U N I V E R S I T Y OF T A R T U

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Oleg Šelajev

The Use of Circuit Evaluation Techniques for Secure Computation

Master's Thesis

Supervisor: Sven Laur, D.Sc. (Tech.)

Author: ..... "....." June 2011

Supervisor: ..... "....." June 2011

Approved for defence

Professor: ..... "....." June 2011

TARTU 2011

Contents

Abstract

Secure two-party computation problem is about two parties that want to compute some function of their private inputs in a way that other party won't learn it. We describe a general way to perform secure two-party computation of a function specified as a boolean circuit, which was proposed by A.A. Yao in 1982. This method is named Yao garbled circuit evaluation and is secure against semi-honest adversaries. We present a new efficient protocol for secure two-party computation Circus, that is secure against malicious adversary in consistency model. Consistency model implies that either both parties will receive correct output and persist privacy of their inputs or a honest party will know, that is was cheated and that adversary potentially have learnt 1 bit of other party's input value. We specify all necessary sub-protocols and their security requirements and prove security of Circus in malicious environment.

Introduction

Digital circuits power all computations in the world. Every processor chip relies on them. Mathematical analog of a digital circuit is a boolean circuit which operates not on existence of electrical power, but on logical values. Generally speaking, boolean circuits are a powerful model and a very interesting area of research about computations. On the other hand, all modern computations are distributed and usually more than one party is involved. We combine our data to produce new results, but at the same time we put our privacy on the risk. Modern society bothers about information being private very much. So computations should be as secure as possible. In general, the problem of secure computations is widely known and a lot of research has been done to discover possibilities to compute while preserving privacy of own data. General ways to do that were known since 1982, when A.C.C.Yao proposed [1] a way how to use boolean circuits to guard computation against the adversary who will behave honestly, but investigate information available to find other parties inputs. However, if we allow the adversary to behave at its own will, Yao's solution does not guarantee security. Later there were many attempts to improve this result and several protocols that are secure against the adversary with any behavior were derived. Those techniques are not only computationally very intense and require a lot of time and computational power, but also it is not trivial to prove that using them preserves security.
In year 2006, Mohassel and Franklin proposed another extension to original Yao solution and described a protocol that could be secure and more efficient. However, they did not give a full proof of security of the proposed construction in their paper [2]. In this thesis, we describe in details the protocol proposed by them, compare it to other existing solutions and give a full proof that it is secure. Additionally, we describe and partially create an implementation of this protocol for the Sharemind platform and give some basic overview of its performance and how much it can be improved. In the next chapter, we describe the necessary concepts and primitives to construct a secure protocol and then transition to defining protocol flow and proving that it is secure.

1 Preliminaries

Cryptographic protocols are complex interactive computations that satisfy some security requirements and have desired security properties. Usually, to discuss security of a protocol quantitatively, a notion of adversary's advantage against the protocol is used. Advantage characterizes probability that a given adversary succeeds at a certain attack. It would be very difficult to analyze even a midsize protocol in detail, for example, prove that some security property holds for every possible adversary, unless we somehow abstract ourselves from low level implementation details. A common way to decrease complexity of proofs is to use well defined cryptographic primitives with given security properties as black box functionality placeholders [3]. In this case, we split complexity of the proof into having a real world implementation of the primitive and mathematical proof that a protocol using this primitive is secure (satisfies needed properties). This chapter describes the concepts and cryptographic primitives that we need to construct protocols for multiparty computation.

1.1 Time complexity

Time complexity of a computation describes how much time it takes to finish the computation. One obvious way to quantify time complexity is to define it in terms of the number of elementary operations needed for the computation. This approach is not very usable, because usually the number of operation is hard to determine exactly and what is more important it is different for different input sizes. Thus, usually when speaks about time complexity they want to asymptotically describe time behavior of a functions as the size of its input goes to infinity. This is a comfortable way to define time complexity, but not suitable for our needs. When we construct a cryptographic primitive we assume that solving a specific problem takes a specific number of elementary operations and conclude that those operations will take certain amount of real world time. Our operations have a fixed input size: key length, length of a number, etc - so we do not care about what could be function complexity on near-infinity sized inputs. Additionally, asymptotic notations tend to hide constant factors, which can result in orders size difference between expected time and real-world time. We need to define some natural ordering of functions based on their time complexity. So if we have some threshold t in the number of elementary operations that can be computed under time t, we can divide all functions with sizes of their inputs in two groups, those that can be computed in less than time t and those which cannot. In this thesis, we mainly use only this aspect of time complexity that given a threshold t, some functions cannot be computed under time t. When we talk on a higher level of abstraction than a single function computation it's not always comfortable to specify this time constraint for every function. So we say that a party is t-time if it can compute functions whose time complexity is not greater than t. Note that this t-time is a bound on a total time of party's computation, so it cannot compute multiple t-time functions in parallel still being a t-time party.
Time complexity is a very convenient way to bound the power of dishonest parties. We say that

if a party that does not follow a protocol or tries to obtain some knowledge it is not supposed to and it can compute functions which time complexity is not bigger than t is a t-time adversary.

1.2 Boolean circuit computation

Boolean function is a predicate, in other words, it is a function that takes some 1 bit values and returns 1 bit. Basic boolean functions are not ($\neg$), and ($\wedge$), or ($\vee$), xor ($\oplus$), eq ($\Leftrightarrow$), symbols show their mathematical notations. A boolean function can be presented as a table every row of which contains information about what is an output of the function for given argument values. Such a table is called a truth table of a boolean (logical) function. For example, following is the truth table for and function.

α	β	$\alpha \wedge \beta$
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: Boolean and truth table

Combination of boolean functions is obviously also a boolean function. This fact is used to combine several basic functions into a greater one. We just join outputs of one function to inputs of another one, this allows us to specify more sophisticated functions. Boolean circuit computation is a way to specify computations that uses directed graphs to represent a combination of boolean functions. Nodes of this graph are called gates and represent simple boolean functions (like and or xor). Edges, also known as *wires*, in case of circuit computation can hold 1 bit value. It is very similar to how hardware computation go, there are electrical wires that at every point of time can either have a current or not. And a physical analogue of a gate is a computation unit that sets current on outgoing electrical wires depending on an input wire's current. So as in a physical device, incoming edges for every node in the graph correspond to input values to the function of the gate. Outgoing edges will hold output of the function after computation.

To compute the circuit one needs to evaluate every gate's value. Consider the following full adder circuit as present in the Figure 1 below.

Circuit's inputs are bits A and B, C_{in} is a curry bit if it exists (if there were previous additions), outputs are S which will hold bit value of $(A + B) \mod 2$, and an output curry bit to propagate addition further C_{out}. Suppose we know the values of A, B, and C_{in}. Then to evaluate the circuit we are evaluating its gates one by one. At each step, we evaluate one gate which inputs are known, so for the example above we cannot evaluate or right away. But we could evaluate gates in this concrete example in a left to right fashion. Evaluation goes straightforwardly, we have a gate g and all its input wires contain bit values. If they do not one needs

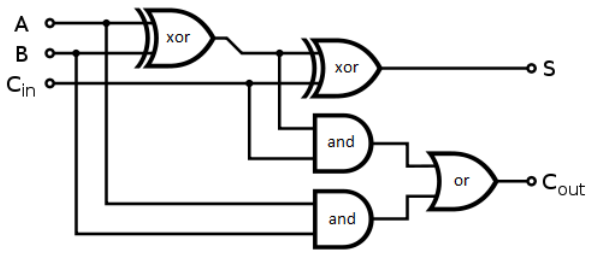

Figure 1: Full adder digital circuit [4]

first to evaluate the gate that has a wire with no value as an outgoing one. This evaluation sets bit values to all outgoing wires of this gate according to the gate's truth table.
Boolean circuit is a very powerful computation model, for example we can do randomised computation with it. All we need is just to specify some input wires that will hold presampled random bit values. We can obviously construct circuits for deterministic functions also. For instance, we will show how to construct a circuit to compute greater than function: $\mathsf{gt}(\alpha, \beta) = \alpha > \beta$, where α and β are 32-bit integers. We name α and β bits by index, α_0 is the least significant bit of α and α_{32} refers to the most significant bit of α. We will use the following gates: and, or, eq and not - all of them in their straightforward logical way. Gate and is true if both arguments are true, or is one of arguments is true, eq if both arguments are of the same value, not inverts the input bit.

First, we define 32 equality gates. Let $eq_i = (\alpha_i \Leftrightarrow \beta_i)$ for all i in $\{0, \ldots, 31\}$. Note that already here we can define $f(\alpha, \beta) = \alpha = \beta$ as and of all eq_i gates. But gt is more interesting. Now α is greater than β if for any bit i: $\alpha_i > \beta_i$ and $\forall j \in [i+1; 31]$ $\alpha_j = \beta_j$. This fact allows us to construct bit-gt gates in a very straightforward manner. We start with 31-th and 30-th bits:

$$\begin{aligned} gt_{31} &= \alpha_{31} \wedge \neg\beta_{31} \\ gt_{30} &= eq_{31} \wedge \alpha_{30} \wedge \neg\beta_{30} \end{aligned}$$

Now we need to introduce more gates which will accumulate the boolean value if the bits greater than the given index are equal for both input values.

$$u_{30} = eq_{31} \wedge eq_{30}$$

And in general:

$$u_i = eq_i \wedge u_{i+1}$$

Now we can proceed with bit-gt gates that are left:

$$gt_i \;=\; u_{i+1} \;\wedge\; \alpha_i \;\wedge\; \neg\beta_i$$

Now we have values if α is greater than β because of any i-th bit, so we just or all these values to obtain the result. In the same way, we can construct function less than lt .

Note that there are two obvious ways how to estimate the time complexity of a boolean circuit. We can define a total time to compute a circuit as a number of gates.
$t_{total} = |\mathcal{C}|$, where $\mathcal{C}$ is a set of all gates of a circuit.

If we allow parallelization of computations then a circuit can generally be computed faster than in a number-of-gates time, so minimal time to compute a circuit depends on a depth of the circuit, cause all gates on a single level can be computed simultaneously:
$t_{minimal} = \mathsf{depth}(C)$ where $\mathsf{depth}(C)$ is depth of the circuit.

2 Cryptographic tools

2.1 Pseudorandom generators and functions

Random in common sense means unpredictable. So, say that we want to have a random string x which is n bits long. The most unpredictable way to get this string is to uniformly sample the set $\{0,1\}^n$. If n is fairly large, it is very hard to get n bits with enough entropy. So we use a pseudorandom generators and functions to emulate source of randomness. They are pseudorandom because they produces result that is difficult to predict. Pseudorandom generator is a deterministic function that takes seed s of size m (which is small comparing to the size of the output) and stretches that to a hard to predict output of the size n. Formally it is defined as $f : \mathcal{S} \to \mathcal{X}$, where $\mathcal{S}$ is the seed space and $\mathcal{X}$ is here stretched output space. Security requirements for a pseudorandom generator f is that its output must be unpredictable, i.e., indistinguishable from a random sampling. Let's define games that describe this behavior for a pseudorandom generator f on Figure 2.

$\mathcal{G}_0^{\mathcal{A}}$
$x \xleftarrow{u} \{0,1\}^n$
return $\mathcal{A}(x)$

$\mathcal{G}_1^{\mathcal{A}}$
$s \xleftarrow{u} \{0,1\}^m$
return $\mathcal{A}(f(s))$

Figure 2: Pseudorandom generator indistinguishability games

In those games, an adversary $\mathcal{A}$ is trying to guess index of the game it is playing. So now we can define security of pseudorandom generator in terms of advantage of adversary $\mathcal{A}$.

$$\mathsf{Adv}_f^{PRG}(\mathcal{A}) = |\mathsf{Pr}[\mathcal{G}_0^{\mathcal{A}} = 1] - \mathsf{Pr}[\mathcal{G}_1^{\mathcal{A}} = 1]|$$

A function f is $(t,\ \varepsilon)$-*pseudorandom generator* if for all t-time adversaries $\mathcal{A}$ advantage $\mathsf{Adv}^{PRG}(\mathcal{A})$ is less or equal to ε.

As real life candidate for pseudorandom generator we can mention modified version of synchronous stream cipher SNOW2.0 [5]. For instance, time complexity of algebraic attacks against modified SNOW2.0 is about 2^{1292} operations. However, no precise estimates for $(t,\ \varepsilon)$ pairs are known and cannot be known without extensive breakthrough in complexity theory.

A *pseudorandom generator* is a deterministic function from cartesian product of key space and message space to ciphertext space: $f : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$. Usually we want to run a function many times so the key part of the argument space can be fixed and referred to implicitly as function index, thus if for instance we fix a key k, pseudorandom function f will be referred as $f_k : \mathcal{M} \to \mathcal{C}$.

Now, let F_{all} be the set of all functions $f : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$. Then we can define a function family $\mathsf{F} \subseteq \mathsf{F}_{all}$ as a multiset of functions $\mathsf{F} = \{f_k : k \in \mathcal{K}\}$ with fixed key part k. Note that it is a

multiset, because there could exist k_1 and k_2 such that $k_1 \neq k_2$, but $f_{k_1} \equiv f_{k_2}$. Let us design two games similarly to the games for pseudorandom generator, see Figure 3.

$\mathcal{G}_0^{\mathcal{A}}$

$f \xleftarrow[u]{} \mathsf{F}_{all}$

for $i \in \{1, q\}$:

$\quad y_i \leftarrow f(x_i)$

return $\mathcal{A}(\boldsymbol{y})$

$\mathcal{G}_1^{\mathcal{A}}$

$k \xleftarrow[u]{} \mathcal{K}$

for $i \in \{1, q\}$:

$\quad y_i \leftarrow f(k, x_i)$

return $\mathcal{A}(\boldsymbol{y})$

Figure 3: Cryptographic games to define pseudorandom function family

Similarly as before with pseudorandom generator, adversary is trying to guess which game is it playing so its advantage is defined in a similar way

$$\mathsf{Adv}_f^{PRF}(\mathcal{A}) = |\mathsf{Pr}[\mathcal{G}_0^{\mathcal{A}} = 1] - \mathsf{Pr}[\mathcal{G}_1^{\mathcal{A}} = 1]|.$$

A function family F is (t, ε)-*pseudorandom function family* if for all t-time adversaries $\mathcal{A}$ advantage $\mathsf{Adv}^{PRF}(\mathcal{A})$ is less or equal to ε.

Pseudorandom permutation is defined similarly, but through families of permutations. Let F_{prm} be a set of all permutations $f : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$, and let $\mathsf{P} \subseteq \mathsf{F}_{prm}$ be a permutation family.

We define pseudorandom permutation games in a similar manner as distinguishing pseudorandom function games above. The differences are only that in game $\mathcal{G}_0^{\mathcal{A}}$ we sample a function not from the set of all function but from the set of all permutations F_{prm}. Advantage is defined in exactly the same way as with pseudorandom functions

$$\mathsf{Adv}_f^{PRP}(\mathcal{A}) = |\mathsf{Pr}[\mathcal{G}_0^{\mathcal{A}} = 1] - \mathsf{Pr}[\mathcal{G}_1^{\mathcal{A}} = 1]|.$$

A permutation family P is (t, ε)-*pseudorandom perumation family* if for all t-time adversaries $\mathcal{A}$ advantage $\mathsf{Adv}^{PRP}(\mathcal{A})$ is less or equal to ε.

Now when we have defined those primitives, let us specify what could be real life candidates for them. For instance block ciphers are usually candidates of pseudorandom permutations by design. Specifically we are interested in AES (Rijndael) cipher which is a pseudorandom permutation family, where we sample a specific function by providing a key to AES.

2.2 Array encryption

Usually encryption schemes are used to encrypt messages, so encryption enc and decryption dec are defined to operate on the following domains:

$$\mathsf{enc} : \mathcal{K} \times \mathcal{M} \longrightarrow \mathcal{C} \ ,$$
$$\mathsf{dec} : \mathcal{K} \times \mathcal{C} \longrightarrow \mathcal{M} \ ,$$

where $\mathcal{K}$, $\mathcal{M}$ and $\mathcal{C}$ are respectively the key, message and ciphertext spaces. To simplify matters in this thesis, we need a modified definition of encryption scheme, as we need to encrypt tables and arrays so a single cell can be revealed by revealing the keys. Assume we have the following setup of four element array arranged in a table, see Figure 4.

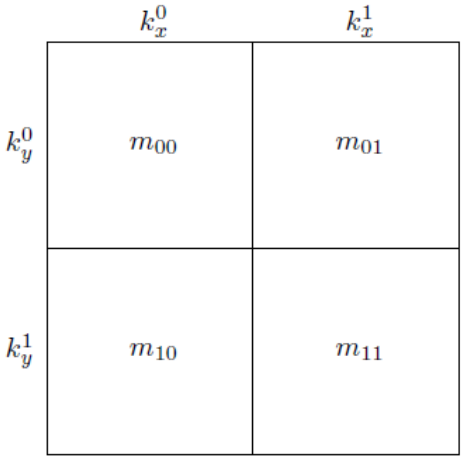

Figure 4: Array encryption table

We have four key values and four messages arranged in the table. Now, we need an array encryption scheme to use two different keys to perform encryption (decryption) operation on each message. To accommodate this fact we modify the domains of the encryption scheme:

$$\mathsf{enc} : \mathcal{K} \times \mathcal{K} \times \mathcal{M} \longrightarrow \mathcal{C} \ ,$$
$$\mathsf{dec} : \mathcal{K} \times \mathcal{K} \times \mathcal{C} \longrightarrow \mathcal{M} \ .$$

Notion $\mathsf{enc}_{k_x,k_y}(m) = c$ means that the message m is encrypted using the keys k_x and k_y to produce the ciphertext c. Decryption operation keys are notioned in the same manner as indexes, not arguments of the operation: $\mathsf{dec}_{k_x,k_y}(c) = m$.
Let us now define a notion for the whole table encryption. Let $\boldsymbol{m} = (m_{00}, m_{01}, m_{10}, m_{11})$, then the following formula describes one of the options how to organize array encryption operation

$$\mathsf{AE}_{k_x^0,k_x^1,k_y^0,k_y^1}(\boldsymbol{m}) = \begin{pmatrix} \mathsf{enc}_{k_x^0,k_y^0}(m_{00}) & \mathsf{enc}_{k_x^0,k_y^1}(m_{01}) \\ \mathsf{enc}_{k_x^1,k_y^0}(m_{10}) & \mathsf{enc}_{k_x^1,k_y^1}(m_{11}) \end{pmatrix} .$$

Note, that opposite to encryption we want decryption operation being called on each cell explicitly. More specifically, security requirement for such array encryption scheme is that adversary which has two keys, one from each pair (k_x^0, k_x^1) and (k_y^0, k_y^1), should be able to decrypt only one cell. The desired property of the encryption scheme is to be secure under chosen plaintext attacks, so now we define formal games for IND-CPA setting, see Figure 5.

$\mathcal{G}_0^{\mathcal{A}}$

$k_x^0, k_x^1 \leftarrow \mathcal{K} \times \mathcal{K}$
$k_y^0, k_y^1 \leftarrow \mathcal{K} \times \mathcal{K}$
$\boldsymbol{m_0}, \boldsymbol{m_1}, b_x, b_y \leftarrow \mathcal{A}$
if $\boldsymbol{m_0}[b_x][b_y] \neq \boldsymbol{m_1}[b_x][b_y]$ then **return** $\perp$
$c_0 \leftarrow \mathsf{AE}_{k_x^0,k_x^1,k_y^0,k_y^1}(\boldsymbol{m_0})$
$c_1 \leftarrow \mathsf{AE}_{k_x^0,k_x^1,k_y^0,k_y^1}(\boldsymbol{m_1})$
return $\mathcal{A}(c_0, k_x^{b_x}, k_y^{b_y})$

$\mathcal{G}_1^{\mathcal{A}}$

$k_x^0, k_x^1 \leftarrow \mathcal{K} \times \mathcal{K}$
$k_y^0, k_y^1 \leftarrow \mathcal{K} \times \mathcal{K}$
$\boldsymbol{m_0}, \boldsymbol{m_1}, b_x, b_y \leftarrow \mathcal{A}$
if $\boldsymbol{m_0}[b_x][b_y] \neq \boldsymbol{m_1}[b_x][b_y]$ then **return** $\perp$
$c_0 \leftarrow \mathsf{AE}_{k_x^0,k_x^1,k_y^0,k_y^1}(\boldsymbol{m_0})$
$c_1 \leftarrow \mathsf{AE}_{k_x^0,k_x^1,k_y^0,k_y^1}(\boldsymbol{m_1})$
return $\mathcal{A}(c_1, k_x^{b_x}, k_y^{b_y})$

Figure 5: Array encryption IND-CPA games

Advantage of an adversary defined as usual as the following difference:

$$\mathsf{Adv}_{\mathsf{AE}}^{IND-CPA}(\mathcal{A}) = |\mathsf{Pr}[\mathcal{G}_0^{\mathcal{A}} = 1] - \mathsf{Pr}[\mathcal{G}_1^{\mathcal{A}} = 1]|.$$

An encryption scheme is (t, ε)-*array encryption* IND-CPA secure if for all t-time adversaries $\mathcal{A}$ advantage $\mathsf{Adv}_{\mathsf{AE}}^{IND-CPA}(\mathcal{A})$ is less or equal to ε.

To build *one-time pad array encryption scheme*, we need 2ℓ bit long keys to encrypt four element array with ℓ bit long messages. Namely, we can split each key k into ℓ bit long blocks and use these blocks sequentially when we need to encrypt message with key k. To get rid of the restriction on the key length, we use pseudorandom generator to stretch keys upto the necessary length. If keys are not long enough, we use pseudorandom generator f to stretch them upto needed length. The result is given in Protocol 1. For clarity, let $f : \mathcal{K} \to \mathcal{M} \times \mathcal{M}$, particularly $f : \{0,1\}^k \to \{0,1\}^\ell \times \{0,1\}^\ell$, and let $f(x)[0]$ denote the first component of $f(x)$ and $f(x)[1]$ the second component.

Lemma 1. *If generator used to stretch keys for array encryption scheme is (t, ε)-pseudorandom generator, one-time pad array encryption scheme is (t, $8 \cdot \varepsilon$)-*IND-CPA *indistinguishable.*

Proof. (Sketch) Protocol 2 describes ideal functionality for array encryption AE°.

Input: $k_x^0, k_x^1, k_y^0, k_y^1$ and message $\boldsymbol{m} = (m_{00}, m_{01}, m_{10}, m_{11})$.
Output: ciphertext $\boldsymbol{c} = (c_{00}, c_{01}, c_{10}, c_{11})$

1. Compute $c_{00} = m_{00} \oplus f(k_x^0)[0] \oplus f(k_y^0)[0]$.
2. Compute $c_{01} = m_{01} \oplus f(k_x^0)[1] \oplus f(k_y^1)[0]$.
3. Compute $c_{10} = m_{10} \oplus f(k_x^1)[0] \oplus f(k_y^0)[1]$.
4. Compute $c_{11} = m_{11} \oplus f(k_x^1)[1] \oplus f(k_y^1)[1]$.

Protocol 1: Array encryption scheme using pseudorandom generator for one-time-pad AE^f

Input: message $\boldsymbol{m} = (m_{00}, m_{01}, m_{10}, m_{11})$
Output: ciphertext $\boldsymbol{c} = (c_{00}, c_{01}, c_{10}, c_{11})$

1. If not provided, generate keys $k_x^0 \leftarrow \{0,1\}^{2\ell}, k_x^1 \leftarrow \{0,1\}^{2\ell}, k_y^0 \leftarrow \{0,1\}^{2\ell}, k_y^1 \leftarrow \{0,1\}^{2\ell}$.
2. Compute $c_{00} = m_{00} \oplus k_x^0[0] \oplus k_y^0[0]$.
3. Compute $c_{01} = m_{01} \oplus k_x^0[1] \oplus k_y^1[0]$.
4. Compute $c_{10} = m_{10} \oplus k_x^1[0] \oplus k_y^0[1]$.
5. Compute $c_{11} = m_{11} \oplus k_x^1[1] \oplus k_y^1[1]$.

Protocol 2: Ideal array encryption scheme AE°

Remember, that the advantage of adversary $\mathcal{A}$ is defined in terms of winning games described in Figure 5. Consider, following intermediate games given on Figure 6.

$\mathcal{G}_2^{\mathcal{A}}$

$k_x^0, k_x^1 \leftarrow \mathcal{M} \times \mathcal{M} \times \mathcal{M} \times \mathcal{M}$
$k_y^0, k_y^1 \leftarrow \mathcal{M} \times \mathcal{M} \times \mathcal{M} \times \mathcal{M}$
$\boldsymbol{m_0}, \boldsymbol{m_1}, b_x, b_y \leftarrow \mathcal{A}$
if $\boldsymbol{m_0}[b_x][b_y] \neq \boldsymbol{m_1}[b_x][b_y]$ then **return** $\perp$
$\boldsymbol{c_0} \leftarrow \mathsf{AE}^\circ_{k_x^0, k_x^1, k_y^0, k_y^1}(\boldsymbol{m_0})$
$\boldsymbol{c_1} \leftarrow \mathsf{AE}^\circ_{k_x^0, k_x^1, k_y^0, k_y^1}(\boldsymbol{m_1})$
return $\mathcal{A}(\boldsymbol{c_0}, k_x^{b_x}, k_y^{b_y})$

$\mathcal{G}_3^{\mathcal{A}}$

$k_x^0, k_x^1 \leftarrow \mathcal{M} \times \mathcal{M} \times \mathcal{M} \times \mathcal{M}$
$k_y^0, k_y^1 \leftarrow \mathcal{M} \times \mathcal{M} \times \mathcal{M} \times \mathcal{M}$
$\boldsymbol{m_0}, \boldsymbol{m_1}, b_x, b_y \leftarrow \mathcal{A}$
if $\boldsymbol{m_0}[b_x][b_y] \neq \boldsymbol{m_1}[b_x][b_y]$ then **return** $\perp$
$\boldsymbol{c_0} \leftarrow \mathsf{AE}^\circ_{k_x^0, k_x^1, k_y^0, k_y^1}(\boldsymbol{m_0})$
$\boldsymbol{c_1} \leftarrow \mathsf{AE}^\circ_{k_x^0, k_x^1, k_y^0, k_y^1}(\boldsymbol{m_1})$
return $\mathcal{A}(\boldsymbol{c_1}, k_x^{b_x}, k_y^{b_y})$

Figure 6: Array encryption IND-CPA games with random number generator

As AE° uses uniformly sampled keys, so messages of the tables are xor-ed with random bit-strings, which results in ciphertext being uniformly sampled from all the possible ciphertexts. Thus, these games are perfectly indistinguishable from each other, which means that there is no

adversary that can distinguish which one of these games is it playing with significant certainty.

Let's investigate what is the computational distance between games $\mathcal{G}_0^{\mathcal{A}}$ and $\mathcal{G}_2^{\mathcal{A}}$. The only significant difference in their description is that encryption is performed in $\mathcal{G}_0^{\mathcal{A}}$ using keys stretched by pseudorandom generator and for $\mathcal{G}_2^{\mathcal{A}}$ using uniformly random bit-strings. Essentially, the only difference is in how $\boldsymbol{c_0}$ is computed (because adversary does not see $\boldsymbol{c_1}$). Now, in AE^f ciphertext $\boldsymbol{c_0} = F(\boldsymbol{m_0}, f(k_x^0), f(k_x^1), f(k_y^0), f(k_y^1))$, where f is pseudorandom generator and F is a deterministic function, which actually does the same as AE^{f}, but is written explicitly with the keys as arguments. On the other hand, in AE° the respective ciphertext is computed as $\boldsymbol{c_0^\circ} = F(\boldsymbol{m_0}, \bar{k_x^0}, \bar{k_x^1}, \bar{k_y^0}, \bar{k_y^1})$, where keys are uniformly sampled from the space $\mathcal{M} \times \mathcal{M}$. To derive the computational distance between these games, consider the following series of hybrid games, where the ciphertext $\boldsymbol{c_0}$ is computed in the following fashion:

$$\begin{aligned}
\mathcal{G}_{00} &: \boldsymbol{c_0} = F(\boldsymbol{m_0}, \bar{k_x^0}, f(k_x^1), f(k_y^0), f(k_y^1)) \ , \\
\mathcal{G}_{01} &: \boldsymbol{c_0} = F(\boldsymbol{m_0}, \bar{k_x^0}, \bar{k_x^1}, f(k_y^0), f(k_y^1)) \ , \\
\mathcal{G}_{02} &: \boldsymbol{c_0} = F(\boldsymbol{m_0}, \bar{k_x^0}, \bar{k_x^1}, \bar{k_y^0}, f(k_y^1)) \ .
\end{aligned}$$

Now the distance between $\mathcal{G}_0^{\mathcal{A}}$ and $\mathcal{G}_{00}$ is clearly ε, as if some adversary $\mathcal{A}$ can achieve a better result, we can construct distinguisher $\mathcal{B}^{\mathcal{A}}$ for PRG games on Figure 2, which achieves the same success rate as $\mathcal{A}$. This leads to a contradiction with the function f being (t, ε)-pseudorandom generator. $\mathcal{B}$ will query PRG game with k_x^0, then form all necessary input for $\mathcal{A}$ by uniformly sampling three other keys, and return whatever $\mathcal{A}$ returns. This strategy obviously gains the same success, as $\mathcal{A}$ in distinguishing $\mathcal{G}_0^{\mathcal{A}}$ from $\mathcal{G}_{00}$. By a similar argument, the distance between $\mathcal{G}_{00}$ and $\mathcal{G}_{01}$ is ε, the distance between $\mathcal{G}_{01}$ and $\mathcal{G}_{02}$ is ε and the distance between $\mathcal{G}_{02}$ and $\mathcal{G}_2^{\mathcal{A}}$ is ε. Thus the total distance between $\mathcal{G}_0^{\mathcal{A}}$ and $\mathcal{G}_2^{\mathcal{A}}$ is $4 \cdot \varepsilon$.

In the same manner we show that the distance between $\mathcal{G}_1^{\mathcal{A}}$ and $\mathcal{G}_3^{\mathcal{A}}$ is $4 \cdot \varepsilon$. Thus, the total computational distance between games $\mathcal{G}_0^{\mathcal{A}}$ and $\mathcal{G}_1^{\mathcal{A}}$ is then $8 \cdot \varepsilon$, so the encryption scheme is $(t, 8 \cdot \varepsilon)$-IND-CPA indistinguishable. □

2.3 Commitment schemes

Commitment scheme is usually a two-phase protocol, which allows a party to send messages to other party without instantly revealing their content and later also reveal the content of messages. Usually, when one party sends a message to the other, it reveals content of this message immediately. When using a commitment, this process of sending the message becomes actually two-phase: sending commitment value and then sending decommitment value. This allows the sender to send a message without immediate revealing of message content. Commitment schemes are widely used in building cryptographic protocols since about 1982, but formally formalized first by Brassard, Chaum, and Crepeau [6]. First of all, there is a generation procedure $\mathsf{Gen}()$, which is used to generate shared parameters pk for a commitment scheme, two main phases of

commitment scheme are: creation Com and opening Open of a commitment. Those are defined as $\mathsf{Com}_{pk} : \mathcal{R} \times \mathcal{M} \to \mathcal{C} \times \mathcal{D}$ and $\mathsf{Open}_{pk} : \mathcal{C} \times \mathcal{D} \to \mathcal{M}$, where $\mathcal{R}$ is a space of randomness, $\mathcal{M}$ is the message space, $\mathcal{C}$ and $\mathcal{D}$ are respectively spaces of commitment and decommitment values. Shared parameters define behavior of the Com and Open procedures, so we usually specify them not as an argument, but as index, like $(c, d) \leftarrow \mathsf{Com}_{pk}(m)$. In the notation above c is called commitment value, d is a decommitment value, and pk is shared parameters value that were produced by Gen. Commitment scheme is functional iff

$$\forall pk \leftarrow \mathsf{Gen}(), \forall m \in \mathcal{M} : \mathsf{Open}_{pk}(\mathsf{Com}_{pk}(m)) \equiv m.$$

The fact that there are two values: commitment and decommitment instead of a single message, is used to hide the message from the receiver until the decommitment value arrives to him. Usually, a party that commits to a message m computes $(c, d) \leftarrow \mathsf{Com}(m)$ and sends c to the receiver, then at some point of time later it sends d also to the receiver, so the last could verify the message was $\mathsf{Open}(c, d)$.

Now from all properties a commitment scheme can have, we are interested in hiding and binding. Hiding means that a commitment value alone does not provide meaningful information

$\mathcal{G}_0^{\mathcal{A}}$

$pk \leftarrow \mathsf{Gen}()$
$(m_0, m_1) \leftarrow \mathcal{A}(pk)$
$(c, d) \leftarrow \mathsf{Com}_{pk}(m_0)$
return $\mathcal{A}(c)$

$\mathcal{G}_1^{\mathcal{A}}$

$pk \leftarrow \mathsf{Gen}()$
$(m_0, m_1) \leftarrow \mathcal{A}(pk)$
$(c, d) \leftarrow \mathsf{Com}_{pk}(m_1)$
return $\mathcal{A}(c)$

Figure 7: Commitment scheme hiding property games

to the verifier. More formally (t, ε)-*hiding property* of the commitment scheme means that for any t-time adversary advantage against commitment scheme hiding property games, defined on Figure 7, is bounded by ε where the advantage is then defined

$$\mathsf{Adv}_f^{hiding}(\mathcal{A}) = |\mathsf{Pr}[\mathcal{G}_0^{\mathcal{A}} = 1] - \mathsf{Pr}[\mathcal{G}_1^{\mathcal{A}} = 1]|.$$

Binding means that the commiter, after publishing a commitment value cannot provide decommitment values that open the commitment to different messages. Formally, it is defined as probability of the adversary to win this binding game (see Figure 8).

The advantage against the binging property is simply defined as the probability of the adversary to win the binding game

$$\mathsf{Adv}_f^{binding}(\mathcal{A}) = \mathsf{Pr}[\mathcal{G}_0^{\mathcal{A}} = 1] \ .$$

Commitment scheme is (t, ε)-*binding* if there exists no t-time adversary $\mathcal{A}$, such that advan-

$\mathcal{G}_0^{\mathcal{A}}$

$pk \leftarrow \mathsf{Gen}()$
$(c, d_0, d_1) \leftarrow \mathcal{A}(pk)$
$m_i \leftarrow \mathsf{Open}_{pk}(c, d_i)$ **for** $i \in \{0, 1\}$
if $m_0 = \perp$ **or** $m_1 = \perp$ **then return** 0
return $[m_0 \neq m_1]$

Figure 8: Commitment scheme binding property game

tage $\mathsf{Adv}^{binding}(\mathcal{A})$ is greater than ε.

2.3.1 Pedersen commitment scheme

Suppose we have a finite cyclic multiplicative algebraic group $\mathbb{G}$. If the discrete logarithm problem in it is hard, then the group is DL-secure. Figure 9 defines the formal security games for discrete logarithm problem. More specifically it is (t, ε)-DL-secure if no t-time adversary $\mathcal{A}$ can

$\mathcal{G}^{\mathcal{A}}$

$x \underset{u}{\leftarrow} \mathbb{G}$
$y = g^x$
$\bar{x} \leftarrow \mathcal{A}(y)$
return $x \stackrel{?}{=} \bar{x}$

Figure 9: Discrete logarithm game

gain advantage $\mathsf{Adv}_f^{DL}(\mathcal{A}) = \mathsf{Pr}[\mathcal{G}^{\mathcal{A}} = 1]$ greater than ε. Which means that the adversary cannot correctly compute discrete logarithm of a random group element with probability more than ε.

Pedersen commitment scheme [7] is set up by choosing $\mathbb{G} = \langle g \rangle$ a q-element DL-group, where q is a prime. Take y uniformly from $\mathbb{G}$, then $(g,\ y)$ will be public parameters of the commitment scheme. To commit to message $m \in \mathbb{Z}_q$, one needs to choose $r \leftarrow \mathbb{Z}_q$ and compute the commitment value $c \leftarrow g^m \cdot y^r$ and decommitment value $d \leftarrow (m, r)$. The pair (m, r) is a valid decommitment value for commitment c with public parameters $pk = (g, y)$ if $c = g^m \cdot y^r$. This construction gives us the following security guarantees for hiding and binding properties. In this thesis we present only sketch proofs of properties of Pedersen commitment scheme, rigorous mathematical proofs about them, interested reader can find in Liina Kamm is work about classification of commitment schemes [8].

Theorem 1. *If $\mathbb{G}$ is a $(t,\ \varepsilon)$-DL-secure group with q elements, where q is a prime, Pedersen*

commitment scheme is perfectly hiding and (t, ε)-binding commitment scheme.

Proof. (Sketch) HIDING. Following is a valid argument that could be used to construct a straightforward proof. Note that y^r is uniformly distributed over $\mathbb{G}$, since r is uniformly chosen from $\mathbb{Z}_q$, g is a generator of $\mathbb{G}$ and $y = g^x$ for some $x \neq 0$. And as $g^m \cdot \mathbb{G} = \mathbb{G}$, so c is uniformly distributed over $\mathbb{G}$. So there is no way for the adversary to determine which hiding game is it playing.
BINDING. We can prove binding property by showing that a valid double opening to a Pedersen commitment will reveal a discrete logarithm of g^y (which is a random group element), so it is impossible for (t, ε)-adversary. Suppose (m_0, r_0) and (m_1, r_1) are valid decommitment values for some commitment $c = g^m \cdot y^r$, note that they must open to different values, but:

$$c \;=\; g^{m_0} \cdot\; y^{r_0} \;=\; g^{m_1} \cdot\; y^{r_1} \iff log_g(y) \;=\; \frac{m_0 - m_1}{r_0 - r_1}$$

As $r_0 \neq r_1$, adversary who is able to break binding property can compute discrete logarithm. It contradicts the assumption that $\mathbb{G}$ is (t, ε)-DL-secure. □

2.3.2 Split receipt commitment

For this thesis, we need another type of commitment scheme with peculiar security properties. It has the same procedures as basic commitment scheme Gen, Com and Open. Generation procedure Gen produces public parameters as usual. However, Com and Open work on slightly different domains

$$\begin{aligned} &\mathsf{Com} : \mathcal{R} \times \mathcal{M} \to \mathcal{C} \times \mathcal{D} \times \mathcal{D} \ , \\ &\mathsf{Open} : \mathcal{C} \times \mathcal{D} \times \mathcal{D} \to \mathcal{M} \ . \end{aligned}$$

The important feature of this type of a commitment is that it produces two decommitment values for each commited message. Requirement for this scheme to be functional is the same as the usual one

$$pk \leftarrow \mathsf{Gen}() \ \forall m : \mathsf{Open}(\mathsf{Com}(m)) = m.$$

Consider values that are produced by Com procedure $c, d_1, d_2 \leftarrow \mathsf{Com}(m)$, where c is a commitment value, d_1 and d_2 are respectively the first and second decommitment values.
Security requirements for the scheme are then the following. The pairs c, d_1 and c, d_2 are hiding, meaning that one cannot learn the commited message by observing only one such pair of values. Formalising this requirement as games, we have the following game, see Figure 10.

With corresponding advantage definition, which is as usual

$$\mathsf{Adv}^{s-hid}(\mathcal{A}) = |\mathsf{Pr}[\mathcal{G}_0^{\mathcal{A}} = 1] - \mathsf{Pr}[\mathcal{G}_1^{\mathcal{A}} = 1]| \ .$$

$\mathcal{G}_0^{\mathcal{A}}$

$pk \leftarrow \mathsf{Gen}()$
$(m_0, m_1, r) \leftarrow \mathcal{A}(pk)$
$(c, d_1, d_2) \leftarrow \mathsf{Com}_{pk}(m_0)$
return $\mathcal{A}(c, d_r, r)$

$\mathcal{G}_1^{\mathcal{A}}$

$pk \leftarrow \mathsf{Gen}()$
$(m_0, m_1, r) \leftarrow \mathcal{A}(pk)$
$(c, d_1, d_2) \leftarrow \mathsf{Com}_{pk}(m_1)$
return $\mathcal{A}(c, d_r, r)$

Figure 10: Split receipt commitment hiding games

Binding property of split receipt commitment stands in the fact that there should exist no double opening possibilities if c and one of the decommitment values are fixed. The following game on Figure 11 formalizes this property.

$\mathcal{G}_0^{\mathcal{A}}$

$pk \leftarrow \mathsf{Gen}()$
$(c, d_1, \hat{d}_1, d_2, \hat{d}_2) \leftarrow \mathcal{A}(pk)$
$m_0 \leftarrow \mathsf{Open}_{pk}(c, d_1, d_2)$
$m_1 \leftarrow \mathsf{Open}_{pk}(c, \hat{d}_1, \hat{d}_2)$
if $d_1 \neq \hat{d}_1 \ \wedge \ d_2 \neq \hat{d}_2$ then **return** $\perp$
if $m_0 = \perp \ \vee \ m_1 = \perp$ then **return** $\perp$
return $[m_0 \neq m_1]$

Figure 11: Split receipt commitment binding game

Advantage $\mathsf{Adv}^{\mathcal{A}}$ of adversary $\mathcal{A}$ is probability that it will win this game

$$\mathsf{Adv}^{s-bind}(\mathcal{A}) = \mathsf{Pr}[\mathcal{G}_0^{\mathcal{A}} = 1].$$

Now, split receipt commitment is (t, ε)-hiding if no t-time adversary $\mathcal{A}$'s advantage $\mathsf{Adv}_f^{s-hid}(\mathcal{A})$ is greater than ε. Also, split receipt commitment is (t, ε)-binding if no t-time adversary $\mathcal{A}$'s advantage $\mathsf{Adv}_f^{s-bind}(\mathcal{A})$ is greater than ε.

Now we describe how to organize a split receipt commitment scheme from a usual commitment scheme CS such that decommitment value d produced by this commitment scheme is an element of Abelian group $(\mathcal{D}, +)$. Then to perform a split receipt commitment $\mathsf{SR\text{-}Com}$ operation on message m, we compute $(c, d) \leftarrow \mathsf{Com}(m)$, and then additively share d as a pair (d_1, d_2), where $d_1 \leftarrow \mathcal{D}$ and $d_1 + d_2 = d$.

Lemma 2. *If* Com *is* (t, ε)*-hiding and* $(t\prime, \varepsilon\prime)$*-binding commitment scheme, decommitment value of which is an element of Abelian group* $(\mathcal{D}, +)$*, split receipt commitment* $\mathsf{SR\text{-}Com}$ *based on it and additive sharing is* (t, ε)*-hiding and* $(t\prime, \varepsilon\prime)$*-binding.*

Proof. HIDING PROPERTY. We prove the hiding property of such a split receipt commitment scheme with a straightforward reduction. If there exists a t-time adversary $\mathcal{A}$ who gains a advantage more than ε, then we can construct adversary $\mathcal{B}$ to break the hiding property of the underlying commitment scheme in the following way. Adversary $\mathcal{B}$ initializes $\mathcal{A}$ and submits messages produced by $\mathcal{A}$ to the hiding game. Upon receiving the commitment value c, adversary $\mathcal{B}$ will generate random $d_1 \in \mathcal{D}$, and return whatever $\mathcal{A}$ returns on (c, d_1, r). Obviously, advantage of $\mathcal{B}$ against underlying commitment hiding is equal to $\mathcal{A}$'s advantage against split receipt commitment hiding game and we have reached a contradiction.

BINDING PROPERTY. Proving binding property of this split receipt commitment scheme is also straightforward. Adversary who can provide a valid double opening triples (c, d_1, d_2) and $(c, d_1, \hat{d}_2)$, can double open underlying commitment with pairs $(c, d_1 + d_2)$ and $(c, d_1, \hat{d}_2)$, since $d_1 + d_2 \neq d_1 + \hat{d}_2$ in case of successful double opening for SR-Com. The similar argument holds if adversary successfully attacks binding of split receipt commitment with triples that share the value of d_2. This implies, that adversary $\mathcal{B}$ against Com binding games that uses $\mathcal{A}$ to produce triples (c, d_1, d_2) and $(c, d_1, \hat{d}_2)$ and returns $(c, d_1 + d_2, d_1, \hat{d}_2)$ will succeed if $\mathcal{A}$ would. And as Com is (t, ε)-binding, $\mathcal{A}$'s advantage against SR-Com binding property is at most ε. □

3 Secure multiparty computation

Consider the following scenario, when we have several parties that have their respective inputs. They want to compute a function on their inputs, but in the same time they want to do that in a way so their inputs will not be learned by the other parties.

3.1 Secure two-party computation

Secure two-party computation problem differs from a multiparty computation by the fact, that there are only two parties involved. Assume that two parties $\mathcal{P}_1$ and $\mathcal{P}_2$ want to compute function $f(x, y) = (z_1, z_2)$ from their respective inputs x and y. They want to compute it in a such way that $\mathcal{P}_1$ receives z_1 as the output and $\mathcal{P}_2$ the output z_2. Privacy requirements state that no party should learn the other's party input. Other requirement for this computation is correctness, the output should be from corresponding to the distribution of function f outputs. This setting is different from a multiparty setting in several ways, most severe is that one of the parties will always be in a dominant position. Suppose that we have two parties and that they want to compute some function in a such manner that both parties will receive some output. They will do it by running a protocol that consists of several messages that should be sent from one party to another. Assume that a protocol run consist on sending four messages: $\alpha_1, \beta_1, \alpha_2, \beta_2$. Messages noted as α_i are sent from $\mathcal{P}_1$ to $\mathcal{P}_2$, messages β_i in another direction from $\mathcal{P}_2$ to $\mathcal{P}_1$. Now after exchanging those messages both parties will learn their outputs. However note, that the last message β_2 does not give any information to $\mathcal{P}_2$, so it can learn its output just from first three messages. That puts it into a dominant position in a sense that $\mathcal{P}_2$ can exit from the protocol at a time when it has received all interesting for it information, and $\mathcal{P}_1$ still has not. It can be shown that any two-party protocol with any number of messages suffers from this vulnerability [10]. So while investigating a two-party protocol we should have this possibility of a dominant party to cancel early.

3.2 Security of a two-party computation

Let's say those parties have their respective inputs x and y, and they want to compute the function f, in a way that $f(x, y) = (z_1, z_2)$ and party one, referred as $\mathcal{P}_1$ gets z_1 as the output and party two which will be referred as $\mathcal{P}_2$ gets output z_2. First, we will construct an idealized version of how can this computation can be organized. Pretend there exists a trusted third party (referred as $\mathcal{T}$), which is totally honest, does not want to learn parties' inputs and is trusted by both parties involved in the computing f. The ideal protocol uses $\mathcal{T}$ to collect inputs from the parties, then $\mathcal{T}$ computes $(z_1, z_2) \leftarrow f(x, y)$ and then it sends z_1 to a first party and z_2 to a second one. To accommodate the possibility of dominant party to abort computation before other party receives its output, ideal protocol incorporates a phase when dominant party sends *continue/abort* signal to $\mathcal{T}$. It happens after $\mathcal{P}_1$ receives its output but before $\mathcal{P}_2$ does. Protocol 3 describes this setting, where $\mathcal{P}_1$ is dominant.

1. $\mathcal{T}$ collects inputs from both parties: f and x_1 from $\mathcal{P}_1$, x_2 from $\mathcal{P}_2$.
2. $\mathcal{T}$ computes $(z_1,\ z_2) \leftarrow f(x_1,\ x_2)$
3. $\mathcal{T}$ sends z_1 to $\mathcal{P}_1$
4. $\mathcal{P}_1$ sends a signal to $\mathcal{T}$. It is either halting signal that corresponds to $\mathcal{P}_1$'s wish not to continue protocol run, or continue signal, that allows $\mathcal{T}$ to release $\mathcal{P}_2$'s output
5. If $\mathcal{T}$ received a signal to continue, it sends z to $\mathcal{P}_2$. Otherwise $\mathcal{P}_2$ receives $\perp$.

Protocol 3: Ideal functionality for two-party computation with $\mathcal{T}$

The outcome of such protocol run depends only on inputs that $\mathcal{T}$ gets from parties. We assume that computation goes magically in $\mathcal{T}$, so it is correct and no information is leaked during computation. Thus, if any attack exist for this ideal functionality, it depends only on submitting specifically crafted input to $\mathcal{T}$ or observing outputs of the computation. Note, that in any real implementation of the protocol both these steps: submitting inputs and receiving outputs are present. Thus any attack possible against ideal functionality is achievable against real protocol, as corrupted party can exploit submitting same input or observe output in the same way as against real protocol. Thus any real protocol will be less or equally secure as ideal one. It is important to note, that it is possible that this ideal protocol could offer very little or no security guarantees at all (we can construct ideal protocol that is totally insecure, for example, in the sense of privacy of inputs). But then, a real protocol that implements the same functionality will also be totally insecure, thus offer equal amount of security, which do not contradicts with point previously stated.

In this thesis an ideal protocols are marked with a circle, like this π°, compared to a real protocol π. A notion $\pi_1 \preceq \pi_2$ means that π_1 is less or equally secure than π_2.

Now we present a general way to proof the security of a protocol. Assume that we have a protocol π_1 that has some security guaranties (in extreme case no security guarantees, which we denote as zero or no security case). And we have a protocol π_2 that we want to prove being at least as secure as π_1. We make an assumption that there is an adversary $\mathcal{A}$ that is good against protocol π_2. Then we show that we can use $\mathcal{A}$ to construct an adversary $\mathcal{A}^\circ$ that is at least as good against protocol π_1 as $\mathcal{A}$ against π_2. Consequently we have shown that we can successfully attack protocol π_1 if there exist adversary which can successfully attack π_2.

Figure 12 defines formal games to determine if a protocol is secure in a standalone model. Standalone model implies that there are no pre- or post-processing context around the protocol, so we just initialize all parties, adversary and run protocol.

Note, that in these games parties inputs ϕ_i does not necessary contains only party's input to the protocol, it also can contain any auxiliary information that this party can possess before protocol execution.

$\mathcal{G}^{\mathcal{A}}_{REAL}$

$$\left[\begin{array}{l} sample : \phi_1, \phi_2, \phi_{\mathcal{A}} \leftarrow \mathcal{D} \\ init : \mathcal{A}(\phi_{\mathcal{A}}),\ \mathcal{P}_1(\phi_1),\ \mathcal{P}_2(\phi_2) \\ run\ protocol : \pi \\ collect\ outputs : \boldsymbol{\varphi} = (\varphi_1, \varphi_2, \varphi_{\mathcal{A}}) \\ \textbf{return}\ \mathcal{B}(\varphi) \end{array}\right.$$

$\mathcal{G}^{\mathcal{A}}_{IDEAL}$

$$\left[\begin{array}{l} sample : \phi_1, \phi_2, \phi_{\mathcal{A}} \leftarrow \mathcal{D} \\ init : \mathcal{A}(\phi_{\mathcal{A}}),\ \mathcal{P}_1(\phi_1),\ \mathcal{P}_2(\phi_2) \\ run\ ideal\ protocol : \pi^{\circ} \\ collect\ outputs : \boldsymbol{\varphi}^{\mathbf{\circ}} = (\varphi_1, \varphi_2, \varphi_{\mathcal{A}}) \\ \textbf{return}\ \mathcal{B}(\varphi^{\circ}) \end{array}\right.$$

Figure 12: Standalone security games

Now in these games $\mathcal{B}$ is a success evaluation predicate, which evaluates how well adversary $\mathcal{A}$ is doing against some pre-specified attack target against real protocol. If adversary $\mathcal{A}$ succeeds in cheating with larger probability against real protocol than any adversary in the ideal world, then intuitively there should exist some distinguisher for outputs of these adversaries. Thus, for existence of reduction $\mathcal{A} \longmapsto \mathcal{A}^{\circ}$ it is important that

$$\forall \mathcal{A}\ \exists \mathcal{A}^{\circ} \forall \mathcal{D} : \boldsymbol{\varphi}^{\mathbf{\circ}} \equiv \boldsymbol{\varphi}\ ,$$

where $\equiv$ means indistinguishability of distributions. However, note that mere indistinguishability is not enough as reduction must be efficient. To be able to prove that protocol is secure, we need the fact that $\mathcal{A}^{\circ}$ running time is not much greater that $\mathcal{A}$ time. So we need reduction $\mathcal{A} \longmapsto \mathcal{A}^{\circ}$ by such that execution times of $\mathcal{A}$ and $\mathcal{A}^{\circ}$ are approximately the same. So along with reduction of adversaries we require that exist relation $t_{\mathcal{A}} \longmapsto t_{\mathcal{A}^{\circ}}$, where $t_{\mathcal{A}}$ is execution time of $\mathcal{A}$ and $t_{\mathcal{A}^{\circ}}$ is execution time of $\mathcal{A}^{\circ}$. And we require that $t_{\mathcal{A}^{\circ}} \leqslant f_s(t_{\mathcal{A}})$, where f_s is execution time of $\mathcal{A}^{\circ}$ and those times satisfy condition of $t_{\mathcal{A}^{\circ}} \leqslant f_s(t_{\mathcal{A}})$, where f_s is a *polynomial function*. So, when distributions φ° and φ coincide, then a protocol is *perfectly secure* with running time overhead f_s. Being a perfectly secure protocol means that for all possible attack targets real protocol is as secure as ideal protocol and we cannot improve any more.

If output distributions are ϵ-indistinguishable $\boldsymbol{\varphi}^{\mathbf{\circ}} \equiv_{\epsilon} \boldsymbol{\varphi}$, then real protocol is statically secure and respectively if distributions are computationally ε-equivalent $\boldsymbol{\varphi}^{\mathbf{\circ}} \equiv_{\varepsilon} \boldsymbol{\varphi}$, protocol is ($f_s$, t, ε)-*secure*. For perfectly secure real protocol no success evaluation predicate can distinguish which game is it playing. For computationally secure protocols a smaller set of predicates cannot distinguish games, so we must put upper bound on predicate evaluation's t-time.

Suppose we have a real world protocol π, its ideal analogue π° real world adversary $\mathcal{A}$ attacking π. Simulator $\mathsf{Sim}^{\mathcal{A}}$ is an adversary against π°, that uses $\mathcal{A}$ internally. Essentially, we can think of $\mathsf{Sim}^{\mathcal{A}}$ as a proxy that sits between $\mathcal{A}$ and $\mathcal{T}$ of the ideal protocol and intercepts communication between them. It is clear, that generally real world adversary $\mathcal{A}$ cannot communicate with $\mathcal{T}$ directly because of incompatibility of their interfaces, but $\mathsf{Sim}^{\mathcal{A}}$ solves this problem.

3.3 Oblivious transfer

Oblivious transfer is a special case of secure two-party computation [12]. Consider setting party one has database of messages $\boldsymbol{m} = (m_0, \ldots, m_n)$. Party two has index i, of which message does it want. The function those parties want to compute is $f(\boldsymbol{m}, i) = (\perp, m_i)$. Security requirements are the same as for general secure two-party computation: every party gets only output intended for it and cannot learn other party's input values. Note that if we assume there is no way to organize oblivious transfer, then general secure two-party computation is obviously also unachievable, cause oblivious transfer is special case of general two-party computation.

One of the most basic settings is one-out-of-two oblivious transfer $\binom{1}{2}$-OT, when message database size is 2. There are many ways to implement oblivious transfer protocol and a lot of research has been done in this area.

3.4 Security of multiparty computation

Consider following games at Figure 13 that describe multiparty computation setting.

$\mathcal{G}^{\mathcal{A}}_{REAL}$

$\lceil$ *sample* : $\phi_1, \phi_2, \ldots, \phi_n, \phi_{\mathcal{A}} \leftarrow \mathcal{D}$
$|$ *init* : $\mathcal{A}(\phi_{\mathcal{A}}),\ \mathcal{P}_1(\phi_1),\ \mathcal{P}_2(\phi_2) \ldots, \mathcal{P}_n(\phi_n)$
$|$ *run protocol* : π
$|$ *collect outputs* : $\boldsymbol{\varphi} = (\varphi_1, \varphi_2, \ldots, \varphi_n, \varphi_{\mathcal{A}})$
$\lfloor$ **return** $\mathcal{B}(\boldsymbol{\varphi})$

$\mathcal{G}^{\mathcal{A}^\circ}_{IDEAL}$

$\lceil$ *sample* : $\phi_1, \phi_2, \ldots, \phi_n, \phi_{\mathcal{A}} \leftarrow \mathcal{D}$
$|$ *init* : $\mathcal{A}^\circ(\phi_{\mathcal{A}}),\ \mathcal{P}_1(\phi_1),\ \mathcal{P}_2(\phi_2) \ldots, \mathcal{P}_n(\phi_n)$
$|$ *run ideal protocol* : π°
$|$ *collect outputs* : $\boldsymbol{\varphi}^\circ = (\varphi_1, \varphi_2, \ldots, \varphi_n, \varphi_{\mathcal{A}^\circ})$
$\lfloor$ **return** $\mathcal{B}(\boldsymbol{\varphi}^\circ)$

Figure 13: Multiparty computation security games

Similar to secure two-party computation setting, real protocol is secure, if distributions of parties' outputs in real and ideal protocol runs are indistinguishable and t-time of ideal adversary $\mathcal{A}^\circ$ is comparable to real world adversary $\mathcal{A}$ t-time. More specifically, real protocol is *perfectly secure* if both following conditions hold:

$$\forall \mathcal{A}\ \exists \mathcal{A}^\circ \forall \mathcal{D} : \boldsymbol{\varphi}^\circ \equiv \boldsymbol{\varphi}\ ,$$
$$t_{\mathcal{A}^\circ} \leqslant f_s(t_{\mathcal{A}})\ ,$$

for a polynomial f_s.

Additionally, in the same way as for secure two-party computation, we can define ϵ-statically security and (f_s, t, ε)-computational security for a real protocol if, respectively, $\boldsymbol{\varphi}^\circ \equiv_\epsilon \boldsymbol{\varphi}$ or $\boldsymbol{\varphi}^\circ \equiv_\varepsilon \boldsymbol{\varphi}$ and $t_{\mathcal{A}^\circ} \leqslant f_s(t_{\mathcal{A}})$ hold.

To get an insight, how proving security of a protocol works in a multiparty computation setting, consider a model of an protocol's communication between several parties on the Figure 14. We have four parties that are represented by circles and double-ended arrows represent

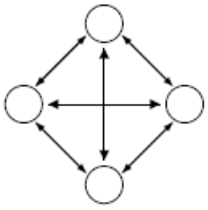

Figure 14: Real protocol's communication model

communication channels that are between every pair of parties. Now to model an ideal analogue of this protocol we substitute direct communications between parties with a communication via trusted third party $\mathcal{T}$ as defined on Figure 15).

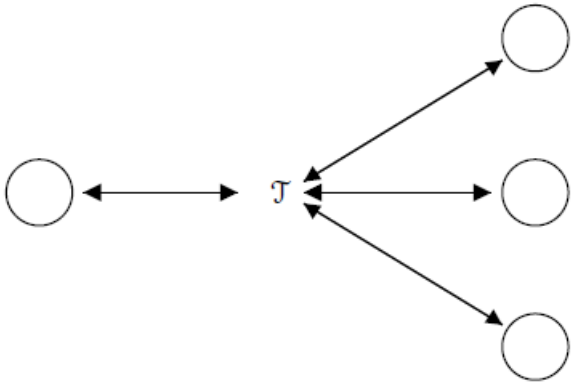

Figure 15: Ideal protocol's communication model with $\mathcal{T}$

An exact way of adversary controlling corrupted parties must be specified by the setting, but one of the most usual settings used is static corruption of at most k parties. This means that adversary chooses upto k parties to corrupt before protocol execution, and it gets all information these parties have if it is a honest-but-curious adversary or gets full control over these parties in a malicious setting. Consider a figure below, suppose parties on the right of $\mathcal{T}$ are corrupted and send all their information to an adversary.

Now, to prove security of the protocol, we need to add a simulator to that model. Simulator has two interfaces: $\mathcal{T}$ interface which is used to organize communication in a ideal protocol, and adversary interface which deals with a real world protocol and adversary implementation. Look at the Figure 16.

Dashed circles are virtual parties that simulator creates to accommodate communication

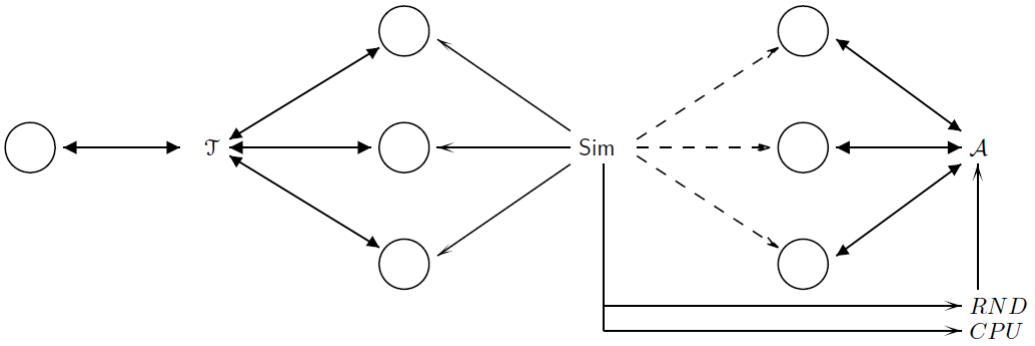

Figure 16: Simulator fits between real world adversary and ideal protocol

between adversaries and corrupted parties. Additionally, simulator controls randomness source and CPU of an adversary to being able to rewind adversary and execute it from the beginning to supply it with a predefined randomness. Now, the goal of simulator construction is to put a real world adversary against an ideal protocol. All the computation is done by ideal protocol and must be as secure as possible. If there exist correspondence $\mathcal{A} \longmapsto \mathsf{Sim}^{\mathcal{A}}$ such that outputs of parties in the run with simulator were indistinguishable from the output of the parties without it. Then adversary can simulate this run by itself and actual engagement of a real party does not add knowledge of an adversary, thus protocol is secure.

4 Introduction to Yao garbled circuits

Consider the following modified scenario of secure two-party computation. Two parties $\mathcal{P}_1$ and $\mathcal{P}_2$ want to compute $(\varnothing, z) \leftarrow f(x_1, x_2)$, where x_1 and x_2 are respective parties inputs. Important difference from a general two-party computation case is that only one party is going to receive meaningful output value, as sender $\mathcal{P}_1$ always receives $\varnothing$ if protocol run completes.

First general solution to such secure two party computation problem was proposed by A. A. Yao in the seminar paper [1]. His idea consists of three main techniques:

- reordering gate's truth table rows to hide content of the gate;
- modifying boolean circuit to operate on encryption scheme keys, not just bit values;
- using oblivious transfer to securely give evaluation party input to the circuit.

In this thesis we use $\mathsf{xor}+PRG$ array encryption scheme AE_f described before. This simplifies description of Yao garbled circuit construction procedure and is somewhat easier to understand for a reader.

4.1 Yao garbled circuit construction

Flipping values on wires. We start securing boolean circuit evaluation with the following procedure. In the evaluation process, every wire in circuit will hold one bit value. We need those values not to provide any meaningful information to the adversary. To do that we will randomly modify the meaning of this value. Procedure to do that is the following, for every wire we with probability one half "flip" its values. Flipped values are then detached from actual meaning of value on the wire, cause a wire which must hold value, for instance, 1, after being flipped can contain either 1 or 0.

If we do not modify anything else in the circuit, obviously evaluation will not compute intended function correctly. Thus, we must modify content of truth-tables of gates accordingly. As an illustration of the said before, suppose we have an and gate with its ordinary truth-table. This gate has two input wires and one output wire. Suppose then that we have flipped the value on first input of the gate (wire x in the Table 2), then 0 on that wire actually means 1. Suppose flipped wire x is denoted by $\bar{x}$, then we modify the table in the following way.

$\bar{x}$	y	$x \wedge y$
0	0	0
0	1	1
1	0	0
1	1	0

Table 2: and gate truth table with flipped x wire

Note, that we have changed contents of result column of that table. If values on wires y or $x \wedge y$ have been also flipped, we would need to also take that into account when computing table.

Extending values on wires. Procedure for this step of securing circuit evaluation is the following, for every wire w of the circuit we generate two random keys for array encryption scheme. We just create two random keys for each wire. Then we use gates' reorganized truth-tables from the previous step, and substitute bit values in those with corresponding wire keys. Let k_x^0, k_x^1, k_y^0, k_y^1 be keys generated for wires x and y respectively. Additionally, let k_z^0, k_z^1 be keys generated for gate's output wire z.

Also, let f be a pseudorandom generator that we will feed to array encryption scheme to stretch keys. Then we put those keys into the truth-table of the gate and encrypt values of the output column (see Table 3). As in the previous example, we have wire x flipped, and denote that

$\bar{x}$	y	$x \wedge y$
k_x^0	k_y^0	$\mathsf{enc}_{k_x^0,k_y^0}(k_z^0 \| \| 0)$
k_x^0	k_y^1	$\mathsf{enc}_{k_x^0,k_y^1}(k_z^1 \| \| 1)$
k_x^1	k_y^0	$\mathsf{enc}_{k_x^1,k_y^0}(k_z^0 \| \| 0)$
k_x^1	k_y^1	$\mathsf{enc}_{k_x^1,k_y^1}(k_z^0 \| \| 0)$

Table 3: and gate encrypted truth table

by using $\bar{x}$ in the gate's truth-table. Note, that output wire key is concatenated with a bit value, so domains for array encryption scheme are then: $\mathcal{K} = \{0,1\}^{128}$, $\mathcal{M} = \{0,1\}^{129}$, $\mathcal{C} = \{0,1\}^{129}$; for AES with 128-bit keys as pseudorandom generator.

Now output column of this table is filled with pseudorandom data. Garbled circuit generation is finished now and description of the circuit will consist of:

- description of wires, from which gate it goes to which;
- description of gates: encrypted flipped truth table result column.

Note that in order to reverse the output of the circuit, which will be a key value, circuit generator, also known as generating party, must remember corresponding key to bit value transitions and if output wire was flipped. This means that if circuit generator has flipped values on output wires of the circuit, it must remember which wire has it flipped, cause then value produced by circuit evaluation could be flipped one.

Transferring keys to evaluator. Now this circuit cannot be evaluated without knowing keys corresponding to a values for input wires. Party that generated circuit can send keys corresponding to its input with circuit description, they do not reveal bits of input, so it is still private. On the other hand, parties need to engage in oblivious transfer protocol to get keys that correspond to circuit evaluating party's input to evaluator.
The setting for Yao garbled circuit protocol is such, that generating party has two keys and evaluating party has a bit value for each input wire. After oblivious transfer circuit evaluator will learn key value corresponding to input value and circuit generator will learn nothing.
To evaluate a garbled circuit, party receives its description and keys corresponding to party

input. Then through oblivious transfer protocol run it obtains keys corresponding to its input and continues with the following evaluation scheme. For every gate which input wires contain key values, decrypt truth table. As every row of truth table is encrypted with different key pairs it will be able to decrypt only one row. Set values from that decrypted row to outgoing wires of this gate. Proceed with this gate by gate evaluation until no non-evaluated gates left. Collect circuit output value from output wires. These values for every output wire w, will be in the form of $out_w = key||b$, where key is one of the keys generated for output wires and b is a bit value of the output. Note, that for input and intermediate wires key values needed to proceed with circuit evaluation. However, we do not use key values of output wires, cause nothing is encrypted with those. So generator of the circuit can put any bit-string of corresponding length there. Array encryption scheme works fine on any bit-strings and actually can be used to encrypt messages of arbitrary length, so it does not limit information that can be put on an output wire. Additional feature of Yao garbled circuit protocol is that output wires can be flipped or not. If output wires are certainly not flipped, evaluator in addition to bit-string information will learn bit value of the output. On the other hand, setting my be fixed so that generating party can flip values on output wires, then, evaluator will learn just information contained in output bit-string and not actual bit values.

5 Security of Yao circuit evaluation

First public results about Yao garbled circuit were available from 1982, but the whole paper by A.A.Yao [1] was not publicly disclosed. Actually only the extended abstract is publicly available. No rigorous proof of security for garbled circuit protocol can be found there. One published proof appeared in 2009 by Lindell and Pinkas [11]. In this paper we provide another proof that Yao garbled circuit protocol is secure against *semi-honest* adversary. Semi-honest adversary follows protocol and just tries to investigate information to learn other party's output.
Note, that despite the fact that there must exist a dominant party in two-party computation, in case of Yao circuit evaluation domination does not provide any additional advantage for parties. Suppose sender $\mathcal{P}_1$ is in dominant position, then after receiving its output it can abort computation. Fortunately, its output is always $\varnothing$, so it cannot learn anything from it and aborting the computation is equivalent to refusing to submit its input: in both cases sender gets null and receiver $\mathcal{P}_2$ receives $\perp$. If receiver is dominant it also does not change anything to abort computation, cause sender then receives $\perp$ instead of $\varnothing$, which does not play any relevant role either. So we deliberately remove that dominance step from two-party computation process and further do not mention it.

Now the setting for secure Yao circuit evaluation is that there are parties sender $\mathcal{P}_1$, receiver $\mathcal{P}_2$ and $\mathcal{T}$ in an ideal model. Let x be sender's input, y receiver's input and $F(x, y)$ function $\mathcal{P}_1$ want to compute. As the result of the protocol, receiver learns $F(x, y)$, sender learns nothing.
Note a very important observation, that then there exists function $f(y) = F(x, y)$, such function where sender's input is specified implicitly. We will insist on the condition that f must be a predicate (have one-bit output) and be representable with a boolean circuit. Additional detail is that, however we need f be a predicate, Yao garbled circuit can hold arbitrary information attached to the output values. We let messages z_0, z_1 be bit-string of appropriate size be attached to output values 0 and 1 respectively.

Now we require this computation to be correct, as in actually computing not to leak information about other parties' inputs, but we allow the receiver to obtain some knowledge about the function f, more specifically it will learn skeleton of boolean circuit that specifies f. We use notion $\mathsf{skeleton}(f)$ to refer to this information. If $f = \perp$ or evaluation of $f(y) = \perp$, then receiver will get $\perp$ as a result. Consider then, the following scheme at Figure 17, that describes the protocol run in an ideal model.

We will name that protocol an ideal conditional oblivious transfer COT°. During COT° one party specifies single bit output function f and two strings z_0 and z_1, other specifies input y and the second party gets $z_{f(y)}$.

Now, when we have specified how Yao circuit evaluation goes in ideal model, let us quickly recall how the protocol is implemented in a real world. Sender prepares garbled circuit, encrypt it and has keys for input wires ready to be transferred to receiver. Then parties are engaged in

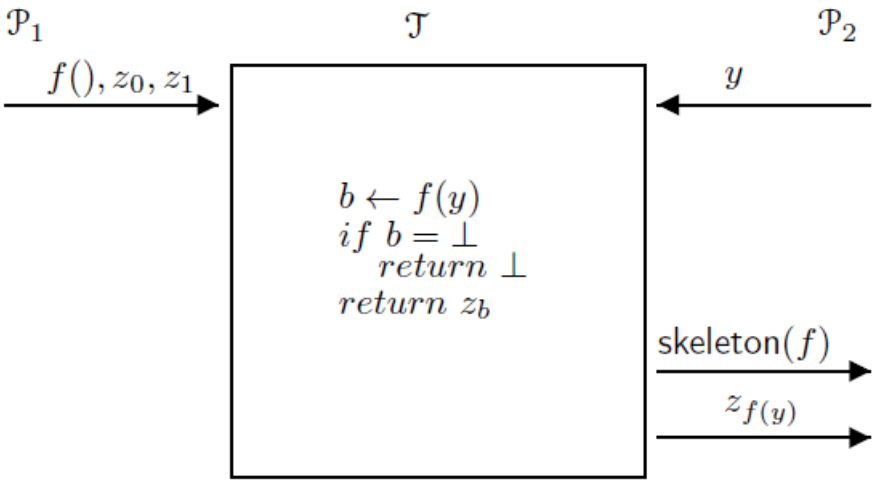

Figure 17: Yao circuit evaluation in ideal model, COT°

series of OT protocols to transfer keys corresponding to receiver's input to receiver. Next, sender sends circuit description to receiver, which then evaluates this garbled circuit and gets output bit b and a bit-string that was used as a key for output wire.

In this chapter we prove the following theorem regarding the security of garbled circuit protocol described above.

Theorem 2. *If AE^f is (t, ε)-array encryption scheme, COT is perfectly secure against malicious sender and $(t, |\mathcal{C}| \cdot \varepsilon)$-secure against malicious receiver, where $|\mathcal{C}|$ is number of gates in the circuit describing function supplied by sender.*

The proof is divided into several parts that we describe in next subsections. We start with proving security against sender and then proceed with proof of security in case where circuit receiver is corrupted.

5.1 Security against corrupted sender

5.1.1 Security against semi-honest sender

The case of corrupted generating party is trivial, but to be consistent we still present a proof of security here.

In current setting in standalone model corrupted party gets input to the protocol as $\phi = (history, f, z_0, z_1)$, where *history* is any information available to a party before protocol run, and f, z_0, z_1 is the intended input.

It is easy to construct a simulator in this model, a $\mathsf{Sim}^{\mathcal{A}}$ receives the same intended input f, z_0, z_1. Then it is engaged in a series of OT protocols with $\mathcal{P}_1$. As a result of each of these OT

protocol, $\mathcal{P}_1$ needs to receive $\varnothing$, so $\mathsf{Sim}^{\mathcal{A}}$ provides these $\varnothing$. Note, that semi-honest sender cannot modify its intended input, so $\mathsf{Sim}^{\mathcal{A}}$ proceeds with sending this f, z_0, z_1 to $\mathcal{T}$ as the input to ideal Yao circuit evaluation. Sender does not receive any more output, so $\mathsf{Sim}^{\mathcal{A}}$ is done. Computation is correct, as it is performed by $\mathcal{T}$.

We proceed then with showing that distributions of the outputs of the parties are indistinguishable. Note that, $\mathcal{P}_1$'s output must be in any case $\varnothing$ as it is $\varnothing,\ldots,\varnothing$ for OT protocols, so it is the same as in ideal protocol. $\mathcal{P}_2$ will receive some output from circuit evaluation, cause circuit generating party is semi-honest, so it follows protocol and circuit will be evaluable. And thus, the output of $\mathcal{P}_2$ coincides with the $\mathcal{P}_2$'s output in ideal model.
Now in both ideal and real models, in the protocol run adversary sees $\mathcal{P}_1$'s input and all values received by $\mathcal{P}_1$. As these coincide in ideal and real model, adversary behavior and output is the same in ideal and real models. Now we showed that outputs of ideal protocol $(\varphi_1^\circ, \varphi_2^\circ, \varphi_{\mathcal{A}}^\circ)$ are equivalent to those of real protocol $(\varphi_1, \varphi_2, \varphi_{\mathcal{A}})$, simulator overhead is negligible, thus protocol is as secure as the ideal functionality provided by $\mathcal{T}$. Note, that formally, we need to prove that t-time of the $\mathsf{Sim}^{\mathcal{A}}$ is at most polynomially larger than t-time of $\mathcal{A}$. However, as $\mathsf{Sim}^{\mathcal{A}}$ does not do any computation by itself its overhead is not significant. Note, that in proofs further in this thesis, we omit explicit specification of simulator's t-time if a trivial intuition correctly suggests that overhead is polynomial.

5.1.2 Security against malicious sender

The case with maliciously corrupted sender is different, case actual input to the protocol in this case can be different from the intended input that simulator knows. Thus, we must first extract the "actual". This is done in the following manner.

Recall, that parties do series of OT protocols to transfer keys to receiver. Parties use ideal OT that includes $\mathcal{T}$. So sender's input to that OT protocols for each input wire w contains both keys, the one that corresponds to value 0 and the one for value 1.

During these OT protocols, simulator $\mathsf{Sim}^{\mathcal{A}}$ collects all keys that correspond to all possible inputs of the receiver and responds with expected $\varnothing$ to the sender. Now, sender provides garbled circuit description. Note, that simulator has now all input keys and thus is able to fully disclose the circuit obtaining both actual output values $\hat{z}_0, \hat{z}_1$ instead of one, and plain description of the f (as it can trace how circuit evaluation actually went). Then $\mathsf{Sim}^{\mathcal{A}}$ submits $(f, \hat{z}_0, \hat{z}_1)$ as the input to ideal Yao circuit evaluation and finishes.

Computation is correct, as it is performed by $\mathcal{T}$ which has received actual input values originated by malicious sender. Distributions of outputs are the same, as $\mathcal{P}_1$ receives $\varnothing$ and several $\varnothing$ as result of OT protocols. $\mathcal{P}_2$ gets one of $\hat{z}_0, \hat{z}_1$ in both ideal and real models. Exact value which it receives depends only on its input, so for honest receiver it will be the same value in both ideal and real models. Adversary, as in semi-honest case, sees $\mathcal{P}_1$ inputs (f, z_0, z_1) and all values seen by $\mathcal{P}_1$ which includes several $\varnothing$ from OT protocol runs and $\varnothing$ as $\mathcal{P}_1$'s output. These values coincide with values seen by adversary in ideal model, so adversary's outputs in real and

ideal models are the same.

Thus, we have shown that Yao circuit evaluation is perfectly secure against malicious sender.

5.2 Security against corrupted receiver

To prove security of Yao garbled circuit protocol against corrupted receiver, we start from an ideal protocol implementation, which uses trusted third party and gradually replace parts of the ideal protocol with parts of the real protocol in a way that preserves security.

We start with a showing ideal functionality of Yao garbled circuit protocol. First of all, let's fix the problem in detail. There are two parties $\mathcal{P}_1$ and $\mathcal{P}_2$ that have their private inputs x and y and they want to compute a function $F(x, y) = (\varnothing, z)$. Moreover to simplify the construction of protocol, let function $f(y) = F(x, y)$ be already presented as a boolean circuit. Refer back to Figure 17, which defines ideal Yao protocol functionality. Now we have honest sender, so the function it incorporates into garbled protocol is the intended one, so $\mathcal{T}$ knows it. Additionally, to enhance readability we explixitly specify sender $\mathcal{P}_1$'s input, despite the fact that it can be hidden in the definition of function f to compute. Consider the following simplified model at Figure 18 that describes this case.

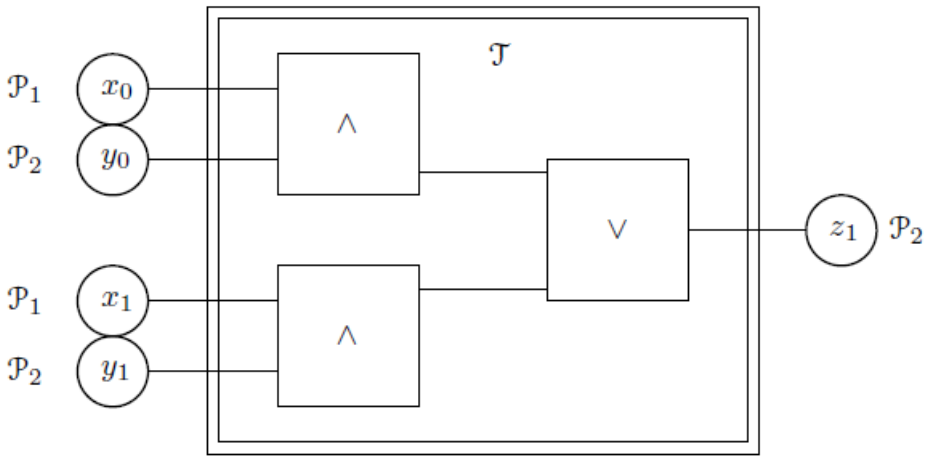

Figure 18: Ideal Yao circuit functionality

We have two bit inputs $x = x_1x_0$ and $y = y_1y_0$ on the left side of a big box and an output z_1 on the right side. Let this box be constructed in a way that no adversary can see what is done inside it. The computation in the box is performed by a trusted third party. This protocol is secure. Now we start modifying this construction by implementing procedures described previously in the section about construction of garbled circuit. We will show that doing so preserves secureness of the protocol.

First of all, we with probability one half flip values on every wire and modify gates of a circuit

to accommodate those flipped values. Let's formally define and investigate properties of flipping procedure. Flipping a wire means that we sample a fair coin and xor its result with value on the wire. Suppose $\sigma \leftarrow \{0,1\}$, then $w_{flipped} = w \oplus \sigma$. Note that probability that there will be a certain value on wire is one half.

$$\mathsf{Pr}[w_{flipped} = 0] = \mathsf{Pr}[w = 0] \cdot \mathsf{Pr}[\sigma = 0] + \mathsf{Pr}[w = 1] \cdot \mathsf{Pr}[\sigma = 1] = \mathsf{Pr}[w = 0] \cdot \frac{1}{2} + \mathsf{Pr}[w = 1] \cdot \frac{1}{2} = \frac{1}{2}$$

$$\mathsf{Pr}[w_{flipped} = 1] = \mathsf{Pr}[w = 0] \cdot \mathsf{Pr}[\sigma = 1] + \mathsf{Pr}[w = 1] \cdot \mathsf{Pr}[\sigma = 0] = \mathsf{Pr}[w = 0] \cdot \frac{1}{2} + \mathsf{Pr}[w = 1] \cdot \frac{1}{2} = \frac{1}{2} = \frac{1}{2}$$

And now we can safely allow adversary to see values on the wires. Figure 19 clarifies this setting. Assuming that $\mathcal{P}_2$ is corrupted, adversary can see now values that are on the wires marked with small $\mathcal{A}$ boxes. We denote gates whose truth-table is modified according to wires being flipped with $\otimes$ symbol.

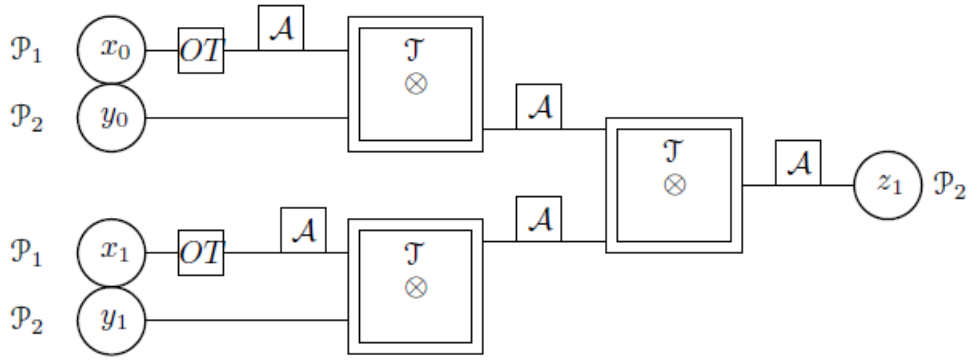

Figure 19: Hybryd Yao protocol with $\mathcal{T}$ gates evaluation

Now, evaluation of the circuit goes essentially as before, but not in one step with $\mathcal{T}$. Instead, we have circuit gates that can be computed by $\mathcal{T}$, and wires that are visible to an adversary. But we have already flipped wires' values and changed gates functionality to mirror those flips to preserve correctness of computation. As in settings before, parties do series of OT protocols to transfer keys corresponding to receiver's input to receiver. Then receiver starts to evaluate ideal gates. Hybrid protocol insists that receiver first evaluate all input gates of the circuit, to incorporate that we will restrict circuits so that every output bit depends on all input bits, so for every bit of output $z_i = F(x_0, \ldots, x_n, y_0, \ldots, y_n)$, where F is some function. This will ensure that all input gates must be evaluated before any output gate while we, obviously, do not exclude any functions from being computable that way.
Then receiver proceeds with opening all other gates of the circuit. Ideal gate evaluation goes in the following manner. Receiver provides inputs for the gate to $\mathcal{T}$ and receives values for output wires of that gate.

Lemma 3. *If circuit's wire has been flipped, gates are implemented as ideal gates and adversary can evaluate every gate only once, then evaluation is correct and perfectly secure against semi-honest receiver.*

Proof. (Sketch) Note, that protocol insists that input gates are evaluated first, so we show that after evaluation of input gates adversary does not learn anything. Adversary cannot learn input directly, as only one party is corrupted and other party sends its input directly to $\mathcal{T}$. Any wire evaluated by receiver contains random value from $\{0, 1\}$ with equal probabilities, so it does not leak information. So while adversary sends values of intermediate gates to $\mathcal{T}$, it receives a value for a wire which does not allow him to learn if the output wire was flipped or not. Evaluation of the output gates reveals outputs to $\mathcal{P}_2$, as output wires are never flipped.

Next, we show a simulator for this protocol. As in previous case, Sim starts engaging in series of OT protocols. With each OT Sim gets one bit of receiver's input and to obtain "actual" input just concatenates them appropriately. Receiver on the other hand for each OT wants to receive a response with a key, that corresponds to his input bit. Simulator generates these keys randomly as a circuit generating party would do when garbling a circuit, and send them to receiver.

After that simulator proceeds with ideal Yao circuit evaluation with $\mathcal{T}$ as now it knows receiver's input. As a result Sim receives from $\mathcal{T}$ $\mathsf{skeleton}(f)$ of the circuit and actual output value. Simulator transfers $\mathsf{skeleton}(f)$ to receiver.
Now receiver starts to evaluate gates of the circuit. For every non-output gate Sim responds to receiver with a randomly generated key value. Note that these values come from the same distribution as original keys that were situated on the circuit wires. In response to evaluation of output gate Sim sends actual output value it got from $\mathcal{T}$ to receiver.

Protocol run with simulator produces correct output as $\mathcal{P}_1$ receives correct output $\varnothing$ and $\mathcal{P}_2$ receives correct output as Sim just proxies output received from $\mathcal{T}$. For every output wire of intermediate gate Sim replies with random values, which have the same distribution as values returned by $\mathcal{T}$ in ideal model. Thus, part of information available to the adversary, which includes $\mathcal{P}_1$'s input, all messages receiver by $\mathcal{P}_1$, which are $\varnothing$'s, is the same in ideal and real models. The other part on information available to the adversary, namely keys for wires of the circuit comes from the same distribution in ideal and real models, so adversary's outputs must be the same in ideal and real models. Thus protocol is correct and perfectly secure against semi-honest receiver. □

Now, note very important detail that $\mathcal{T}$ will evaluate every gate only once. If we allow evaluating party to evaluate gate more than once (for different input values), then party can evaluate gate for all possible input sets, determine if output value was flipped or not and determine other party's input for this gate.
Now we need to implement this one-time evaluating gate as a real functionality. For that, consider the Figure 20 that describes the circuit evaluation model for next step of the proof of security. Now all wires and truth-tables of gates marked with a small box with $\mathcal{A}$ are visible to adversary.

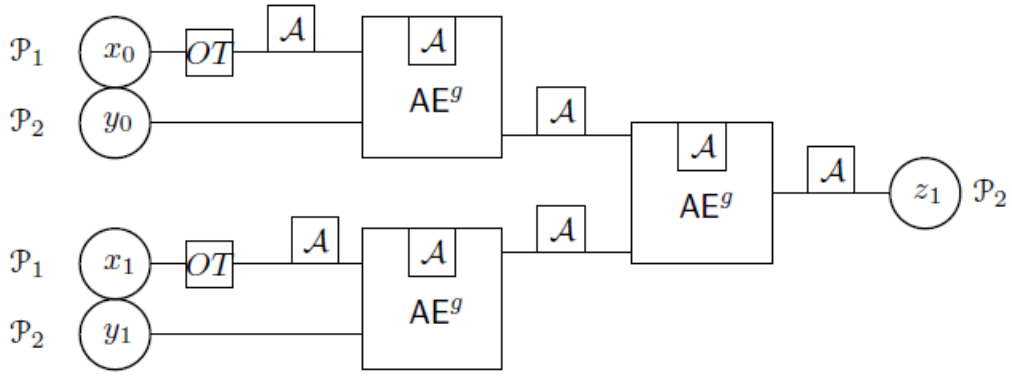

Figure 20: Real circuit visibility model

As adversary now knows content of truth-tables it could evaluate gates more than once, thus it can deduct information about inputs. To overcome this problem we substitute bit values on wires to a randomly generated keys for array encryption scheme (described in previous chapter) and use input wire values to encrypt output wires values.
Let g be a pseudorandom generator, for instance we can use AES's key stretching procedure as such.

Suppose we will use AES with 128 bit keys as pseudorandom generator in array encryption scheme described previously. Array encryption scheme needs to operate then on the following domains:

$$\begin{aligned}\mathsf{enc} &: \{0,1\}^{128} \times \{0,1\}^{128} \times \{0,1\}^{129} \longrightarrow \{0,1\}^{129} \quad , \\ \mathsf{dec} &: \{0,1\}^{128} \times \{0,1\}^{128} \times \{0,1\}^{129} \longrightarrow \{0,1\}^{129} \quad .\end{aligned}$$

We concatenate keys of the output wires with probably flipped bit values of wires to ensure that evaluator knows which cell of truth-table to decrypt.

As described previously array encryption scheme, when using good pseudorandom generator, is fairly secure against chosen plaintext attacks, we now have the result that adversary can only open one cell of the encrypted truth-tables of each gate.
Now we have all needed results to state the following compiler theorem.

Theorem 3. *If AE^f is (t, ε)-array encryption scheme, Yao garbled circuit protocol that uses AE^f to encrypt truth-tables of circuit's gates is perfectly secure agaist malicious sender and (t, $|\mathcal{C}| \cdot \varepsilon$)-secure against malicious receiver, where $|\mathcal{C}|$ is number of gates in the circuit.*

Proof. (Sketch) As usual we start with the construction of an appropriate simulator Sim. Sim-

ulator starts with series of OT protocols to transfer necessary keys to receiver. During every OT run, receiver reveals one bit of his input to Sim, on the other hand Sim generates a key in the same manner as sender would generate a wire key and send it to receiver. After all OT protocols are completed receiver has a certain set of keys $\boldsymbol{k}$. Simulator, on the other hand, knows receiver's actual input y and proceeds with ideal evaluation of Yao garbled circuit with $\mathcal{T}$, receiving $\mathsf{skeleton}(f)$ of the circuit and actual output value z.
Then simulator populates gates of $\mathsf{skeleton}(f)$ with arbitrary boolean functions, for example $\wedge$, thus getting a full circuit for function $\hat{f}$ with the same skeleton, i.e., $\mathsf{skeleton}(\hat{f}) = \mathsf{skeleton}(f)$. Then simulator uses functionality of a honest garbled circuit sender with inputs $(\hat{f}, z, z)$. Additionally Sim makes sure that keys for input wires that correspond to receiver's input are the ones from $\boldsymbol{k}$. Then Sim reveals description of this garbled circuit to receiver.

Note, as receiver knows keys from $\boldsymbol{k}$ it is able to evaluate that circuit (open one value of each gate's truth-table until the output wires). Moreover, as circuit was prepared with both output values equal to z, after evaluation receiver will gets z as the output.

It is not trivial to rigorously prove, that receiver cannot distinguish between description of garbled circuit received in real world run and the one supplied by simulator. However note, that output values of receiver in both models coincide, as they are equal to z. Additionally, $\mathsf{skeleton}(\hat{f}) = \mathsf{skeleton}(f)$, so receiver cannot easily distinguish between those. An actual proof can be done using mathematical induction by showing that after evaluation of every gate receiver does not learn information that helps him to distinguish those circuits with probability more than ε. We leave complete proof of this fact as an exercise for a interested and intelligent reader[1]. These probabilities are accumulated for every gate producing in total $|\mathcal{C}| \cdot \varepsilon$ probability that these circuits can be distinguished. Note, that $\mathcal{P}_2$ in the real protocol does not interact with $\mathcal{P}_1$ except for OT-s, which are handled by $\mathcal{T}$, and input submission. So there is no interaction where malicious behavior can achieve any benefits compared to semi-honest behavior. So while we have not said about adversary being malicious, the conclusion does not change and distributions of the outputs for both parties and adversary remains indistinguishable from distribution of outputs in ideal model.

Having shown that real protocol is simulatable, provides correct computation and its outputs are indistinguishable from an ideal protocol run, we conclude that Yao garbled circuit protocol is secure against malicious receiver. □

5.3 Remark on security against malicious adversary

Note an interesting fact, in the previous section we have proven, that Yao garbled circuit evaluation shaped as we did it (recall Figure 17) is secure against malicious adversary. Careful reader would notice that it is different from the original problem setting for secure two-party computation. For secure two-party computation problem the setting is usually fixed in such way, that both parties know what function $F(x, y)$ they want to compute. However, we have modelled

[1] An interested and intelligent reader who has roughly 8 hours of free time and lots of tea or coffee to spare.

ideal world so, that only sender specifies a function that will be computed. So, actually, Yao garbled circuit protocol described above does not solve secure two-party computation problem even for receiver only output functions $(\varnothing, z) \leftarrow F(x, y)$. The reason is that sender can specify its input (f_x, z_0, z_1) such that $f_x(y) \neq F(x, y)$.

However, modelling Yao garbled circuit in this way allows us to present a protocol using symmetric Yao circuit evaluation to achieve security against malicious adversary in standard secure two-party computation setting.

6 Consistent Yao circuit evaluation, Circus protocol

There exist several ways to increase security of Yao Circuit evaluation protocol to achieve security against malicious adversary. Usually we sacrifice performance for that. Here we would like to describe one common way to do that by including randomness into protocol by using technique is called *cut-and-choose*.

Main idea of this technique is that circuit generator must prepare many circuits instead of one. Evaluating party will choose some of them to evaluate and request sender to prove that leftover circuit were produced correctly. After that receiver evaluates chosen circuits and majority of theirs outputs is taken as final output value. Mohassel and Franklin state [2] that this approach allows circuit evaluation to be secure with inverse exponential probability. However obviously amount of computation and network resources needed to perform such protocol grow linearly with number of circuits generator constructs. There are constant attempts to invent modified protocols to secure Yao circuit evaluation against malicious adversary, but their performance tend to suffer. For instance Abhi Shelat and Chih-Hao Shen derive [15] protocol that is faster than usual cut-and-choose approach, but still with severe time overhead over plain Yao garbled circuit evaluation protocol. In this thesis we present a protocol with better theoretical computational performance.

6.1 Security models

First of all, we need to specify a security model that we will use. Recall a description of standard ideal model presented earlier in Protocol 3. This is a strong model, where no information is leaked to an adversary. We want to describe two more models that are weaker in the sense that there exists possibility for adversary to get some additional information regarding other party's input.

K-leaked model. A weaker model that allows adversary, which we will denote as $\mathcal{P}_1$ to be consistent with figures below, to learn k bits of other party's input. This model was introduced by Mohassel and Franklin [2], when they presented a protocol idea that became a base for protocol presented in this thesis. Therefore for historical reasons and overall better understanding of security properties of protocol presented further, we need to describe such model here. More specifically, we show a *1-leaked* model, where adversary gets to know 1 bit about other party's output. The computation in this model goes as described at Figure 21.

Note that we speak about *1-leaked* model, if function g that $\mathcal{P}_1$ submits to $\mathcal{T}$ is a predicate. If it is instead a k-bit output function this figure then describes *1-leaked* model. Essentially, the main difference from the standard ideal model lies in the fact, that corrupted party asks $\mathcal{T}$ a predicate about other party's input, receives the output of specified predicate. Presumably, $\mathcal{P}_1$ is in the dominant position here, as it can abort computation early, in which case $\mathcal{P}_2$ will receive $\perp$ as the output.

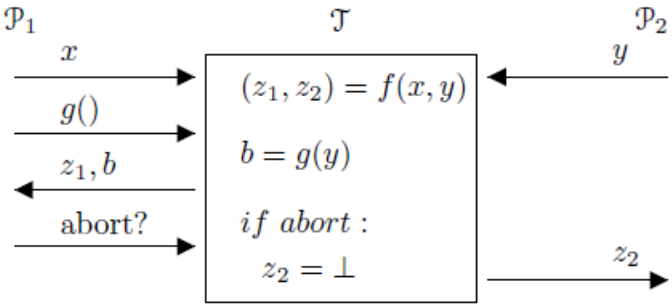

Figure 21: Two-party computation in k-leaked model

Consistency model. Consistent computation guarantees that malicious participants cannot learn anything beyond their intended outputs and honest participants can detect malicious behavior that alters their outputs [16]. Consider the Figure 22 for description of computation in the consistency model. The predicate $\pi()$ is called *halting predicate*, cause if it is true $\mathcal{T}$ halts

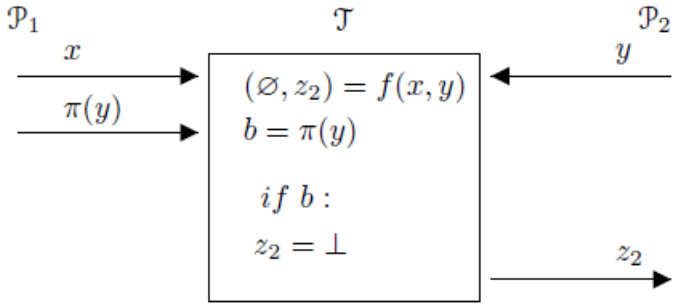

Figure 22: Two-party computation in consistency model

and broadcasts $\perp$ for every output value. Note, that it is very similar to a function $g(y)$ in the *1-leaked* model, however we note it differently to emphasize it's predicate nature of returning a single bit. Additionally, note that this model is stronger than *1-leaked* model described before, as in case of consistent computation, corrupted party may learn 1 bit about other party's input, however it may occur that he does not. For symmetric function evaluation, where both parties need to receive meaningful outputs, not $\perp$, consistency model is equivalent to *1-leaked* model, as sender which received $\perp$ will know that $\pi(y)$ was evaluated to true.

Note, that if we restrict ourselves to asymmetric functions, where sender receives $\varnothing$ as the output, then consistent model satisfies privacy requirements of standard ideal model against corrupted sender. As the only way for corrupted party to learn information is to submit a specific predicate to $\mathcal{T}$ and observe its further behavior. If $\mathcal{T}$ sends normal value as the output of the protocol, then predicate was evaluated to false, also, sender will receive $\perp$ if predicate

was true. However, as sender receives no output, then he cannot determine to which value was predicate evaluated. Another interesting property is that if specified predicate is true, receiver will also get $\perp$ instead of a normal output value, which will indicate that sender was malicious.

In this thesis, we present a protocol for secure two-party computation problem that is secure against malicious adversary in consistency model. To describe the protocol, we need several subprotocols be defined as a black-box ideal functionality.

6.2 Subprotocols

To simplify analysis of the protocol we assume that several protocols exist and are implemented either by specification provided with description below or using $\mathcal{T}$.

Commitment scheme. First of all, we need a split receipt commitment scheme SR-Com. Detailed description of security properties for such scheme based on some usual commitment scheme Com was provided earlier in this thesis, so now we just specify how SR-Com implementation works. For futher analysis, assume that SR-Com split receip commitment scheme is $(t_1, \varepsilon_{hiding})$-hiding and $(t_2, \varepsilon_{binding})$-binding. Previously, we have shown how to construct a split receipt commitment scheme from usual commitment scheme Com, which is $(t_1, \varepsilon_{hiding})$-hiding and $(t_2, \varepsilon_{binding})$-binding. Namely, to commit to message m, a party generates $(c, d) = \mathsf{Com}(m)$, then splits decommitment value d additively into two parts (d_1, d_2) such that $d = d_1 + d_2$. To commit to a message, party sends c to a receiver, and a triple (c, d_1, d_2) is a decommitment value. Sender will leave one part of the decommitment d_1 to himself, thus allowing to open the commitment only when he provides it.

Disclose when equal. Another block needed for protocol construction is modified conditional disclosure of secrets with equality check protocol DWE. For this protocol parties fix some condition and specify inputs. If condition specified is met by input values, then parties receive output, which can be either directly specify by them or received during computation inside DWE.

We need a specific version of DWE, where parties specify parts of split receipt commitment values in hope that they both can open other's party commitment to the same value. In this case, they will receive this value as the output. This behavior is described at Figure 23.

Notion of the variables used in Figure 23 is that barred value, for instance, $\bar{c}^{m_0}$ is originally produced by $\mathcal{P}_2$, while normal variables without a bar, for instance $d_1^{m_1}$ are originally produced by $\mathcal{P}_1$. The upper index on the variable shows during commitment of which message was this value produced, for example, c^{m_0} was produced in $\mathsf{SR\text{-}Com}(m_0)$.

Now, this version of DWE° protocol does the following, each party provides parts of two decommitment triples for SR-Com and the second decommitment value d_2. Trusted third party tries to open each of the commitment values of a party with decommitment value d_2 provided by other party. If $\mathcal{T}$ finds a pair of decommitment values that open to the same message, this message becomes an output value. If parties fail to open each other's split receipt commitment to

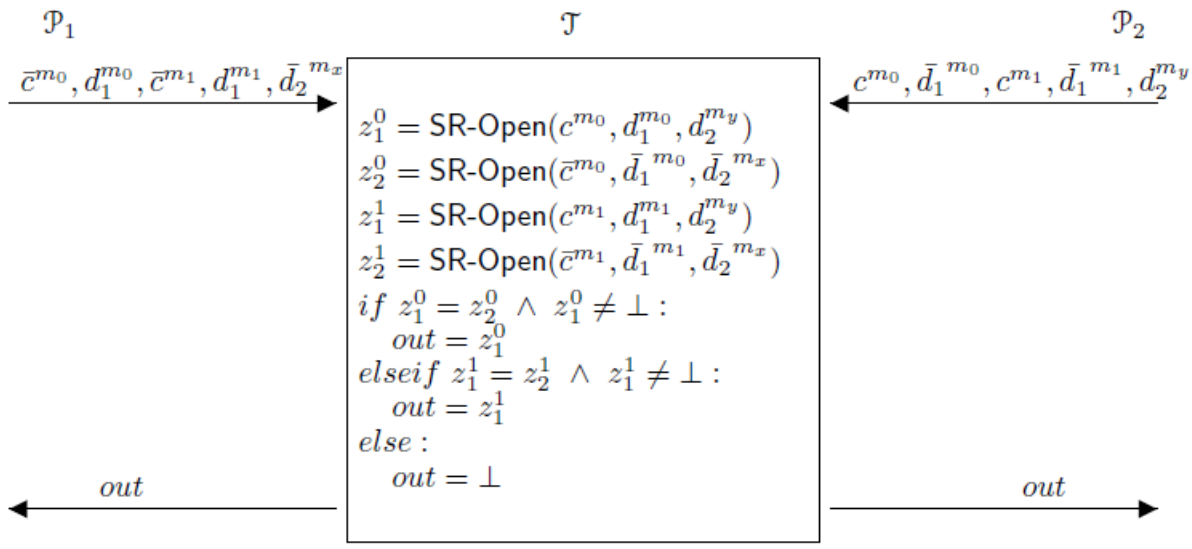

Figure 23: DWE° protocol

the same value they both will receive $\perp$. Note, that for consistency model we need asymmetric version of DWE°, where sender do not receive any output values, as opposite to a symmetric DWE° pictured in the figure above.

6.3 Circus protocol description

Word circus refer to a series of choreographed acts given by travelling clowns, acrobats, trained animals etc. The word also describes the performance that they give, which is usually a series of acts that are choreographed to music and introduced by a "ringmaster". We hope that such name properly resembles the flow of values in the protocol, that we will define further, and provides appropriate mnemonic experience.

We assume that all primitives mentioned above exist and treat them as an ideal black-box implementations. Additionally parties have fixed a function $F(x, y)$ they want to compute before protocol run. From a bird's eye view Circus protocol can be describes as following. Parties execute split receipt commitment protocols and obtain commitments to messages "0" and "1". Then they execute two rounds of COT protocols $\mathcal{P}_1$ with $\mathcal{P}_2$ on receiving end and the other way around (some details on this will follow). Then parties use DWE providing commitments from the beginning and results of corresponding COT-s. Then they output whatever DWE output was.

Figure 24 describes Circus with black-box implementations of subprotocols defined above.

Consistently with the notion used before, every value with a bar, i. e. $\bar{f}()$ is originated at the receiver's side, and values without bar on the sender's side. Additionally note, that however

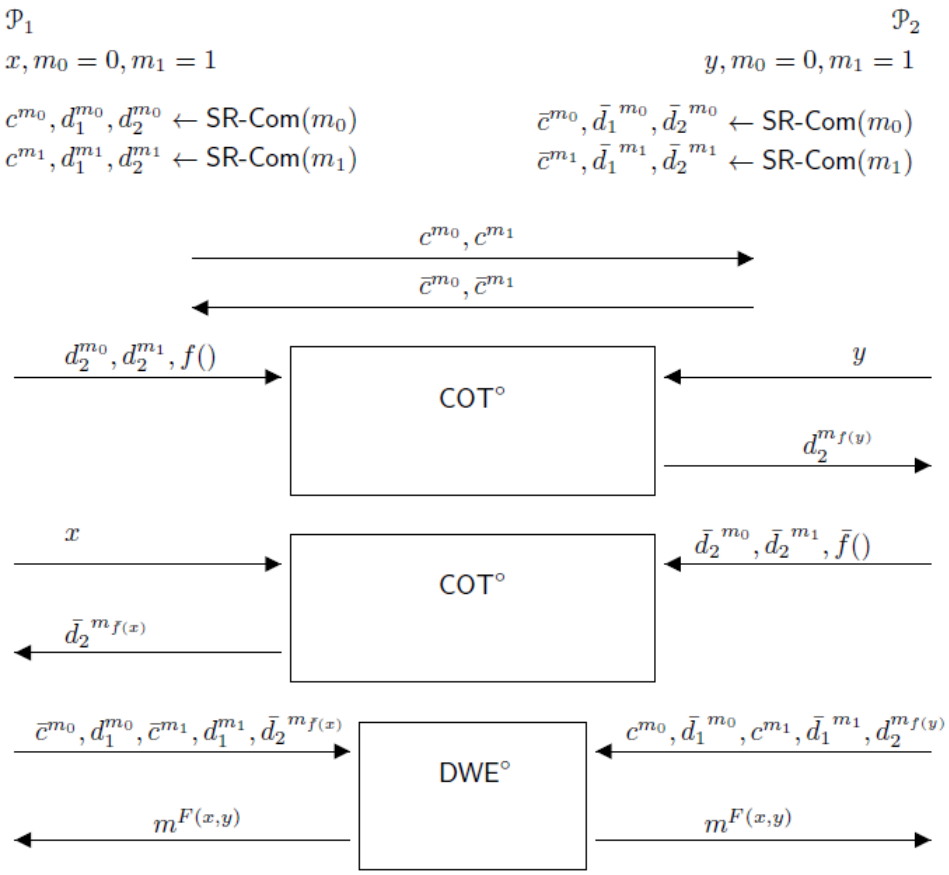

Figure 24: Circus protocol

parties have fixed a boolean function $F(x, y)$ to compute, they submit other functions to COT° protocols. A honest $\mathcal{P}_1$ will submit function $\bar{f}(y) = F(x, y)$ and a honest $\mathcal{P}_2$ is expected to submit function $\hat{f}(x) = F(x, y)$. Let us also assume that the function $F(x, y)$ is such, that both parties want to obtain the same value: $z_i = F(x, y)$. Also both parties know that function $F(x, y)$ is a predicate, so output can be either 0 or 1, so honest parties will specify $m_0 = 0$ and $m_1 = 1$. Now we have fixed all important details and can prove the following lemma.

Lemma 4. *If* $\mathsf{SR\text{-}Com}$ *is a* $(t, \varepsilon_{binding})$*-split receipt commitment scheme, the following statements hold. If* $\mathcal{P}_1$ *is malicious,* $\mathcal{P}_2$ *receive either* $F(x, y)$ *or* $\perp$*, where* x *is the message submitted to second instance of* COT°*, with probability* $1-\varepsilon_{binding}$*. If* $\mathcal{P}_2$ *is malicious,* $\mathcal{P}_1$ *receive either* $F(x, y)$ *or* $\perp$*, where* y *is the message submitted to first instance of* COT°*, with probability* $1 - \varepsilon_{binding}$*.*

Proof. First, we investigate malicious $\mathcal{P}_1$ case. Consider the following decommitment triples that

are involved in DWE° protocol:

$$
\begin{aligned}
t_0 &= (\bar{c}^{m_0}, \bar{d_1}^{m_0}, \bar{d_2}^{m_0}) \ , \\
t_1 &= (\bar{c}^{m_0}, \bar{d_1}^{m_0}, \hat{d}_2^{m_{\bar{f}(y)}}) \ , \\
t_2 &= (\bar{c}^{m_1}, \bar{d_1}^{m_1}, \bar{d_2}^{m_1}) \ , \\
t_3 &= (\bar{c}^{m_1}, \bar{d_1}^{m_1}, \hat{d}_2^{m_{\bar{f}(y)}}) \ .
\end{aligned}
$$

Decommitment triples t_0 and t_2 are constructed by honest $\mathcal{P}_2$ and thus open to respectively m_0 and m_1. As split receipt commitment scheme $\mathsf{SR\text{-}Com}$ is $\varepsilon_{binding}$-binding, each of the triples t_1 and t_2 can only be opened to respectively m_0, m_1 or $\perp$. If malicious $\mathcal{P}_1$ can, for instance, open triple t_1 to another value, we can straightforwardly construct adversary winning split receipt commintment binding game. Additionally, both those triples can be open to non-$\perp$ values with negligible probability, so only one of these triples will produce a non-$\perp$ value.

Thus, using value provided by malicious $\mathcal{P}_1$, which is $\hat{d}_2^{m_{\bar{f}(y)}}$, DWE° can open exactly one of the triples generated by $\mathcal{P}_2$ to exactly the value $\mathcal{P}_2$ has commited to.

Now, note that, $\mathcal{P}_2$ is honest, so when it provides value $d_2^{m_{\bar{f}(x)}}$ it opens right commitment triple generated by $\mathcal{P}_1$ as $\bar{f}(x) = F(x, y)$.

So now, DWE° will produce correct output, cause $\mathcal{P}_1$'s commitment will be opened using correct decommitment value computed by $\mathcal{P}_2$ and $\mathcal{P}_1$ cannot open commitment values originated at the receiver's side to anything else, but intended values with probability more than $\varepsilon_{binding}$. In case $\mathcal{P}_1$ tries to maliciously open commited values, DWE° will result in $\perp$ as an output for $\mathcal{P}_2$.

Exactly the same reasoning shows that in case of malicious $\mathcal{P}_2$, $\mathcal{P}_1$ will receive correct output or $\perp$ from DWE°, if $\mathcal{P}_2$ does not succeed in double opening commitment values which occurs with probability $\varepsilon_{binding}$. □

6.4 Security of symmetric circuit evaluation

There are opinions that this protocol proposed by Mohassel and Franklin [2] is flawed and does not provide security against malicious adversary even in *k-leaked* model. Kiraz and Schoenmakers [14] propose the following case where malicious adversary can learn a bit of other's party input without revealing being corrupted. Assume parties want to compute which 3-bit input is greater. Let $\mathcal{P}_1$ be corrupted and $\mathcal{P}_2$ a honest one. Suppose they have respectively inputs $x = 4 = 100_b$ and $y = 7 = 111_b$, where notion n_b stands for number n in binary. Now $\mathcal{P}_1$ constructs a garbled circuit and submits 100_b as its input there. Then, when it receives another garbled circuit (constructed by $\mathcal{P}_2$) for evaluation it sets $5 = 101_b$ as its input. Both circuits evaluations will return that $\mathcal{P}_2$'s input is greater, so conditional disclosure of secrets won't find corruption and $\mathcal{P}_2$ will not know that $\mathcal{P}_1$ was cheating. On the other hand $\mathcal{P}_1$ now concludes that $\mathcal{P}_2$'s input is greater than 100_b and 101_b and thus second bit of $\mathcal{P}_2$'s input is 1. Information leaks, protocol is flawed [14]. However, we must notice, that this is not actually an attack, and this trick can be

pulled also in the ideal model.

Let's say that adversary want to use this modifying input technique to learn a bit from other party's input in consistency model. So let an adversary $\mathcal{A}$ be engaged in computing function $\mathsf{gt}(x, y)$, where x and y are three bit numbers. Adversary submits $4 = 101_b$ as its input value to $\mathcal{T}$. Now it forms a halting predicate $\pi(y) = \mathsf{gt}(4, y) \neq \mathsf{gt}(5, y)$ and checks if $\mathcal{T}$ response afterwards is $\perp$ or not. As receiver's input was 7, predicate is false, and parties will know that receiver's input is greater. But by the same reasoning as in real protocol attack described above, adversary knows that $\mathcal{P}_2$'s input is greater that 5 and can deduce information about receiver's input. So this attack exists also in ideal model and thus do not reduce security of real protocol compared to ideal one.

Now we have shown that most straightforward accusations of insecurity of the protocol do not hold, so we proceed with proving security of Circus protocol and proving that it provides consistency of computation.

6.4.1 Simulator construction

In this section we show how to construct simulator $Sim^{\mathcal{P}_1}$ with which $\mathcal{P}_1$ could communicate such that the output of protocol run is indistinguishable from a run without simulator. Let's say parties want to compute function $F(x, y)$ of their respective inputs x and y. As we are dealing with consistency model, Sim relies on halting predicate for $\mathcal{T}$ to provide consistency of outputs in real model.

Before anything else, Sim generates randomness r_1 to initialize malicious $\hat{\mathcal{P}}_1$, that Sim will use internally and randomness r_2 that will be used to simulate actions that real world $\mathcal{P}_2$ would have taken in the real protocol run. These values will ensure, that we can align every real protocol run to a specific run of ideal protocol, where parties are initialized with exactly those values as randomness source.

Protocol starts with parties creating commitments for messages $m_0 = 0$ and $m_1 = 1$ and then proceeds with parties sending commitment values to each other. As the adversary is malicious it can send arbitrary values instead of properly generated ones, so Sim receives from $\mathcal{P}_1$ values $\hat{c}^{m_0}$, $\hat{c}^{m_1}$. We will use hatted values to denote that this message can contain arbitrary value. Then Sim generates valid commitment triples $(\bar{c}^{m_0}, \hat{d_1}^{m_0}, \hat{d_2}^{m_0}) \leftarrow \mathsf{SR\text{-}Com}(0)$ and $(\bar{c}^{m_1}, \hat{d_1}^{m_1}, \hat{d_2}^{m_1}) \leftarrow \mathsf{SR\text{-}Com}(1)$ and sends $\bar{c}^{m_0}$ and $\bar{c}^{m_1}$ to $\mathcal{P}_1$.

At this point sender is ready to start with COT° protocols, so Sim receives $\hat{d_2}^{m_0}, \hat{d_2}^{m_1}, f()$. $\mathcal{P}_1$ does not want anything in response here, so Sim just proceeds with second instance of COT°.

For the second instance of COT° sender specifies its input x. Note, that sender is malicious, and this submitted value can be different from intended input, but that just makes this x an "actual" input of sender. In response Sim sends $\bar{d_2}^{m_0}$ to $\mathcal{P}_1$. As split receipt commitment scheme is ε_{hiding}-hiding, adversary do not know to which bit value was $f(y)$ evaluated. Additionally, note that it is not important which decommitment value will be sent to $\mathcal{P}_1$, as following DWE° is also simulated and not actually computed.

At this point Sim knows sender's actual input, so Sim can perform ideal protocol run with $\mathcal{T}$. So Sim submits x to $\mathcal{T}$ as sender's input. Then Sim constructs the following halting predicate:

$$\pi(y) = \mathsf{Circus}(\hat{\mathcal{P}}_1(x_i, r_1), \mathcal{P}_2(y, r_2)) \stackrel{?}{=} (?, \perp) \ ,$$

where ? means, that we are not interested in sender's output value, predicate just makes sure that receiver will get $\perp$ as the result, and x_i is the intended input for $\mathcal{P}_1$. Note, that this predicate is efficient as it takes the same time to evaluate predicate as Circus protocol run takes, so predicate's overhead is polynomial, which is acceptable.

This predicate can be constructed cause Sim knows description of corrupted $\mathcal{P}_1$ and description of how honest $\mathcal{P}_2$ must behave. Now if we are in *1-leaked* model, then Sim will receive either an output value or $\perp$ from the $\mathcal{T}$. Then Sim proceeds with receiving five values from the sender for DWE°. To finish the protocol Sim responds to $\mathcal{P}_1$ with whatever value it received from $\mathcal{T}$ at the previous step.

Similarly, if we are talking about consistency model where Sim does not receive anything from $\mathcal{T}$ after submitting halting predicate. But, as we are in the consistency model, sender does not want to receive anything as a result of final DWE° protocol run. So after submitting halting predicate to $\mathcal{T}$, Sim just receives five values as sender's input to DWE° and does not do anything else.

Such simulator obviously implements interface to successfully communicate with real world $\mathcal{P}_1$, so to prove security of the protocol we need to prove that outputs of parties coincide in real and ideal models. Proving equivalence of outputs is trivial due to the specific nature of halting predicate. Note, that after receiving such halting predicate $\mathcal{T}$ evaluates if this protocol run must finish with output values or $\perp$ values. Note, that in any case $\mathcal{P}_2$ will receive the same output value. If the protocol run must end with $\perp$ value as output for $\mathcal{P}_2$, halting predicate will be true, so $\mathcal{T}$ halts and sends $\perp$ to $\mathcal{P}_2$. If protocol must end with an actual value output for $\mathcal{P}_2$, halting predicate will be false, and $\mathcal{T}$ will return actual value $F(x, y)$ as the output for $\mathcal{P}_2$. Note, that as we have aligned randomness before protocol run, every run of the real protocol is matched to exactly one run of the ideal protocol with equivalent outputs for receiver. Thus distribution of receiver's output coincide in both real and ideal models.

To prove equivalence of the sender's outputs, note, that as the result of DWE° simulator submits whatever value $\mathcal{T}$ responded as protocol output for $\mathcal{P}_1$. So $\mathcal{P}_1$'s output is the same in real and ideal models. Note, that before DWE° protocol, $\mathcal{P}_1$ sees only commited values, which, as split receipt commitment is ε_{hiding}-hiding, ensure, that does not know actual values involved in current computation, so its output in real model will coincide with the output in ideal model. Adversary sees all values corrupted $\mathcal{P}_1$ knows, but except $\mathcal{P}_1$' inputs and output, all intermediate values are generated randomly by Sim. So adversary's output must also be the same in real and ideal models. Thus distributions of the outputs coincide in real and ideal models, which proves that protocol is secure.

Similarly for consistency model, $\mathcal{P}_2$'s output in the same in both real and ideal models.

Sender does not receive any output, so its output $\varnothing$ is the same in both models. Due to the same argument as for *1-leaked* model, output of the adversary also coincides in both models.

Note, that 1 bit about other party's input is leaked due to halting predicate nature. If function $F(x, y)$ is symmetric, the same halting predicate leaks 1 bit to the adversary in the consistency model also.

Note that, however we do not provide exact specification of simulator for the corrupted receiver case, one can construct a simulator similar to one above that uses the same technique for halting predicate construction to ensure equivalence of outputs in real and ideal models.

6.5 Circus construction scheme

Constructing a general scheme implementing Circus for predicates is straightforward. We need a split receipt commitment scheme, which can be constructed from a usual commitment scheme. Asymmetric Yao garble circuit protocols boxes can be constructed as described above with one-time-pad array encryption scheme, using, for example, AES as pseudorandom generator.

We also can construct a DWE protocol, as a special case of Conditional Disclosure of secrets protocol, for reference, how to construct that refer to Sven Laur and Helger Lipmaa paper [17].

Additionally note, that however we do not present a rigorous proof, but, intuitively, Circus protocol is parallelly composable. To prove this fact, we must show a simulator construction for parallel execution of several Circus protocol runs. Such a simulator can be constructed using several simulators for a single Circus protocol run, however exact prove of such simulator existence is left beyond the scope of current thesis to preserve its space and readability. Next, we show the Circus protocol can be constructed and provide experimental result of its implementation.

7 Experimental results

We investigate possible implementation of Circus protocol as a case study. First, we describe how we could have implemented necessary subprotocols and then discuss the implementation and its performance.

7.1 Adaptation for MPC-platform

Descrined by Dan Bogdanov, Sven Laur and Jan Willemson [18], Sharemind is implemented as a platform for privacy preserving computations [19]. Basically, Sharemind is a virtual machine that consist of three *miner* machines that can perform math operations on additively shared integers. It does so in a such way, that private values (those that no individual miner must know) before being put into a virtual machine are shared between miners. Additive sharing of integer n between miners $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ means that three random integers s_0, s_1 and s_2 such that $n = s_0 + s_1 + s_2$ are created (s_0 and s_1 are generated randomly and s_2 computed as $s_2 = n - s_0 - s_1$) and sent to respective miners. Now it is easy to perform basic math operations on shared values. For instance to add two shared values x and y shared as s_0^x, s_1^x, s_2^x for x and s_0^y, s_1^y, s_2^y for y each miner just computes its share of $z = x + y$ as sum of its shared for x and y. Indeed, $s_0^x + s_0^y + s_1^x + s_1^y + s_2^x + s_2^y = x + y = z$. In a similar manner subtraction and multiplication can be performed. However, computing more difficult functions on additively shared values, consider division for example, needs trickier protocols. We will use envelope notation $[\![x]\!]$ to denote operations performed on shared value x. A result of operation on shared values is also a shared value, so, for example, notion $[\![z]\!] = [\![x]\!] + [\![y]\!]$ means that parties compute shared value z as a sum of shared values x and y.
A shared value becomes publicly available when all miners send their shares for this value to all other miners. This is known as publishing a value, which we will denote as $\mathsf{pub}_{\mathcal{P}_i}([\![x]\!])$, where index shows to which miner value is published. If no index is specified, it means that value is made public and available to all miners.

Having this multiparty computation platform we can construct protocols with more ease than by using other cryptographic primitives. Note, that due to the fact that consistent computation cannot be ensured in three-party computation against maliciously corrupted party [16], this protocol cannot be securely implemented in current Sharemind platform.

Obviously, slow versions of protocols described below, that are based on some two-party computation schemes and involve only two out of three miners are possible in Sharemind, however such implementation does not interest us in the scope of current thesis.

7.1.1 Commitment scheme in MPC-platform

Commitment among miners that additively share values goes in the following manner. Assume that miner $\mathcal{P}_1$ wants to commit message m. It generates random $d \leftarrow \mathbb{Z}_{2^n}$, where n is length

of value that can be shared. Sharemind operates on 32-bit integers, so in its case $n = 32$. Additionally, every commitment operation gets a public identifier id, just for miners to be sure that they all operate on the same commitment. When this id and d are ready, $\mathcal{P}_1$ shares d and m, and notifies other miners what is current commitment id.

To open a commitment miner, let's say it is $\mathcal{P}_2$, shares its version of decommitment value d^*, then miners compute message as:

$$\begin{aligned}&[\![m^*]\!] = [[\![d]\!] \stackrel{?}{=} [\![d^*]\!]] \cdot [\![m]\!] \quad , \\ &\mathsf{pub}_{\mathcal{P}_2}(m^*) \quad .\end{aligned}$$

So now, if $\mathcal{P}_2$ knew or guessed correctly decommitment value it gets message m. This commitment scheme is only statistically hiding, as $\mathcal{P}_2$ can guess value of d.

7.1.2 Oblivious transfer in MPC-platform

Oblivious transfer in MPC platform that uses additive shares can be achieved in the following way. Let's name three miners of Sharemind as $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$. Suppose $\mathcal{P}_1$ has two values x_0 and x_1 and $\mathcal{P}_2$ knows bit b. We want to perform an oblivious transfer so $\mathcal{P}_2$ learns x_b, but not x_{1-b} and no miner learns b. To do so, $\mathcal{P}_1$ shares x_0 and x_1 between miners and $\mathcal{P}_2$ shares b. Then miners compute message m.

$$[\![m]\!] = [\![x_0]\!] \cdot ([\![1]\!] - [\![b]\!]) + [\![x_1]\!] \cdot [\![b]\!] = \begin{cases} [\![m_0]\!] & \text{if } b = 0 \\ [\![m_1]\!] & \text{if } b = 1 \end{cases}$$

After this computation all miners have their shares of m and after $\mathsf{pub}_{\mathcal{P}_2}(m)$, $\mathcal{P}_2$ learns m_b.

7.1.3 Conditional disclosure of secrets in MPC platform

Conditional disclosure of secrets protocol also is available between machines that additively share values. For Circus protocol we need *disclose when equal* DWE version of conditional disclosure of secrets, which releases secret if both parties can open other party commitment for the same value. Due to the fact that opening a commitment leaves message shared between miners makes DWE very straightforward. Assume that commitment was done as described above, c^m is just a publicly available identifier of the commitment, message m is shared between miners into $[\![m]\!]$, and a decommitment value d^m is generated. Now this decommitment values is additively shared into two parts $d^m = d_1^m + d_2^m$, and values d_1^m and d_2^m are shared between all miners. A party that wants to open commitment using decommitment value d_2^*, shares it, then parties compute $[\![m^*]\!] = [[\![d_2^m]\!] \stackrel{?}{=} [\![d_2^*]\!]] \cdot [\![m]\!]$ and publish m^* to that party.

So $\mathcal{P}_1$ has commitment values $c^{m_0}, d_1^{m_0}$ and $c^{m_1}, d_1^{m_1}$, and decommitment value d_2^{*1} to some $\mathcal{P}_2$'s commited value. On the other hand, $\mathcal{P}_2$ has its commitment values $\bar{c}^{m_0}, \bar{d_1}^{m_0}$ and $\bar{c}^{m_0}, \bar{d_1}^{m_0}$ and decommitment for $\mathcal{P}_1$'s commitment d_2^{*2}. Parties then share all these values between all min-

ers and proceed with opening four possible triples of decommitment values and obtain commited messages.

$$[\![z_1^0]\!] = [[\![\bar{d}_2{}^{m_0}]\!] \stackrel{?}{=} [\![\bar{d}_2{}^{*_1}]\!]] \cdot [\![m_0]\!]$$
$$[\![z_2^0]\!] = [[\![d_2^{m_0}]\!] \stackrel{?}{=} [\![\bar{d}_2{}^{*_2}]\!]] \cdot [\![m_0]\!]$$
$$[\![z_1^1]\!] = [[\![\bar{d}_2{}^{m_1}]\!] \stackrel{?}{=} [\![\bar{d}_2{}^{*_1}]\!]] \cdot [\![m_1]\!]$$
$$[\![z_2^1]\!] = [[\![d_2^{m_1}]\!] \stackrel{?}{=} [\![\bar{d}_2{}^{*_2}]\!]] \cdot [\![m_1]\!]$$
$$[\![m]\!] = [[\![z_1^0]\!] \stackrel{?}{=} [\![z_2^0]\!] \wedge [\![z_1^0]\!] \neq 0] \cdot [\![z_1^0]\!] + [[\![z_1^1]\!] \stackrel{?}{=} [\![z_2^1]\!] \wedge [\![z_1^1]\!] \neq 0] \cdot [\![z_1^1]\!]$$

Now, $[\![m]\!]$ is shared value of the output of DWE so miners either publish it to one of the parties $\mathcal{P}_1$ and $\mathcal{P}_2$ or to both, depending on intentions of DWE.

7.2 Implementation of COT

One of the most important parts of Circus protocol is COT$^\circ$ subprotocol, which implements, actually, Yao garbled circuit protocol functionality.

Practical part consists of two main problems: implement COT and incorporate COT functionality into Sharemind platform. Here we provide an overview of how we solved these problems, for finer details refer to Appendix A, which includes source code of our solution and documentation about it.

We have implemented COT in Python 2.7 as a single program that implements functionality for both sender and receiver parties involved in COT. Despite the fact that this thesis mainly concerns evaluation of predicates, our solution can be used to evaluate functions with more than 1 bit output.

Boolean circuit description for a specific function is provided with a file that follows following format, which is best described on an example. Fulladder function takes input bits a, b and curry bit c and evaluates expression $a + b + c$. Such evaluation produces two-bit result. Consider the following circuit description for Fulladder function.

$$g1: \ xor\ a\ b$$
$$g2: \ xor\ g1\ c$$
$$g3: \ and\ a\ b$$
$$g4: \ and\ c\ g1$$
$$g5: \ or\ g3\ g4$$
$$adder\ g2\ g5$$

Elementary boolean functions, for example, and are specified using prefix notation and literal

name of the function and , so to specify expression $a \wedge b \wedge c$ one can write *andandabc* into circuit description. Supported elementary functions are and, or, xor, eq, not and nand. For every such elementary function a gate in the final circuit will be created. Each line of the circuit description file starts with some gate name followed by ":", where gate name is a single strings that does not contain spaces. On the right side of such ":" goes the description of gate functionality. Line $g1 : \ xor \ a \ b$, specifies that gate $g1$ will compute function $a \oplus b$. As circuit file parser have no information what a and b are, it will assume these are names of input wires of the circuit. Gate description can contain input wires names and names of other gates, so gate $g5 : \ or \ g3 \ g4$ computes or function of outputs of gates $g3$ and $g4$. Additionally, every gate can be specified to compute more complex boolean functions, for example $g1 : \ or \ and \ x \ y \ not \ z$ is a valid declaration of a gate. Last line of circuit description file is a space delimited string, where first element is circuit name and all other elements specify outputs of which gates are considered circuit outputs. In the example above, circuit name is *adder* and it outputs two bits: one from gate $g2$ and one from gate $g5$. Such format allows to construct sufficiently complex boolean circuits.

Our solution parses such description file and if a given file does not follow the convention described above will notify user about it. If circuit file is successfully parsed, we construct a boolean circuit and can proceed with garbling it.

Implemented procedure for construction of garbled circuit precisely follows the description of Yao garbled circuit construction given previously in this thesis. Namely, for every wire of the circuit we generate two random bit-strings 128 bit long. We flip every wire, except output wires with probability one half. And then encrypt truth-tables of the gates according to flipped wires values.

We use array encryption scheme with $128 - \text{bit AES}$ as pseudorandom generator. Sender then can send description of garbled circuit and input keys that correspond to his input values to the receiver.

Receiver needs to get keys that correspond to his input values. To accommodate that, we have incorporated into our solution some sort of OT protocol that uses RSA encryption mechanism to encrypt keys. We do not claim that this mechanism is secure, however, it is quite intense computationally, which is desired property for us to be able be more sure in our performance results and serves its goal to transfer keys to the receiver. Note, that because of this you will need freely available *rsa-2.0-py2.7.egg* Python module to be able to run our solution.

Evaluation of the garbled circuit also follows the description given previously in this thesis. Receiver evaluate the circuit gate by gate until all output wires will get values. Then receiver collects those values and outputs them.

Now we describe how to integrate our python solution into Sharemind platform. Sharemind is a stack based virtual machine. This enables easy integration of several protocols together. Protocol takes input values from virtual machine stack, performs necessary computation and puts output values to stack. Additionally, as Sharemind essentially consists of three miner machines that run the same code, it provides *MPI*-like syntax and templates for sending and

receiving messages between miners and for synchronization of miners.

Sharemind platform is not publicly available, so we provide just source files that integrate with our solution. For more details, refer to Appendix A.

To use Python code from C++ code of the Sharemind one needs to provide link to "Python.h" header file at compilation time of the Sharemind and link it to a Python dynamic libraries. For information about how exactly this must be done refer to Python specification. Further integration with Sharemind is very straightforward, we call Python functions by name and can transfer information from Python objects to C++ code.

For testing purposes Sharemind provides to our solution just a platform for networking between miner machines. The setting is then such that first miner is the sender $\mathcal{P}_1$, second miner is the receiver $\mathcal{P}_2$. Third miner does not perform computation.

7.2.1 Performance results and notes about implementation

Tests were performed on a single machine with Intel(R) Core(TM) i5 CPU M560 @2.67GHz 2.67Ghz processor and 3.83 GB of usable RAM. Python version is 2.7.1, Sharemind version is 2.0.0.

Tests include construction of garbled circuit for function $\mathsf{gt}(\mathsf{x},\mathsf{y})$, which outputs 1, if $x > y$ and 0 otherwise. Both inputs x and y are 32-bit integers. Giver boolean circuit for this function consists of about 250 gates. Tests were performed as standalone application, when one Python process performed actions of both parties, and in Sharemind platform, where one of the miners was playing role of the sender and other of the receiver.

Standalone application. Tests were run by execution of the main program from file *expressions.py*, which results in performing actions of both parties of the protocol by a single Python process. Additionally, it provides some profiling information which parts of the program take most of the time.

Tests included 10 independent runs with randomly generated input values, whose average time to complete was 4.5 seconds. Almost half of this time were spent in OT protocols to transfer keys to the receiver and about 20 percent of the time were spent by stretching keys for array encryption scheme.

Sharemind platform. Tests on Sharemind platform were run using file *TestingProtocol.cpp*. We count only time spent in the protocol itself, without any virtual machine or miners initialization time. Similar to standalone application tests, 10 independent runs of evaluation $\mathsf{gt}(\mathsf{x},\mathsf{y})$ were performed. Average time taken by protocol run was about 8.2 seconds. We conclude that about 4 seconds were spent on networking and serialization of information about circuit. Network consumption were about 200KB per test. This consists mainly of description of garbled circuit and sender's and receiver's keys.

Notes. First of all, we need to say, that timing results that we produced are low compared to results of benchmarking solution Shelat and Shen solution [15]. There are two big reasons for that, first of all we have used a function which is almost twice smaller than the one used by Shelat and Shen. Additionally, we have benchmarked only a part of the protocol that is secure in malicious model. The whole protocol will consist of two such evaluations and some additional overhead for other subprotocols. Despite the fact that efficient implementation of those subprotocols is possible, as we described above, they still will provide noticeable overhead. Another important detail to note is that Python is considered about hundred times slower that C++. So without doubt, performance of the solution created within this thesis can be dramatically improved by switching to a faster programming language. Additionally, our code is, probably, not as efficient as possible, so some improvement can be gained there. Moreover, created application implements a secure protocol, but by no means claims to be secure. There are number of reasons why it is insecure, for instance, it uses Python's module *pickle* for serialization, which leads to security issues by design.

This is a proof-of-concept piece of software and it needs to be treated as such. However, performance results that we provided above were stable enough and can be used as rough estimate of performance of our protocol.

8 Conclusion and future work

In this thesis we presented an independent proof that Yao garbled circuit protocol is secure against semi-honest adversary. We proposed a new protocol Circus, based on Yao garbled circuit protocol, for secure two-party computation problem. We proved that Circus is secure against malicious adversary in consistency model. We implemented the trickiest of subprotocols for Circus in Python and integrated this solution with Sharemind platform. Performance results, that we achieved, give hope, that properly implemented Circus can be compared by speed with the newest and fastest protocols known before. Obvious continuation of this paper will be improving performance of the implemented COT protocol and implementation of other subprotocols for Circus that we left beyond scope of this thesis.

Tõeväärtusskeemide evalueerimise tehnikad turvaliste arvutuste jaoks

Magistritöö (30 EAP)
Oleg Šelajev

Resümee

Tõeväärtusskeemide arvutamine on võimas arvutuste mudel, milles arvutamiseks kasutatakse tõeväärtustabelitega väravaid, mis realiseerivad elementaarseid loogika funktsioone nagu ja, või jne. 1982. aastal pakkus A. A. Yao välja uldise protokolli, kuidas saab kasutada modifitseeritud skeemi kahe osapoole turvaliseks arvutamiseks. See sai nimeks Yao krüpteeritud arvutusskeem. Kahe osapoole vahel turvalise arvutamise ülesanne koosneb sellest, et kaks osapoolt tahavad arvutada funktsiooni privaatsete sisenditega nii, et kumbki osapool ei saa teada, milline oli teise osapoole sisend.

Yao krüpteeritud arvutusskeem on turvaline pool-ausas mudelis, kus ründaja käitub protokolli järgi ning lihtsalt uurib saadud andmeid vastasosapoole sisendi leidmiseks. Juhul, kui ründaja ei järgi protokolli, saab ta turvalisust rikkuda ja informatsiooni sisendi kohta leida. Viimase vältimiseks on leiutatud erinevaid tehnikaid, mis tavaliselt kasutavad mitme Yao krüpteeritud arvutusskeemi korraga.

Antud töö põhitulemus on Yao krüpteeritud arvutusskeemi protokollil põhineva Circus protokolli kirjeldus ning tõestus, et ta on turvaline konsistentses mudelis. Konsistentses mudelis arvutamisel kas mõlemad osapooled saavad korrektse tulemuse ja nende sisendid jäävad privaatseteks või aus osapool saab teada, et teine osapool ei käitunud protokolli järgi.

Töös esitatakse range tõestus, et Yao krüpteeritud arvutusskeem on turvaline pool-ausas mudelis ning kirjeldatakse kõik vajalikud alamprotokollid Circus protokolli koostamiseks. Lisaks tõestatakse antud töös Circus protokolli turvalisuse omadused ja nende olemasolu jaoks vajalikud eeldused. Teoreetiliselt ei ole Circus protokoll aeglasem, kui kõige paremad protokollid kahe osapoole vahel turvalise arvutamise puhul, kus üks osapool on kuritahtlik.

Praktilises osas luuakse Yao krüpteeritud arvutusskeemi implementatsioon, mis võib olla aluseks Circus protokolli realisatsiooni jaoks. Vaatamata sellele, et töö raames luuakse kontseptsiooni tõestav rakendus, siis selle implementatsiooni jõudlustestid näitavad, et Circus protokoll võib olla tõesti kiirem, kui olemasolevad lahendused. Lisaks luuakse Sharemind platvormi moodul, mille abil saab integreerida krüpteeritud skeemide arvutamise protokolli Sharemindiga.

References

[1] Andrew Chi Chih Yao. *Protocols for secure computations (extended abstract)*. In FOCS, pp. 60164, 1982.

[2] Payman Mohassel, Matthew Franklin. *Efficiency Tradeoffs for Malicious Two-Party Computation*. In the 9th PKC conference, Springer-Verlag (LNCS 3958), pp. 458-473, 2006.

[3] Sven Laur. *Cryptographic Protocol Design*. Doctoral dissertation, Helsinki University of Technology, 2008. TKK Dissertations in Information and Computer Science, TKK-ISC-D2

[4] Wikipedia. *Image of Full Adder circuit*. `http://en.wikipedia.org/wiki/File:Full_Adder.svg`, last checked 19.05.2011

[5] Hadi Ahmadi, Esmaeili Salehani. *A Modified Version of SNOW2.0*. In ICSICC'07, 2007

[6] Gilles Brassard, David Chaum, Claude Crepeau *Minimum Disclosure Proofs of Knowledge*. Journal of Computer and System Sciences, vol. 37, pp. 156-189, 1988.

[7] Torben Pryds Pedersen. *Non-Interactive and Information-Theoretic Secure Veriable Secret Sharing*. In Advances in Cryptology - CRYPTO 91, 11th Annual International Cryptology Conference, pp. 129-140, 2009.

[8] Liina Kamm. *Homological Classification of Commitment Schemes*. Tartu University, Master's thesis, 2007

[9] Ahto Buldas and Sven Laur. *Knowledge-binding commitments with applications in time-stamping*. Lecture Notes in Computer Science, Volume 4450/2007, pp. 150-165, 2007.

[10] Donald Beaver. *Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority*. In Journal of Computing, Vol 4, pp. 75-122, 1991

[11] Yehuda Lindell, Benny Pinkas *A Proof of Security of Yao's Protocol for Two-Party Computation*. In the Journal of Cryptology, 22(2): pp. 161-188, 2009.

[12] Michael Rabin *How to Exchange Secrets by Oblivious Transfer*. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981.

[13] Tal Rabin, Michael Ben-Or *Verifiable secret sharing and multiparty protocols with honest majority*. STOC '89 Proceedings of the twenty-first annual ACM symposium on Theory of computing, New York, 1989.

[14] Mehmet Kiraz, Berry Schoenmakers. *Securing Yaos Garbled Circuit Construction Against Active Adversaries*. Online Proceedings 1st Benelux Workshop on Information and System Security, WISSEC 2006, Belgium, 2006

[15] Abhi Shelat, Chih-Hao Shen *Two-Output Secure Computation with Malicious Adversaries*. Lecture Notes in Computer Science, Volume 6632/2011, pp. 386-405, 2011.

[16] Sven Laur, Helger Lipmaa *On the Feasibility of Consistent Computations*. Lecture Notes in Computer Science, Volume 6056/2010, pp. 88-106, 2010.

[17] Sven Laur, Helger Lipmaa *Additive Conditional Disclosure of Secrets and Applications*. ACNS 2007, 2005.

[18] Dan Bogdanov, Sven Laur and Jan Willemson. *Sharemind: A Framework for Fast Privacy-Preserving Computations*. Lecture Notes in Computer Science, Volume 5283/2008, pp. 192-206, 2008.

[19] AS Cybernetica. *Introduction to Sharemind*.
`http://sharemind.cyber.ee/introduction-to-sharemind`, last checked 19.05.2011

Appendices

Appendix A

Source code for experimental part of this work is available from a CD attached to a hard copy of this thesis. The CD contains "README" file, which explains how to treat this piece of software.