SECURE CIRCUIT EVALUATION PROTOCOLS IN PRACTICE

THESIS

CHRISTIAN KOLLER NIELSEN , 201303527

MASTER'S DEGREE

April 2020

Advisor: Ivan Bjerre Damgård

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

SECURE CIRCUIT EVALUATION PROTOCOLS IN PRACTICE

CHRISTIAN KOLLER NIELSEN

Thesis

Master's Degree
Department of Computer Science
Science & Technology
Aarhus University

April 2020

ABSTRACT

In modern society we live in a digital information age where we through electronic media interact with a large number of parties whom we have never met and who might have interests which are divergent from our own. Many people have private data online. *These private data provide value when they are used*. Hence, we need to find ways of controlling the leakage of confidential data, while the data are stored, communicated, or computed on, even when the owner of the data does not trust the parties he or she communicates with. *Secure Function Evaluation* is a possible solution to this and therefore it is the subject of interest in this work. This thesis explores Secure Function Evaluation protocols in theory and in practice. More specifically speaking, it analyzes existing protocols and looks into ways of making them work in practice. This is done by implementing the protocols and evaluating in practice, in a real world environment. The findings of experiments carried out ins this work connects the theory with practice.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Secure multiparty computation (MPC) is a field in cryptography where n players can securely evaluate an agreed function based on their inputs and only learn the output of that function whilst keeping their inputs private.

The topic of MPC was formally introduced for the first time by Yao[7] in 1982, where he presented and motivated MPC through the millionaire problem: Assume two millionaires wish to know who is richer, but they do not want to reveal any additional information about each other's wealth. The two millionaires are interested in learning the result of a function that calculates which number is bigger, given their wealth as a private input.

In modern society we live in a digital information age where we through electronic media interact with a large number of parties whom we have never met and who might have interests which are divergent from our own. Many people have private data online, such as information about loans, taxes, diseases or medicine. These private data provide value when they are used. Hence, we need to find ways of controlling the leakage of confidential data, while the data are stored, communicated, or computed on, even when the owner of the data does not trust the parties he or she communicates with.

In many cases added value can be obtained by combining information from several confidential sources. This could motivate competing companies to compute some result from each others data without revealing sensitive data to their competitors. A motivational example for MPC are multiple parties who are interested in computing an intersection of their sets. Let us say, multiple banks are doing fraud detection and therefore want to know if they have any shared customers. Another example of MPC is online voting where n parties vote yes or no and a function determines if the majority voted yes, without revealing what other people voted for and whilst making sure that the result of the function is correct. Yet another example could be online auctions, where n parties make bids and a function correctly evaluates what the highest bid is and who has it, without revealing any bids. In fact, Danish sugar beet farmers have used an MPC action system to trade production rights[8]. A final example is doing distributed confidentiality. The idea is sharing the key and doing distributed encryption and decryption. This eliminates the single point of failure of doing it on one computer and thereby protects the key. The question that arises here is how can this be done without revealing confidential information while ensuring correctness of the result.

One possibility would be that all parties privately give their input(s) to a trusted party, who then does the computation, announces the result, and forgets about all private data he or she has seen. One problem of having a trusted party, however, is that it creates a single point of attack from where all data could be stolen. Another problem is that all parties must trust the trusted party. The reason why there is privacy concerns in the first place is because the parties do not trust each other, so why should we believe that they can find a new party they all trust? This leads us to the question of how this problem can be solved without relying on a trusted party. MPC protocols are one possible answer to this question, and are therefore the subject of this work.

In detail, the aim of this thesis is to explore basic concepts, as well as progress in the field of MPC. More specifically, this work looks at protocols for *secure function evaluation* (SFE), meaning protocols that can evaluate any function. Various MPC concepts and protocols will be explored through both theory and implementation. The aim of this thesis is to compare implementations of different SFE protocols and investigate how the practical implementations compares to the theory.

2 PRELIMINARIES

This chapter present notations and definitions that will be used throughout the thesis.

2.1 DEFINITIONS

Informally an entity is *negligible* if it is too small to matter [3].

Definition 1 (Negligible). *$\epsilon(k)$ is negligible in k for any positive polynomial p where $\epsilon(k) \leqslant \frac{1}{p(k)}$ for all large enough k.*

A distinguisher D plays a game where two probabilistic algorithms U and V run on the same input x. D gets x and now has to guess if the output y is generated from U or V on input x. The output distributions of U and V on input x, this will be refereed to as U_x and V_x.

Definition 2 (Statistical Distance). *Given two probability distributions, P and Q, the statistical distance between them is defined as $SD(P, Q) = \frac{1}{2}\sum_y |P(y) - Q(y)|$, where $P(y)$ is the probability P assigns to y.*

Definition 3 (Statistical indistinguishable). *Given two probabilistic algorithms U and V they are statistical indistinguishable if $SD(U_x, V_x)$ is negligible in the length of the string x. We write this $U \sim^s V$.*

2.2 NOTATION

The parties or players that participate in the execution of the protocol are called $P_1, P_2, ..., P_n$. Each player P_i holds a secret input x_i and the agreed function takes n inputs. The goal is to compute $y = f(x_1, x_2, ..., x_n)$ while making sure that the *correctness* of the protocol is satisfied, meaning that the correct value of y is computed. Furthermore, it is required that there is *privacy*, meaning that y is the only new information released. Computing f such that privacy and correctness are satisfied is refereed to as computing f *securely*[1].

We define a protocol execution in an *ideal world*, where the protocol is executed in an ideal and secure way. Imagine a protocol execution in the ideal world to be the following: All parties privately send their inputs to a box, the box correctly evaluate the agreed upon function and sends the result to the parties. After the protocol the explodes and eliminates all traces of data it has seen.

When you want to look at the security of a protocol you do it with regards to some adversary attacking. The protocol execution in the ideal world is then compared to a protocol execution in the *real world*. Then for all adversaries there exists a simulator, so that the adversary in the real world and the simulator in the ideal world achieve the same effect. Any attack that is done in the ideal world can also be done in the real world, but in the ideal world we have something ideal and secure and thus the affect of any attack in the ideal world is acceptable. Hence we are happy if any attack achieve the same affect in the real world. It is now clear what we must look at the capabilities of the adversary attacking the protocol.

2.3 ADVERSARIES

In MPC protocols multiple players can be corrupted. C is the set of corrupted players and t the maximum number of parties that is allowed to be corrupt whilst security can still be guaranteed. Think of an adversary as a malicious entity which can corrupt a subset of players and which aim to prevent the users of the protocol to achieve their goal. All the corrupted parties could cooperate or be controlled by one entity and therefore a single adversary is modelled to control all corrupted parties[1].

Definition 4 (Passive adversary)**.** *A passive adversary obtains the complete information obtained by the corrupted players but the players still follow the protocol as specified. It will use the information to try to learn more than it should.*

Definition 5 (Active adversary)**.** *An active adversary take full control of the corrupted players and does not need to follow the protocol as specified.*

An adversary can also vary how he corrupts the players. Both passive and active adversaries can be static or adaptive.

Definition 6 (Static adversary)**.** *For a static adversary the set of corrupted players are set before the protocol starts and will remain the same throughout the execution of the protocol.*

Definition 7 (Adaptive adversary)**.** *An adaptive adversary can at any time during the protocol choose to corrupt new players.*

2.4 COMMUNICATION

We will also model what the adversary is allowed to do communication-wise.

Definition 8 (Synchronous model)**.** *In the synchronous model there is clocks that are synchronized such that when a message is sent it will arrive before some time bound. We assume that there is pairwise secure channels, such that the adversary cannot interfere with the network traffic.*

2.5 Complexity

Efficiency of protocols are measured in *computational complexity* and *communication complexity*. Computational complexity is a measure of how many steps is required in the worst case for an input of a given size. The number of steps is measured as a function of that size. Communication complexity is the number of bits sent between the parties on the network.

2.6 Privacy and Robustness

We also define what the players are *allowed* to do and see, as well as what they *actually* can do and see during an execution of a protocol[1].

Definition 9 (Leaked Values)**.** *The leaked values* $\{view_j\}_{P_j \in C}$ *are exactly the information leaked to the corrupted parties during an execution of the protocol.*

Definition 10 (Allowed Values)**.** *The allowed values* $\{x_j, y_j\}_{P_j \in C}$ *are the corrupted players own input and output that they will learn during the execution of the protocol.*

Intuitively, a protocol is private if the leaked values contain no more information than the allowed values. In cryptography you often rely on hard problems were no polynomial time algorithm is known to solve the problem. It therefore make sense to say that a protocol is private if the leaked values can be computed efficiently from the allowed values. The program that efficiently computes the leaked values from the allowed values is called a simulator S. Since parties in MPC protocols can make random choices it is defined that:

Definition 11 (Privacy)**.** *A protocol is private if there exists an efficient simulator* S *such that the simulated values* $S(\{view_j\}_{P_j \in C})$ *and the leaked values* $\{view_j\}_{P_j \in C}$ *have the same distribution.*

We define robustness of a protocol to model the influence an active adversary has on the output of a protocol. The influence an attacker has on the protocol execution, we call the *actual influence* and the influence we allow we call the *allowed influence*. We reuse the approach from privacy as define robustness as such:

Definition 12 (Robustness)**.** *A protocol is robust if there exists an efficient simulator* S *such that for every adversary attacking the protocol,* S *can efficiently compute an allowed influence with the same effect.*

We require one single simulator that simultaneously can demonstrate both privacy and robustness.

The Universally Composable (UC) Model describes security for a cryptographic protocol π. In this model a protocol can be proven secure regardless of the context. We will model protocols through interactive systems and interactive agents[1].

Definition 13 (Interactive agent (IA))**.** *An interactive agents is a computational device that receives and sends messages on named ports and that holds an internal state.*

A set of interactive agents can become an *interactive system* (IS) by connecting the in-port and out-ports with the same name as seen in 1

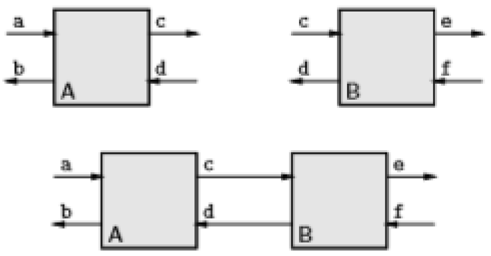

Figure 1: An interactive agent A with $\mathrm{In}(A) = a, d$ and $\mathrm{Out}(A) = b, c$. An interactive agent B with $\mathrm{In}(B) = c, f$ and $\mathrm{Out}(B) = d, e$. And the interactive system $IS = A \diamond B$ with $\mathrm{In}(IS) = a, f$ and $\mathrm{Out}(IS) = b, e$ [1].

Definition 14 (Behavioral Equivalence)**.** *Two interactive systems having different internal structure are behavioral equivalent if they are indistinguishable for and outside observer, meaning that they give the same outputs on the same open out-ports whenever they get the same input on the same open in-ports.*

Ideal Functionality

Imagine an ideal world with an ideal protocol. This ideal protocol is a specification of what we would like the real protocol to do. In the ideal world we model the protocol as an interactive agent and call this an *ideal functionality* F. In the real world we model the protocol π as an interactive system and then the goal is to proof that π is at least as secure as F by showing that they are *behavioral equivalent*.

Informally speaking, secure function evaluation F_{SFE} works like this: All the players will give their input to F_{SFE} which will always calculate the correct result according to its specification, without leaking any information other than the outputs it is supposed to send to the players. F_{SFE} has an input and output port for every player. Furthermore, it has two special ports called the leakage port and influence port which are used for communication with the adversary. When a protocol is executed the in-ports and out-ports are used, but if a player P_i is corrupted then F_{SFE} stops using the i'th in/out-ports and uses the leakage and influence port instead. In this case the adversary will communicate on behalf of P_i. The real protocol and F_{SFE} have different open-port structures and hence cannot be behavioral equivalent. To solve this a simulator S is introduced. If F_{SFE} is composed with this simulator and S works as it is supposed to, you cannot tell the difference between π and F_{SFE} anymore.

The environment

π and F should be equivalent no matter what the context is. The protocol could for example be used as a subroutine in a bigger system which is part of an environment. The environment can be modelled as an interactive system which chooses inputs of the players, uses the protocol and finally receives the result from the protocol. We denote the environment as Z. The environment gets two play with either pi or F. It should not be able to guess which one it is, except with negligible probability close to ½.

2.8 SECRET SHARING

Secret sharing is a method for distributing a secret to multiple parties who each get a share of the secret that reveals nothing about the secret. The secret can be reconstructed when a sufficient number of shares are combined together[1].

Lagrange Interpolation

Given a set C of size m consisting of $t+1$ or more points, which are evaluations of some polynomial f of degree t, we can compute any point on the polynomial like this:

$$f(x) = \sum_{i \in C} f(i)\delta_i(x)$$

where δ_i is

$$\delta_i(x) = (\prod_{j \in C, j \neq i} \frac{x-j}{i-j})$$

Note that one can construct a recombination vector $\mathbf{r_j} = (r^j_1, ..., r^j_m) = (\delta_1(j), ..., \delta_m(j))$ that can be used to compute the value $f(j)$. This recombination vector works for all polynomials of degree lower than $m-1$.

Shamir's Secret Sharing Scheme

This scheme is based on polynomials over a finite field $\mathbb{F}$. To share a secret $s \in \mathbb{F}$ a dealer has to choose a random polynomial f_s over $\mathbb{F}$ of degree at most t where $f_s(0) = s$. Then n shares are generated by evaluating f_s on the points $f_s(1), ..., f_s(n)$. The n shares can then be privately distributed to the n parties. This works because any set of t or fewer shares contains no information on s, but if you have $t + 1$ or more shares the polynomial can be reconstructed by using *Lagrange Interpolation*. This means that if t or less corrupt parties work together and pool their information they can gain no extra information about the secret s.

A secret sharing is defined as a vector:

$$[s; f_s]_t = (f_s(1), f_s(2), ..., f_s(n))$$

Note that this notation states that $deg(f_s) \leqslant t$, $f_s(0) = s$ and it describes the shares $(f_s(1), f_s(2), ..., f_s(n))$. We will vary bet ween the notation $[s; f_s]_t = [s]_t$.

We can do arithmetic operations on the shared secrets by using entrywise addition, scalar multiplication and Schur product on the shares as such:

$$\begin{aligned} [a; f]_t + [b; g]_t &= (f(1) + g(1), ..., f(n) + g(n) &&= [a + b; f + g]_t \\ \alpha[a; f]_t &= (\alpha(1), \alpha f(2), ..., \alpha f(n) &&= [\alpha a; \alpha f]_t \\ [a; f]_t \cdot [b; g]_t &= (f(1) \cdot g(1), ..., f(n) \cdot g(n) &&= [ab; fg]_{2t} \end{aligned}$$

2.9 ALGEBRA

In the MPC protocols explored in this thesis, computations are done in a field $\mathbb{F}$, because they use Shamir's Secret Sharing. This means that the input from the players must be elements in the field and the result will also be an element in the field. Hence, we need to understand how to do arithmetics in a field. To understand this, we need understanding of algebraic structures such as groups, rings and fields[4].

Groups

A pair $(G, \circ)$ consisting of a set G and a composition $\circ$ is called a group if it satisfies the following properties:

(i) The composition is associative: $a \circ (b \circ c) = (a \circ b) \circ c$
for every $a, c, b \in G$.

(ii) There is a neutral element $e \in G$ such that: $e \circ a = a$ and $a \circ e = a$
for every $a \in G$.

(iii) For every $a \in G$ there is an inverse element $t \in G$ such that $t \circ a = e$ and $a \circ t = e$.

A group G is called abelian if $a \circ b = b \circ a$ for every $a, b \in G$. For any natural number n, $(Z_n, +)$ is a group where $+$ means addition modulo n. For multiplication the neutral element is 1 and thereby $(Z_n, \cdot)$ is not a group because 0 does not have an inverse. $(Z_n \backslash \{0\}, \cdot)$ is also not a group. If you look at the element $2 \in Z_6$ and multiply it with any number in Z_6 you would get an even number, and thereby not the neutral element 1. So that means that there is no inverse of 2 mod 6. The solution to making multiplication modulo n be a group operation is to get rid of all the the numbers where $gcd(a, n) \neq 1$.

$$Z_n^* = \{a \in Z_n \mid gcd(a, n) = 1\}$$

Rings

A ring is an abelian group $(R, +)$ with an additional composition called multiplication. Multiplication satisfies the following for every $a, b, c \in R$:

(i) $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

(ii) There is a neutral element $1 \in R$ such that: $1 \cdot a = a$ and $a \cdot 1 = a$

(iii) $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(b + c) \cdot a = b \cdot a + c \cdot a$

Fields

A field is a ring R such that the second operation also satisfies all group properties. i.e $R^* = R \setminus \{0\}$ is called a field. A finite field also called a Galois field only exists if $|\mathbb{F}| = p^n$ where p is a prime and $n \geqslant 1$. When we write $GF(q)$ we mean a finite field of size q. One example would be $GF(11)$. In this example $p = 11$ and $n = 1$, so it is compliant with the requirements of a finite field. Another example is the binary field $GF(256)$=$GF(2^8)$.

Finite fields can be further subdivided into prime fields and extension fields. If $n = 1$ then we have a prime field $GF(p)$ and if $n > 1$ then we have an extension field.

Prime Field Arithmetic

Elements of a prime field $GF(p)$ are the integers $\{0, 1, ..., p-1\}$. For $a, b \in GF(p)$. For $a, b \in GF(p)$ arithmetics are done in the following way:

(i) Addition: $a + b \equiv c \mod p$

(ii) Subtraction: $a - b \equiv d \mod p$

(iii) Multiplication: $a \cdot b \equiv e \mod p$

(iv) Inversion: The Inverse a^{-1} must satisfy $a \cdot a^{-1} \equiv 1 \mod p$. So how do we compute a^{-1}? It can be computed with the extended Euclidean Algorithm[6].

Extended Field Arithmetic

In cryptography we are interested in fields where $GF(2^n)$. The elements of $GF(2^n)$ are polynomials $a_{n-1} \cdot x^{n-1} + ... + a_1 \cdot x + a_0 = A(x) \in GF(2^n)$, where the coefficients $a_i \in GF(2) = \{0, 1\}$. An example is $GF(2^3)$, where we know that any element must have the form $A(x) = a_2x^2 + a_1 + a_0$. We can view that as a 3 bit vector and we can express 8 different numbers with 3 bits which makes sense since we are working with $GF(2^3) = GF(8)$. The elements in the group are $GF(8) = \{0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1\}$. In an extension field $A(x), B(x) \in GF(2^m)$ with the coefficients a_i and b_i addition is done by regular polynomial addition:

$$C(x) = \sum_{i=0}^{n-1} c_i \cdot x_i, \text{ where } c_i \equiv a_i + b_i \mod 2$$

So one just adds or subtracts the coefficients of the polynomials in $GF(2) = \{0, 1\}$. Addition in $GF(2^3)$ with the elements $A(x) = x^2 + x + 1$ and $B(x) = x^2 + 1$ would be:

$$\begin{aligned} A(x) + B(x) &= ((1+1) \mod 2)x^2 + x + ((1+1) \mod 2) \\ &= 0x^2 + x + 0 \\ &= x \end{aligned}$$

For multiplication in $GF(2^m)$ one might think that you can do regular polynomial multiplication as such:

$$\begin{aligned} A(x) \cdot B(x) &= (x^2 + x + 1) \cdot (x^2 + 1) \\ &= x^4 + x^3 + x^2 + x^2 + x + 1 \\ &= x^4 + x^3 + ((1+1) \mod 2)x^2 + x + 1 \\ &= x^4 + x^3 + x + 1 = C'(x) \end{aligned}$$

Here we have a problem since $C'(x) \notin GF(2^3)$. The solution is to reduce $C'(x)$ modulo a polynomial that "behaves like a prime" in the sense that the polynomial can not be factorized. These are called irreducible polynomials. An irreducible polynomial for $GF(2^3)$ is $P(x) = x^3 + x + 1$. Let $A(x), B(x) \in GF(2^m)$ and let $P(x) \equiv \sum_{i=0}^{m} p_i x^j$, where $p_i \in GF(2)$ be an irreducible polynomial, then multiplication is performed as $C(x) \equiv A(x) \cdot B(x) \mod P(x)$. If we for example look at the earlier example we get: $\dfrac{x^4 + x^3 + x + 1}{x^3 + x + 1}$. First we want to get rid of x^4, so we multiply $P(x)$ by x and we get $x^4 + x2 + x$ and then we add it to $A(x) \cdot B(x) = C'(x)$:

$$C'(x) + (P(x) * x) = (x^4 + x^3 + x + 1) + (x^4 + x2 + x) = x^3 + x^2 + 1$$

but this is still not in our field so we reduce again, but only with p(x) this time because we already got rid of x^4.

$$(x^3 + x^2 + 1) + (x^3 + x + 1) = x^2 + x \in GF(2^3)$$

Now we have an element in our field and we have done.
For every field $GF(2^m)$ there are several irreducible polynomials and the result of the arithmetic in the fields depends on that irreducible polynomial is being used. In the widely used encryption algorithm AES the irreducible polynomial used is $P(x) = x^8 + x^4 + x^3 + x + 1$[6].

In $GF(2^m)$ again the inverse $A^{-1}(x)$ of an element $A(x) \in GF(2^m)$ must satisfy that $A(x) \cdot A(x)^{-1} \equiv 1$ and what we use to find it again is the extended Euclidean Algorithm.

2.10 THE ADVANCES ENCRYPTION STANDARD

In cryptosystems where plaintexts x encrypted with the same key k as such $E_k(x_1)E_k(x_2)\ldots$ are called *block ciphers*. The Advances Encryption Standard (AES) is a block cipher used world wide. AES has a block length of 128 bits and it can have key lenght of 128 bits, 192 bits and 256 bits. AES is an iterated cipher meaning that the number of rounds depends on the length of the key. In each round the AES performs the following:

1. It performs operation *Add-RoundKey* which XOR the RoundKey and State, where State is an initialization of the plaintext x.

2. *SubBytes* using an S-box it performs a permutation called ShiftRows on State, it performs mixColumns on State and it performs AdRoundKey on State.

3. Perform subbytes, perform shiftrows and perform addRoundKey.

4. define the cipher text y to be State.

Definition 15 (Invertible matrice)**.** *An* $n \times n$ *square matrix* A *is* invertible *if there exist another* $n \times n$ *matrix* B *such that:* $AB = BA = I$*, where* I *is the identity matrix. If the determinant of a square matrix is* 0 *then it has no inverse.*

A Van der Monde matrix with r rows and c columns is a matrix is on the form:

$$\begin{bmatrix} 1 & x_1^1 & x_1^2 & \dots & x_1^{c-1} \\ 1 & x_2^1 & x_2^2 & \dots & x_2^{c-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_r^1 & x_r^2 & \dots & x_r^{c-1} \end{bmatrix}$$

We use the notation $Van^{(r,c)}$ for a *Van der Monde matrix* with r rows and c columns, where all the elements $x_1, ..., x_n$ are distinct. If all numbers in a sqaure Van der Monde matrix are distinct then the determinant is non-zero and thereby it has an inverse. For a matrix $M = Van^{(r,v)}$ with $r > c$ we can create a new matrix M_r from the subset of the rows $R \subseteq \{1, ..., r\}$ where $|R| = c$. This matrix will also be invertible as it is a square Van der Monde matrix. This means that we can choose any c rows from M and get an invertible matrix. We say that the matrix is *super-invertible*.

Secret Sharing

A secret sharing $[s; f_s]_d$ can be done with Van der Monde matrices by constructing a Van der Monde matrix that can evaluate polynomials when multiplied with a vector of points. In the vector we set $x_0 = s$ and $(x_1 \dots x_d) \in_R \mathbb{F}$. The Van der Monde matrix will be on the form $M^{(d)} = Van^{(n,d+1)}(1, \dots, n)$.

$$\underset{(n \times d+1)}{M^{(d)}} \cdot \underset{(d+1 \times 1)}{(x_0, \dots, x_d)} = \underset{(n \times 1)}{(f_s(1), f_s(2), ..., f_s(n))}$$

Normally, when we do Shamir's secret sharing with $[s; f_s]_d$, we evaluate a polynomial on a point for each player and distribute $f_s(x_i)$ to player P_i. Below is an example where the degree $d = 2$

$$f(x) = 5x^2 + 4x + 3, \quad f(1) = 12, f(2) = 31, f(3) = 60$$

The same can be achieved with a Van Der Monde matrix:

$$M^{(2)} \cdot (x_0, x_1, x_2) = \begin{bmatrix} 1 & 1 & 1^2 \\ 1 & 2 & 2^2 \\ 1 & 3 & 3^2 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 12 \\ 31 \\ 60 \end{bmatrix}$$

If $d = t$ then we will refer to it as a t-sharing and if $d = 2t$ we will refer to it as a 2t-sharing.

Van der Monde matrices allow you to do multiple secret sharings in parallel by using a matrix describing m secret sharings, instead of using a vector which describes one secret sharing as above. For example:

$$\begin{aligned} f(x) &= 5x^2 + 4x + 3, \quad f(1) = 12, f(2) = 31, f(3) = 60 \\ f(x) &= 2x^2 + 9x + 4, \quad f(1) = 15, f(2) = 39, f(3) = 49 \end{aligned}$$

$$M^{(2)} \cdot (x_0, x_1, x_2) = \begin{bmatrix} 1 & 1 & 1^2 \\ 1 & 2 & 2^2 \\ 1 & 3 & 3^2 \end{bmatrix} \begin{bmatrix} 3 & 4 \\ 4 & 9 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} 12 & 15 \\ 31 & 30 \\ 60 & 49 \end{bmatrix}$$

Row r_i of the result is then the shares assigned to player P_i.

Randomness Extraction

Van der Monde Matrices can be used for randomness extraction. For $M = Van^{(r,c)^T}$ with $r > c$. We can compute $M \cdot (x_1, \dots, x_r) = (y_1, \dots, y_c)$ and extract randomness from the vector $X = (x_1, \dots, x_r)$, even when only c of the x values are chosen uniformly random.

$$\underset{(c \times r)}{M} \cdot \underset{(r \times 1)}{X} = \underset{(c \times 1)}{Y}$$

To see this, we denote M^R to a square matrix for some subset of c rows which we multiply with a vector X^R with c uniformly random chosen values of x_i's.

$$\underset{(c \times c)}{M^R} \cdot \underset{(c \times 1)}{X^R} = \underset{(c \times 1)}{Y^R}$$

Since M^R is invertible and the vector X^R by definition is uniformly random, we get that the vector Y^R is uniformly random. We denote M^T to be a matrix with the t rest of the rows in M that is not in M^R and multiply it with a vector X^T with t values x_j that is not uniformly random, but still chosen independently from the x_i's.

$$\underset{(c\times t)}{M^T} \cdot \underset{(t\times 1)}{X^T} = \underset{(c\times 1)}{Y^T}$$

Now we have that $M \cdot R = Y^R + Y^T = Y$, where Y^R is uniformly random and Y^T is chosen independently from Y^R and thus Y is uniformly random as well.

2.12 CIRCUITS

Secure Function Evaluation protocols can evaluate any function f with i inputs and j outputs, where i and j are positive integers. We look at protocols where the mapping $f(x_1, x_2, ..., x_i) = y_1, y_2, ..., y_j$ is described using an arithmetic circuit.

An arithmetic circuit is an acyclic direct graph where each node is a gate and each gate has input wires and output wires. There is i input gates with no incoming wires and any number of outgoing wires. The players know on forehand which gate they have to supply their secret input to. There is j number of output wires with one input wire and one output wire. Internally there can be addition and multiplication gates which both have two input wires and one output wire. There is also multiply-by-constant gates which have one input wire and one output wire. Evaluating a circuit can be done by first evaluating all gates in the first layer of the circuit and continuing this layer by layer until they have all been evaluated. Boolean circuits are like arithmetic circuits, except one difference: The internal gates can be an AND gate, OR gate, XOR gate or a NOT gate. Furthermore, the inputs and outputs are bits. If you want to evaluate a boolean circuit with an SFE protocol for arithmetic circuits it can be done by emulation. This is done by using the analytical representations of boolean gates. $f(a) = 1 - a$ is the analytical representation of a NOT gate:

- f(0) = 1-0 = 1
- f(1) = 1-1 = 0

$f(a, b) = a \cdot b$ is the analytical representation of an AND gate:

- $f(0, 0) = 0 \cdot 0 = 0$
- $f(0, 1) = 0 \cdot 1 = 0$
- $f(1, 0) = 1 \cdot 0 = 0$
- $f(1, 1) = 1 \cdot 1 = 1$

$f(a, b) = (1 - (1 - a)(1 - b)) = a + b - a \cdot b$ is the analytical representation of an OR gate:

- $f(0,0) = 0 + 0 - 0 \cdot 0 = 0$
- $f(0,1) = 0 + 1 - 0 \cdot 1 = 1$
- $f(1,0) = 1 + 0 - 1 \cdot 0 = 1$
- $f(1,1) = 1 + 1 - 1 \cdot 1 = 1$

$f(a,b) = a + b - 2 \cdot a \cdot b$ is the analytical representation of an XOR gate, since the XOR it addition without a carry $a + b$ is the addition in first position and $-2ab$ removes the carry in second position.

- $f(0,0) = 0 + 0 - 2 \cdot 0 \cdot 0 = 0$
- $f(0,1) = 0 + 1 - 2 \cdot 0 \cdot 1 = 1$
- $f(1,0) = 1 + 0 - 2 \cdot 1 \cdot 0 = 1$
- $f(1,1) = 1 + 1 - 2 \cdot 1 \cdot 1 = 0$

3 CIRCUIT EVALUATION WITH PASSIVE SECURITY

This chapter presents two protocols for doing secure function evaluation with passive security for any number of $n \geqslant 3$ players and with $t < n/2$ corrupt players. In both protocols the parties learn no more than their own inputs and the outputs they were supposed to receive, even if their computing power is unbounded.

3.1 PROTOCOLS WITH PASSIVE SECURITY

It is assumed that all players can communicate by using a perfectly secure channel, so if two players exchange data the third player knows nothing about what is sent. We require that there will be *perfect correctness* meaning that with probability 1 all players receive correct outputs based on the inputs supplied. We also require *perfect privacy* meaning that the corrupt players C can learn no more than their allowed values. Privacy is shown through the simulation paradigm presented in the sections 2.2 and 2.7. To recap, with the simulation paradigm we show that if a party only learns information X, then everything that party sees can be efficiently recreated or simulated if only given X. Privacy has formally been defined in 11. To recap, we said that privacy is ensured if an efficient probabilistic algorithm exists. This algorithm is a simulator S, which - given the allowed values - can produce the output whose distribution is the same as the dataset of leaked values seen by the corrupt parties. This means the following must hold for the protocols:

$$S(\{x_j, y_j\}_{P_j \in C}) \overset{\text{perf.}}{\equiv} \{view_j\}_{P_j \in C}$$

3.2 PROTOCOL CEPS

This protocol consist of three phases: input-sharing, computation and output reconstruction. For simplicity the protocol will be described for a circuit where each player P_i has a secret input $x_i \in \mathbb{F}$[1]. The communication complexity is $\mathcal{O}(C \cdot n^2)k$, where C is the number of gates and n is the number of parties and k is the bit-length of the elements over the field. The communication complexity of a MPC protocol is the total number of bits sent and received by the honest parties in the protocol.

In the case that players have multiple or no inputs it is assumed that the players have agreed on forehand or have some way of knowing

who is in charge of which input gate. We require that the invariant is held. For this, consider any gate with $\alpha \in \mathbb{F}$ assigned to its input or output wire. Then, if the gate has been processed the players hold $[\alpha; f_\alpha]$.
Input Sharing Phase: Each player P_i has a secret input $x_i \in \mathbb{F}$ and they distribute $[x_i, f_{x_i}]_t$.

Computational Phase: Go through the gates in computational order and do the following for the different kinds of gates:

- **Addition gate:** The players hold $[a; f_a]_t$, $[b; f_b]_t$ the players compute $[a; f_a]_t + b; f_b]_t = [a + b; f_a + f_b]_t$.
- **Multiplication-by-constant gate:** The players hold $[a; f_a]_t$ and α and compute $\alpha[a; f]_t = [\alpha a; \alpha f]_t$.
- **Multiplication gate:** The players hold $[a; f_a]_t$, $[b; f_b]_t$
  1. The players compute: $[a; f]_t * [b; g]_t = [ab; fg]_{2t}$.
  2. The players create a new polynomial and thereby 'reduce the degree' by letting $h = f_a f_b$ where $h(0) = f_a(0) f_b(0) = ab$ each party already holds $h(i)$ and distributes $[h(i); f_i]_t$.
  3. Remember that $deg(h = f_a f_b) = 2t \leqslant n - 1$. Remember also that we introduced the notion of a recombination vector $\mathbf{r} = (r_1^j, \dots r_m^j)$ that can be used to compute a value $f(j)$ of a polynomial f and that this recombination vector works for all polynomials of degree lower than $m - 1$. Hence we can let $\mathbf{r} = (r_1, \dots r_n)$ be a recombination vector for $h(0) = \sum_{i=1}^{n} r_i h(i)$ for any polynomial h where $deg(h) \leqslant n - 1$. The players can compute:

$$\begin{aligned}\sum_i r_i[h(i); f_i)]_t &= [\sum_i r_i h(i); \sum_i r_i f_i)]_t \\ &= [h(0); \sum_i r_i f_i)]_t \\ &= [ab; \sum_i r_i f_i)]_t\end{aligned}$$

     This means that we can use Lagrange to recombine the shares $[h(i); f_i)]_t$ and get $h(0) = ab$.

Output Reconstruction Phase: All gates have been processed and now the players hold $[y_j; f_{y_j}]$ for each output gate. The players securely send their shares to each other and after having received shares for an output gate, they can use Lagrange Interpolation to compute $y = f_y(0)$ from any of the $t + 1$ or more shares.

Correctness

If the invariant is maintained, the players hold the correct shares for the correct value on every wire hold once the whole circuit has been evaluated. The invariant for input sharing holds because of what we have seen in the Lagrange Interpolation section 2.8. In the computation phase the invariant is held because of previously discussed polynomial arithmetic's. Moreover, we have seen that the invariant for the multiplication gate is held.

Privacy

The corrupted parties C get two types of messages. *Type 1* are messages from the input sharing where they receive random sharings $[x_i; f_{x_i}]_t$ and from the multiplication-gate where they receive random sharings $[h(i), f_i]_t$ from the honest parties. *Type 2* is in the output reconstruction phase where they receive all shares in $[y; f_y]_t$ for each output gate. We can also make two observations. *Observation 1* is that in input sharing and multiplication values are shared using random polynomials f_{x_i} and f_i, which are both of degree at most t. Since the are at most t corrupt players, they can only get t shares if they pool their information. Hence, they can not reconstruct the secret and their shares are just uniformly random values. *Observation 2* is that the values sent by the honest parties to the corrupt parties in the output reconstruction phase could be computed by the corrupted parties if they were given the value $f_y(0) = y$, as it would give them and $t + 1$ shares.

The values seen by the corrupted players reveal no more information than their own inputs and outputs. In input sharing and multiplication they just receive uniformly random values and in the output reconstruction phase they receive values they could have computed themselves from their own output y.

Remember that a simulator, given inputs and outputs of the corrupt players, will try to efficiently generate a view that is perfectly indistinguishable from the corrupted players view, and thereby show that the corrupted players have learned no more than their own inputs and outputs.
The simulator S will run the corrupted party on their inputs and outputs. In the input sharing and multiplication it samples uniformly random values and sends them from the corrupted parties to the honest parties. This gives us the right output distribution by observation 1. Remember S is given all inputs and outputs of the corrupted parties, this means that S was given $f(0) = y$ for each output gate. Hence, in the output reconstruction phase S can use the t shares plus y to

calculate the shares in $[y; f_y]$ to be constant with the corrupted parties shares. This gives the right distribution by observation 2.

3.3 PROTOCOL CEPS SPEED

This multiparty computation was introduced by Damgaard et. al. in 2007 [2]. It is unconditionally secure against passive adversaries and it has a communication complexity of $\mathcal{O}(C \cdot n)k$. It consist of two phases. A preprocessing phase and an evaluation phase. The preprocessing phase uses three sub-protocols: Protocol *Double-Random(l), Open(l)* and *Triples(l)*. Computations are done in a finite field $\mathbb{F}$ where $k = \log_2(|\mathbb{F}|)$ is the bit-length of the elements. Note that the notation for secret shares will vary between $[s; f_s]_t = [s]_t$ 2.8.

Random Sharings

A protocol called *Double-Random(l)* is used to create l pairs of random sharings of uniformly random values $r_i \in \mathbb{F}$ of degree t and $2t$. This is called a t-sharing and a 2t-sharing. In other words *Double-Random(l)* creates the t-sharings $[r_1]_t, \dots, [r_l]_t$ and the 2t-sharings $[r_1]_{2t}, \dots, [r_l]_{2t}$ and it outputs the pairs $([r_1]_t, [r_1]_{2t}) \dots, ([r_l]_t, [r_l]_{2t})$. Note that it is the same secret in each pair. The matrix $M = Van^{(n,n-t)^T(1,\dots,n)}$ is used in the protocol and creates $l = n - t$ pairs, so for larger l the protocol is run a number of times in parallel.

Protocol Double-Random(l)

1. Each $P_i \in P$: Pick a uniformly random value $s^{(i)} \in_R \mathbb{F}$ and deal a t-sharing $[s^{(i)}]_t$ and a 2t-sharing $[s^{(i)}]_{2t}$.
2. Compute: $[r_1]_t, \dots, [r_l]_t = M([s^{(1)}]_t, \dots, [s^{(n)}]_t)$
3. Compute $[r_1]_{2t}, \dots, [r_l]_{2t} = M([s^{(1)}]_{2t}, \dots, [s^{(n)}]_{2t})$.
4. Output $([r_1]_t, [r_1]_{2t}) \dots, ([r_l]_t, [r_l]_{2t})$.

There is $m = n - t$ sharings of $[s^{(i)}]_t$ distributed by the honest parties that are unknown to the corrupted parties. The matrix M with m rows is a superinvertible matrix and therefore the m sharings of $[r_i]_t$ are independent uniformly random t-sharings unknown by the corrupted parties. In the same way $[r_i]_{2t}$ are uniformly random values unknown to the corrupted parties. In the randomness extraction section 2.11 it is shown in detail why we get uniformly random values under these circumstances.
Another protocol called Random(l) is like the protocol Double-Random(l), except from the fact that it does not create the 2t-sharings.

The communication complexity of the dealings in step 1 is $\mathcal{O}(n^2k)$, because everyone has to send shares to each other. Per pair generated the communication complexity is $\mathcal{O}(nk)$. So, if l pairs are generated, the communication complexity is $\mathcal{O}(nlk + n^2k)$.

If $l > n - t$, then we need to run the protocol in parallel by utilizing the idea of using Van der Monde matrices introduced in the preliminaries 2.11. In step 1. the players will send row vectors of shares to each other. In step 2. we again use a Van der Monde matrix and can reuse the same technique and multiply M by a matrix of secret shares instead of a vector. The same can be done in step 3, which will still give the same amount of messages sent in the protocols.

Opening Sharings

To open sharings the players choose a party $P_{king} \in P$ which does the reconstruction and sends the result to the rest of the parties.

Protocol Open(d, $[x]_d$)

1. Each $P_i \in P$: Send x_i of $[x]_d$ to P_king.
2. P_king: Use Lagrange Interpolation to calculate $f(0) = x$ and send it to all parties.
3. Each $P_i \in P$ output x.

If $[x]$ is a correct sharing of degree d and everyone follows the protocol, all parties output x. In this protocol there is $2(n-1)$ messages sent, because n players send their shares to P_{king} and P_{king} send x to n players. Hence, the communication complexity is $\mathcal{O}(n-1)$.

Multiplication Triples

With protocol *Triples(l)* the parties can generate l triples $([a]_t, [b]_t, [c]_t)$, where a, b and c are uniformly random values and with $c = ab$.

Protocol Triples(l)

1. All parties run the protocols:
   a) $\mathtt{Random}(2l) \to (([a_1]_t, [b_1]_t), \ldots, ([a_l]_t, [b_l]_t)$
   b) $\mathtt{DoubleRandom}(l) \to (([r_1]_t, [r_1]_{2t}), \ldots, ([r_l]_t, [r_l]_{2t})$.
   c) The outputs are grouped into the triples $([a_i]_t, [b_i]_t, ([r_i]_t, [r_i]_{2t}))$ for $i = \{1, \ldots l\}$
2. For each triple do the following:
   a) Compute $[D]_{2t} = [a]_t[b]_t + [r]_{2t}$.
   b) Run protocol $\mathtt{Open}(2t, [D]_{2t}) \to D$.
   c) Compute $[c]_t = D - [r]_t$
   d) Output $([a]_t, [b]_t, [c]_t)$

In step (2.a) we know from the preliminaries section about shamir secret sharing, 2.8, that: $[a]_t[b]_t = [ab]_{2t}$ and $[ab]_{2t} + [r]_{2t} = [ab + r]_{2t} = [D]$ and therefore $D = ab + r$. In step (2.b) opening [D] leaks no information on a or b, because r is created such that it is uniformly random and independent and of a and b. Therefore the protocol is private.

In step (2.c) we subtract shares of r of degree t and therefore the shares of c are of degree t. Hence, we are left with triples $([a]_t, [b]_t, [c]_t)$ with $ab = c$.

There is only communication through the protocols Open(d, $[x]_d$), Random(l) and DoubleRandom(l) and therefore the communication complexity for generating l triples is $\mathcal{O}(nlk + n^2k)$.

Preprocessing Phase

In the preprocessing phase some sharings are generated for the input gates and multiplication gates. Let i be the number of input gates and m be the number of multiplication gates. To preprocess a circuit do the following steps in parallel for all gates:

1. **Input Gates**:
   a) Run $\mathtt{Random}(i)$ and associate $[r_{gid}]_t$ to each input gate in the circuit.
   b) Send share $[r_{gid}]_t$ to the player P_j who has to provide input for the gate with the id gid.
   c) P_j computes r_gid

.

2. **Multiplication Gates**: Run $\mathsf{Tripes}(m)$ and associate one multiplication triple with each gate.

In step (1.a) the protocol $\mathsf{Random}(l)$ is run, which has a communication complexity of $\mathcal{O}(nlk+n^2)k$ and in step (1.b) everyone has to talk to everyone. In step 2 the protocol $\mathsf{Tripes}(m)$ is run which has a communication complexity of $\mathcal{O}(nlk+n^2)k$. This gives us a total communication complexity of $\mathcal{O}(nlk+n^2)k$, where l is the number of input gates plus the number of multiplication gates.

The preprocessing phase is independent on the circuit that is to be evaluated in the sense that the player can preprocess a number of gates 'over night', such that the players can create a list of preprocessed gates ready for use. Therefore, one can argue that the preprocessing phase is not affecting the running communication complexity or run-time of protocol *ceps speed*.

Evaluation Phase

For an input gate (gid, inp, P_j) the player P_j has to provide a secret input $x_gid \in \mathbb{F}$ for the gate with the id gid. This is done with shamir secret sharing such that each player now holds $[x_{gid}]_t$.

1. **Input Gates**: For each (gid, inp, P_j) in the circuit:
   a) P_j send $\delta_{gid} = x_{gid} + r_{gid}$ to all parties.
   b) All parties then compute $[x_{gid}] = \delta_{gid} - [r_{gid}]$.
2. **scalar-multiplication gates**: Do as in protocol CEPS.
3. **multiplication gate**: For each multiplication gate (gid, mul, gid_1, gid_2) in the circuit:
   a) Compute $[\alpha_g id] = [x_g id1] + [a_g id]$ and $[\beta_g id] = [x_g id2] + [b_g id]$.
   b) Run $\mathsf{Open}([\alpha_{gid}]) \to \alpha_{gid}$
   c) Run $\mathsf{Open}([\beta_{gid}]) \to \beta_{gid}$
   d) Calculate $[x_{gid}] = \alpha_{gid}\beta_{gid} - \alpha_{gid}[b_{gid}] - \beta_{gid}[a_{gid}] + [c_{gid}]$.
4. **Output**: $\mathsf{Open}([x_{gid}]) \to x_{gid}$

The correctness follows from protocol CEPS as we use the same principles.

The privacy comes from r_{gid} in the input and a_{gid}, b_{gid} in the multiplication is uniformly random elements in $\mathbb{F}$. Therefore, $\delta_{gid} = x_{gid} + r_{gid}$, $\alpha_{gid} = x_{gid1} + a_{gid}$ and $\beta_{gid} = x_{gid2} + b_{gid}$ leak no information on $x_g id$. No gates leak except the output gates, but that also leaked in CEPS and it in fact has to leak. The Communication complexity is $\mathcal{O}(nCk + n^2k)$, where C is the number of gates in the circuit.

4 CIRCUIT EVALUATION WITH ACTIVE SECURITY

This chapter presents a modification of protocol ceps in order to make it secure against active adversaries for any number of $n \geqslant 3$ players and with $t < n/2$ corrupt players.

The basic idea is modify protocol CEPS such that all players are committed to all shares they hold. We use the notaion $[[x_i; f_{xi}]]_t = (< f(1) >_1, \dots, < f(n) >_n)$ means that P_i chooses a polynomial and then commits to the coefficients of that polynomial.

Input Sharing Phase: Each player P_i has a secret input $x_i \in \mathbb{F}$ and they distribute $[[x_i, f_{x_i}]]_t$. If this fails then P_i is corrupt and 0 is assigned as default input for P_i

Computational Phase: Go through the gates in computational order and you the following for the different kind of gates:

- **Addition gate:** The players hold $[[a; f_a]]_t$, $[[b; f_b]]_t$ the players compute $[[a; f_a]]_t + b; f_b]]_t = [[a + b; f_a + f_b]]_t$.
- **Multiplication-by-constant gate:** The players hold $[[a; f_a]]_t$ and α and compute $\alpha[[a; f]]_t = [[\alpha a; \alpha f]]_t$.
- **Multiplication gate:** The players hold $[[a; f_a]]_t$, $[[b; f_b]]_t$
  1. The players compute: $[[a; f]]_t * [[b; g]]_t = [[ab; fg]]_{2t}$.
  2. The players create a new polynomial and thereby 'reduce the degree' by letting $h = f_a f_b$ where $h(0) = f_a(0) f_b(0) = ab$ each party already holds $h(i)$ and distributes $[[h(i); f_i]]_t$.
  3. Remember that $deg(h = f_a f_b) = 2t \leqslant n - 1$. Remember also that we introduced the notion of a recombination vector $\mathbf{r} = (r_1^j, \dots r_m^j)$ that can be used to compute a value $f(j)$ of a polynomial f and that this recombination vector works for all polynomials of degree lower than $m - 1$. Hence can let $\mathbf{r} = (r_1, \dots r_n)$ be a recombination vector for for $h(0) = \sum_{i=1}^{n} r_i h(i)$ for any polynomial h where $deg(h) \leqslant n - 1$. The players can compute:

$$\begin{aligned} \sum_i r_i [[h(i); f_i)]]_t &= [[\sum_i r_i h(i); \sum_i r_i f_i)]]_t \\ &= [[h(0); \sum_i r_i f_i)]]_t \\ &= [[ab; \sum_i r_i f_i)]]_t \end{aligned}$$

This means that we can use Lagrange to recombine the shares $[[h(i); f_i)]]_t$ and get $h(0) = ab$.

Output Reconstruction Phase: All gates has been processed and now the players hold $[[y_j; f_{y_j}]]$ for each output gate. The players securely sends their shares to each other and after receiving on shares a output gate they can use Lagrange Interpolation to compute $y = f_y(0)$ from the any of the $t+1$ or more shares.

4.1 COMMITMENTS

The players create an object $[[a; f]]_t$ that they all agree upon, if they don't agree upon this then P_j is corrupt. This is known as verifiable secret sharing.

What if actively corrupt players decide to send the wrong shares back or what if the dealer is corrupt? In that case Shamir's secret sharing has a problem.

A dealer distributes shares of a secret to other people in such a way that the honest players are guaranteed to get consistent shares of a secret, or they will agree that the dealer failed. The correct secret can always be reconstructed, even if the dealer does not participate. This is because if $t < n/2$ we can open with the other shares of commitments. We can fail up to t times because there is t corrupt players.

We can also do arithmetic operations like we did on CEPS, but now the arithmetic operation are done on committed values.

4.2 PERFECT COMMITMENT PROTOCOL

This protocol ensures all honest parties are holding up consistent shares of some value. It also preservers privacy if the committer remains honest. The idea is that the committer P_i not only secret share the value he commits to, but also shares some redundant information that players can check. This can be done by creating *bivariate symetric polynomais* on the as such:

$$f_a(X, Y) = \sum_{\sigma,\tau}^{t} = \alpha_{\sigma,\tau} X^{\sigma} Y^{\tau},$$

where $\alpha_{\sigma,\tau} \in_R \mathbb{F}$ and $\alpha_{0,0} = a$.

This polynomial can be described my a matrix containing the coefficients like in the following example:

$$f_a(X,Y) = \begin{array}{c|cccc} & Y^0 & Y^1 & Y^2 & Y^3 \\ \hline X^0 & 4 & 26 & 5 & 47 \\ X^1 & 26 & 18 & 15 & 14 \\ X^2 & 5 & 15 & 34 & 20 \\ X^3 & 47 & 14 & 20 & 25 \end{array}$$

here the secret is $f_a(0,0) = a = \alpha_{0,0} = 4$ and $t = 3$. After the committer has created this polynomial he then send the polynomial in one variable to each player P_k:

$$f_k(X) = f_a(X,k) = \sum_{\sigma=0}^{t}(\sum_{\tau}^{t} \alpha_{\sigma,\tau}k^{\tau})X^{\sigma}$$

If we look at the concrete example of $f_a(X,Y)$ and use this formula we can create the following shares for P_1:

$$\begin{aligned} 4*1^0+22*1^1+46*1^2+32*1^3 &= 104 \\ 22*1^0+17*1^1+1*1^2+28*1^3 &= 68 \\ 46*1^0+1*1^1+7*1^2+26*1^3 &= 80 \\ 32*1^0+28*1^1+26*1^2+16*1^3 &= 102 \end{aligned}$$

P_1's share is the following polynomial: $104X^0 + 68X^1 + 80X^2 + 102X^3$. We can do the same to get shares for P_2 and P_3 as well and describe their polynomials through vectors containing the coefficients as such:

$$\begin{aligned} P_1&: \quad [104, 68, 80, 102] \\ P_2&: \quad [112, 96, 96, 85] \\ P_3&: \quad [79, 39, 109, 124] \end{aligned}$$

The connection to standard secret sharing can be seen in the following polynomial:

$$g_a(X) = f_a(X,0) = \sum_{\sigma=0}^{t} \alpha_{\sigma,0}X^{\sigma}$$

This polynomial is a polynomial of degree t such that $g_a(0) = a$. Now since f_a is symmetric we have that $g_a(k) = f_a(k,0)f_a(0,k) = f_k(0)$. This means that by secret sharing ..

The idea is then that players can check consistency by evaluating and comparing their shares. To check consistency any two players P_i and P_k can does the following:

$$f_k(j) = f_a(j,k) = f_a(k,j) = f_j(k)$$

The idea is that after P_k get his share $f_k(X)$ he checks pairwise that $f_k(j) = f_j(k)$ with all other players P_j. If $t+1$ honest players has consistent shares then we know that there is enough information to determine a polynomial of degree at most t, that all honest player agree on.

If we continue the example above P_1 would check with P_2 and P_3 for consistency as such:

$$\begin{aligned} f_1(2) &\mod 47 = 13 \\ f_2(1) &\mod 47 = 13 \\ \text{and} & \\ f_1(3) &\mod 47 = 22 \\ f_3(1) &\mod 47 = 22 \end{aligned}$$

4.3 THE PROTOCOL

COMMUNICATION

Each protocol is located in their own package where a $\mathtt{routes.py}$ file is located. Here API endpoints for that specific protocol are located and are used for the players to communicate through.

The following approach to communication was used throughout all protocols in the implementation. Below is a concrete example of the endpoints used *A Perfect Commitment Protocol* that is a protocol used in protocol CEAS:

1. On input $(\mathrm{commit}, i, \mathrm{cid}, a)$, P_i samples a bivariate symmetric polynomial $f_a(X, Y)$ of degree at most t, such that $f_a(0, 0) = a$. He sends the polynomial $f_k(X) = f_a(X, k)$ to each P_k (therefore P k also learns $\beta_k = f_k(0)$).

2. Each P_j computes $\beta_{k,j} = f_j(k)$ and sends $\beta_{k,j}$ to each P_k.

3. Each P_k checks that $\mathrm{deg}(f_k) \leqslant t$ and that $\beta_{k,j} = f_k(j)$ for $j = 1, \ldots, n$. If so, he broadcasts success. Otherwise, he broadcasts $(\mathrm{dispute}, k, j)$ for each inconsistency.

4. For each dispute reported in the previous step, P_i broadcasts the correct value of $\beta_{k,j}$.

5. If any P_k finds a disagreement between what P_i has broadcast and what he received privately from P_i, he knows P_i is corrupt

and broadcasts (accuse, k).

6. For any accusation from P_k in the previous step, P_i broadcasts $f_k(X)$.

7. If any P_k finds a new disagreement between what P_i has now broadcast and what he received privately from P_i, he knows P_i is corrupt and broadcasts (accuse, k).

8. If the information broadcast by P_i is not consistent, or if more than t players have accused P_i, players output fail. Otherwise, players who accused P_i and had a new polynomial $f_k(X)$ broadcast will accept it as their polynomial. All others keep the polynomial they received in the first step. Now each P_k outputs success and stores $(\mathrm{cid}, i, \beta_k = f_k(0))$. In addition P_i stores the polynomial $g_a(X) = f_a(X, 0)$.

5 IMPLEMENTATION

A proof-of-concept implementation of *Protocol CEPS* and *Protocol CEPS SPEED* has been created with the purpose of comparing the two protocols in practice. This chapter shall provide an overview of the implementation.

The implementation can be found here in the following git repository: github.com/ckoller/secretsharing.

5.1 INTRODUCTION

The implementation of the protocols has been developed as a framework that allows one to do multiparty computation with three or more participants. Furthermore, tests have been created to insure the correctness of the protocols. On top of that, some performance tests have been created to compare the protocols in practice.

This implementation is not suited for a production environment as it was created with the purpose of comparing the protocols, and hence, the sent messages between the users are not encrypted. Therefore, it is not recommended that the application is used as sub-protocols in other applications. Some examples of faster MPC implementations are: VIFF or SPDZ.

The implementation has been developed in the object-oriented programming language Python and it uses a micro-framework for web services called Flask. These tools have been chosen because they enable a developer to quickly create a server and with a RESTful API with few lines of code. This means that with little effort and few boiler plate code I was able to setup an environment where players can communicate with each other rather quickly, and thereby had more time to focus on the protocols to be implemented. Another alternative to REST would be using Sockets, which has significantly less overhead per transmission, but require more setup.

Furthermore a for doing perfect commitment has been implemented, which is a sub protocol in CEAS.

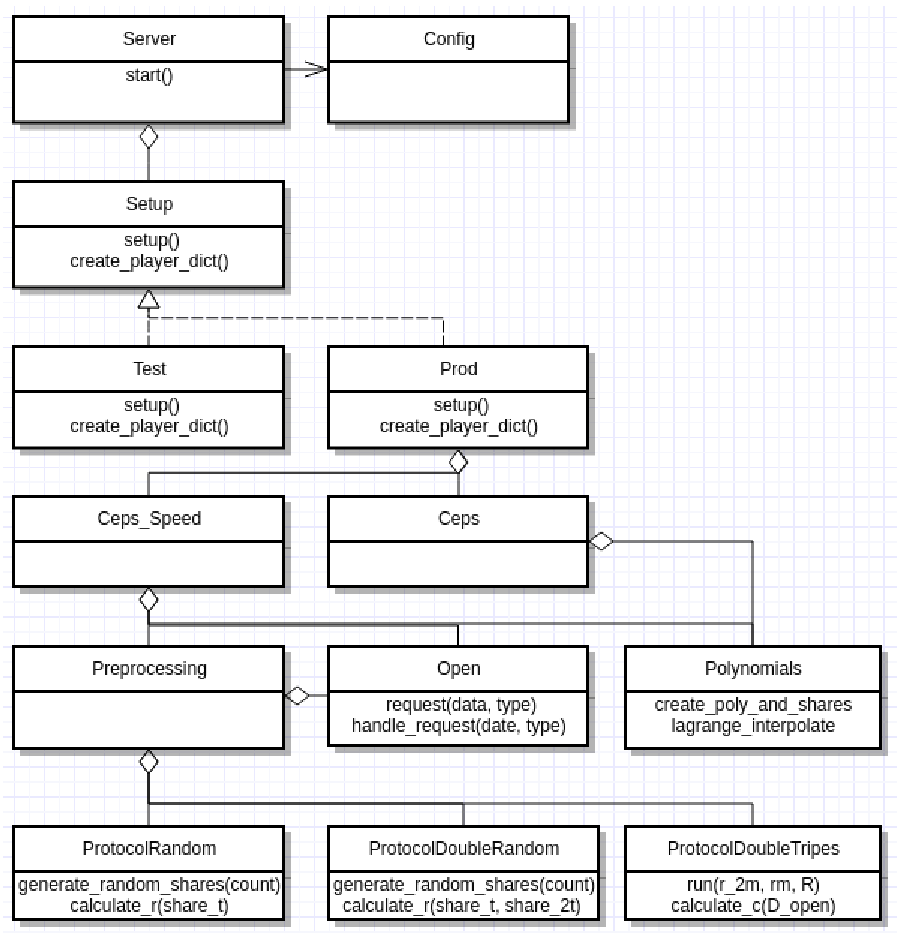

Figure 2: UML Class Diagram showing an overview of the implementation of Protocol Ceps and Protocol Ceps Speed

In figure 2 we see the class *Server* which is in charge of starting the Flask server with the setup and input parameters provided by either class Test or class Prod. In this setup-phase the players ip, port and id are initialized and a given list of known players is read. In the config file config.py a setup type is chosen specifying whether the application shall be run with the production setup or the test setup. Additionally, within the config.py there is specified a path to a folder that contains circuits which can be evaluated.

On run time both protocol ceps and protocol ceps speed are available for use. The class *Ceps* and *Ceps_Speed* are in charge of controlling the communication flow of the respective protocols.

Polynomials

The polynomial math has been encapsulated in the class $\mathsf{Polynomials}$. This includes the creation of shares, normal polynomials, bivariate symetric polynomials, creation of Van der Monde matrices and polynomial recreation through Lagrange Interpolation. Polynomial computations are done in a prime field with a prime specified by a field variable in this class.

Preprocessing

The class $\mathsf{Preprocess}$ is in charge of generating some sharings for the input gates and multiplication gates. Let i be the number of input gates and m be the number of multiplication gates. Remember that each time any of the preprocessing protocols are run we generate $l = n - t$ outputs, so if there is more than l gates we have to run the protocols in parallel. It has been described earlier 3.3 that these protocols can be run in parallel by working with matrices instead of vectors. The multiplication gates each need one multiplication gates triple associated with them. This means class $\mathsf{ProtocolTriples}(m)$ has to be run $\mathsf{ceil}(m/l) + 1$ times. Where the ceil function always rounds up. ProtocolTriples needs $\mathsf{ProtocolRandom}(2l)$ and $\mathsf{ProtocolDoubleRandom}(l)$. This means that the class $\mathsf{ProtocolDoubleRandom}$ has to be run $\mathsf{ceil}(m/l) + 1 = c$ times. For each input gate the class $\mathsf{PrototolRandom}$ is run and this means that the protocol has to be run $(c * 2) + \mathsf{ceil}(i/2) + 1$.

Boolean Circuit Evaluation

It is interesting to test the implementation on a larger circuit that could potentially be used in practice. Nigel Smart[5] has created a number of binary test circuits useful for this purpose. The circuits consist of AND's, XOR's and inverters and the gates are in topological order. The circuits have been created by a tool and hand-optimized circuits can achieve much smaller AND gate counts.

The initial implementation can evaluate arithmetic circuits. To evaluate these boolean circuits we use the analytical representations of the respective boolean gates. Furthermore, these circuits have a different format than my initial arithmetic circuits and thus a converter had to be created.

Two circuits from [5] have been used: A circuit describing a 32-bit adder has been used to test for correctness of the implementation. If the adder can add correctly, then the implementation most likely works as it should. To evaluate for performance with a real world example a circuit that describes an encryption function for AES has been used. The AES circuit takes two inputs: a 128 bit plaintext and a 128 bit key. The AES circuit consist of 36000 gates and evaluating the gates one by one resulted in a flooding of messages. We either got timeout or "Max retries exceeded with URL", meaning that the server refuses the connection because there was sent too many requests from same ip address in a short period of time. To combat this, the evaluation had to be done layer by layer, such that all messages that can be sent in one layer were grouped together and sent as one message. This means that now instead of sending about one message for each gates AND or XOR gates which is about 30.000 messages, we would only sent about 300 messages, one message for each layer.To do this, the circuit needed to know which layer a gate was in. Therefore, the circuits are *parsed* and an extra attribute is added to each circuit, namely which layer it is in. Since the gates are in topological order we could go through the gates from top to bottom and update they gates with a layer attribute. Then look at what wire predecessor id and update the current gates id.

Also input gates and output gates are added, because the initial implementation expects these gates to be there. The circuits used in the implementation is an extension of the circuit from [5].

The above introduced variation of how to handle input sharing and evaluation. These variations are encapsulated and delegated into different strategies. An example of this can be seen in figure 3 below.

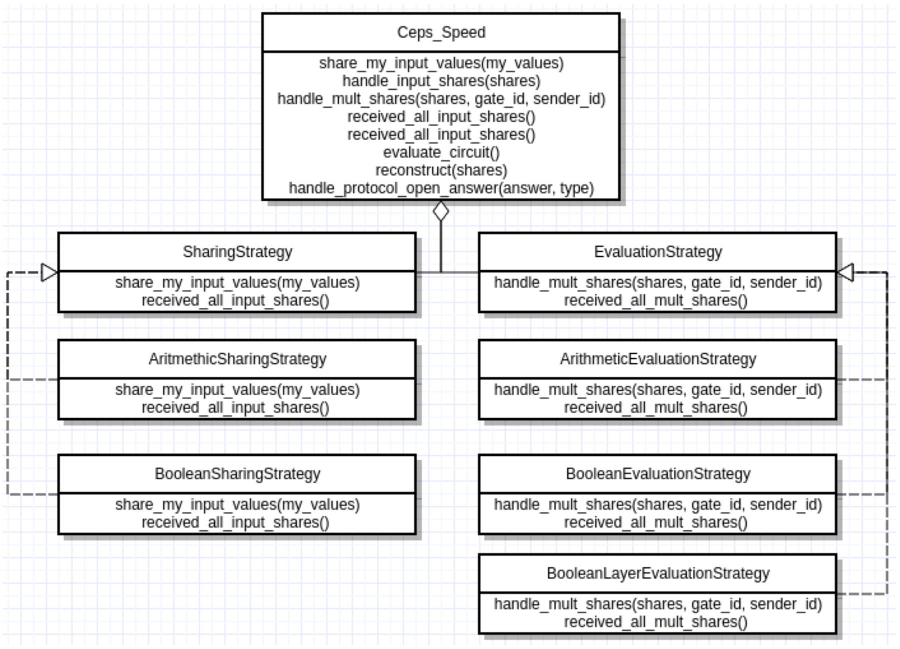
Ceps_Speed
share_my_input_values(my_values)
handle_input_shares(shares)
handle_mult_shares(shares, gate_id, sender_id)
received_all_input_shares()
received_all_input_shares()
evaluate_circuit()
reconstruct(shares)
handle_protocol_open_answer(answer, type)
SharingStrategy
share_my_input_values(my_values)
received_all_input_shares()
EvaluationStrategy
handle_mult_shares(shares, gate_id, sender_id)
received_all_mult_shares()
AritmethicSharingStrategy
share_my_input_values(my_values)
received_all_input_shares()
ArithmeticEvaluationStrategy
handle_mult_shares(shares, gate_id, sender_id)
received_all_mult_shares()
BooleanSharingStrategy
share_my_input_values(my_values)
received_all_input_shares()
BooleanEvaluationStrategy
handle_mult_shares(shares, gate_id, sender_id)
received_all_mult_shares()
BooleanLayerEvaluationStrategy
handle_mult_shares(shares, gate_id, sender_id)
received_all_mult_shares()

Figure 3: UML Class Diagram showing an overview of the implementation of Protocol Ceps speed and its different variants

Each protocol is located in their own package where a `routes.py` file is located. Here API endpoints for that specific protocol are located and are used for the players to communicate through.

The following approach to communication was used throughout all protocols in the implementation. Below is a concrete example of the endpoints used *A Perfect Commitment Protocol* that is a protocol used in protocol CEAS:

1. On input $(\mathtt{commit}, i, \mathrm{cid}, a)$, P_i samples a bivariate symmetric polynomial $f_a(X,Y)$ of degree at most t, such that $f_a(0,0) = a$. He sends the polynomial $f_k(X) = f_a(X,k)$ to each P_k (therefore P k also learns $\beta_k = f_k(0)$).

HTTP Method:	POST
API Endpoint:	/API/commit
Data:	[prover_id, commit_id, share]
Explanation:	give shares to other players

2. Each P_j computes $\beta_{k,j} = f_j(k)$ and sends $\beta_{k,j}$ to each P_k.

HTTP Method:	POST
API Endpoint:	/API/<commit_id>/consistency
Data:	[prover_id, commit_id, sender_id, consistency_value]
Explanation:	send consistency values to each other

3. Each P_k checks that $\deg(f_k) \leqslant t$ and that $\beta_{k,j} = f_k(j)$ for $j = 1, \ldots, n$. If so, he broadcasts success. Otherwise, he broadcasts $(\mathtt{dispute}, k, j)$ for each inconsistency.

HTTP Method:	POST
API Endpoint:	/API/<commit_id>/dispute
Data:	[prover_id, commit_id, sender_id, status, disputer_id]
Explanation:	check consistency values and broadcast results

4. For each dispute reported in the previous step, P_i broadcasts the correct value of $\beta_{k,j}$.

HTTP Method:	PUT
API Endpoint:	/API/<commit_id>/consistency
Data:	[prover_id, commit_id, sender_id, consistency_value, disputer_id1, disputer_id2]
Explanation:	p_i send correct con_vals

5. If any P_k finds a disagreement between what P_i has broadcast and what he received privately from P_i, he knows P_i is corrupt and broadcasts $(accuse, k)$.

HTTP Method:	POST
API Endpoint:	/API/<commit_id>/accuse
Data:	[prover_id, commit_id, sender_id, accusation]
Explanation:	p_k broadcast accusations on disagrements con_vals

6. For any accusation from P_k in the previous step, P_i broadcasts $f_k(X)$.

HTTP Method:	PUT
API Endpoint:	/API/<commit_id>/commit/
Data:	[prover_id, commit_id, sender_id, accuser_id, share]
Explanation:	p_i broadcast share for accusations

7. If any P_k finds a new disagreement between what P_i has now broadcast and what he received privately from P_i, he knows P_i is corrupt and broadcasts $(accuse, k)$.

HTTP Method:	PUT
API Endpoint:	/API/<commit_id>/accuse/
Data:	[prover_id, commit_id, sender_id, accusation]
Explanation:	p_k broadcast accusations on disagreements

8. If the information broadcast by P_i is not consistent, or if more than t players have accused P_i, players output fail. Otherwise, players who accused P_i and had a new polynomial $f_k(X)$ broadcast will accept it as their polynomial. All others keep the polynomial they received in the first step. Now each P_k outputs success and stores $(cid, i, \beta_k = f_k(0))$. In addition P_i stores the polynomial $g_a(X) = f_a(X, 0)$.

When a player wishes to communicate through an endpoint they use the *requests* library. Throughout the application the same approach has been used to sent data to players API endpoints. An example of this is shown in the code snippet below:

```
import requests
    ...
    def send_input_shares(self, input_shares):
        for player_id, player in config.players.items():
            url = "http://" + player + "/api/ceps/share/"
            data = {"shares": json.dumps(input_shares[player_id]),
                "sender_id": json.dumps(config.id)}
        requests.post(url, data)
```

Unfortunately the HTTP requests are synchronous, meaning that the code will block or wait until a response is returned from the API. A better approach would be to use asynchronous communication or/and multithreading, because this allow for computation do be done in parrallel by the different servers.

TEST

To run the application with n players, a test environment has been created, such that the application can be run locally. In this way the code has been developed through tests to ensure *correctness* of the implementation. Furthermore, the application has been run in a production environment on Amazon EC2 servers to test *performance* and compare the MPC protocols in practice.

Correctness

The correctness test was made in order to have an easy way to run the application locally, but also to ensure that the new functionality worked as intended. When you run a local test, $n - 1$ instances of the application are started in gnome-shells and 1 instance is started in a thread and the tests are then only run on that single instance.

Performance

To test for performance tests were run in a production environment on amazon EC2 servers. The performance tests requires configuration of the `config.py` file. The variable *circuit_folder_path* should be set to a string describing the full path of a folder containing circuits and the variable *setup_type* should set to "prod".

6 EXPERIMENTS AND RESULTS

This chapter describes how performance tests of the implementation of protocol CEPS and protocol CEPS SPEED was done, and it presents and discuss the result and findings hereof.

EXPERIMENTS

Environment and Hardware

The tests was run on Amazon Elastic Compute Cloud (EC2), that is a platform where users can rent virtual computers on where they can run their own applications. For testing, a machine was chosen which was running Ubuntu 16.04 and had 1 CPU, 2.5 GHz, Intel Xeon Family and 1 GiB memory. On the machine the application was initialized, and replicated to n duplicates containing the same code. The machines public DNS' (IPv4) was then listed in a the *player_config.py* file.

Circuits

The various circuits used in the performance tests all have one gate per layer, except for the first layer that contains all input gates, as seen in figure 4 below:

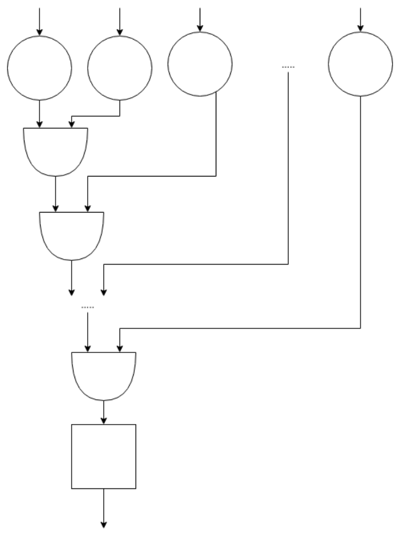

Figure 4: An example of a test circuit with input gates (circles) in the first layer, one AND gate in each internal layer and one output gate (square).

In the following tests we write a circuit with g gates, but mean a circuit with g internal gates and $g + 1$ input gates in the first layer and 1 output gate in the last layer.

The application has been optimized in such way that it sends as much information as possible each time it sends a message. This means that the circuits are evaluated layer by layer and all information from one layer is grouped and forward warded to the next layer. If our test circuits had all their gates in one layer, then the players would only have to send one message to each player in protocol CEPS or one message to the king if protocol CEPS SPEED is run. For this reason, the circuits used for testing have one gate in each internal layer.

Who provides input for the input wires in the tests? For every input wire a new player provides input such that the implementation is tested with as much communication as possible.

Measure

In the experiments the running time has been measured and the running time of the slowest machine has been noted down. Each experiment has been conducted 5 times and what is shown in the results is an average of this.

The time is measured with the python `time` module. On protocol CEPS the is started when the first share is is to be send or, the first input share is received and the time is stopped when the last share for the output gate is received, such that the result of the evaluation can be evaluated. On protocol CEPS SPEED the time is started and stop like in protocol CEPS, this we refer to as the time without preprocessing. There is also a time for the execution of the whole protocol, which is started when the preprocessing starts and end when the protocol is finished.

RESULTS

In the following there will be presented an analyzed two types of test. One where the number of gates will vary and one were the number of player will vary.

Performance Tests by Number of Gates

The performance of Protocol CEPS and Protocol CEPS SPEED has been tested with a constant number of players and a varying number of gates to test which effect the number of gates has on the performance of two the MPC protocols. In this section tests are done 5 players and g number of gates.

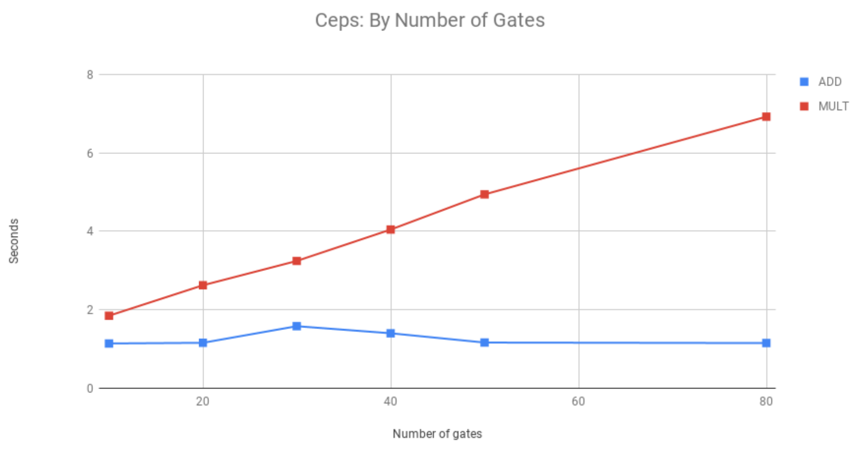

Figure 5: Trends of protocol CEPS evaluating two different circuit types: One with g MULT gates and one with g ADD gates.

Note that in figure 5 each point on the graph an evaluation of another circuit. This means that here CEPS has been evaluated on circuits 10 different circuits. 5 circuits containing MULT gates and 5 circuits containing ADD gates. The blue line labelled ADD describes the trend when increasing ADD gates in a circuit and likewise for the red line labelled MULT.

In figure 5 we have no incremental trend for evaluation of circuits with ADD gates. Since the evaluation of ADD gates requires no communication, it is expected that an increase of ADD gates has few effect on the running time. The MULT gates require all players to send one message to each other for each layer, so for each layer that is n^2 messages being sent. The communication complexity is exponential, but since we only linearly increment the number of gates and keep n as a constant, one can expect a linear growth as seen in 5.

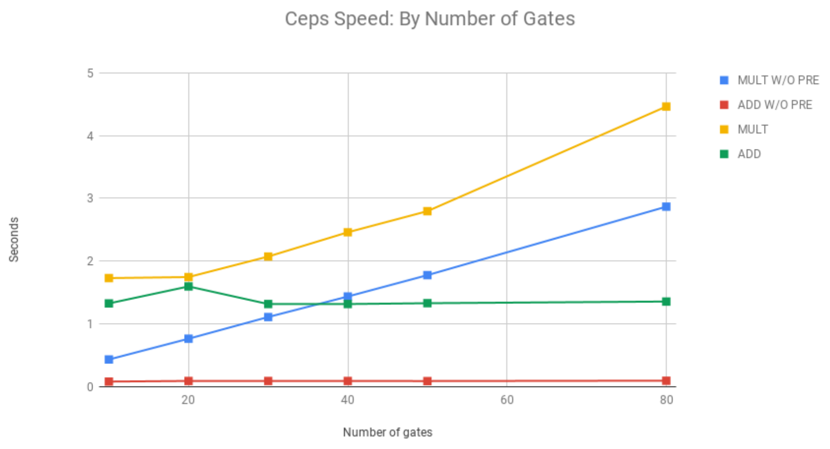

Figure 6: Trends of protocol CEPS SPEED evaluating two different circuit types: One with g MULT gates and one with g ADD gates.

In figure 6 we see the running times of doing the previous experiment on protocol CEPS SPEED. Only now the time is measured both with and without the preprocessing phase. This is done because, in theory, the preprocessing phase could be done overnight, whilst the evaluation phase could then be done alone when needed. In figure 6 we see the same trends as for ceps in figure 5. For every MULT gate n messages have to be sent to the king and then the king has to send n messages back, so that we have $2n$ messages for each layer and in this case each gate. This means that if we increase the number of gates linearly we also expect the number of messages and thereby the execution time to increase linearly. For ADD gates no messages are sent between the players. The reason why it takes time is because the input gates require some communication for preprocessing and evaluation. But since they are all in the same layer this is packaged and sent in a single messages. For this reason we expect a non-increasing flat curve describing the execution time for addition gates, which we do see in our results. The execution time without the preprocessing phase cuts down on execution time, which was expected. The running time without the preprocessing phase are parallel lines, just with lower values, for the respective gates. Hence preprocessing time is constant. This may be because the messages in the preprocessing phase are packaged and we don't send more messages when there are more gates to be evaluated.

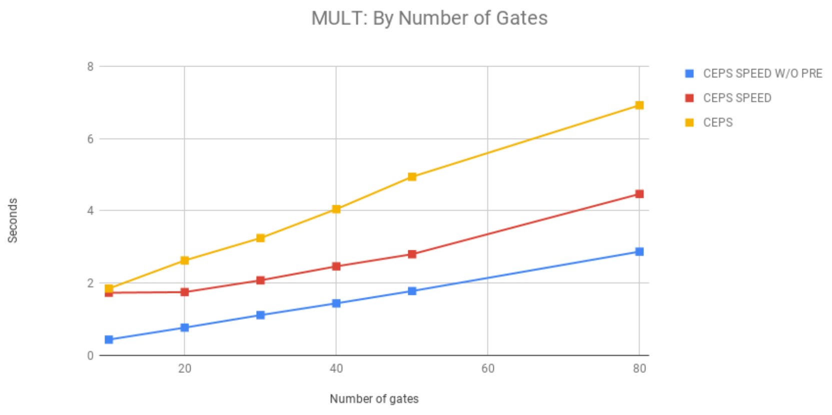

Figure 7: Graphs showing running time of different MPC protocols evaluating circuit with g AND gates.

In figure 7 we see that protocol CEPS SPEED is faster which what we expected from the theory, since there is sent a total of n messages per layer instead of n^2 messages.

We also see from the trend that CEPS running time is steeper, meaning that it gains more time when more gates are added. Now since we have synchronous communication is makes sense that CEPS is a lot slower because there is more communication and every time we send a message we stop and wait until that player has answered. If it implementation has asynchronous communication I would expect the run times to be closer, because requests can be computed in parallel. But still the communication complexity of CEPS is a lot higher compared to CEPS SPEED.

We also see that CEPS SPEED without the preprocessing phase again is faster and that the preprocessing time is constant and thereby does not increase with the number of gates, which (as earlier discussed) makes sense since the messages are packaged.

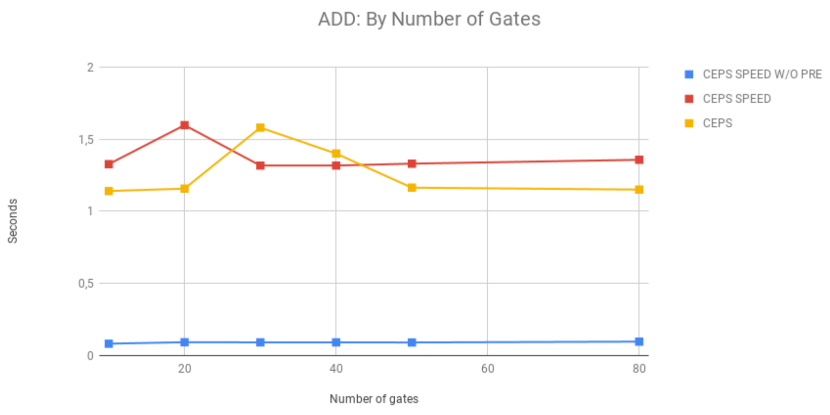

Figure 8: Graphs showing running time of different MPC protocols evaluating circuits with g OR gates.

In CEPS addition gates require no communication apart from the input sharing and output reconstruction. This means that in CEPS $2n^2$ messages are needed to evaluate the circuits consisting of only ADD gates. In CEPS SPEED addition gates also only require communication for the input and the output reconstruction. The input gates need preprocessing which requires a run of protocol Random(i). Here a t-sharing is dealt and this requires n^2 messages. Then, in the preprocessing phase $[r_{gid}]$ is sent to the P_j who has to provide input to the gate with that id. That is another n messages sent. In the evaluation phase phase n^2 message are sent for the input gates, and n messages are sent for the output gates. That is a total of $n^2 + n^2 + n + n$ messages sent, which would explain why it takes a little longer to evaluate CEPS SPEED with circuits consisting of only OR gates, as seen in figure 8.

Performance Tests by Number of Players

These tests, like the tests in the previous section, use the same type of circuits. The only difference is that the number of gates is kept constant whilst the number of players is varying.

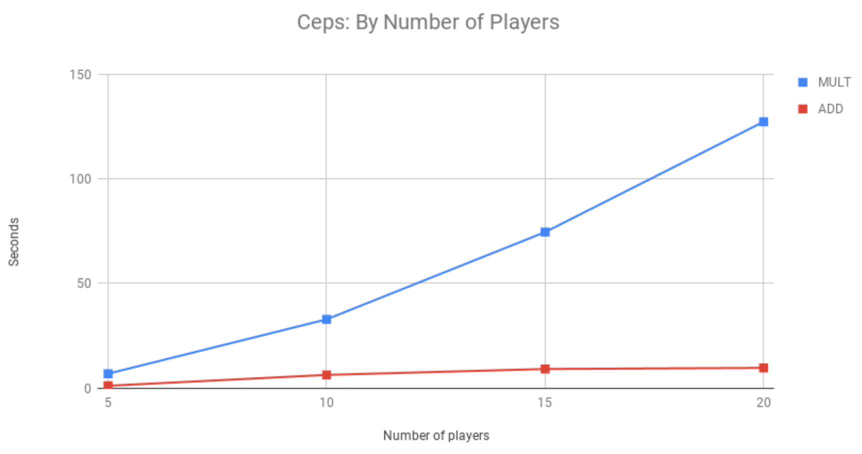

Figure 9: Execution time of Ceps of circuits with 5 internal gates and n players.

In figure 9 one can see that the circuit with ADD gates has a nearly constant execution time. When the number of players is increased, we increase the number of messages sent in the input sharing phase and the output reconstruction phase and, therefore, we would expect a small increase in the number execution time. But since there is no communication needed to evaluate ADD gates, the increase is expected to be small.

In figure 9 it can be seen that in a circuit consisting of only internal MULT gates the running time increases exponentially with the number of players. This is because they send n^2 messages in each layer. It heavily affects because of the the synchronous communication.

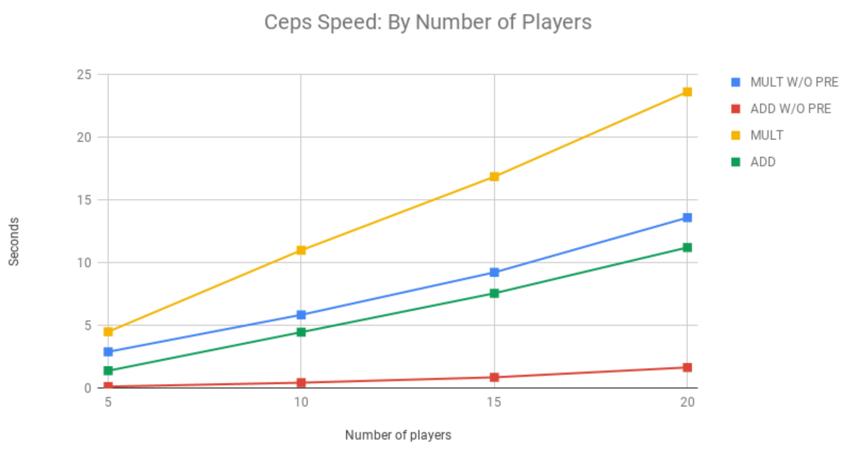

Figure 10: Execution time of Ceps Speed for circuits with 5 internal gates and n players.

Figure 10 shows that ADD is increasing with the number of players. As previously discussed, this is because that there now is more communication.

MULT is increasing linearly with the number of players. This is because n messages are sent in every layer.

By increasing the number of players instead of the number of gates, we see that the preprocessing time is no longer constant. Again this is caused because more people messages sent. The full running times increase steeper than the the time without the preprocessing phase. Again this is caused by more messages between people.

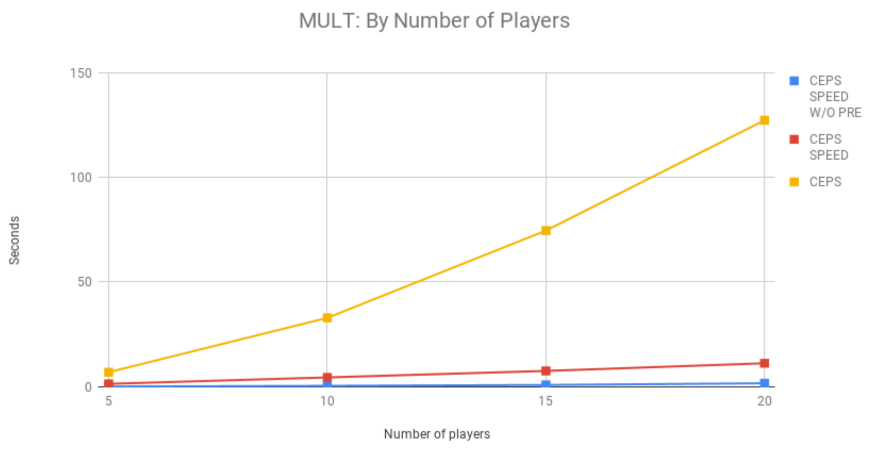

Figure 11: Comparison of MPC protocols execution time with constant number of MULT gates and n players.

In figure 11 it is seen that we save a lot of time by using CEPS SPEED when we evaluate circuits with a lot of MULT gates, with a large amount of players.

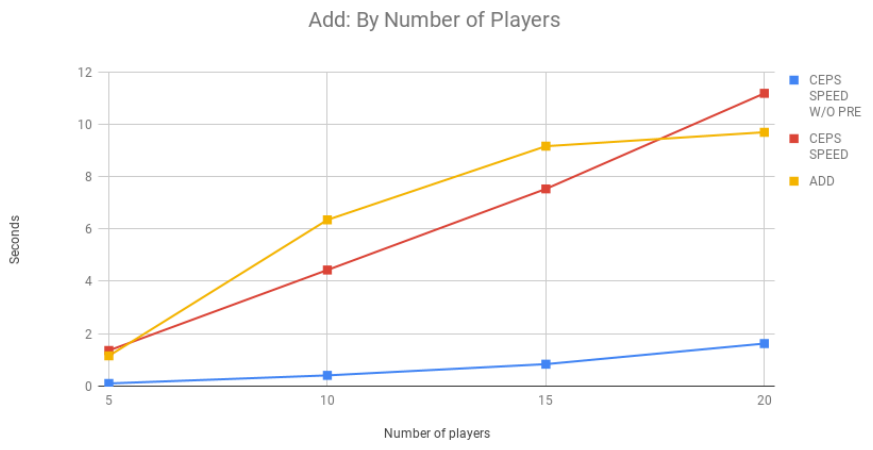

Figure 12: Comparison of MPC protocols execution time with constant number of ADD gates and n players.

In figure 12 we see that we do not gain anything by using ceps speed over ceps, unless ceps speed is preprocessed over night. Then gain a great amount of time.

AES TESTS

The test was done on an AES circuit with 25124 XOR gates, 6800 AND gates, 1692 INV gates, 467 INPUT gates and 128 OUTPUT gates. The same circuit was executed and the number of player was varied.

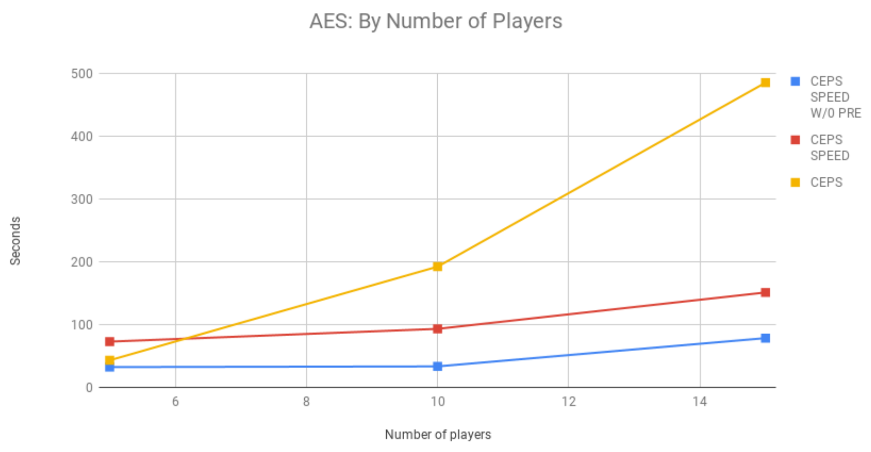

Figure 13: Execution time of AES circuit with n players.

Here we see that CEPS SPEED greatly outperforms protocol CEPS. Protocol CEPS' run time increases exponentially with the number of people in the protocol. It is not realistic to use in real time as it is quite slow. This is caused by the implementations lack of asynchronous HTTP request and the lack of multi-threading.

LOAD BALANCING TESTS

To try to improve the performance we have tested out load balancing. When working with circuits as big the AES circuit, computational complexity is a limiting factor. In protocol CEPS SPEED everything is sent to one king who then uses computing power whilst the rest of the players are doing nothing. If we make multiple kings and thereby balance the load, maybe we can make the protocol even faster. This is a trade-off between communication and computation. If we for every single gate in a layer send this shares to a new king we spend a lot of effort communicating for few computational power (since each king only does work for one gate). Therefore we collect a batch of gates. If the batch size it set to 100, we give the first 100 gates to the first king, and then the next 100 to the next king and so forth. In this way if there is only 10 gates that needs processing, we don't send 10 messages, as the amount of messages seems to be the bottleneck of this synchronous implementation.

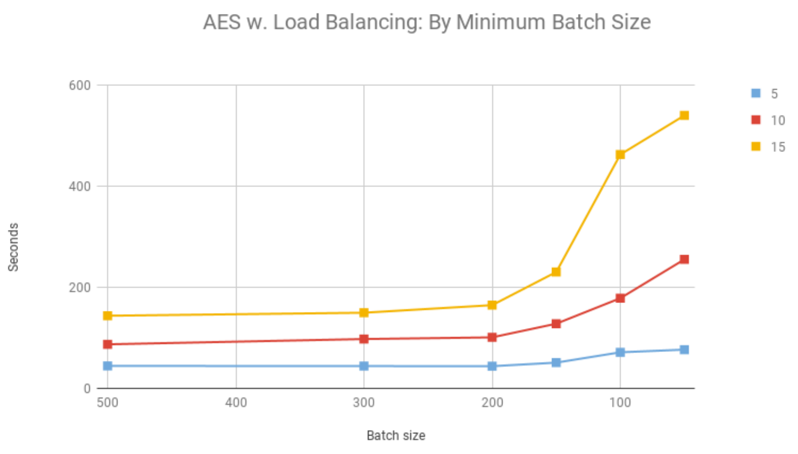

Figure 14: Comparison of execution time of CEPS SPEED with load balancing with 5, 10 and 15 players.

In figure 14 it can be seen which influence the minimum batch size has on the execution time. The 3 lines describe what the influence is when 5, 10 or 15 players participate in the protocol. All three graphs show the same trend. Namely, that the execution time increases when we lower the minimum batch size. This can be caused by the cost of communication, which does not outweigh the cost of computation, unless there is data enough, or in other words, unless the batch is big enough.

It is interesting to look at how loads balancing performs compared to an execution of CEPS speed without loads balancing, because we want to see if we made an improvement. Therefore, the next three figures compare executions by load balancing with an execution without load balancing.

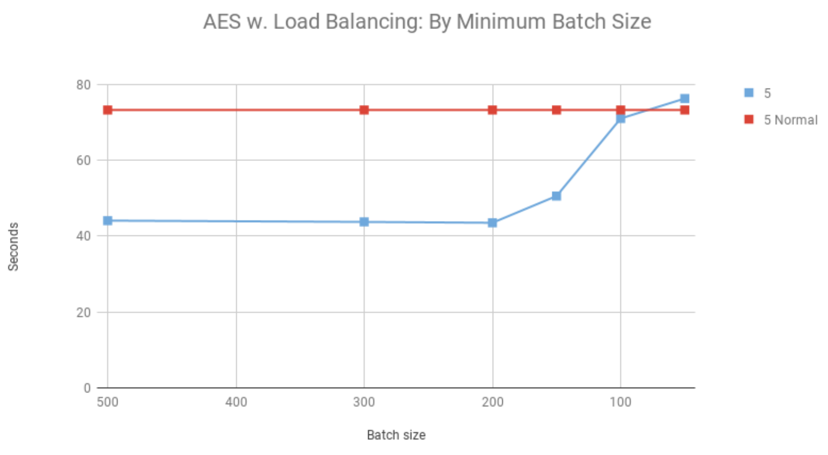

Figure 15: A comparison of load balancing vs no load balancing with 5 players.

In figure 15 we see that we only benefit from load balancing if the load is higher than 100 and we see that the benefit is constant when the batch size is higher than 200. If the AES circuit has no or very few layers with more than 200 gates in, then we do not benefit from splitting the data up, because all data is just sent to one player anyway.

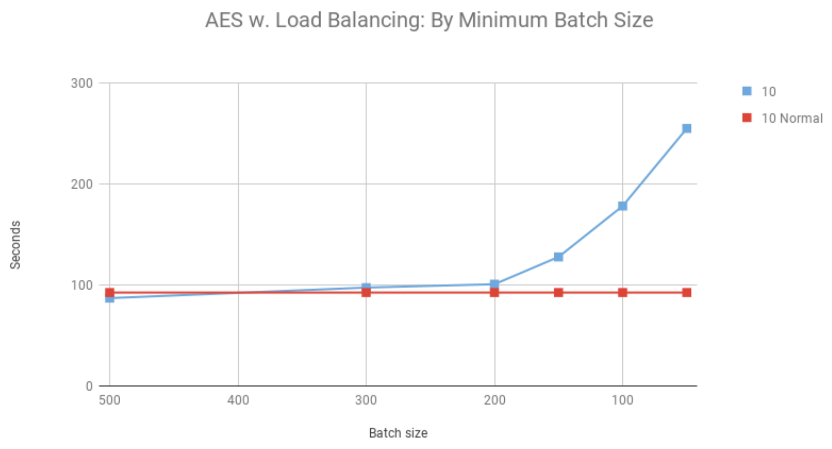

Figure 16: A comparison of load balancing vs no load balancing with 10 players.

In figure 16 we see no benefit from using load balancing. When there are more players, more messages have to be sent to more players. Thus, the computational benefit no longer outweights the communications trade-off.

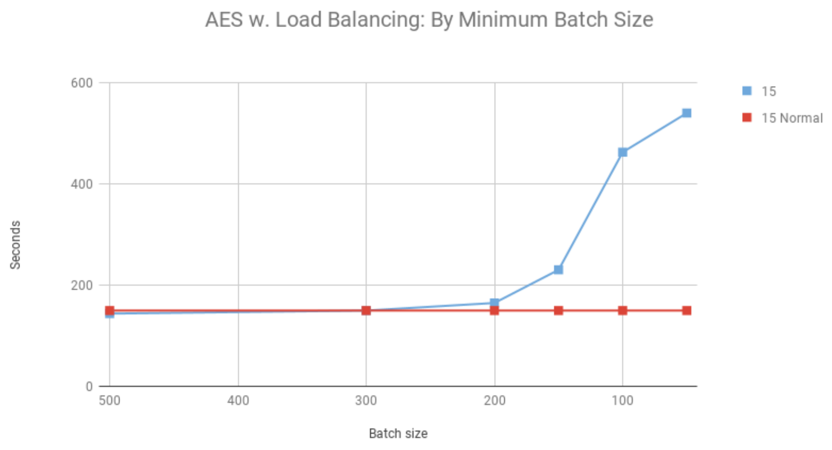

Figure 17: A comparison of load balancing vs no load balancing with 15 players.

In figure 17 we see the same trend as in figure 16 with the only difference that the execution time is considerably longer.

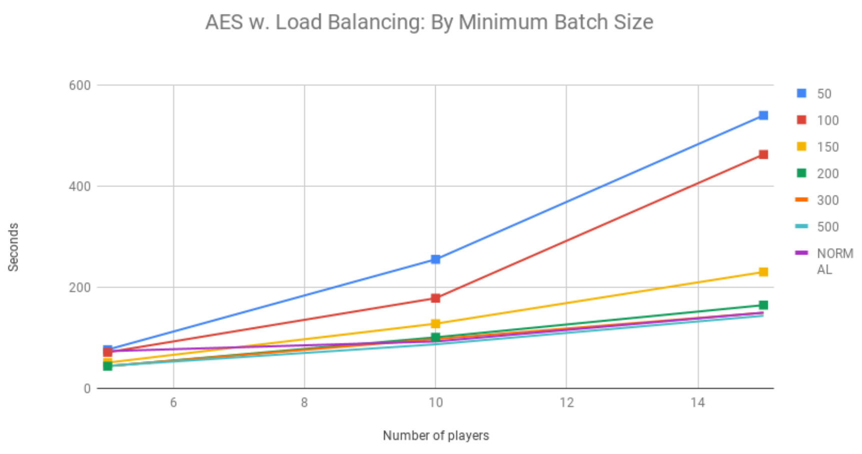

Figure 18: Load balancing with different batch size by number of players.

In figure 18, shown above, we see the earlier load balancing data all grouped into one single diagram, showing that the execution time depends on two factors: the number of players and the batch size. Whilst we could detect a positive relationship between the execution time and the number of players, i.e. the more players the longer the execution time, the correlation between the batch size and the execution time is negative, i.e.the lower the batch size the longer the execution time.

7 CONCLUSION

This thesis has explored the theory of Secure Multiparty Computation, short MPC, by closely looking at two protocols for doing MPC. One protocol for MPC with passive security called CEPS and another variant hereof which does the same but with a lower communication complexity, this we call CEPS SPEED. The theory of the security of these protocols has been explored and analyzed.

Furthermore, implementations of the two protocols have been made. Through the implementations of the two protocols I was able to acquire knowledge of the correctness of the protocols and gain some insight into how to make it work in practice. However, the implementation has been done in an interpreted programming language, namely python, and this has given it some performance overhead. The emphasis in this thesis was placed on the cryptography part of the implementations and thereby less time was spend on "boiler-plate code". With hindsight it appears, however, that it would have been more interesting to be able to look at the results of an implementation that supports multi threading and asynchronous communication. Nonetheless, it has been interesting to do the implementations and comparing the protocols has motivated a closer look on how to optimize cryptographic processes instead of only looking at the security thereof.

It has been investigated how the two protocols work in practice through experiments carried out Amazon servers where the running time of the two implementations has been compared. Experiments were done on many different circuits among others a circuit that would make sense to use in practice, namely a circuit for AES encryption. More specifically the test investigated the affect the number of players had, and what affect the number gates has in the different protocols.

The results from the experiments have shown that the practical implementation holds up to theory and that the lower communication complexity of CEPS SPEED results in a lower running time as well.

Overall it has been interesting to explore the theory, and implementing the protocols has been helpful for gaining understanding of them.

7.1 FUTURE WORK

The findings in this thesis can be of great use for future work, as it would be interesting to see how they compare to an implementation that better supports parallelism. It would also be interesting to use the experience gained do the implementations in in a lower level language, such that they are faster and could be used in practice. Then it would also be necessary to look how to best secure communication between the parties. Another future work is active security as it would have been interesting to see how they perform in practice and see much having active security affect the running time.

BIBLIOGRAPHY

[1] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. "Secure Multiparty Computation and Secret Sharing." In: (2015).

[2] Ivan Damgaard and Jesper Buus Nielsen. "Scalable and Unconditionally Secure Multiparty Computation." In: (2007).

[3] Ivan Bjerre Damgård and Jesper Buus Nielsen. "Commitment Schemes and Zero-Knowledge Protocols." In: (2017).

[4] Niels Lauritzen. "Concrete Abstract Algebra." In: (2002).

[5] Nigel Smart. *Circuits of Basic Functions Suitable For MPC and FHE*. URL: https://homes.esat.kuleuven.be/~nsmart/MPC/ (visited on 01/15/2019).

[6] Douglas Stinson. "Cryptography Theory and Practice." In: (2006).

[7] Andrew C. Yao. "Protocols for Secure Computations." In: (1982).

[8] Peter Bogetoft et. al. "Secure Multiparty Computation Goes Live." In: (2008).