protocol. Now, Alice sends her n messages to Charlie, and Bob sends his input value b to Charlie too. Then, Charlie computes the function and sends the output x_b to Bob. We discussed this was the most secure way of implementing the protocol, because Bob has no choice of receiving any extra information than the output. If Bob were a malicious player, Alice's security would be guaranteed if there exists a simulator S that interacts with Charlie and its output is computationally indistinguishable from Bob's output. That is because in the ideal model, where we use the simulator and Charlie participates, the sender's security is guaranteed. Therefore, if we had a real world run of the protocol where Bob's output is computationally indistinguishable from Charlie's output, then the real world run of the protocol would be computationally indistinguishable to its run in the ideal world. We define sender's security differently from receiver's security because the receiver gets some information from the sender, and we do not want the receiver to obtain any extra information from the sender.

If a protocol does not verify the above requirements, it is not useful for communication because it would not be secure. Therefore, across this chapter, we prove that each defined protocol verifies all properties. The material presented along the chapter has been consulted in [3, 8, 13, 18].

4.2 1-out-of-n OT Protocols

We will assume that now on, it is $n = 2^l$ for simplicity, because we will use pairs of elements. To implement a 1-out-of-n OT protocol, we can call the 1-out-of-2 OT protocol several times, but the operating cost would be huge. This implementation would have a difference with the one we described in the previous chapter, in Algorithm 3.7. Since the idea is that Bob receives $n = 2^l$ messages, in step number 2, in which Alice sends Bob her encrypted messages, she will send him the n encrypted messages instead of just two. Then, Bob will select the one he wishes and decrypt it.

We present two 1-out-of-n protocols in this chapter, and both will verify *correctness*, *sender's security* and *receiver's security*. One of the protocols is closely similar to ElGamal's protocol, but Alice encrypts n messages. The other protocol uses hash keyed functions, whose complexity is lower and they are defined along the project. In both protocols, we consider a sender Alice and a receiver Bob, with no trusted third party. Alice's inputs are $x_1, x_2, \dots, x_n \in \{0, 1\}^k$, Bob's input is $b \in \{1, 2, \dots, n\}$, and it is $n = 2^l$, $l \in \mathbb{N}$.

4.2.1 1-out-of-n OT Protocol as a Variant of ElGamal

Let \mathbb{F}_p^* be a multiplicative cyclic group of order p , where p is a large prime. Consider two generators, fixed by Alice, $g, h \in \mathbb{F}_p^*$, $g \neq h$. It is evident that both generators are different to 1. Moreover, since Alice is a semi-honest player, she will not select h as a power of g , so the discrete logarithm $\log_g h$ is unknown to all participants in the communication. Therefore, g and h are going to be used repeatedly over the communication. Also, since we consider all players as semi-honest, they will follow the steps of the protocol and they will not try to cheat. With this ideas, we can describe the first 1-out-of-n OT protocol.

Algorithm 4.1 (1-out-of-n OT protocol).

- **Inputs:** Alice's inputs are $x_1, x_2, \dots, x_n \in \{0, 1\}^k$, and Bob's input is $b \in \{1, 2, \dots, n\}$.
- **Outputs:** Alice gets no output, and Bob's output is $x_b \in \{0, 1\}^k$.

1. Alice chooses $g, h \in \mathbb{F}_p^*$ and sends them to Bob.
2. Bob, that has received $g, h \in \mathbb{F}_p^*$, verifies that $g \neq 1$ and $h \neq 1$.
3. Bob selects a random value $k \in \mathbb{F}_p$, and computes $y = g^k h^b \text{ mod } p$. Then, Bob sends y to Alice.

4. Alice computes, for all $i \in \{1, 2, \dots, n\}$, $c_i = (g^{k_i}, x_i(yh^{-i})^{k_i})$, where $k_i \in \mathbb{F}_p^*$ for all $i \in \{1, 2, \dots, n\}$, and all k_i are chosen randomly by Alice. Then, Alice sends all c_i to Bob.
5. Bob receives c_1, c_2, \dots, c_n , and selects $c_b = (a, r)$. To decrypt the message, Bob computes $x_b = ra^{-k}$.

As we can see, this protocol is the same as computing ElGamal's protocol several times. The encryption and decryption of the messages are done as in ElGamal, the only difference is that Alice encrypts n messages, not just one. As encryption and decryption follow the same steps as ElGamal, it is easy to see that Bob receives the initial message correctly. That is, Bob receives the encrypted message

$$c_b = (a, r) = (g^{k_b}, x_b(yh^{-b})^{k_b}) \quad (4.1)$$

Then, to decrypt the message, Bob computes

$$ra^{-k} = x_b(yh^{-b})^{k_b}(g^{-k_b})^k = x_b \quad (4.2)$$

Therefore, since Bob obtains the initial value of x_b correctly, then *correctness* is verified by the previous protocol.

Theorem 4.2 (*Receiver's security*, [18]). *Bob's choice b is unconditionally secure.*

Proof. As we defined, Bob's input is an element $b \in \{1, 2, \dots, n\}$, and he computes $y = g^k h^b$, so there exists a random value $k \in \mathbb{F}_p$ that Bob selects and satisfies that $y = g^k h^b$. Thereupon, following the DDH assumption, even if Alice had unlimited computer power, she could not get any information about Bob's input b . In conclusion, *receiver's security* is guaranteed. \square

Theorem 4.3 (*Sender's security*, [18]). *If the DDH assumption is verified, then for all $i \in \{1, 2, \dots, n\}$, where $i \neq b$, each $e_i = (g, h, c_i)$ is computationally indistinguishable from $e_b = (g, h, c_b)$ with $g, h \in \mathbb{F}_p$ and $g \neq 1, h \neq 1$, and the sender's security is verified.*

Proof. Following the notation we used in the protocol, we can denote $c_i = (a_i, r_i)$, and $c_b = (a, r)$, where $a_i, r_i, a, r \in \mathbb{F}_p$.

We know that Bob has two values, his input b and a random value k . By the DDH assumption, it is easy to see that Bob can not select any different pair (k', b') verifying that $y = g^k h^b = g^{k'} h^{b'}$. If Bob could do this, he would be able to compute the discrete logarithm $\log_g(h)$. This would be a contradiction because we have assumed that the discrete logarithm is unknown by both, Alice and Bob.

Again, by the DDH assumption, we can see that for all $i \in \{1, 2, \dots, n\}$, $i \neq b$, $e_i = (g, h, c_i)$ is computationally indistinguishable from a variable calculated as $E_i = (g, h, g^{k_i}, x_i(g^k h^{b-i})^{k_i})$, where $k_i \in \mathbb{F}_p, g, h \in \mathbb{F}_p$ two generators.

Let $R = (r_1, r_2, r_3, r_4)$ be a set of random elements where $r_1, r_2 \in \mathbb{F}$ are generators, and $r_3, r_4 \in \mathbb{F}_p$. If E_i and R are computationally distinguishable by a probabilistic

polynomial time machine M , then some random elements of the form (g, g^a, g^b, g^{ab}) would be distinguishable from elements of the form (g, g^a, g^b, g^c) , where $g \in \mathbb{F}_p$ is a generator, and $a, b, c \in \mathbb{F}_p$. This is a contradiction with the DDH assumption, so the mentioned PPT machine M will never exist. Consequently, for any $i \in \{1, 2, \dots, n\}, i \neq b$, it verifies that each $e_i = (g, h, c_i)$ is computationally indistinguishable from $e_b = (g, h, c_b)$. Thereupon, we conclude that *sender's security* is also guaranteed. \square

4.2.2 1-out-of-n OT Protocol Using Pseudo-random Functions

For this protocol we have to consider some new definitions that we are going to use. In these following definitions we need to use the concept of black box. A black box is a system from which we do not know its functioning, we can just know its inputs and outputs, but not how it operates with them.

Definition 4.4 (Random functions). A random function or random oracle is a theoretical black box which answers to every question with a random response from its output domain.

Definition 4.5 (Pseudo-random functions). A pseudo-random function is a function that can not be distinguished from a random function by an observer that has access to the function in a black box manner. That is to say, the observer does not know how the function works, just its inputs and outputs. A pseudo-random function answers the same to the same question.

Definition 4.6 (Hash functions). A hash function is a function which takes an string of bits of an arbitrary length as input and transforms it into a fixed-size string of bits. A hash function is a one-way function, i.e., it is easy to compute, but its inverse is difficult to calculate.

In the 1-out-of-n OT protocol we present in this section, we use pseudo-random functions in order to encrypt the messages. We use those functions because their complexity is lower than the complexity of any PKCS, so the protocol described here has a lower complexity than the one presented in the previous section. We will assume that one-way hash functions can be modeled as pseudo-random functions, which is justified in [8].

Consider $F : \{0, 1\}^m \rightarrow \{0, 1\}^r$ to be a hash function, and k to be a key that we will use for encrypting the input of the function F , which we call to key the function. The keyed function is denoted F_k . Moreover, we have mentioned that the inverse of a hash function is difficult to calculate, so we have the same for keyed hash functions. That is, with keyed hash functions we can encrypt a value with a key, as we do with PKCS, but the operations are easier, so the complexity is lower, and those functions verify *Correctness* and *Security* (for the sender and the receiver). These latter properties will be studied after the definition of the protocol. By keying the function F , we construct a pseudo-random function that is applied to a value x from the input domain of F , and it verifies that $F_k(x) = F(x, k) \in \{0, 1\}^r$. That is the encryption of value x under key k .

We will encrypt each input with a combination of keys, such that for each input x_i , $1 \leq i \leq n$, $n \in \mathbb{N}$, we will encrypt x_i by computing $F_k(i)$ and xoring it with x_i . To

simplify, we assume that i is represented by $\log_2 n$ bits, $\{i_1, i_2, \dots, i_n\}$. This representation can be, for example, the binary representation of $i - 1$. Therefore, for this protocol Alice sends Bob a set of keys, and her inputs encrypted with those keys. Bob selects one of the keys, and he can calculate the inverse of the function keyed with the same key he has chosen.

Algorithm 4.7 ($1 - out - of - n$ OT protocol with pseudo-random functions).

- **Inputs:** Alice's inputs are $x_1, x_2, \dots, x_n \in \{0, 1\}^m$, and Bob's input is $b \in \{1, 2, \dots, n\}$.
- **Outputs:** Alice gets no output, and Bob's output is $x_b \in \{0, 1\}^m$.

1. Alice generates l random pairs of keys. That is,

$$(K_1^0, K_1^1), (K_2^0, K_2^1), \dots, (K_l^0, K_l^1)$$

2. Alice computes $y_i = x_i \oplus (\oplus_{j=1}^l F_{K_j^{i_j}}(i))$ for all $0 \leq i \leq n$, where $i_j \in \{i_1, i_2, \dots, i_n\}$, the bit representation of i .
3. Alice and Bob participate in a $1-out-of-2$ OT protocol on the pairs of strings (K_j^0, K_j^1) , for all $j \in \{1, 2, \dots, l\}$. Bob selects $K_j^{i_j}$.
4. Alice sends y_1, y_2, \dots, y_n to Bob.
5. Bob computes $x_b = y_b \oplus (\oplus_{j=1}^l F_{K_j^{i_j}}(b))$.

In the protocol, each K_j^r is a t -bit key to the pseudo-random function F_K , for all $1 \leq j \leq l$, and $r \in \{0, 1\}$. Note that Bob could not compute any other value $x_i \neq x_b$ because he just has the key $K_j^{i_j}$ selected on step 3, and that is the only key with which he can calculate the function.

The implementation of this protocol allows Bob to obtain any value he wishes. Moreover, since we defined $n = 2^l$, then, as we have l pairs of keys which are exchanged in a $1-out-of-2$ OT protocol, we perform that protocol $\log_2 n$ times in the $1-out-of-n$ OT algorithm. We already know that the $1-out-of-2$ OT protocol verifies the *receiver's security*, thus, as we implement that protocol $\log_2 n$ times, we can conclude that it also maintains the *receiver's security*. We have to prove that the *sender's security* is also maintained, and in order to prove that we need to introduce a lemma.

Lemma 4.8. ([13]). *If the sender's security is not preserved in the previous protocol, then either the 1-out-of-2 OT protocol does not provide sender's security or the function F is not a pseudo-random function.*

Proof. Suppose the $1-out-of-2$ OT protocol guarantees the *sender's security*. Thus, since the *sender's security* is verified in the $1-out-of-2$ OT protocol, then the receiver can just look one of the two inputs of the sender. From the definition of the *sender's security*, we have to make a comparison between the real and ideal model. We have to prove that, for every PPT machine M_R substituting the receiver in the real model, there exists a PPT

machine M_I in the ideal model playing the receiver role such that M_R and M_I 's outputs are computationally indistinguishable.

We are going to show how, given a black-box access to the receiver M_R , we can extract the indices of the keys he has learned in step 3 of Algorithm 4.7. Knowing these indices allows us to identify the item from the server's inputs that has been learned by M_R in Algorithm 4.7. Thus, we can construct a PPT machine M_I in the ideal model that plays the receiver's role and whose output is computationally indistinguishable from M_R 's output.

To extract the mentioned indices, we will implement the protocol until the end of step 3. Then, fix the state of M_R by that time, and run different experiments (closely similar to the algorithm) that start in that state. To find the keys that the receiver learned, we will run $2l$ experiments, and those will be called $\{E_{i,j}/i \in \{0,1\}, 1 \leq j \leq l\}$.

In each experiment $E_{i,j}$ choose a random key r and replace the values $F_{K_j^i}(I)$ in the generation of the y_I 's (Alice's encrypted inputs), with $F_r(I)$. Suppose the receiver learns K_j^i in Step 3 of the protocol. Then, his view of the experiment, where the random key is used, is different by a non-negligible difference from his view in the runs of the protocol that use the original keys. That is because the functions are pseudo-random. Note that, for every j , this will happen at most to one of the experiments $E_{0,j}$ and $E_{1,j}$ (if this happened for both experiments, then the *sender's security* were not guaranteed in the 1-out-of-2 OT protocol). Therefore, in at most one of the experiments $E_{0,j}$ and $E_{1,j}$ the output's distribution of the receiver is different (by a non-negligible difference) than his output in the original protocol.

Let $t_0^j \in \{0,1\}$ be equal to the index $i \in \{0,1\}$ for which the above phenomenon happens. After we finish the experiments, we can define $t_0 \in \{1, \dots, n\}$ as the concatenation of the bits t_0^j , being $j \in \{1, \dots, l\}$.

Now, with this procedure and the algorithm of M_R , we will construct an algorithm for the PPT machine M_I that runs in the ideal model and acts as the receiver. Since it is the ideal world, then Charlie can participate in the protocol. It behaves as follows:

1. M_I generates a set of keys as in step 1 from the protocol, and then it engages in a 1-out-of-2 OT protocol with M_R .
2. M_I extracts t_0 , as we described above, which corresponds to the keys that M_R learned.
3. M_I asks Charlie for the value of x_{t_0} .
4. M_I sends y_1, y_2, \dots, y_n to M_R . Here, y_{t_0} is the encryption of the keys whose index is t_0 , and any other y_J is an encryption of a random value with the keys corresponding to the index J .
5. M_I sends y_1, y_2, \dots, y_n to M_R . y_{t_0} is the encryption of x_{t_0} with the keys corresponding to t_0 , and the rest of the values are encryption of random values with keys corresponding to their index.

We have to prove that M_R and M_I 's outputs are computationally indistinguishable. Suppose they are computationally distinguishable.

There exist l keys that M_R did not learn, let's say, $K_1^{u_1}, \dots, K_l^{u_l}$. We can define $l + 1$ elements, which we call hybrids, H_1, \dots, H_{l+1} . A hybrid H_i corresponds to running the protocol and, in Step 4, sending the values y_1, \dots, y_n to the receiver with one difference: for all $J \neq t_0$, y_J is generated with random values instead of F 's outputs keyed by $K_1^{u_1}, \dots, K_l^{u_l}$.

Hence, H_{l+1} corresponds to running the receiver in the original protocol. Then, if F is pseudo-random, then H_1 corresponds to running M_R using the algorithm M_I . This is due to the fact that the difference between two instances is that in H_1 each y_J , $J \neq t_0$, is generated by xoring x_J with a random value, and in the invocation of M_R , those values are generated by xoring the real outputs of F with random values.

Consequently, H_1 and H_{l+1} 's outputs are computationally distinguishable. Thus, there are two hybrids H_i and H_{i+1} , $1 \leq i \leq l$, which are computationally distinguishable. This means that there exists a distinguisher between random values and the output of F when it is keyed by $K_i^{u_i}$. This way, F would not be a pseudo-random function. \square

With this, we conclude that the protocol defined above provides *sender's security*. Therefore, it is proven that the protocol is secure and correct.

Furthermore, the complexity of the protocol corresponds to $n \log_2 n$ evaluations of the function F_K in the preprocessing of the step 1, and $\log_2 n$ calls of the *1-out-of-2 OT* protocol in the step 3.

Chapter 5

OT On Commitment Protocols and Zero-knowledge Proofs

References from this chapter have been taken from [9].

5.1 Introduction

In this chapter, we introduce some new concepts to define commitment protocols and zero-knowledge proofs. As in the previous chapters, we consider a sender and a receiver in the communication, Alice and Bob.

A commitment protocol is a procedure in which two players, Alice and Bob, participate. An activity is going to be performed by one of the parties. Before the activity occurs, one of the parties is going to commit to a result and is going to send the commitment to the other party, but this party will not be able to know that commitment until the end. At the end of the action, they both will know the result of the activity. For example, suppose that event is to flip a coin. If they are in the same room, face to face, Bob can make a prediction (a commitment) about how the coin is going to end, that is, heads or tails, while Alice is flipping the coin. At the end of the procedure, Alice will show Bob the result and they will see if it matches the commitment that Bob made. They can not cheat, because they are in the same room, seeing how the event happens, so they will see the result together. But if they were in different rooms, cities, or even countries, Bob could select a value without telling Alice and, after Alice told him the result, Bob could cheat and say that it is what he had thought or not. Or contrary, after Bob told Alice the selected value, Alice could lie to Bob and say that it was the result obtained on the activity or it was not, such that she could benefit from it. Therefore, with a commitment protocol we achieve the goal of performing an action between two participants providing that none of the participants cheat to the other one.

A zero-knowledge proof is a protocol in which one of the participants (the prover) proves to the other participant that a statement is true (or false) without revealing any extra information about the statement. This second participant will just be able to know that the statement is true (respectively, false). For instance, using again the coin flipping example, Alice could prove Bob that she knows the result of the action of flipping the coin, but she would not tell him the result. That would be a zero-knowledge proof.

At the end of this chapter we explain two different *oblivious transfer* protocols: *Oblivious circuit evaluation* and *Oblivious decommitment of strings*. Both of them use also Algorithm 3.7, or any of the *1-out-of-n OT* protocols we presented in Chapter 4. Before this, we need to define some new ideas and properties that we are going to use in order

to implement these mentioned protocols.

5.2 Width 5 Permutation Branching Programs

In order to implement the commitment protocol and zero-knowledge proofs we described above, we need first to define an element that is going to be used. That is the *width 5 permutation branching programs* (denoted W5PBP) of length k , [9], in which we are going to use permutations of 5 elements, denoted S_5 .

Also, we will have a function $E : S_5 \rightarrow \{0, 1\}$ that is used for the evaluation of the W5PBP and it verifies that

$$E(x) = \begin{cases} 0 & \text{if } x = Id \\ 1 & \text{if } x \neq Id \end{cases} \quad (5.1)$$

where Id is the identity permutation of order 5.

Moreover, we will use two functions Π and V such that:

$$\Pi : \{1, 2, \dots, k\} \times \{0, 1\} \rightarrow S_5 \quad \text{and} \quad V : \{1, 2, \dots, k\} \rightarrow \{1, 2, \dots, n\} \quad (5.2)$$

Definition 5.1 (Width 5 permutation branching programs). A width 5 permutation branching program M of length k with inputs $x_1, x_2, \dots, x_n \in \{0, 1\}$

$$M : \{0, 1\}^n \rightarrow \{0, 1\} \quad (5.3)$$

whose evaluation is defined as:

$$M(x_1, x_2, \dots, x_n) = E(\Pi(1, x_{V(1)}) \circ \Pi(2, x_{V(2)}) \circ \dots \circ \Pi(k, x_{V(k)})) \quad (5.4)$$

Since it is of length k , there exist k variables from the input domain of the W5PBP that will be seen. W5PBP are used because they can recognize any language in class NC^1 , which is proved in Barrington's theorem in [1]. Therefore, if we have a function of order NC^1 , there exists at least one *width 5 permutation branching programs* whose evaluation is the same as evaluating the mentioned function.

Π designates the permutations of 5 elements in each level $(1, 2, \dots, k)$ of the W5PBP, and V designates which k variables are looked by an observer. That is, $\Pi(i, x_{V(i)})$ is a permutation of five elements in level i , and $x_{V(i)}$ is the i th variable which is looked. That means, we have n variables which are the inputs of the W5PBP, and the program has length $k \leq n$, so an observer from the communication scheme will see just k variables.

By using a vector $R = \{r_1, r_2, \dots, r_{k-1}\}$, $r_i \in S_5$, the permutations defined by Π can be randomized such that the evaluation of its randomization is the same as the evaluation of the W5PBP. Let us consider a W5PBP M , a new one can be defined, called the randomized program of M .

Definition 5.2 (Randomized W5PBP). The randomized program of M , denoted $M_R = (\Pi_R, V)$, is a W5PBP such that

$$\Pi_R(i, a) = r_{i-1}^{-1} \circ \Pi(i, a) \circ r_i \quad (5.5)$$

for $i \in \{1, 2, \dots, k\}$, $a \in \{0, 1\}$, and r_0, r_k are defined as the identity permutation.

Let us see if with this definition it is verified that $M(x_1, x_2, \dots, x_n) = M_R(x_1, x_2, \dots, x_n)$. We know, from Definition 5.1, that

$$M(x_1, x_2, \dots, x_n) = E(\Pi(1, x_{V(1)}) \circ \Pi(2, x_{V(2)}) \circ \dots \circ \Pi(k, x_{V(k)})) \quad (5.6)$$

Also, from Definition 5.2, we have that

$$\begin{aligned} M_R(x_1, x_2, \dots, x_n) &= E(\Pi_R(1, x_{V(1)}) \circ \Pi_R(2, x_{V(2)}) \circ \dots \circ \Pi_R(k, x_{V(k)})) = \\ &E(r_0^{-1} \Pi(1, x_{V(1)}) r_1 \circ r_1^{-1} \circ \Pi(2, x_{V(2)}) \circ r_2 \circ \dots \circ r_{k-1}^{-1} \circ \Pi(k, x_{V(k)}) \circ r_k) = \\ &E(r_0^{-1} \circ \Pi(1, x_{V(1)}) \circ \dots \circ \Pi(k, x_{V(k)} \circ r_k)) = \\ &E(\Pi(1, x_{V(1)}) \circ \Pi(2, x_{V(2)}) \circ \dots \circ \Pi(k, x_{V(k)})) \end{aligned} \quad (5.7)$$

So, as we can see, the result is the same in both cases.

5.3 Committal Protocol Using Oblivious Transfer

Each committal protocol has two phases: a commit phase and a decommit one. During the commit phase, Alice commits to a value x , and Bob learns nothing about that value during the protocol execution of that phase. It is in the decommit phase when Bob learns the value of x and is sure that the value he has learned is indeed the value of x .

5.3.1 Simple Committal Protocol

First of all, we present a simple committal protocol, which is used in order to commit to a single bit b . This protocol consists on two algorithms, one for the commitment to the bit, and another one for the decommitment.

In the commit algorithm we will use an OT channel, which is a channel by which not all the bits are received by the sender. It is called OT channel because some of the bits will not be received, but Alice will not know which of them were correctly received or not by Bob, and Bob will know nothing about the bits he did not receive. We will denote p the probability of a bit to be transmitted correctly through the OT channel, and k will be the security parameter, a large enough number that verifies that the protocol is secure. A set of bits will be sent through an OT channel.

In the decommit phase we will use a clear channel, which is a channel by which all elements are received correctly by Bob. In this decommit phase, Bob will know the value of the bits he did not receive.

Algorithm 5.3 (Simple-Commit(b, k, p)).

- **Inputs:** Alice's input is b , the bit to which she commits. Bob has no input. p is the probability for a bit to be transmitted correctly through the OT channel.
 - **Outputs:** Alice gets no output, and Bob's output is the value of the vector received by the OT channel, $\vec{b}' \in \{0, 1, \#\}^k$, where $\#$ means that that bit was not received.
1. Alice selects a random vector $\vec{b}_c = b_1, b_2, \dots, b_k$, such that $b = b_1 \oplus b_2 \oplus \dots \oplus b_k$.
 2. Alice sends \vec{b}_c to Bob by the OT channel.
 3. Bob receives $\vec{b}' = b'_1, b'_2, \dots, b'_k$.

Algorithm 5.4 (Simple-Decommit(\vec{b}_c, \vec{b}', k)).

- **Inputs:** Alice's input is $\vec{b}_c = b_1, b_2, \dots, b_k$, the vector she sent on the simple-commit. Bob's input is $\vec{b}' = b'_1, b'_2, \dots, b'_k$.
 - **Outputs:** Alice gets no output, and Bob's output is the value of b .
1. Alice sends $\vec{b}_c = b_1, b_2, \dots, b_k$ to Bob through a clear channel.
 2. Bob receives $\vec{b}_c = b_1, b_2, \dots, b_k$. If $b_i \neq b'_i$ for any $i \in \{1, 2, \dots, k\}$, $b'_i \neq \#$, then Bob will stop the communication because Alice would have changed the bit b_i .
 3. If $b_i = b'_i$ for every $i \in \{1, 2, \dots, k\}$, then Bob computes $b = b_1 \oplus b_2 \oplus \dots \oplus b_k$. The bits he did not receive are substituted by the bits he receives now.

We would like that the probability of Bob recovering b before the decommit phase to be zero, and the same with the probability of Alice causing Bob to compute a different bit from the one she committed to. Unfortunately, this is almost impossible because it always exists a small probability of any of the parties cheating (they are semi-honest), so we would like both probabilities to be low.

Lemma 5.5. *At the end of Simple-Commit(b, k, p), the probability that Bob can recover b is p^k . In Simple-Decommit(\vec{b}_c, \vec{b}', k), the probability that Alice can cause Bob to compute a different bit (without Bob noticing about this) than the one she committed to is at most $1 - p$.*

Proof. In Simple-Commit(b, k, p), Alice sends $\vec{b}_c = b_1, b_2, \dots, b_k$ through the OT channel to Bob. The probability that Bob receives b_i , $1 \leq i \leq k$, is p for all i . Thus, the probability that Bob receives all bits b_i is p^k .

In Simple-Decommit(\vec{b}_c, \vec{b}', k), Alice sends \vec{b}_c to Bob through a clear channel. As p is the probability that a bit is transmitted by the OT channel, then $1 - p$ is the probability that a bit is not transmitted by the OT channel.

If Alice decides to change the value of a bit b_i , for some i such that $1 \leq i \leq k$, then, since the probability that Bob did not receive that bit is $1 - p$, then the probability for Alice cheating on that bit is $1 - p$.

If Alice decides to change the value of n bits, $1 \leq n \leq k$, the probability for Alice to be successful is at most $(1 - p)^n$.

Therefore, as $(1 - p)^n \leq 1 - p$, the probability that Alice can cause Bob to recover a bit different than what it committed is at most $1 - p$. \square

Therefore, when k is bigger, the previous probabilities are lower.

5.4 Permutation Randomizers, Condensers, and Tableaus

Some new definitions and ideas are introduced in this section, [9]. These ideas will let us define some other commitment protocols along with zero-knowledge proofs.

Definition 5.6 (Permutation randomizer). Fix $m, n \in \mathbb{N}$, with $n = 2^k, k \in \mathbb{N}$. A permutation randomizer, denoted $R \in \mathcal{R}_n^m$, is an $(m(n - 1) + 1) \times n$ array of permutations in S_5 . We denote $R_{i,j}$ the element in the i th row and j th column from the randomizer R , and \mathcal{R}_n^m is the set of randomizers with the mentioned dimensions. $R \in \mathcal{R}_n^m$ is a randomizer if it verifies that for all $0 \leq i < m$, and $0 \leq j < n - 1$ then:

1. If $k \neq j, j + 1$, then
$$R_{i(n-1)+j,k} = R_{i(n-1)+j+1,k}. \quad (5.8)$$

2. For $l = i(n - 1) + j$, then
$$R_{l+1,j}R_{l+1,j+1} = R_{l,j}R_{l,j+1}. \quad (5.9)$$

Definition 5.7 (Distribution on permutation randomizers). Let a sequence of elements of S_5 be denoted as $S = s_0, s_1, \dots, s_{n-1}$. The uniform distribution of permutation randomizers is defined as

$$\mathcal{R}_n^m[S] = \{R/R \in \mathcal{R}_n^m, R_{0,i} = s_i \quad \forall i, 0 \leq i < n\} \quad (5.10)$$

Definition 5.8 (Permutation condensor). Fix $n = 2^k$, where $k \in \mathbb{N}$. A permutation condensor, $C \in \mathcal{C}_n$, is an $(k + 1) \times n$ array of permutations in S_5 verifying that

$$C_{i+1,j} = C_{i,2j}C_{i,2j+1} \quad (5.11)$$

where $0 \leq i \leq k$ and $0 \leq j < \frac{n-1}{2}$.

Definition 5.9 (Permutation tableau). Fix $m, n = 2^k$, where $m, k \in \mathbb{N}$. A permutation tableau, $T \in \tau_n^m$, is a tuple $T = (R, C)$, where $R \in \mathcal{R}_n^m$, $C \in \mathcal{C}_n$, and it verifies that for every i , $0 \leq i < n$, $C_{0,i} = R_{m(n-1),i}$.

Definition 5.10 (Distribution on permutation tableaux). Let a sequence of elements of S_5 be denoted as $S = s_0, s_1, \dots, s_{n-1}$. The uniform distribution of the permutation tableaux is defined as

$$\tau_n^m[S] = \{T/T = (R, C) \in \tau_n^m, R \in \mathcal{R}_n^m[S]\}. \quad (5.12)$$

5.5 Introducing Zero-knowledge Proofs on Commitment Protocols

In this section, we show another commitment protocol, with a commit and a decommit phase, as before, and a zero-knowledge proof. Due to the fact that we also perform a zero-knowledge proof, this protocol will be separated into three phases: a commit phase, a decommit phase, and a check proof. Moreover, this time it is called *Strong-commit/decommit*, because if the zero-knowledge proof is not accepted, then the commitment either. Also, in contrast to the first protocol described in this chapter, the *Simple-commit* protocol, now Alice performs a *Strong-commit* algorithm along with a zero-knowledge proof over those bits, that are denoted $X = \{x_1, x_2, \dots, x_n\}$. Later, a *Strong-decommit* algorithm is performed, so Bob will receive the decommitment to the set of bits that Alice had committed to. Finally, the zero-knowledge proof is verified (or not), and if it fails, then Bob rejects the complete protocol. Which Alice proves is a predicate (or property) of the mentioned bits, and Bob verifies that proof on the last phase of the protocol. Proving this predicate using zero-knowledge proof has order NC^1 , and we will say that it is the order of the predicate. That predicate, denoted P , could be, for example, that the bits of X represent an even number. Furthermore, in order to verify the zero-knowledge proof that Alice proves to Bob, the randomizers, condensers and tableaux that we have previously defined are used, and Alice will select a tableau as the ones we explained above.

Each bit x_i from X can be written as a *XOR* operation over a sequence of bits, that is, $x_i = x_{i,1} \oplus x_{i,2} \oplus \dots \oplus x_{i,k}$. We will denote $[X]$ the matrix, that we will call bit-committal matrix, formed by the values $x_{i,j}$, where $1 \leq i \leq n$ and $1 \leq j \leq k$.

$$[X] = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,k} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,k} \\ \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,k} \end{pmatrix}$$

Let us consider a predicate P on X , and let us consider that its proved has a complexity of order NC^1 . The corresponding predicate on $[X]$ can be denoted P' , and its proved has also a complexity of order NC^1 , because the order of the *XOR* operation $x_i = x_{i,1} \oplus x_{i,2} \oplus \dots \oplus x_{i,k}$ is $O(1)$, so it does not increase the order of the predicate. Thus, because of Barrington's theorem mentioned at the beginning of the chapter, [1], there exists at least one W5PBP that evaluates P' on $[X]$. That W5PBP can be denoted $M_{P'} = (\Pi, V)$, and it has size $m = 2^l$, $nk \leq m \leq (nk)^c$, for some $c \in \mathbb{N}$. The W5PBP

can be forced to have its length as a power of 2 by padding it. Π is the same as it was before, and V is a function such that denotes the element from the matrix that is looked at, defined as

$$V : \{1, 2, \dots, m\} \rightarrow \{1, 2, \dots, n\} \times \{1, 2, \dots, k\} \quad (5.13)$$

Definition 5.11. ($S(M, [X])$). Let $[X]$ be a $n \times k$ matrix whose elements are bits, and $M = (\Pi, V)$ a W5PBP on $[X]$ of length $m = 2^l$. We define $S(M, [X])$ as the sequence s_1, s_2, \dots, s_m , where $s_i = \Pi(i, x_{V(i)})$.

We now can describe the *Strong-commit/decommit* protocol. We define, as before, a set of bits $X = \{x_1, x_2, \dots, x_n\}$, the predicate P' on $[X]$, and the W5PBP that evaluates P' on $[X]$, $M_{P'} = (\Pi, V)$, of length $m = 2^l$. Again, we are going to be using an oblivious transfer channel along the protocol. In this case, the probability of a bit to be transmitted across the OT channel is p .

Algorithm 5.12 (*Strong-Commit*(X, k, p)).

- **Inputs:** Alice's input is $X = \{x_1, x_2, \dots, x_n\}$. Bob has no inputs. The probability that a bit is transmitted through the OT channel is p .
 - **Outputs:** Alice gets no output. Bob's output is a $n \times k$ boolean matrix $[X']$ and a tableau T' , from which some elements have been lost in the transmission.
1. Alice selects a random $n \times k$ boolean matrix $[X]$, where $x_i = x_{i,1} \oplus x_{i,2} \oplus \dots \oplus x_{i,k}$, and a random tableau $T \in \tau_m^{m'}[S]$, where $m' \in \mathbb{N}$.
 2. Alice sends $([X], T)$ to Bob through the OT channel.
 3. Bob receives $([X'], T')$.

Algorithm 5.13. (*Strong-Decommit*($i, [X], [X']$)).

- **Inputs:** Alice's inputs are the the rows of the boolean matrix, $X_i = \{x_{i,1}, x_{i,2}, \dots, x_{i,n}\}$, for all $i \in \{1, 2, \dots, n\}$, and the tableau $([X], T)$. Bob's input is $[X']$.
 - **Outputs:** Alice gets no output. Bob's output is $x_i = x_{i,1} \oplus x_{i,2} \oplus \dots \oplus x_{i,k}$, for all $i, 1 \leq i \leq n$.
1. Alice sends, now through a clear channel, the rows $x_i = x_{i,1}, x_{i,2}, \dots, x_{i,k}$, for all $i, 1 \leq i \leq n$, to Bob.
 2. Bob receives the rows clearly. If $x_{i,j} \neq x'_{i,j}$ for any $i, 1 \leq i \leq n$, and $j, 1 \leq j \leq k$, then he stops the protocol. If not, he computes $x_i = x_{i,1} \oplus x_{i,2} \oplus \dots \oplus x_{i,k}$ for every i .

As we mentioned, this protocol included a zero-knowledge proof. For that reason, there is another part of the protocol, which is a check proof. Entry $x'_{i,j}$ from the matrix $[X']$ will be called again, and it may or not be defined, because it could have or not been received by Bob through the OT channel. Elements $R'_{i,j}$ and $C'_{i,j}$ may be defined, if Bob has received all their bits, or undefined, if Bob has received just some bits or none of them. This check protocol will be run by Bob, because Alice is the one who made the proof, and Bob tries to verify it.

Check-Proof($[X'], T', M_{P'}$): Bob checks if:

- for $k \neq j, j + 1$, then $R_{i(n-1)+j,k} = R_{i(n-1)+j+1,k}$.
- for $q = i(m^2 - 1) + j$, then $R_{q+1,j}R_{q+1,j+1} = R_{q,j}R_{q,j+1}$.
- $C_{i+1,j} = C_{i,2j}C_{i,2j+1}$, $1 \leq i \leq m - 1$, $1 \leq j \leq \frac{m-1}{2}$.
- $C_{0,i} = R_{m^2(m-1),i}$, $1 \leq i \leq m - 1$.
- $R_{0,i} = \Pi(i, x_{V(i)})$, $1 \leq i \leq m - 1$.
- $C_{l,0} \neq Id$, $1 \leq l \leq k + 1$.

If all of these properties are verified, then Bob accepts the proof and the commitment. If any of them is not verified, Bob does not accept the proof neither the commitment.

Let us now consider that Bob always receives k -bits, and that he can designate in advance $Q = q_1, q_2, \dots, q_{k-1}$ positions that he will look. Those positions can be in $[X]$ or in the tableau T . In those positions, Bob can see the content in the matrix and the tableau.

Definition 5.14 (Bob's view). Let $[X]$ be a bit-committal boolean matrix chosen by Alice, as we described before. Let $Q = q_1, q_2, \dots, q_{k-1}$ be a set of positions that can be in the bit-committal matrix or in a tableau T . $View([X], Q)$ is the distribution of views that Bob sees, obtained by choosing the tableau T according to the strong-commit protocol, and sending Bob the values of $[X]$ and T at the position specified by predicate Q .

5.6 Oblivious Transfer

After having discussed all of these ideas, let us show how some *oblivious transfer* protocols, different to those that we have previously defined, can be performed. Unfortunately, the bibliography was limited and the extended paper of [9] was not found, so we show a brief explication of the procedures.

5.6.1 Oblivious Circuit Evaluation

Randomized branching programs will be used for evaluating functions in an oblivious transfer way, that is what we call *Oblivious circuit evaluation*.

Suppose f is a boolean function whose complexity is of order NC^1 , and Bob and Alice wish to compute $f(x_1, \dots, x_n)$, from which Alice knows some inputs, and Bob knows the rest. Since the function is of order NC^1 , there exists a W5PBP for evaluating the function. The evaluation of the function using the W5PBP is the same as computing $f(x_1, \dots, x_n)$.

Therefore, Alice and Bob agree on a W5PBP for evaluating f , of size k , and we denote it as $M^f = (\Pi, V)$. This W5PBP is defined by a permutation in S_5 . Its corresponding randomized program is $M_R^f = (\Pi_R, V)$, chosen randomly by Alice. Then, for all i , $1 \leq i \leq k$, such that Alice knows $x_{V(i)}$, she sends over $\Pi_R(i, x_{V(i)})$ to Bob. Besides, for all i , $1 \leq i \leq k$, such that Bob knows $x_{V(i)}$, Alice and Bob invoke a *1-out-of-2 OT* protocol, for example Algorithm 3.7. With this procedure, Alice's secrets are $\Pi_R(i, 0)$ and $\Pi_R(i, 1)$, for all i , $1 \leq i \leq k$. Bob selects just $\Pi_R(i, x_{V(i)})$, from which Bob knows the value of $x_{V(i)}$, for all i , $1 \leq i \leq k$. This way, Alice and Bob learn nothing about each other's inputs because they are using a *1-out-of-2 OT* protocol.

Thereupon, for every i , $1 \leq i \leq k$, such that Bob knows $x_{V(i)}$, Bob receives the value of $\Pi_R(i, x_{V(i)})$. These values are permutations, and Bob can just multiply all of them and verify if the result is the same as the W5PBP they both had agreed on, the canonical one.

Since they invoke a *1-out-of-2 OT* protocol for every i , $1 \leq i \leq k$, then Alice does not know the values that Bob selected, neither the result of the product. Moreover, let us informally show that Bob does not learn any other value. Let us denote $View_B(R, x_1, x_2, \dots, x_n)$ to the random variables that Bob learns after invoking the *1-out-of-2 OT* protocol, that is, the values $\Pi_R(i, x_{V(i)})$, for i , $1 \leq i \leq k$, such that Bob knows the value of $x_{V(i)}$, and the result of $f(x_1, x_2, \dots, x_n)$. It is easy to see that Bob's view corresponds to those variables he knows (his inputs), the ones he learns, and the evaluation of the function, $f(x_1, x_2, \dots, x_n)$. Thereupon, he knows no information about Alice's inputs.

In conclusion, any function f in NC^1 can be evaluated using this method, in which Alice learns nothing about Bob's inputs and the output of the function, and Bob learns nothing about Alice's inputs.

5.6.2 Oblivious Decommittal

We can define an *Oblivious decommittal* of strings. It is a decommitment algorithm in which Bob receives the decommitment from Alice, and Alice does not know what decommitment Bob receives. We can do it by combining the *Strong-committal* protocol and the *1-out-of-2 OT* protocol or the *1-out-of-n OT* protocol.

Alice commits to a set of strings (it can be 2 strings or n) using the *Strong-commit* Algorithm, and a property over those strings that she will prove to Bob by a zero-knowledge proof. Then, Alice and Bob engage in a *1-out-of-2 OT* protocol or a *1-out-of-n OT* protocol, depending on the number of strings she sends. Therefore, Bob will choose one string from the set he has received. We know that Bob will know nothing about the rest of strings, and Alice will know nothing about the string that Bob chooses.

Alice will later decommit the strings she had sent using the *Strong-decommit* algorithm and, again, a *1-out-of-2 OT* protocol or a *1-out-of- n OT* protocol. Therefore, Bob will be able to select the decommitment to the string he had selected on the commit phase, and he will receive the bits of the string he has. He will then perform a check-proof for accepting the string (or not). Moreover, since they have engaged on an *oblivious transfer* protocol, Alice will know nothing about the value that Bob selected, and Bob will know nothing about the values he did not select.

Chapter 6

Dealing with Malicious Players

Until now, we have considered all players to be semi-honest. We are now going to focus on malicious players, that have to be studied in a different way because of their behaviour. Just to remember, a malicious player is a player that does not follow the protocol and tries to cheat. It can send incorrect inputs, try to discover private information from other players, or even try to change the behaviour of the rest of players. Consequently, any player that acts differently from the protocol instructions is considered as malicious.

A malicious player can also take control over other parties, and those parties are called corrupted parties. In case there exists any corrupted party, it will be a fixed number that will not change over the execution of the protocol, and will maintain corrupted. Also, the malicious players will maintain their behaviour along the execution too. The malicious players can also be called adversaries. The parties that are not corrupted or malicious are going to be semi-honest along all the protocol execution.

The bibliography for this chapter has been taken from [6, 11].

6.1 Two-party Secure Computation Against Malicious Players

We showed that functions can be implemented with *oblivious transfer*. In order to deal with malicious players, we want to firstly define a protocol secure in the ideal model, so we will consider the evaluation of a function, defined as follows:

$$\begin{aligned} f : \{0, 1\}^n \times \{1, 2, \dots, n\} &\longrightarrow \{0, 1\} \\ (a, b) &\longmapsto f(a, b) = x_b \end{aligned}$$

where $a = (x_1, x_2, \dots, x_n)$ is a string of bits and b is a value $1 \leq b \leq n$. Alice will get no output and Bob will receive the value of $f(a, b) = x_b$.

Security will be defined as before, considering the real and ideal world. That is, we want the real world execution of the protocol to be the same as the ideal world execution of the protocol. We already know that in the ideal world a trusted third party can participate in the protocol. When we deal with malicious players we have to add that Charlie is also an incorruptible party, not just a trusted one, and no malicious player can take control over him. Let us suppose that f is the function that the sender and the receiver wish to compute. Each one has its own inputs, and they send them to Charlie, the trusted third party. Then, Charlie will evaluate the function with the values he receives from Alice and Bob. In this ideal protocol, a malicious player can deviate from the protocol by choosing

a different input and sending it to Charlie. Therefore, the output of the function would be computed with that different choice made by the malicious player, but, since Charlie is honest, the computation would be calculated correctly. Moreover, the malicious party could abort the execution of the protocol at any time.

Since just two parties participate in this protocol, we are going to consider that one of them is honest and the other is malicious. What we pretend is to protect the honest party against its adversary (the malicious party). To achieve this, we must first consider all the possible actions that a malicious player can do. That is, a malicious player can decide not to participate in the protocol, it can change its local inputs, and it can abort the protocol execution before it is finished. These are unpredictable actions that the malicious players can perform in an ideal model. Consequently, we describe an ideal model run of the protocol for malicious players that includes this unpredictable actions.

Algorithm 6.1 (Ideal model for malicious players).

1. Both parties send their inputs to Charlie. If Alice is a honest party (respectively Bob), she will send a (respectively b). If Alice is a malicious party (respectively Bob), she can send a or a' (respectively b or b').
2. If Charlie does not receive a pair (a, b) , then he sends a special symbol \perp to both parties. If he receives a pair (a, b) , then he sends $f(a, b)$ to Bob.
3. If Alice (respectively, Bob) is the malicious party, she might decide to stop Charlie before the end of the protocol. If this happens, Charlie sends Bob (respectively, Alice) the special symbol \perp . If Alice (respectively, Bob) decides not to stop Charlie, he sends $f(a, b)$ to Bob (respectively, Alice).

Suppose the function f is as we described above, and consider that $M = (M_1, M_2)$ is a pair of PPT machines that represents both parties participating in the protocol. Since one of the parties is honest, for a $i \in \{1, 2\}$, M_i will follow the protocol instructions, that is, it is the honest party. These pairs of machines, in which one of them is honest but not the other, are said to be admissible. Then, we will denote as $\text{IDEAL}_{f,M}(a, b)$ the joint distribution of f under M in the ideal model. This represents the output from the execution of the above Algorithm 6.1. For example, if we suppose that the first player (M_1) is the malicious party, we know that it can change its input, abort before finishing, or continue until the end with its real input or another different. If it decided to change its input and to send a' to Charlie, then the joint distribution would be $(M_1(a', \cdot), f(a', b))$. If it decided to abort before the end of the protocol, the joint distribution would be $(M_1(a, \cdot), \perp)$. Also, if it decided not to abort the execution, the joint distribution would be $(M_1(x, \cdot), f(x, b))$, where x is the input that M_1 decided to send, and it can be the real input or a different one. If the malicious player were the second party, it would be analogous.

In the real world, Charlie does not exist, so Alice and Bob will run the protocol. A real world protocol is considered to be secure if its output is the same as in the ideal model. We know that no malicious player can compute a successful attack in the ideal world. Therefore, it does not exist any secure protocol run in the real world where a malicious

player cheats successfully, because the output would be different from the output in the ideal world.

In the real world, malicious parties can follow any strategy, so we are not able to make a list of their possible actions. Therefore, we can not describe a precise algorithm with all their possible responses. We will have the same function f that both parties wish to compute, and we will consider the same pair of PPT machines $M = (M_1, M_2)$ that represents the first and second party in the real model, in which one of them is honest and the other one is malicious. A protocol \mathcal{P} will be followed by the honest party in order to perform the evaluation of the function f . As before, we denote the execution of \mathcal{P} by M in the real model as $\text{REAL}_{\mathcal{P},M}(a, b)$, and it is the output that results from the evaluation of f following protocol \mathcal{P} in the real model by M_1 and M_2 .

After we have defined how real and ideal models are implemented, we can finally define the security of a two-party computation with malicious players. We discussed above that a two-party computation in the real world is secure if its output is similar to the output on the ideal model, so we can now write a formal definition. Therefore, let us consider a function f and a protocol \mathcal{P} as above.

Definition 6.2 (Security in the malicious model). \mathcal{P} computes f securely in the malicious model if for every pair of admissible PPT machines $A = (A_1, A_2)$ in the real model, there exists a pair of admissible PPT machines $B = (B_1, B_2)$ in the ideal model, such that

$$\{\text{IDEAL}_{f,A}(a, b)\} \approx \{\text{REAL}_{\mathcal{P},B}(a, b)\}. \quad (6.1)$$

Therefore, we have defined the security for a two-party secure computation against malicious players. Note that this two-party computation is the same as an *oblivious transfer* protocol. What we want is to define an algorithm for a protocol like this against malicious players in the real world. Remember that we mentioned that we do not know the possible actions the malicious party can perform, so to solve this problem we will use the GMW compiler.

6.2 GMW Compiler

GMW compiler will let us implement a protocol in which the malicious players participate.

Definition 6.3 (GMW compiler). The GMW compiler is the set of steps that the participants must follow if they wish to perform a protocol. They will not be able to perform the protocol if they do not follow the steps.

To implement the compiler, we first need a protocol secure against semi-honest players. This protocol will be the compiler's input. Along the project we have defined several protocols and all of them are secure against semi-honest players. Therefore, any of those protocols could be used as the input of the compiler.

The idea of the compiler is to force the malicious parties to behave properly and follow the protocol. Again, we have two parties in the computation, and each one has, at least, one input, which is a string of bits.

Definition 6.4 (Coin-tossing protocol). A coin-tossing protocol is a protocol that generates a random bit with uniform distribution.

The compiler starts by having each party committed to its own input. Then, a coin-tossing protocol is run by the parties. That coin-tossing protocol must be secure against malicious players, and it is explained in [2]. The possible results of the coin-tossing protocol, heads or tails, are associated with a bit. For example, if the result is heads, it will be a number 1, and if it is tails, it will be a number 0 (or contrary). The protocol will be repeated several times for both parties, and the result will be a string of bits owned by Alice, and a string of bits owned by Bob. Alice will know nothing about Bob's string, and Bob will know nothing about Alice's string. Then, each party will commit to its string. Therefore, each party will have a commitment to the other party's input, its string of bits, and a commitment to the other party's string generated in the coin-tossing protocol. Thus, with the committed input and random string, each party can be forced to follow the protocol. In order to force them to behave correctly, the parties will use zero-knowledge proofs in each step to verify that they are following the protocol. Therefore, a malicious player could not deviate from the protocol because it would be detected. Thereupon, if the protocol is secure against semi-honest players, it will be secure against malicious players.

Let us now present the compiler procedure in a more precise way:

1. **Input commitment:** Let $x \in \{0, 1\}^k$ be the party's input string. This party computes $C(x)$, which is the commitment to the value of x , and sends it to the other party, along with a zero-knowledge proof of x . This proof will be denoted as λ . That is, the parties execute the following function

$$x \mapsto (\lambda, C(x)) \quad (6.2)$$

2. **Coin-tossing:** Let us assume that to commit to a t -bit string, the commitment protocol needs $t \cdot n$ random bits. The parties generate the mentioned string by computing a coin-tossing protocol

$$(t, t \cdot n) \mapsto (U_t, C(U_{t \cdot n})) \quad (6.3)$$

where they generate a t -bit long random string (U_t), and its corresponding commitment, $C(U_t)$. That is, one party receives a t -bit long random string, and the other party receives the commitment to that value, $C(U_{t \cdot n})$.

3. **Protocol perform:** Now, the parties can run any protocol secure against semi-honest players such as the ones we have been showing along all the project, any of the *1-out-of-2 OT* protocols, *1-out-of- n OT* protocols, or two-party secure computation. They will perform the chosen protocol sending zero-knowledge proofs of each step, their input strings and random strings generated in the coin-tossing protocol.

Each step in the compiler is committed or proven by zero-knowledge proof. Therefore, a malicious player could not cheat in this protocol, because if it changed one step, it would be detected. Hence, it can just behave as a semi-honest player or abort the execution of the protocol.

In addition, the commitment protocols mentioned along the chapter can be any of the proven commitment protocol, or the *Simple-commitment* protocol described in the previous chapter. Also, as a zero-knowledge proofs are also executed in some steps of the compiler procedure, the *Strong-commitment* protocol could be used too.

The protocols already described for semi-honest players in the previous chapters can be implemented this way, where each party's input would be its initial input, the string of bits. Therefore, we can conclude that, for implementing an OT protocol against malicious players, we can use the GMW compiler and, on its third step, implement the selected oblivious transfer protocol: the *1-out-of-2 OT*, *1-out-of- n OT* protocols, and two-party computation using OT.

Chapter 7

Uses of Oblivious Transfer

After the definition of some OT protocols, we will now illustrate some application of this method. Bibliography was taken from [13, 14].

7.1 PIR to SPIR Transformation

Suppose there is a person who owns a database that consists in n words, and another person, a searcher, wants to search in that database, without the owner noticing about it. This is called *Private Information Retrieval*, or PIR. The difference with *oblivious transfer* is that in PIR schemes there is a database where the values the searcher wants, it is not the owner who has them. If the searcher could just look one (or a fixed number) of those words, it is called *Symmetric Private Information Retrieval*. It is called symmetric because the database and the searcher have a privacy requirement. This can be constructed using a *1-out-of- n OT* protocol combined with a PIR one.

PIR schemes has been studied a lot, and has received much attention over the last years, due to the development of technology. At first it was constructed such that the searcher had to communicate with several servers, achieving that those servers could not send information between them, neither to the owner of the database. Kushilevitz and Ostrovsky proposed a PIR scheme, in [10], in which the searcher had to communicate just with one server. After that, they suggested to adapt their protocol with a single server in which the searcher could look as many words as it wanted, to a protocol with a single server in which the searcher could just look one word (or a fixed number of words smaller than the amount of words in the database). This is a SPIR scheme.

Since PIR schemes are focused on communication complexity (of order $O(n)$), and *1-out-of- n OT* protocols are focused on sender's privacy, combining both protocols we achieve the idea of having a communication complexity of order $O(n)$ in which *searcher's privacy* is guaranteed. Therefore, for a SPIR scheme we can use Algorithm 4.7.

Following the notation we have been using along the project, let us suppose that Alice is the owner of the database and Bob is the one who wants to select a word. Also, we will denote the words on the database as x_1, x_2, \dots, x_n , and Bob wants to look at x_b . In the SPIR protocol, Bob does not need to receive all the values y_1, y_2, \dots, y_n that Alice encrypted. Now it is enough for Bob to receive x_b . Therefore, in step 4 of Algorithm 4.7, Alice and Bob perform a PIR reading for y_1, y_2, \dots, y_n , [10]. With this, Bob has information enough for receiving y_b , that he can decrypt and guess the value of x_b .

7.2 Polynomial Evaluation

Oblivious transfer is also used for *Oblivious polynomial evaluation*. Again, there are a sender, Alice, and a receiver, Bob.

Let us consider a finite field \mathbb{F}_q , where $q \in \mathbb{N}$ is the number of elements of the field.

Alice and Bob want to evaluate a polynomial $P(\cdot)$ of degree d_P over \mathbb{F}_q , and Alice is the one who knows the polynomial. Bob has an input $b \in \mathbb{F}_q$. The output of the evaluation will be $P(b)$, and Bob is the one who will receive it. Alice will receive no output, and none of them will know each other's input.

If we consider a small integer n , the idea for evaluating polynomial $P(b)$ is that Alice chooses another bivariate polynomial for hiding hers, and then performs a *1-out-of- n OT* protocol with Bob. Therefore, let us suppose that the random polynomial selected by Alice for hiding P is $Q(x, y)$, also over the considered field. This new polynomial verifies that:

- The degree of variable y on Q , denoted $d_{Q,y}$, is the same as d_P , the degree of P .
- The degree of x in Q is $d_{Q,x}$, a multiple of d_P .
- $\forall y \in \mathbb{F}_q$, it is $P(y) = Q(0, y)$.

As Alice hides the polynomial, Bob will also hide his input b in another polynomial. Let us suppose that he selects a polynomial $S(x)$ of degree d_S over the finite field \mathbb{F}_q verifying that:

- $S(0) = P(b)$.
- $d_S = d_{Q,x}/d_P$.

The polynomial in which Bob wants to hide b is defined as $R(x) = Q(x, S(x))$, and, from the previous properties, it is easy to see that

$$R(0) = Q(0, S(0)) = S(0) = P(b) \quad (7.1)$$

and then the degree of polynomial $R(x)$ is $d_R = d_{Q,x} + d_P d_S = 2d_{Q,x}$.

Then, Bob chooses $m = d_R + 1$ points, all different to zero. Those points are denoted as x_1, x_2, \dots, x_m . For all i , $1 \leq i \leq m$, Bob chooses $n - 1$ random elements $y_{i,1}, y_{i,2}, \dots, y_{i,n-1}$ from \mathbb{F}_q . After this selection, he sends to Alice the values of x_i and the list $(y_{i,1}, y_{i,2}, \dots, y_{i,n-1}, S(x_i))$, for all $1 \leq i \leq m$. The elements from the list are permuted randomly, that is, Bob sends them with a different order. Let us denote the permuted list as $(z_{i,1}, z_{i,2}, \dots, z_{i,n-1}, z_{i,n})$.

Now, they perform a *1-out-of- n OT* protocol in which Alice sends Bob the n values of $Q(x_i, z_{i,1}), Q(x_i, z_{i,2}), \dots, Q(x_i, z_{i,n})$, for all $1 \leq i \leq m$. From these values, Bob chooses to look one $Q(x_i, S(x_i)) = R(x_i)$. This procedure is done m times, until Bob receives m messages. Thereupon, at the end Bob will know $\{R(x_i)\}_{i=1}^m$. With the m values, Bob can interpolate polynomial $R(x)$, and then calculate $R(0) = P(b)$.

With this process, Alice has not revealed the polynomial she owned, and Bob has not revealed his input value b , but he has guessed the value of $P(b)$.

Chapter 8

Conclusions

Oblivious transfer is a developed scheme that has been studied along the last 30 years. Michael O. Rabin presented a revolutionary idea for exchanging secrets, and it gave rise to some different studies and different models from the initial one. Moreover, a really important aspect from the OT protocol is that *participants' security* and *correctness* is verified in all the algorithm.

The most studied branches of this protocol are the ones of *1-out-of-2 OT* and *1-out-of- n OT*, that have been explained in Chapters 3 and 4. Both of those protocols have been implemented in different ways, from which I personally selected those that I considered interesting to expose in this project. Nevertheless, there exist more definitions of the mentioned protocols that were also considered to expose. Moreover, for those representations we have considered semi-honest players, because presenting the protocol in the first place with players like that is much easier (to explain and to understand).

In addition, the ideas described in Chapter 5 show more possibilities in which *oblivious transfer* can be performed, with the use of more complex arguments. Also, if the extended version of the consulted paper, [9], had been found, some precise protocols could have been described. Nevertheless, the main ideas for those protocols have been exposed.

With the complete ideas about the main protocols, malicious players could finally be considered. This type of players are basically forced to follow the steps in the communication, with the GMW compiler, so they can not make any illegal movement without the other player noticing it. The main ideas that helped us to force malicious players to follow the steps were the coin-tossing protocol along with the commitment protocols and zero-knowledge proofs the participants have to perform in each step, and which are very useful in cryptography.

As well, we showed some implementations of the *oblivious transfer* protocol. It is very useful for selecting a value from a database, and also for polynomial evaluation. These implementations have evolved along with technology and digitization, and more applications are developed nowadays, such as signing contracts, particular cases of zero-knowledge proofs, or constructions of black-boxes.

Due to the available space in the project, some other studies about *oblivious transfer* were not explained. For example, that is the case of *k -out-of- n OT*. Following the explained ideas and protocols, it is easy to see that it refers to a communication scheme between Alice and Bob in which Alice sends n messages to Bob, and Bob can just look k of those messages. For this protocol, one of the ideas is to repeat Algorithm 4.7, but the complexity would be huge. The other idea is to define a *1-out-of- n OT* protocol with a lower complexity than the ones we have studied, and to repeat the first step of that algorithm until a desired value is reached.

In conclusion, there are a lot of possibilities in order to implement *oblivious transfer*.

Although it is a recent communication scheme, it has evolved fruitfully and there are different studies about how to implement it.

Bibliography

- [1] David A Barrington, *Bounded-width polynomial-size branching programs recognize exactly those languages in $nc1$* , Journal of Computer and System Sciences **38** (1989), no. 1, 150–164.
- [2] Amos Beimel, Eran Omri, and Ilan Orlov, *Protocols for multiparty coin toss with dishonest majority*, Annual Cryptology Conference, Springer, 2010, pp. 538–557.
- [3] Gilles Brassard, Claude Crépeau, and Jean-Marc Robert, *All-or-nothing disclosure of secrets*, Conference on the Theory and Application of Cryptographic Techniques, Springer, 1986, pp. 234–238.
- [4] Nicholas Carlini, *Lecture 9: Public key encryption*, October 2014.
- [5] Shimon Even, Oded Goldreich, and Abraham Lempel, *A randomized protocol for signing contracts*, Communications of the ACM **28** (1985), no. 6, 637–647.
- [6] Oded Goldreich, Silvio Micali, and Avi Wigderson, *How to play any mental game, or a completeness theorem for protocols with honest majority*, Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali, 2019, pp. 307–328.
- [7] Jeffrey Hoffstein, Jill Pipher, Joseph H Silverman, and Joseph H Silverman, *An introduction to mathematical cryptography*, vol. 1, Springer, 2008.
- [8] Russell Impagliazzo and Steven Rudich, *Limits on the provable consequences of one-way permutations*, Proceedings of the twenty-first annual ACM symposium on Theory of computing, 1989, pp. 44–61.
- [9] Joe Kilian, *Founding cryptography on oblivious transfer*, Proceedings of the twentieth annual ACM symposium on Theory of computing, 1988, pp. 20–31.
- [10] Eyal Kushilevitz and Rafail Ostrovsky, *Replication is not needed: Single database, computationally-private information retrieval*, Proceedings 38th annual symposium on foundations of computer science, IEEE, 1997, pp. 364–373.
- [11] Yehuda Lindell, *Foundations of cryptography 89-856*, Electronic document (April 2006) (2010).
- [12] Wenbo Mao, *Modern cryptography: theory and practice*, Pearson Education India, 2003.
- [13] Moni Naor and Benny Pinkas, *Oblivious transfer and polynomial evaluation*, Proceedings of the thirty-first annual ACM symposium on Theory of computing, 1999, pp. 245–254.

-
- [14] Abhishek Parakh, *Oblivious transfer based on key exchange*, Cryptologia **32** (2008), no. 1, 37–44.
- [15] Francisco José Plaza Martín, *Manual de criptografía: fundamentos matemáticos de la criptografía para un estudiante de grado*, Manual de criptografía (2021), 1–116.
- [16] Michael O Rabin, *How to exchange secrets with oblivious transfer*, (1981).
- [17] Sanjeev Reddy, Tom Yurek, Sourav Das, and Jong Chan Lee, *Special topics in cryptography: lecture 06 & lecture 07*, University of Illinois, sep 2019.
- [18] Wen-Guey Tzeng, *Efficient 1-out-of- n oblivious transfer schemes with universally usable parameters*, IEEE Transactions on Computers **53** (2004), no. 2, 232–240.