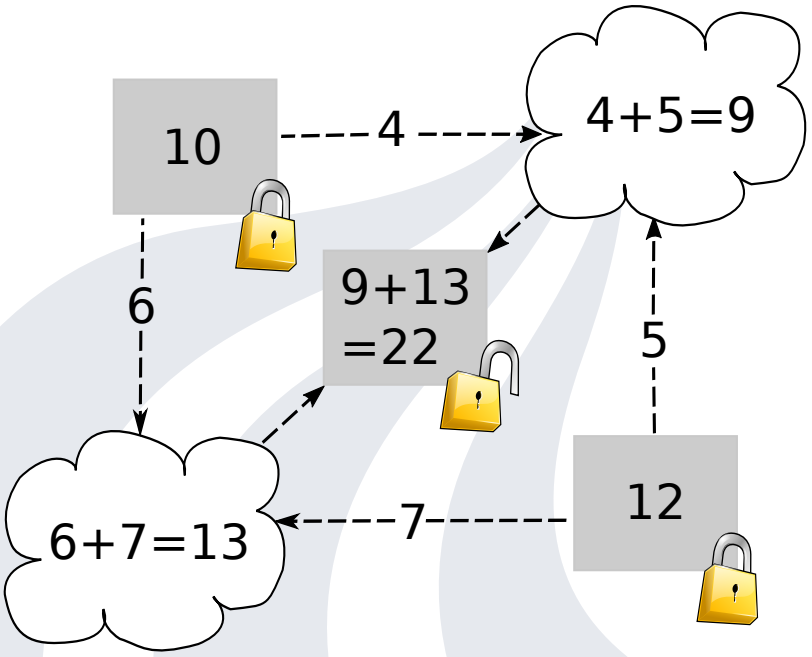
Privacy Preserving Control Using Multiparty Computation



Master's Thesis

Katrine Sofie Tjell

Aalborg University
Mathematics and Technology

Department of Electronics and IT
& Department of Mathematical Sciences
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Privacy Preserving Control Using Multiparty Computation

Theme:
Security

Project Period:
Autumn 2017 and Spring 2018

Project Group:
5.213b

Participants:
Katrine Sofie Tjell

Supervisors:
Rafael Wisniewski
Ignacio Cascudo Pueyo

Copies: 1

Page Numbers: 96

Date of Completion:
June 6, 2018

Synopsis:

The tendency in the technological world today, is that *things* should be *smart* and able to make decisions on their own. One example is the future smart-home, which will autonomously regulate the heat in the house, such that the temperature is appropriate when residents are home, but resources are minimized when the house is empty. It could even be so that the house will unlock, if some of its residents are approaching the front door. For the smart-home to make these decisions it needs data. However, people may be hesitant to reveal these private data. For this reason, there is a need for privacy preserving algorithms, that can carry out calculations on data hidden by encryption.
This thesis investigates the potential of using results from the field of secure multiparty computation to create such privacy preserving algorithms.
The findings are that known algorithms, such as the gradient decent method and the recursive least squares equations, can be formulated as secure multiparty protocols. It can be concluded that there are grounds for further research within this topic.

Preface

This thesis is written by me, Katrine Sofie Tjell, a student at the Mathematical Engineering masters education at Aalborg University. The project period started 1st September 2017 and ended 7th June 2018.

I would like to thank my supervisors Rafael Wisniewski and Ignacio Cascudo Pueyo for their exceptional effort in guiding me throughout the project period.

Aalborg Universitet, June 6, 2018

Katrine Sofie Tjell
<ktjell12@student.aau.dk>

Danish Abstract

Teknologiudviklingen går i dag i retning af at *ting* skal være *kloge*. Dette er i tråd med IoT (Internet of Things), hvor det ønskes at *ting* skal være forbundet og i nogen omfang være i stand til at tage beslutninger. Som eksempel kan nævnes fremtidens *smart-home*, som regulere varme/lys/aircondition i huset, sådan at når beboerne er hjemme er der behageligt at være og når huset er tomt skal der spares på ressourcerne. Det kan ydermere udvikles til at omfatte andre funktion, som for eksempel at huset låser dørene op når beboerne kommer hjem. For at alt dette kan lade sig gøre, er der brug for data. Huset skal vide hvor dets beboer er og hvad de ønsker. Dette kan betyde at nogle mennesker vil føle sig overvåget og de er modvillige til at afgive disse data. Derfor er der brug for metoder til at lave sikre beregninger på data som skal hemmeligholdes.

Denne specialeafhandling dokumenterer en undersøgelse om hvorvidt metoder fra research-feltet *sikkert distribueret beregninger (SDB)*, kan kombineres med algoritmer ofte anvendt indenfor regulering og automatisering. Formålet er at kunne lave estimeringer og optimeret løsninger når det tilgængelige data ønskes hemmeligholdt. Fremgangsmåden til dette er at studere kendte algoritmer og forsøge at omdanne disse til SDB protokoller ved brug af metoderne fra området.

Rapportens første halvdel har fokus på at opnå en basis forståelse for problemstillingen og betingelserne i SDB. Herunder introduceres to *secret sharing ordninger*, henholdsvis en *additiv ordning* og *Shamir's secret sharing ordning*. Begge ordninger er baseret på endelige legemer, hvilket sætter visse begrænsninger. Der introduceres ligeledes to SDB protokoller, som kan beregne henholdsvis summen og produktet af et sæt hemmelige værdier. Disse protokoller er blandt dem der i dag anvendes i praksis. Ideen er at alle andre SDB protokoller til udregning af mere generelle funktioner på hemmeligt data, kan skabes ud fra disse to protokoller.

Sidste halvdel af rapporten beskæftiger sig med at anvende den opnåede viden til at skabe nye SDB protokoller. Konkret undersøges 3 forskellige algoritmer fra hver deres felt, nærmere bestemt: optimerings algoritmen *gradient metoden* med backtracking linjeafsøgning, en *reguleringsalgoritme* til at regulere trykket i et vandnetværk og estimeringsmetoden *mindste kvadraters metode*. For at konverter disse metoder til SDB protokoller skal der blandt andet bruges operationerne sammenligning og division. Det undersøges derfor hvordan disse kan beregnes som en SDB protokol.

Gennem rapporten ønskes det at sammenligne resultater opnået med de tre nævnte algoritmer med resulter opnået med de tilsvarende SDB protokoller. Derfor er alle metoder implementeret i matematik software systemet, SageMath, således at simulering er muligt. Disse simuleringer viser at approksimativt opnås de samme resultat fra begge implementeringer.

Slutteligt konkluderes det at der er mulighed for at bruge metoder fra SDB til at opstille protokoller, som gøre det muligt at udfører en hel algoritme på hemmeligholdt data. Protokollerne som er forslået til sikker beregning af de tre algoritmer, skal ikke ses som færdige protokoller, men derimod fungerer de som et proof-of-concept. Afhandlingen viser derfor, at der er god grund til videre research indenfor dette område.

Reading Guide

This thesis is divided into two parts. The first part consists of two chapters, where the first is concerned with introducing the motivation and objectives of the thesis, while the second provides an introduction to secure multiparty computation (MPC). Thus, the reader who is familiar with secure MPC can skip chapter 2. It is assumed that the reader is familiar with finite fields and finite field arithmetic.

The second part of the thesis consist (besides of the conclusion) of three chapters, where each one is devoted to a particular algorithm, which is of interest to the field of control and automation. The goal is to convert each of the three algorithms into secure MPC protocols. To evaluate the success of the conversion, algorithms and secure protocols are implemented in the mathematics software system SageMath, such that comparison by simulations are possible. The implementations can be seen using the following links:

- Chapter 3, Gradient Descent as a Secure MPC Protocol:
  http://sage.math.gordon.edu/home/pub/132
- Chapter 4, Pressure Control Algorithm as a Secure MPC Protocol:
  http://sage.math.gordon.edu/home/pub/133
- Chapter 5, The Methods Of Least Squares as a Secure MPC Protocol:
  http://sage.math.gordon.edu/home/pub/131/

The concern of the second part is to investigate whether results from secure MPC can be used to create privacy preserving control algorithms.

References will be denoted by [author surname, year, page number].

Notation and Terminology

$[x_1, \ldots, x_n]$	A $1 \times n$ vector.
$[x_1, \ldots, x_n]^\top$	A $n \times 1$ vector.
$[x, f_x]_t$	Shamir sharings of a secret, x, using a t degree polynomial f_x.
$[x]_t$	Same as $[x, f_x]_t$.
$[x]$	Sharings of a secret, x, using any secret sharing scheme having secure protocols for addition and multiplication.
$[x]^B$	Bitwise sharings of a secret x.
$\boldsymbol{x}$	A vector.
$\boldsymbol{X}$	A matrix.
$\boldsymbol{I}_k$	The $k \times k$ identity matrix.
$\mathbf{0}_k$	The $k \times 1$ vector of zeros.
$\mathbf{1}_k$	The $k \times 1$ vector of ones.
$\vee$	Bitwise OR-operator.
$\oplus$	Bitwise XOR-operator.
$\mathbb{F}_p$	The finite field of p elements.
Secret	A value in $\mathbb{F}_p$, which is to remain hidden in any calculations.
Shares	"Parts" of secrets s, such that it takes some or all "parts" to reconstruct s.
Wrap-around p	The term *wrap-around p* is used to denote the situation where for instance $(a + r) \mod p < a$, for $a, r \in \mathbb{F}_p$. Hence, the situation where $a \in \mathbb{F}_p$ is increased but the outcome is less than a.
Wrap-around zero	The term *wrap-around zero* is used to denote the situation where for instance $(a - r) \mod p > a$, for $a, r \in \mathbb{F}_p$. Hence, the situation where $a \in \mathbb{F}_p$ is decreased but the outcome is larger than a.

Contents

Preface iii

Danish Abstract iv

Reading Guide vi

Notation and Terminology vii

I Introduction 1

1 Objectives and Motivation 3
1.1 Motivation: The *Smart*-Projects . . . 3
1.2 Objectives and Scope . . . 4

2 Secure Multiparty Computation 5
2.1 Introduction to Secure Multiparty Computation . . . 5
2.1.1 Adversaries . . . 6
2.1.2 Definition of Security . . . 7
2.2 Secret Sharing . . . 9
2.2.1 Additive Sharing Scheme . . . 10
2.2.2 Shamir's Secret-Sharing Scheme . . . 14
2.2.3 A Compact Notation for Shares by Shamir's Scheme . . . 20
2.3 Improving Efficiency by Preprocessing . . . 25
2.4 Actively Secure MPC Protocols . . . 28
2.4.1 Error-Correcting Algortihm . . . 28
2.4.2 Verification of Shares . . . 31
2.5 Summary . . . 37

II Application **39**

3 Gradient Descent as a Secure MPC Protocol **43**
3.1 Method of Gradient Descent . . . 43
3.2 Converting the Gradient Descent Algorithm to a Secure MPC Protocol . 46
3.2.1 Secure MPC: Comparison . . . 47
3.2.2 Secure MPC: Bit Decomposition . . . 50
3.2.3 Secure MPC: Division By Public Constant . . . 53
3.2.4 Secure MPC: Gradient Descent . . . 53
3.3 Simulation . . . 55
3.3.1 Simulation of **Protocol 3.8** with Fixed γ . . . 56
3.3.2 Simulation of **Protocol 3.8** using Backtracking Line Search . . . 57
3.4 Summery . . . 58

4 Pressure Control Algorithm as a Secure MPC Protocol **59**
4.1 Introduction to the Control Algorithm . . . 59
4.2 Converting the Pressure Control Algorithm to a Secure MPC Protocol . . 62
4.2.1 Secure MPC: the max-function . . . 62
4.2.2 Secure MPC: Pressure Control Algorithm . . . 63
4.3 Simulations . . . 64
4.4 Summary . . . 65

5 The Method of Least Squares as a Secure MPC Protocol **67**
5.1 Introduction to Recursive Linear Regression using the Method of Least Squares . . . 67
5.2 Converting the Recursive Least Squares Equations to a Secure MPC Protocol . . . 70
5.2.1 Modifying the Recursive Least Squares Equations to use Integer Division . . . 71
5.2.2 Secure MPC: Integer Division . . . 73
5.2.3 Secure MPC: Integer Division Protocol . . . 75
5.2.4 Secure MPC: Detecting wrap-around zero . . . 76
5.2.5 Secure MPC: Recursive Least Squares Equations . . . 78
5.3 Simulations . . . 79
5.3.1 Estimation of Parameters of a One Dimensional System . . . 80
5.3.2 Estimation of Parameters of a Two Dimensional System . . . 82
5.3.3 Estimation of Parameters of a Three Dimensional System . . . 84
5.4 Summary . . . 86

6 Conclusion and Future Work **87**
6.1 Future Work . . . 88
6.1.1 Comparison . . . 88
6.1.2 Scaling in Connection With Secure Integer Division . . . 88
6.1.3 Other Secret Sharing Schemes or Encryptions . . . 88

6.1.4 Designing Algorithms for the Purpose of Privacy Preservation . . . 89

Bibliography **91**

A Creating Hyper-invertible Matrices **93**

B PRE-OR of Integer Without Bit-Decomposition **95**

Part I

Introduction

1

Objectives and Motivation

The objective of this thesis is to investigate the possibility of combing theory form the field of cryptography, specifically secure multiparty computation (MPC), with methods within the field of control and automation. In particular, the focus is on the problem of making calculations on data, which are hidden by encryption. Why this problem is of interest to the field of control and automation, is described in the following motivational section.

1.1 Motivation: The *Smart*-Projects

During the last decade several *smart*-projects has started to develop. These are projects like smart-homes, smart-grids, and smart-transportation, [Alaa et al., 2017], [Khattak et al., 2012], [Paul et al., 2017]. The purposes of these projects are for instance to cut down on resource losses, optimize the use of renewable energy sources, optimize solutions from an economic perspective, and simply to increase living standards.

For instance, some of the challenges of the modern power grid, is to optimize power production to balance power consumption and at the same time use as much renewable energy as possible. However, the renewable energy sources are intermittent, as they may produce anything from too-little to too-much power. Thus, one of the ideas in the future smart-grid is to reverse the situation where power production adapts to consumption and instead make consumers adapt to production. To accommodate this, the future smart devices, like smart-washing machine, smart-dishwasher, and smart-meters must be able to decide when to turn on and when not to. To make this decision-making possible, the idea is to make the price of power reflect the production, such that the price is low when production is large. The smart-devices can then be set to turn on only when the price of energy is low, which is also beneficial for the consumer. The future electrical smart-car will also attempt to charge its battery only when the price of power is low. In fact, the electrical cars can help the problem of fluctuating power production, as they can be used

as small power banks, that is filled when power production is high and drained when production is low, [Khattak et al., 2012].

Another way to solve the issue of producing too much energy, is to use smart-heat pumps in district heating facilities, [Fischer and Madani, 2017]. These facilities must then adapt to heating consumption and power production. In this way, the future power-grid, smart-transportation, smart-house, and smart-heating facilities are closely connected and they are all controlled by data from multiple sources.

The future smart-transportation system will also depend on data from many sources. Consider for instance autonomous vehicles as public transportation, [Pereira et al., 2017]. The vehicles should be connected with different traffic controller units, that, for instance, will enable the autonomous vehicle to adjust its speed to match traffic light and find the route that is least occupied. Besides of being convenient for the passenger, this has the potential to increase road and intersection capacities, while reducing pollution, traffic accidents, and so on.

A common challenge these smart-projects faces is the *secure* exchange of data. In the mentioned examples, a lot of different decisions are made based on various calculations on various data. For instance, the smart grid needs to decide the price of power based on the estimated consumption and production. The consumption is estimated based on the consumption profile of the consumers. Regarding the smart transportation, the route of each vehicle is calculated based on traffic and also the passengers location and destination. The problem arise as the users may be hesitant to reveal such data. For instance, homeowners may not want to share their individual consumption profile and the passengers of an autonomous car may not want to reveal their stopovers or destination with other parties. Hence, the obstacle is to do calculations on private data, while ensuring no information leakage.

1.2 Objectives and Scope

The first objective of this thesis is to obtain a basic understanding of secure MPC and to gain knowledge on the state-of-the-art methods within secure MPC. The second objective is to exploit this knowledge to create secure MPC protocols, for doing various calculations on private (encrypted) data. It should be pointed out that the concern is to investigate whether this can be done, and thus very little focus is put on improving efficiency of any kind.

The approach is to consider different algorithms, which are of interest to the field of control. In particular, these are algorithms for optimization, estimation, and control. The idea is to use the methods from secure MPC to convert these well-known and frequently used algorithms into secure MPC protocols.

The aim of this report can be summarized as follows:

- *How can results from secure multiparty computation be used to create privacy preserving control algorithms?*

2

Secure Multiparty Computation

In this chapter the subject of *secure multiparty computation* is presented. Only some methods within this field are introduced, these are so-called BGW-like methods. BGW stands for Ben-Or, Goldwasser and Wigderson, which is the names of the authors of one of the most fundamental results of secure multiparty computation, [Asharov and Lindell, 2011]. This result was published in 1988 and since then some modifications has been introduced. This report focuses on these modified BGW-like methods, which are introduced in [Cramer et al., 2015], for instance. Since 1988 completely new methods has been created, but at this moment it is assumed that the modified BGW-like methods suits the need for this thesis the best.

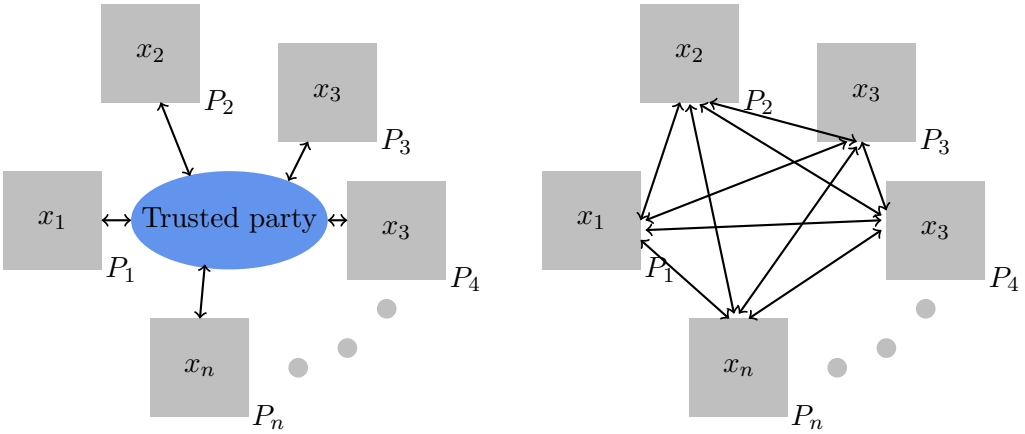
Section 2.1 provides an introduction to the setup in secure MPC, hereunder the term *adversaries* are introduced and it is discussed how to define *security*. In **Section 2.2** two secret sharing schemes are introduced. **Section 2.3** presents the idea in using a preprocessing phase before executing a secure MPC protocol. **Section 2.4** states two methods to deal with active adversaries. Finally, **Section 2.5** provides a summery of the chapter.

2.1 Introduction to Secure Multiparty Computation

This section is based on the paper [Orlandi, 2011] and the book [Cramer et al., 2015]. Secure multiparty computation (MPC) is concerned with creating methods that let a set of parties jointly compute a function, while ensuring the privacy of certain values. An intuitive example would be a set of parties, $P_1, \ldots, P_n$, that want to compute some function that takes one input from each party. That is, each party has a value, $x_1, \ldots, x_n$, which is input to the function $y = f(x_1, \ldots, x_n)$ and all parties want to know the output of the function. The parties do not trust each other, thus they want to keep their input value private.

A simple solution to this problem is to employ a trusted third party to do the computation. In this way, the parties would not have to reveal their private value to anyone but the trusted party. However, in secure MPC it is assumed that a trusted party can only exist in an ideal world, since it is a strong assumption that an incorruptible party exists. Thus, in the real world of secure MPC the parties must compute the function themselves in order to achieve privacy. To do this, secure protocols are used and it is the aim of secure MPC to create these secure, privacy preserving protocols.

The ideal and real worlds of secure MPC are illustrated in **Figure 2.1**.



(a) Ideal-world. **(b)** Real world

Figure 2.1: Illustration of the two worlds in secure MPC; the ideal-world and the real-world. The gray boxes represents parties with a private value and arrows illustrates secure channels for communication.

The concern in secure MPC is not only to do computations on secret data, but also to keep the computations secure against malicious behavior.

2.1.1 Adversaries

The aim of malicious behavior can be to learn private information or to cause the result of the computation to be incorrect. Any entity behaving maliciously is considered an *adversary*.

The adversary works by taking control of a subset of the parties, which are then referred to as *corrupted* parties. Conversely, a party that is not corrupted is referred to as an *honest* party.

An adversary may either be external or internal, and furthermore there may be multiple adversaries attacking the same protocol. However, the adversaries attacking the same protocol are assumed to cooperate, thus without loss of generality it suffices to consider one adversary who is in control of all the corrupted parties.

The adversary can behave in two ways; it can either be *active* or *passive*. In the case of a passive adversary, all parties follow the protocol, but the adversary learns all values that the corrupted parties learn and attempts to use this to learn private information.

An active adversary takes one step further as he may also instruct the corrupted parties to deviate from the protocol attempting to manipulate the result of the computation.

2.1.2 Definition of Security

Once a protocol is created it must be proved that the protocol is secure. For this purpose, a definition of security is necessary. As described previously, the behavior of an adversary can either be passive or active. Therefore, it is also common to distinguish between active and passive security, such that a protocol, which is only secure against a passive adversary, can be stated. It should be obvious that a protocol that is secure against an active adversary is also secure against a passive one. In section 2.4 two approaches to deal with active adversaries are presented, until then the focus is on security against a passive adversary.

Intuitively, a secure protocol ensures that no adversary is able to learn any private information, that the result of the computation is correct, that the input values are independent and so on. As can be seen, this list of properties that a secure protocol must satisfy could turn out to be extremely long, given that adversaries may attack in numerous ways. Thus, instead of attempting to produce such a list, the so-called *simulation paradigm* is used to show that a given protocol is secure. In this paradigm, a protocol execution in the real-world is compared to doing the corresponding computation in the ideal-world. The idea is that, if the knowledge of the real-world adversary can be generate in the ideal-world, then the real-world adversary has learned nothing more than he can in the ideal-world, and hence the protocol is secure.

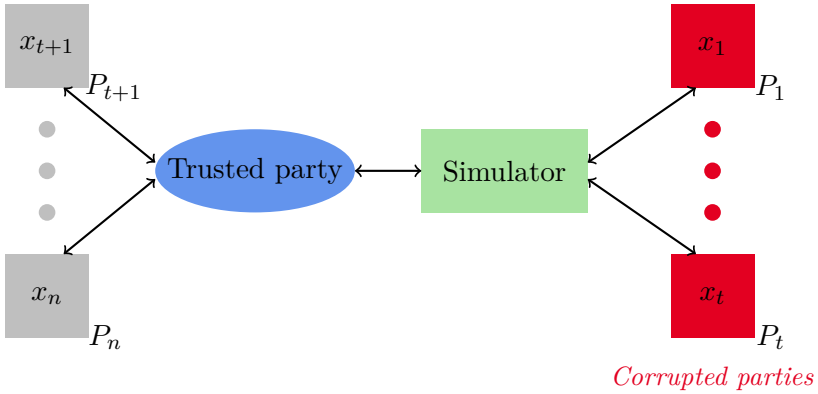


Figure 2.2: Illustration of an ideal-world computation with the parties $P_1, \ldots, P_n$, each having a private value $x_1, \ldots, x_n$. In the figure, the first t parties are corrupted. The illustration gives intuition about the *simulation paradigm*, where the simulator generates a view for the ideal-world adversary, based on information received by the trusted party and the corrupted parties. Arrows illustrates secure channels for communication.

Figure 2.2 illustrates how the simulator carries out the communication between the corrupted parties and the trusted party. If the simulator is capable of generating a view for the adversary, so that the adversary cannot distinguish between being in the ideal-

world and the real-world, the protocol must be secure since the ideal-world is secure by definition. Before a formal definition of security can be stated, the terms *view* and *simulator* must be defined.

Definition 2.1 (View)
The view, $\boldsymbol{v_i}$, of a party, P_i, is a list of the values that P_i get to know during protocol execution. That is, $\boldsymbol{v_i}$ is a vector with the private value, x_i, of P_i in the first entry. Each time P_i calculates a value it is added to $\boldsymbol{v_i}$ and each time P_i receives a message it is added to $\boldsymbol{v_i}$. The view of multiple parties, $P_p, \ldots, P_q$, is written as

$$\boldsymbol{v_C} = \begin{bmatrix} \boldsymbol{v}_p \\ \vdots \\ \boldsymbol{v}_q \end{bmatrix}, \tag{2.1}$$

where C denotes the set of indicies of the parties.

Later it will be seen that when P_i calculates a value it is based on random choices and similarly when P_i receives a message from party P_j, the message will also be based on random choices made by P_j. The view, $\boldsymbol{v_i}$, is therefore a random vector and cannot be calculated deterministically. Thus, when saying that the view of the ideal-world and real-world adversary must be indistinguishable, it is not meant that the two vectors must be identical, but rather that their distributions must be identical. To define when two random vectors are indistinguishable, the term *statistical distance* is used.

Definition 2.2 (Statistical Distance)
Define X_0 and X_1 as two random variables on the same probability space and let D be their common sample space. The statistical distance, $\delta(X_0, X_1)$ is defined as

$$\delta(X_0, X_1) = \frac{1}{2} \sum_{d \in D} |\Pr[X_0 = d] - \Pr[X_1 = d]|, \tag{2.2}$$

[Cramer et al., 2015, p. 16]

Furthermore, the set $\{X(\kappa)\}_{\kappa \in \mathbb{N}}$ is called a *family of random variables* and now the term *indistinguishability* can be defined.

Definition 2.3 ((Perfect) Indistinguishability)
Let X_0 and X_1 be as in **Definition 2.2**. X_0 and X_1 are perfectly indistinguishable, denoted $X_0 \equiv X_1$, if for all $\kappa \in \mathbb{N}$

$$\delta(X_0(\kappa), X_1(\kappa)) = 0, \tag{2.3}$$

[Cramer et al., 2015, p. 21].

Remark, indistinguishability in **Definition 2.3** is in most literature, including [Cramer et al., 2015], referred to as *perfect indistinguishability*. However, in this report this is the only type of indistinguishability that is considered, thus the word *perfect* is omitted.

Now, a *simulator* can be defined, using the definitions just presented, see **Definition 2.4**.

Definition 2.4 (Simulator)
Let $P_1, \ldots, P_n$ be a set of parties each with a secret value, such that P_i has the value x_i for $i = 1, \ldots, n$. Let $C \subset \{1, \ldots, n\}$ denote the indices of corrupted parties, such that $|C| \leq t$, for an integer $t < n$. Furthermore, let $\boldsymbol{v_C}$ be the view of the corrupted parties and let $y = f(x_1, \ldots, x_n)$, where f is some function. A *simulator* is an efficient probabilistic algorithm, that given $\{x_j, y\}_{j \in C}$, generates a random vector, $\boldsymbol{s}(\{x_j, y\}_{j \in C})$, with values that are indistinguishable from the view of the corrupted parties, $\boldsymbol{v_C}$. That is,

$$\boldsymbol{s}(\{x_j, y\}_{j \in C}) \equiv \boldsymbol{v_C}, \tag{2.4}$$

[Cramer et al., 2015, p. 40]

Finally, a protocol that is secure against a passive adversary is defined in **Definition 2.5**.

Definition 2.5 (Security against a passive adversary)
Let $P_1, \ldots, P_n$ be parties, and assume that at most $t < n$ of them are corrupted by a passive adversary. A protocol is said to be *secure against a passive adversary*, if the protocol generates a view for the corrupted parties, which can also be generated by a simulator given in **Definition 2.4**. That a protocol is secure against a passive adversary, means that the adversary will not gain knowledge on any secret values belonging to honest parties, and that the protocol ensures that all parties learn the correct output, [Prabhakaran and Sahai, 2013, p. 9].

The setup of secure MPC has been presented, the focus is now on the building block of the secure MPC protocols presented in this report, namely *secret sharing*.

2.2 Secret Sharing

Secret sharing covers methods for allowing a party to distribute a secret amongst a set of parties without revealing the secret. It is done in such a way, that each participant

receives a piece of information that reveals nothing about the secret. In fact it takes some or all of the pieces of information to recreate the secret. The piece of information is referred to as a *share*, see **Definition 2.6**.

Definition 2.6 (Share)
Let s be a secret value and k an integer. Assume that there exist an algorithm, which produces the values, s_i, $i = 1, \ldots, k$, such that s_i is known for some or all i, s can be calculated. Each s_i value is referred to as a share of s.

It is assumed that secure channels exist between each pair of participants, such that a particular share is learned by solely the participant it was meant for. In regards to this, **Definition 2.7** states what is meant by *distributing shares* and *broadcasting* a value.

Definition 2.7 (Distribution and Broadcasting)
Let $P_1, \ldots, P_n$ be parties. Assume that there exist a secure channel between every pair of parties and that $\{s_{j_i}\}_{i=1,\ldots,n}$ are shares of a secret value s_j, which belong to party P_j. When a party, P_j, *distributes the shares of* s_j, it means that P_j securely sends the value s_{j_1} to party P_1, s_{j_2} to party P_2 and so on and so forth till party P_n has received s_{j_n}.

On the other hand, when a party P_j *broadcasts* a value a, it means that P_j makes a known to all parties.

Note that distribution is much more costly than broadcasting, since it requires the use of secure channels to make sure that the message is solely learned by the intended receiver. Conversely, when broadcasting a value, anyone is allowed to learn the value, therefore it could for instance be announced using the Internet.

It is often in secret sharing schemes assumed that the secret is an element of a finite field, $\mathbb{F}_p$, where p, the cardinality of the field, is a prime. One reason why it is advantageous to consider finite fields is that, it is often necessary to have some uniformly random numbers available, these can easily be sampled from a finite field.

There exist various secret sharing schemes, two well known ones are the *additive sharing scheme* and *Shamir's secret sharing scheme*.

2.2.1 Additive Sharing Scheme

As the name suggests the additive scheme defines the shares of a secret, such that the sum of all shares equals the secret. To be precise, if $s \in \mathbb{F}_p$ is a secret and $s_1, \ldots, s_n$ are shares of s, then

$$s = \sum_{i=1}^{n} s_i \mod p. \tag{2.5}$$

The protocol for creating additive shares of a secret is stated in **Protocol 2.1**.

Protocol 2.1 (Additive Sharing Scheme)
Given a secret $s \in \mathbb{F}_p$.
Outputs the shares $s_1, \ldots, s_n$ of s, where $s_i \in \mathbb{F}_p$ for $i = 1, \ldots, n$.

1. Draw $n-1$ values from a uniform distribution on $\mathbb{F}_p$. The $n-1$ values are referred to as $s_2, \ldots, s_n$. Furthermore, choose a value $s_1 \in \mathbb{F}_p$, such that

$$s = \sum_{j=1}^{n} s_j \mod p. \tag{2.6}$$

To see why the additive sharing scheme is of interest to secure MPC, suppose that n parties each have a secret value, and they want to compute the sum of all values. A secure MPC protocol for this, is stated in **Protocol 2.2**.

Protocol 2.2 (Addition *(Additive Sharing Scheme)*)
Given that party P_i, for $i = 1, \ldots, n$, holds the shares $a_{j_i} \in \mathbb{F}_p$ for $j = 1, \ldots, n$, of the secrets $a_1, \ldots, a_n$, where $a_j \in \mathbb{F}_p$ belongs to party P_j. The shares are created using **Protocol 2.1**.
Outputs $\sum_{i=1}^{n} a_i$ to all parties.

1. Each party, P_i, $i = 1, \ldots, n$, computes the value

$$d_i = \sum_{j=1}^{n} a_{j_i} \mod p. \tag{2.7}$$

2. Each party, P_i, $i = 1, \ldots, n$, broadcasts the value d_i.
3. All parties compute

$$y = \sum_{i=1}^{n} d_i \mod p. \tag{2.8}$$

That **Protocol 2.2** is secure under **Definition 2.5** is showed in the following proof.

Proof. First of all, by associativity of addition it is trivial that the protocol ensures correct function evaluation in the passive adversary case. To prove that all input values are kept private, it must be shown that there exists a simulator according to **Definition 2.4**.

Assume that the parties $P_1, \ldots, P_t$ are corrupted and that $P_{t+1}, \ldots, P_n$ are honest parties. Let $T = \{1, \ldots, t\}$ and $N = \{1, \ldots, n\}$ be index sets. Table 1 gives the view of the real-world and ideal-world adversary.

	Real-world	Ideal-world
Values known to the adversary	$\{x_i\}_{i\in T}$ y $\{a_{i_j}\}_{(i,j)\in T\times N}$ $\{a_{i_j}\}_{(i,j)\in\{t+1,\dots,n\}\times T}$ $\{d_i\}_{i\in N}$	$\{x_i\}_{i\in T}$ y

Table 2.1: View of the real- and ideal-world adversary for **Protocol 2.2**.

Now it must be shown that all values the real-world adversary sees can be generated by the simulator in the ideal-world.

The first values that must be generated are $\{a_{i_j}\}_{(i,j)\in T\times N}$. As seen in **Figure 2.2**, in the ideal-world the corrupted parties communicate with the simulator as if it was the trusted party. Thus, the simulator receives $\{x_i\}_{i\in T}$ from the corrupted parties and then it can generate $\{a_{i_j}\}_{(i,j)\in T\times N}$ according to **Protocol 2.1**. Hence, the distribution of $\{a_{i_j}\}_{(i,j)\in T\times N}$ in the ideal-world is indistinguishable from the distribution of the corresponding values in the real-world.

The next values are $\{a_{i_j}\}_{(i,j)\in\{t+1,\dots,n\}\times T}$. These values give no information about x_i and they can be simulated as uniformly random values on $\mathbb{F}_p$. To show that this is true, consider the map $\phi : {\mathbb{F}_p}^{n-t} \to \mathbb{F}_p$,

$$\phi(a_{i_{t+1}},\dots,a_{i_n}) = \left(\sum_{j=t+1}^{n} a_{i_j} + c\right) \mod p = x_i, \tag{2.9}$$

where $c = \sum_{j\in T} a_{i_j}$ is a known constant.

Now, if every $x_i \in \mathbb{F}_p$ has the same number of preimages of ϕ, it is proved that no information is gained from knowing c.

Consider the following sets

$$S_0 = \{(a_{i_{t+1}},\dots,a_{i_n}) \in \mathbb{F}_p \mid x_i = 0\} \tag{2.10}$$
$$S_1 = \{(a_{i_{t+1}},\dots,a_{i_n}) \in \mathbb{F}_p \mid x_i = 1\} \tag{2.11}$$
$$\vdots \tag{2.12}$$
$$S_{m-1} = \{(a_{i_{t+1}},\dots,a_{i_n}) \in \mathbb{F}_p \mid x_i = m-1\}. \tag{2.13}$$

Now, if all sets have equal cardinality, then $x_i \in \mathbb{F}_p$ has the same number of preimages. To show that this is the case, consider the map $\psi\colon S_k \to S_l$

$$\psi(a_{i,t+1},\dots,a_{i_n}) = (a_{i_{t+1}},\dots,(a_{i_n}+l-k) \mod p), \quad k<l. \tag{2.14}$$

By observing that ψ is bijective, it is clear that the sets, $S_0,\dots,S_N$, have equal cardinality. This means that given $\{a_{i_j}\}_{(i,j)\in\{t+1,\dots,n\}\times T}$, it is not possible to guess x_i with a higher probability than not knowing these values. Thus, the values can be simulated as uniformly random variables on $\mathbb{F}_p$, which again makes the real-world and ideal-world distribution of $\{a_{i_j}\}_{(i,j)\in\{t+1,\dots,n\}\times T}$ indistinguishable.

The last values for the simulator to generate are $\{d_i\}_{i\in N}$. It is seen that no information about private values can be gained from any of these values, because they give no information about a_{i_j} for any i, j. This means that in the view of the real-world adversary, $\{d_i\}_{i\in N}$ are uniformly random variables, with the dependence that they sum to y. The simulator can calculate $\{d_i\}_{i\in T}$ from $\{a_{i_j}\}_{(i,j)\in N\times T}$, which all are values previously generated by the simulator. $d_{t+1}, \ldots, d_{n-1}$ can be drawn uniformly on $\mathbb{F}_p$ and d_n is chosen such that $y = (\sum_{i\in N} d_i) \mod p$. Hence, a simulator according to **Definition 2.4** exist.

□

Now, if the parties instead wishes to multiply all their secrets, there is no obvious way to do that using the additive sharing scheme. Since the additive sharing scheme and **Protocol 2.2** works because of the associativity of addition, an idea that comes to mind is using the associativity of multiplication, to create a secret sharing scheme and secure protocol for multiplication. Suppose that a *multiplicative sharing scheme* is defined similarly to the additive sharing scheme in **Protocol 2.1**, with the distinction that the shares, $s_1, \ldots, s_n$ of the secret, $s \in \mathbb{F}_p$, are such that

$$s = \prod_{i=1}^{n} s_i \mod p. \tag{2.15}$$

Using this, a protocol for multiplying secrets is stated in **Protocol 2.3**.

Protocol 2.3 (Multiplication *(Multiplicative Sharing Scheme)*)
Given that party P_i for $i = 1, \ldots, n$, holds the shares $a_{j_i} \in \mathbb{F}_p$ for $j = 1, \ldots, n$, of the secrets $a_1, \ldots, a_n$, where $a_j \in \mathbb{F}_p$ belongs to party P_j. The shares are created using the multiplicative sharing scheme.
Outputs $\prod_{i=1}^{n} a_i$ to all parties.

1. Each party, P_i, $i = 1, \ldots, n$, computes the value

$$d_i = \prod_{j=1}^{n} a_{j_i} \mod p. \tag{2.16}$$

2. Each party, P_i, $i = 1, \ldots, n$ broadcasts the value d_i.

3. All parties compute

$$y = \prod_{i=1}^{n} d_i \mod p. \tag{2.17}$$

Unfortunately, **Protocol 2.3** is not secure. To see this, consider **Example 2.1** which gives an example of how a party can learn information about a private value of another party.

Example 2.1
Let P_1, P_2, P_3, be participants in the execution of **Protocol 2.3**. Let $p = 3$ and the private value of P_i be $a_i \in \mathbb{F}_3$ for $i = 1, 2, 3$. For the protocol to be secure, no party must gain any information about the private value of other parties. Consider the case where $y = 0, a_1 = 0, a_2 = 1$ and $a_3 = 2$. Then P_1 and P_2 knows that at least one of the other parties has private value equal to zero. This does not make the protocol insecure since this would also happen in the ideal-world. However, consider **Equation (2.15)**. Since $y = 0$, then either a_{1_1}, a_{1_2} or a_{1_3} must be equal to zero. Say that $a_{1_2} = 0$, then P_2 learns that $x_1 = 0$, which makes the protocol insecure.

Even if **Protocol 2.3** was secure, it would not really help with the fact that, when doing secure MPC it will often be necessary to compute more general functions than the sum or product of all secrets. Hence, a secret sharing scheme that can be used in secure protocols for both addition and multiplication will be needed. Remark, it is possible to introduce secure multiplication for the additive scheme, however it involves complicated tricks, which are out of the scope of this report. Instead the report proceeds by introducing Shamir's secret sharing scheme.

2.2.2 Shamir's Secret-Sharing Scheme

Shamir's secret sharing scheme uses Lagrange polynomials to create shares of a secret $s \in \mathbb{F}_p$. A polynomial $f(x) \in \mathbb{F}_p$ of degree $t < n$ is chosen such that $f(0) = s$ and the rest of the coefficients are random. The shares of s is then determined as $s_1 = f(1), \ldots, s_n = f(n)$ and the shares of the secret is distributed according to **Definition 2.7**. Shamir's scheme uses exclusively finite field arithmetic, thus any arithmetic in any stated protocol or proof of a protocol using Shamir's scheme uses finite field arithmetic.

Protocol 2.4 states Shamir's secret-sharing scheme formally.

Protocol 2.4 (Shamir's Secret-Sharing Scheme)
Given a secret $s \in \mathbb{F}_p$ and an integer $t < n$, where n is the number of participants.
Outputs the shares $s_1, \ldots, s_n$ of s.

1. Choose coefficients $\{a_1, \ldots, a_t\} \in {\mathbb{F}_p}^t$ uniformly at random.
2. Define the polynomial $f(x) = s + a_1 x + \cdots + a_t x^t$ in $\mathbb{F}_p$.
3. Define the shares of s as $s_1 = f(1), \ldots, s_n = f(n)$.

The protocol is from [Cramer et al., 2015, p. 244].

Protocol 2.5 states how a secret is reconstructed given at least $t + 1$ shares of the secret.

Protocol 2.5 (Output Reconstruction)
Given at least $t+1$ shares, s_i for $i \in C$, where $C \subseteq \{1, \ldots, n\}$, is the set of indices of the shares.
Outputs the reconstructed secret s.

1. Construct the polynomial
$$h(x) = \sum_{i \in C} s_i l_i(x), \tag{2.18}$$
where $l_i(x)$ is called a Lagrange basis polynomial and is given by
$$l_i(x) = \prod_{j \in C, j \neq i} \frac{x-j}{i-j}. \tag{2.19}$$
2. The secret s is given as $s = h(0)$.

The protocol is from [Cramer et al., 2015, p. 244].

The technique used in **Protocol 2.5** is called Lagrange interpolation. According to this theory, a polynomial $f(x) \in \mathbb{F}_p$ of degree at most t, can be expressed through $t+1$ points of $f(x)$. That is,
$$f(x) = \sum_{i \in C} f(i) l_i(x), \tag{2.20}$$
where $C \subseteq \mathbb{F}_p$ with cardinality $|C| = t+1$ and $l_i(x)$ are given in **Equation (2.19)**. When the $l_i(x)$ polynomial is evaluated in $i \in C$ it equals one and when evaluated in $k \in C$, where $k \neq i$, it equals zero. This is seen as
$$l_i(k) = \prod_{j \in C, j \neq i} \frac{k-j}{i-j} = \frac{k-1}{i-1} \cdots \frac{k-k}{i-j} \cdots \frac{k-(t+1)}{i-(t+1)} = 0, \quad \text{for } k \neq i, \tag{2.21}$$
$$l_i(i) = \prod_{j \in C, j \neq i} \frac{i-j}{i-j} = 1. \tag{2.22}$$

Hence, $\sum_{i \in C} f(i) l_i(x)$ evaluates to $f(x) \forall x \in C$. That **Equation (2.20)** holds for $x \notin C$, is seen as each $l_i(x)$ polynomial has degree at most t, which means that $\sum_{i \in C} f(i) l_i(x)$ is a polynomial of degree at most t. Therefore, $g(x) = f(x) - \sum_{i \in C} f(i) l_i(x)$ is zero on all points in C. Since it is assumed that $|C| > t$, the polynomial $g(x)$ has more zeros than its degree and thus according to the fundamental theorem of algebra, it must be the zero polynomial. Hence, it follows that $f(x) = \sum_{i \in C} f(i) l_i(x) \, \forall \, x \in \mathbb{F}_p$.

Note that in the classical reconstruction phase of Shamir's scheme, the whole polynomial is reconstructed, as shown in **Protocol 2.5**. However, when the interest is in calculating the secret $s = f(0)$, the reconstruction phase can be optimized for this purpose. Since $f(0)$ is all that is needed, the Lagrange basis polynomials in **Equation (2.19)**

$l_i(x)$ can be simplified to

$$r_i = l_i(0) = \prod_{j \in C, j \neq i} \frac{-j}{i-j}. \tag{2.23}$$

The vector

$$\boldsymbol{r} = [r_1, \ldots, r_n]^\top, \tag{2.24}$$

is called the recombination vector and note that it does not depend on $h(x)$. Hence, the same recombination vector $\boldsymbol{r}$ works for all polynomials of degree $t < n$. Using this, **Protocol 2.5** can be simplified to directly compute $h(0)$,

$$h(0) = \sum_{i \in C} r_i s_i. \tag{2.25}$$

Theorem 2.1 gives two important facts about Shamir's secret-sharing scheme.

Theorem 2.1
Let $s_1, \ldots, s_n$ be shares of a secret, $s \in \mathbb{F}_p$, which has been determined using Shamir's secret-sharing scheme, **Protocol 2.4**. Let $t < n$ be the degree of the polynomial used in the secret-sharing. Then the following holds;

1. s can be reconstructed if $t+1$ or more of the shares are known.
2. Each share is a uniform random variable and thus any set of fewer than $t+1$ shares contains no information about s.

Proof. This proof is from [Cramer et al., 2015, p. 34]. That s can be reconstructed from $t+1$ or more shares are given by Lagrange interpolation theory.

For the second hypothesis, assume that only t shares are known and that $C \subseteq \mathbb{F}_p$ with cardinality $|C| = t$ and $0 \notin C$. According to Lagrange interpolation theory, the original polynomial $f(x)$ cannot be reconstructed from less than $t+1$ points. However, it must be shown that no information is gained from the t shares. Since the uniform distribution is independent from the secret s, it suffices to prove that the shares $s_1, \ldots, s_t$ are distributed uniformly. Recall that the coefficients $a = \{a_1, \ldots, a_t\} \in {\mathbb{F}_p}^t$ are uniformly distributed and $f(x) = s + \sum_{j=1}^{t} a_j x^j$. This can be seen as an evaluation map from ${\mathbb{F}_p}^t$ to ${\mathbb{F}_p}^t$, as a is mapped to $\{f(i)\}_{i \in C}$.

To see that this map is invertible, take any $\{f(i)\}_{i \in C} \in {\mathbb{F}_p}^t$. Then $f(x)$ is known on $t+1$ points since it is also known that $f(0) = s$. This means that Lagrange interpolation can be used to compute $f(x)$ and $a \in {\mathbb{F}_p}^t$. That is, the evaluation map is invertible. Any invertible map from ${\mathbb{F}_p}^t$ to ${\mathbb{F}_p}^t$ maps the uniform distribution on ${\mathbb{F}_p}^t$ to the uniform distribution on ${\mathbb{F}_p}^t$, [Cramer et al., 2015, p. 34]. Thus, the shares are uniformly distributed and since it takes $t+1$ shares to reconstruct s, any less shares reveals no information about s. □

Using Shamir's secret sharing scheme to produce shares of a secret, allows a function of secrets to be computed directly on the shares of the secrets. Consider **Protocol 2.6** that securely computes the sum of n secrets.

Protocol 2.6 (Addition *(Shamir's Secret-Sharing Scheme)*)
Given that each party, P_i for $i = 1, \ldots, n$, has a secret input value $x_i \in \mathbb{F}_p$.
Outputs $y = \sum_{i=1}^{n} x_i$, where $y \in \mathbb{F}_p$.

1. All parties agree on a polynomial degree $t < n$.
2. Each party, P_i, $i = 1, \ldots, n$, uses Shamir's secret-sharing scheme in **Protocol 2.4** and distributes the shares of their private value, x_i. **Table 2.2** shows the polynomial of each party and which shares each party receives in this step.

Party	Polynomial	Received Shares
P_1	$f_1(x) = x_1 + a_{1,1}x + \ldots + a_{1,t}x^t$	$f_1(1), f_2(1), \ldots, f_n(1)$
P_2	$f_2(x) = x_2 + a_{2,1}x + \ldots + a_{2,t}x^t$	$f_1(2), f_2(2), \ldots, f_n(2)$
$\vdots$	$\vdots$	$\vdots$
P_n	$f_n(x) = x_n + a_{n,1}x + \ldots + a_{n,t}x^t$	$f_1(n), f_2(n), \ldots, f_n(n)$

Table 2.2: The polynomial and received shares of each party.

3. Each party, P_i, $i = 1, \ldots, n$, computes the value, d_i

$$d_i = \sum_{j=1}^{n} f_j(i). \tag{2.26}$$

4. Each party, P_i, $i = 1, \ldots, n$, broadcasts the value d_i.
5. Each party computes the recombination vector, $\boldsymbol{r}$, defined by **Equation (2.23)** and **(2.24)**.
6. Each party computes

$$y = \sum_{i=1}^{n} r_i d_i. \tag{2.27}$$

The protocol is from [Cramer et al., 2015, p. 39].

That **Protocol 2.6** is secure in the sense of **Definition 2.5** is given in the following proof.

Proof. To see that the protocol ensures that all parties learn $y = x_1 + \cdots + x_n$, define the polynomial $h(x) = f_1(x) + \cdots + f_n(x)$. It can be seen that $h(0) = y$ and that $h(i) = d_i$.

Thus, when $d_1, \ldots, d_n$ are shared between all parties, all parties have n shares of the secret y. Since $t < n$, each party can reconstruct $h(x)$ and compute $y = h(0)$.

To see that the protocol is secure, assume that $P_1, \ldots, P_t$ are corrupted parties and that $P_{t+1}, \ldots, P_n$ are honest parties. To improve readability, define the index sets $T = \{1, \ldots, t\}$ and $N = \{1, \ldots, n\}$. Furthermore, let $h(x)$ be the degree t polynomial with constant term y. To show that **Protocol 2.6** is secure, there must exist a simulator capable of generating a view for the ideal-world adversary that is indistinguishable from the view of the real-world adversary. **Table 2.3** gives the view of the real-world and ideal-world adversary.

	Real-world	**Ideal-world**
Values known to the adversary	$x_1, \ldots, x_t$ y $\{f_j(i)\}_{(i,j)\in N\times T}$ $\{f_j(i)\}_{(i,j)\in T\times\{t+1,\ldots,n\}}$ $\{d_i\}_{i\in N}$	$x_1, \ldots, x_t$ y

Table 2.3: Overview of the view of the real-world and ideal-world adversary.

The first values that the simulator must generate are $\{f_j(i)\}_{(i,j)\in N\times T}$. The simulator receives $x_1, \ldots, x_t$ from the corrupted parties and hence it can generate $\{f_j(i)\}_{(i,j)\in N\times T}$ according to **Protocol 2.4**.

The next values to be generated are $\{f_j(i)\}_{(i,j)\in T\times\{t+1,\ldots,n\}}$. According to **Theorem 2.1**, any share is a uniform random variable and any set of fewer than $t+1$ shares holds no information. Hence, $\{f_j(i)\}_{(i,j)\in T\times\{t+1,\ldots,n\}}$ can be simulated as being uniformly random on $\mathbb{F}_p$.

Finally, the simulator must generate$\{d_i\}_{i\in N}$ defined in **Equation (2.26)**. These values are all shares of the polynomial $h(x) = f_1(x) + \cdots + f_n(x)$ of degree t. The simulator knows $\{d_i\}_{i\in T}$ from $\{f_j(i)\}_{(i,j)\in T\times N}$ and it knows that $h(0) = y$, thus the simulator knows $t+1$ shares of $h(x)$. The simulator can use Lagrange interpolation to construct $h(x)$ and afterwards it can calculate $\{d_i\}_{i\in\{t+1,\ldots,n\}} = \{h(i)\}_{i\in\{t+1,\ldots,n\}}$.

□

As mentioned, Shamir's scheme also allows the creation of a secure MPC protocol for multiplication of secrets. Unfortunately, such a protocol is more complicated than **Protocol 2.6**. This is because **Protocol 2.6** takes advantage of the fact that when two degree t polynomials are summed the resulting polynomial is also of degree t, whereas when two degree t polynomials are multiplied the resulting polynomial is of degree $2t$. The degree of the polynomials must always be smaller than the number of parties, n, otherwise there will not be enough shares to reconstruct the polynomial. Hence, the idea is to first compute the product between two secrets and then multiply that result with the next secret and so on. This is stated in **Protocol 2.7**, which for obvious reasons requires that $t < n/2$.

Protocol 2.7 (Multiplication *(Shamir's Secret-Sharing Scheme)*)
Given that each party, P_i for $i = 1, \ldots, n$, has a secret input value $x_i \in \mathbb{F}_p$.
Outputs $y = \prod_{i=1}^{n} x_i$, where $y \in \mathbb{F}_p$.

1. All parties agree on a polynomial degree $t < n/2$.
2. Each party, P_i, $i = 1, \ldots, n$, uses Shamir's secret-sharing scheme in **Protocol 2.4** and distributes the shares of their private value, x_i, see **Table 2.2**.
3. Define $h(x)$ as the $2t$ degree polynomial $h(x) = f_1(x)f_2(x)$. Note that $h(0) = x_1x_2$ and furthermore that P_i holds the share $h(i)$ of $h(x)$. Refer to the share $h(i)$ as $k_i = f_1(i)f_2(i)$.
4. Each party computes the recombination vector, $\boldsymbol{r}$, defined by **Equation (2.23)** and **(2.24)**.
5. Repeat steps (a) till (e) for $b = 3, \ldots, n$.
   (a) Each party P_i, distributes a share of k_i to each party using **Protocol 2.5**. An overview of this step is seen in **Table 2.4**.

Party	Polynomial	Recieved Shares
P_1	$h_1(x) = k_1 + b_{1,1}x + \ldots + b_{1,t}x^t$	$h_1(1), h_2(1), \ldots, h_n(1)$
P_2	$h_2(x) = k_2 + b_{2,1}x + \ldots + b_{2,t}x^t$	$h_1(2), h_2(2), \ldots, h_n(2)$
$\vdots$	$\vdots$	$\vdots$
P_n	$h_n(x) = k_n + b_{n,1}x + \ldots + b_{n,t}x^t$	$h_1(n), h_2(n), \ldots, h_n(n)$

Table 2.4: Overview of step **(a)**.

   (b) Each party, P_i, $i = 1, \ldots, n$, calculates

$$d_i = \sum_{j=1}^{n} r_j h_j(i). \tag{2.28}$$

   Remark that d_i is the i'th share of some degree t polynomial, $g(x)$, which evaluates to $h(0)$ in zero.
   (c) Redefine $h(x)$ from item 3, as the $2t$ degree polynomial $h(x) = g(x)f_b(x)$.
   (d) Each party, P_i, $i = 1, \ldots, n$, redefines k_i as

$$k_i = d_i f_b(i). \tag{2.29}$$

   Note that k_i is a share of $h(x)$.
6. Each party, P_i, $i = 1, \ldots, n$, broadcasts the value d_i.

7. Each party computes
$$y = \sum_{i=1}^{n} r_i d_i. \tag{2.30}$$
The protocol is from [Cramer et al., 2015, p.39].

Equation (2.28) should be explained further than a remark. What happens is that $h(0)$ can be reconstructed by
$$\begin{aligned} h(0) &= r_1 k_1 + r_2 k_2 + \cdots + r_n k_n && (2.31)\\ &= r_1 \left(\sum_{i=1}^{n} r_i h_1(i) \right) + \cdots + r_n \left(\sum_{i=1}^{n} r_i h_n(i) \right) && (2.32)\\ &= r_1 \left(\underbrace{\sum_{i=1}^{n} r_i h_i(1)}_{d_1} \right) + \cdots + r_n \left(\underbrace{\sum_{i=1}^{n} r_i h_i(n)}_{d_n} \right). && (2.33) \end{aligned}$$
This shows that d_i must be a share of a polynomial, $g(x)$, that has constant term $h(0)$. To see that $g(x)$ is of degree t, consider $g(i)$,
$$g(i) = r_1 h_1(i) + r_2 h_2(i) + \cdots + r_n h_n(i), \tag{2.34}$$
where $\{r_i\}_{i\in\{1,\ldots,n\}}$ can be seen as scaling factors. Hence, $g(i)$ is the sum of n polynomials of degree t evaluated in i. Thus, $g(x)$ is itself of degree t. That **Protocol 2.7** is secure is proved at the end of **Protocol 2.10**.

Secure multiplication as stated in **Protocol 2.7** seems complicated and the protocol itself is difficult to read, because one has to keep in mind all shares of all the secrets. Furthermore, from now on, all secure MPC protocols stated in this report uses Shamir's secret sharing scheme unless otherwise stated. Thus, there is motivation for introducing a new notation for shares, which will improve readability.

2.2.3 A Compact Notation for Shares by Shamir's Scheme

From **Protocol 2.6** and **2.7**, it is seen that the first steps, which could be referred to as an input-sharing phase, and the final step, which could be referred to as an output-reconstruction phase, is identical. Therefore, it is beneficial to state a general protocol with these two phases and a computation-phase in the middle. Before doing so, it is convenient to introduce a new notation, which will improve readability.

Definition 2.8
Let $a \in \mathbb{F}_p$ and $f(x)$ be a random polynomial over $\mathbb{F}_p$ with $f(0) = a$ and degree at most t. Define $[a; f]_t$ as n shares of the secret a, calculated from $f(x)$,
$$[a; f]_t = [f(1), \ldots, f(n)]^\top, \tag{2.35}$$

with n being a positive integer.

With this notation follows some trivial, yet very important and useful facts, see **Lemma 2.1**.

Lemma 2.1
Let $a, b, c \in \mathbb{F}_p$ and $f(x)$ and $g(x)$ be polynomials over $\mathbb{F}_p$ with degree at most t. Then it holds that

$$[a; f]_t + [b; g]_t = [a + b; f + g]_t, \tag{2.36}$$
$$c[a; f]_t = [ca; cf]_t, \tag{2.37}$$
$$[a; f]_t * [b; g]_t = [ab; fg]_{2t}, \tag{2.38}$$

where $*$ denotes the Schur-product, meaning entrywise multiplication, [Cramer et al., 2015, p. 38].

Proof. The proof of the first two equalities is trivial, thus the focus is on the last equality.

$$[a; f]_t * [b; g]_t = [f(1), \ldots, f(n)] * [g(1), \ldots, g(n)] \tag{2.39}$$
$$= [f(1)g(1), \ldots, f(n)g(n)] \tag{2.40}$$
$$= [(fg)(1), \ldots, (fg)(n)] \tag{2.41}$$
$$= [ab; fg]_{2t}, \tag{2.42}$$

where the last equality is seen to be true by the following observation;

$$(fg)(x) = (a + a_1 x + \ldots + a_t x^t)(b + b_1 x + \ldots + b_t x^t) \tag{2.43}$$
$$= ab + (a_1 b_2 + a_2 b_1)x + \ldots + (a_t b_t)x^{2t}, \tag{2.44}$$

that clearly shows that $(fg)(x)$ has constant term ab and is of degree $2t$.

□

With the new notation it is convenient to define a compact way to express the actions the parties can do with and on the shares. This is introduced in **Definition 2.9**.

Definition 2.9
Let P_i denote a party with secret $a \in \mathbb{F}_p$ and let $[a; f]_t$ be as in **Definition 2.8**. To remind the reader about *distribution* of shares, when P_i *distributes* $[a; f]_t$, it means that P_i securely sends the share $f(j)$ to party P_j for $j = 1, \ldots, n$. Conversely, when all parties have received a share of a secret a computed from the polynomial f, it is said that the parties *hold* $[a; f]_t$.

To describe the situation where the parties hold $[a; f_a]_t$ and $[b; f_b]_t$ and each party P_i computes $f_a(i) + f_b(i)$ or $f_a(i)f_b(i)$, it is said that the parties computes $[a; f_a]_t + [b; f_b]_t$ or $[a; f_a]_t * [b; f_b]_t$, respectively.

When a value x is *opened*, it means that each party P_i broadcasts $f_x(i)$, and afterwards all parties can do Lagrange interpolation to learn the value of x.

Protocol 2.8 states a general protocol for secure MPC using Shamir's secret sharing scheme.

Protocol 2.8
Given that each party, P_i for $i = 1, \ldots, n$, has a secret input value $x_i \in \mathbb{F}_p$.
Outputs $y = g(x_1, \ldots, x_n)$, where $y \in \mathbb{F}_p$.

- INPUT-SHARING: Each party, P_i, $i = 1, \ldots, n$ with private value, $x_i \in \mathbb{F}_p$, distributes $[x_i; f_{x_i}]_t$.
- COMPUTATION: This phase depends on g and thus it cannot be described in detail. However, generally the parties hold $[x_1; f_{x_1}]_t$,..., $[x_n; f_{x_n}]_t$ and by dividing g into simpler functions the parties can evaluate g. In the end the parties hold $[y; f_y]_t$.
- OUTPUT-RECONSTRUCTION: Each party, P_i, $i = 1, \ldots, n$, broadcasts $f_y(i)$. Then all parties can use Lagrange interpolation to compute $y = f_y(0)$.

The idea in **Protocol 2.8**, is that the input-sharing and output-reconstruction phases are the same for all protocols using Shamir's secret sharing scheme. Thus, from now, when defining a protocol it is only necessary to specify the computation phase, since the remaining two phases are given in **Protocol 2.8**. Furthermore, it will only be necessary to prove security of the computation phase, since the proof of **Protocol 2.6** already has proved security of the input-sharing and output-reconstruction phases. This way of describing protocols is demonstrated by **Protocol 2.9** and **Protocol 2.10**, which are protocols for addition and multiplication, respectively.

Protocol 2.9 (Computation-phase: Addition)
Given that the parties hold $[a; f_a]_t$ and $[b; f_b]_t$ and that $t < n$.
Outputs $[y; f_y]_t$, where $y = a + b$.

1. The parties compute $[y; f_y]_t = [a; f_a]_t + [b; f_b]_t$, where $f_y = f_a + f_b$.

Note, that **Protocol 2.9** is essentially the same as **Protocol 2.6**, but using the

introduced notation and **Protocol 2.8**, it can be stated much more compactly. The proof of security of **Protocol 2.9** is already given in the proof of **Protocol 2.6**.

Protocol 2.10 (Computation-phase: Multiplication)
Given $t < \frac{n}{2}$ and that the parties hold $[a; f_a]_t$ and $[b; f_b]_t$.
Outputs $[y; f_y]_t$, where $y = ab$.

1. The parties compute $[ab; h]_{2t} = [a; f_a]_t * [b; f_b]_t$, where $h = f_a f_b$.
2. Each party P_i distributes $[h(i); f_i]_t$.
3. The parties compute

$$\sum_{i=1}^{n} r_i [h(i); f_i]_t = [\sum_{i=1}^{n} r_i h(i); \sum_{i=1}^{n} r_i f_i]_t \tag{2.45}$$

$$= [h(0); \sum_{i=1}^{n} r_i f_i]_t \tag{2.46}$$

$$= [ab; \sum_{i=1}^{n} r_i f_i]_t \tag{2.47}$$

By defining $f_y = \sum_{i=1}^{n} r_i f_i$, it is seen that the parties now hold $[y; f_y]_t$, [Cramer et al., 2015, p. 39].

Note that **Protocol 2.10** can be repeated if more than two private inputs are to be multiplied and thus **Protocol 2.10** is essentially the same as protocol **Protocol 2.7**. It is now showed that **Protocol 2.10** (and thus also **Protocol 2.7**) is secure.

Proof. Assume that t parties are corrupted and that these are referred to as $P_1, \ldots, P_t$. Let $N = \{1, \ldots, n\}$, $T = \{1, \ldots, t\}$ be index sets. To see that **Protocol 2.10** is secure under **Definition 2.5**, consider the view of the real-world and ideal-world adversary.

	Real-world	**Ideal-world**
Values known to the adversary	$\{f_a(i)\}_{i \in T}$ $\{f_b(i)\}_{i \in T}$ $\{f_i(j)\}_{(i,j) \in T \times N}$ $\{f_i(j)\}_{(i,j) \in \{t+1,\ldots,n\} \times T}$ $\{f_y(i)\}_{i \in T}$	$\{f_a(i)\}_{i \in T}$ $\{f_b(i)\}_{i \in T}$ $\{f'_y(i)\}_{i \in T}$

Table 2.5: View of ideal-world and real-world adversary of **Protocol 2.10**.

There must exist a simulator that can generate $\{f_i(j)\}_{(i,j) \in T \times N}$ and $\{f_i(j)\}_{(i,j) \in \{t+1,\ldots,n\} \times T}$ and $\{f_y(i)\}_{i \in T}$ must be indistinguishable from $\{f'_y(i)\}_{i \in T}$.

First of all, the simulator receives the values $\{f_a(i)\}_{i\in T}$ and $\{f_b(i)\}_{i\in T}$ from the corrupted parties. Given that $h = f_a f_b$, means that the simulator can calculate $\{h(i)\}_{i\in t}$ and from these it can easily generate $\{f_i(j)\}_{(i,j)\in T\times N}$, by choosing random polynomials $f_1, \ldots, f_t$.

$\{f_i(j)\}_{(j)\in T}$ for a fixed i, are t shares of $[h(i); f_i]_t$. According to **Theorem 2.1**, these are uniformly random variables and can be simulated as such.

Finally, the $\{f_y(i)\}_{i\in T}$ are also uniformly random variables just as $\{f'_y(i)\}_{i\in T}$ are. Hence these sets are indistinguishable.

□

That protocols for addition and multiplication has been specified is not accidental. The reason is that all finite functions over a finite field can be expressed as a polynomial, and to evaluate a polynomial all that is needed is the operations of addition and multiplication. Thus, by the addition and multiplication protocols, all finite functions can be computed. This fact is stated more formally in **Proposition 2.1**.

Proposition 2.1
Every function $g : \mathbb{F}_p \to \mathbb{F}_p$ can be represented as a polynomial over $\mathbb{F}_p$.

Proof. Let $g : \mathbb{F}_p \to \mathbb{F}_p$ be any function. Since $\mathbb{F}_p$ is a finite field, it contains a finite number of elements, $F_1, \ldots, F_q$, where q is the number of elements in $\mathbb{F}_p$. For the proposition to be true, there must be a q degree polynomial h, such that $g(i) = h(i) \forall i \in \mathbb{F}_p$. Determining h, can be done by solving the following linear system of equations

$$a_0 + a_1F_1 + a_2F_1^2 + \cdots + a_qF_1^q = g(F_1) \tag{2.48}$$
$$a_0 + a_1F_2 + a_2F_2^2 + \cdots + a_qF_2^q = g(F_2) \tag{2.49}$$
$$\vdots \tag{2.50}$$
$$a_0 + a_1F_q + a_2F_q^2 + \cdots + a_qF_q^q = g(F_q). \tag{2.51}$$

This system can also be written as

$$\boldsymbol{V}\boldsymbol{a} = \boldsymbol{g}, \tag{2.52}$$

where $\boldsymbol{a} = [a_0, a_1, \ldots, a_q]^\top$, $\boldsymbol{g} = [g(F_1), \ldots, g(F_q)]^\top$ and

$$\boldsymbol{V} = \begin{bmatrix} 1 & F_1 & F_1^2 & \cdots & F_1^q \\ 1 & F_2 & F_2^2 & \cdots & F_2^q \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & F_q & F_q^2 & \cdots & F_q^q \end{bmatrix}. \tag{2.53}$$

Clearly, p exists if **Equation (2.52)** can be solved for $\boldsymbol{a}$, which is equivalent to saying that h exists if $\boldsymbol{V}$ has a nonzero determinant. $\boldsymbol{V}$ has the Vandermonde-structure and hence the determinant of $\boldsymbol{V}$ is $\prod_{i<j}(F_i - F_j)$. Since $F_i \neq F_j$ when $i \neq j$, the determinant of $\boldsymbol{V}$ is nonzero and thus the proof is complete.

□

2.3 Improving Efficiency by Preprocessing

Given that all finite functions defined on a finite field can be represented as polynomials, the protocols for addition and multiplication has the potential of being used frequently. For this reason it is of course desirable that these protocols are as time efficient as possible at the time of execution. To speed up protocol evaluation, it turns out that doing some preprocessing before the actual protocol execution, improves efficiency significantly. To see what this preprocessing consists of it is important to note that it is usually not local computations done by each party, which is time consuming in practice, but rather communication. When a party, P_i, distributes $[a; f_a]_t$, where a is some secret, the communication between P_i and each of the other parties must be secure to ensure that only the intended receiver gets the message. However, communication in the form of broadcasting is much less costly since anyone is allowed to learn the message. Therefore, if any distribution can be exchanged with broadcasting, the protocol will be more efficient.

When considering **Protocol 2.9** and **Protocol 2.10**, it is noticed that the former requires only local computations, whereas the latter requires each party to distribute a value. Thus, **Protocol 2.10** could be improved by turning the distribution into broadcasting using a preprocessing phase. To do this, three random values, referred to as a *Beaver's triplet*, is introduced.

Definition 2.10 (Beaver's Triplet)
Let $a, b \in \mathbb{F}_p$ be uniformly random and unknown to all parties. Then the triple $[a; f_a]_t$, $[b; f_b]_t$, $[c; f_c]_t$ where $c = ab$ is called a Beaver's triplet. Saying that the parties hold a Beaver's triplet means that the parties hold $[a; f_a]_t$, $[b; f_b]_t$ and $[c; f_c]_t$, [Cramer et al., 2015, p. 164].

Before explaining how a Beaver's triplet can improve time efficiency for the multiplication protocol, consider how the parties can create a triplet. The method is referred to as *Beaver's trick* and is introduced in **Protocol 2.11**.

Protocol 2.11 (Beaver's Trick)
Given $t < \frac{n}{2}$.
Outputs $[a; f_a]_t$, $[b; f_b]_t$ and $[c; f_c]_t$, where a and b are unknown random numbers and $c = ab$.

1. Each party P_i, $i = 1, \ldots, n$, distributes $[a_i; f_{a_i}]_t$ and $[b_i; f_{b_i}]_t$, where $a_i, b_i \in \mathbb{F}_p$ are uniformly random values chosen by P_i.
2. The parties compute $[a; f_a]_t = \sum_{i=1}^{n} [a_i; f_{a_i}]_t$.
3. The parties compute $[b; f_b]_t = \sum_{i=1}^{n} [b_i; f_{b_i}]_t$.

4. The parties invoke **Protocol 2.10** to compute $[c; f_c]_t = [a; f_a]_t * [b; f_b]_t$.

The protocol is from [Cramer et al., 2015, p. 164].

The preprocessing phase of a protocol execution consists of the creation of a large number of Beaver's triplets, which can take a lot of time given the communication needed. The reason why a large number of triplets must be created, is because each can only be used in one calculation. **Protocol 2.12** demonstrates how a Beaver's triplet can be used to speed up an execution of the multiplication protocol.

Protocol 2.12 (Computation-phase: Multiplication Using Beaver's Triplet)
Given that the parties hold $[a; f_a]_t$ and $[b; f_b]_t$ and furthermore, that the parties hold a Beaver's triplet, $[\alpha; f_\alpha]_t$, $[\beta; f_\beta]_t$, $[\gamma; f_\gamma]_t$.
Outputs $[y; f_y]_t$, where $y = ab$.

1. The parties compute $[d; f_d]_t = [a; f_a]_t - [\alpha; f_\alpha]_t$.
2. The parties compute $[e; f_e]_t = [b; f_b]_t - [\beta; f_\beta]_t$.
3. The parties open d and e.
4. The parties compute

$$[y; f_y]_t = de + d[\beta; f_\beta]_t + e[\alpha; f_\alpha]_t + [\gamma; f_\gamma]_t. \tag{2.54}$$

The protocol is from [Cramer et al., 2015, pp. 164-165].

Proof. To see that **Protocol 2.12** ensures that all parties learn $y = ab$, consider the computations throughout protocol execution. First the parties compute $[a; f_a]_t - [\alpha; f_\alpha]_t$ and $[b; f_b]_t - [\beta; f_\beta]_t$, meaning that the parties hold $[d; f_d]_t$ and $[e; f_e]_t$, respectively, by **Equation (1)** and **(2)**. Consider the following calculation

$$\begin{aligned} y &= ab && (2.55)\\ &= (a - \alpha + \alpha)(b - \beta + \beta) && (2.56)\\ &= (d + \alpha)(e + \beta) && (2.57)\\ &= de + d\beta + e\alpha + \alpha\beta && (2.58)\\ &= de + d\beta + e\alpha + \gamma, && (2.59) \end{aligned}$$

which shows that **Equation (2.54)** indeed computes the product of a and b.

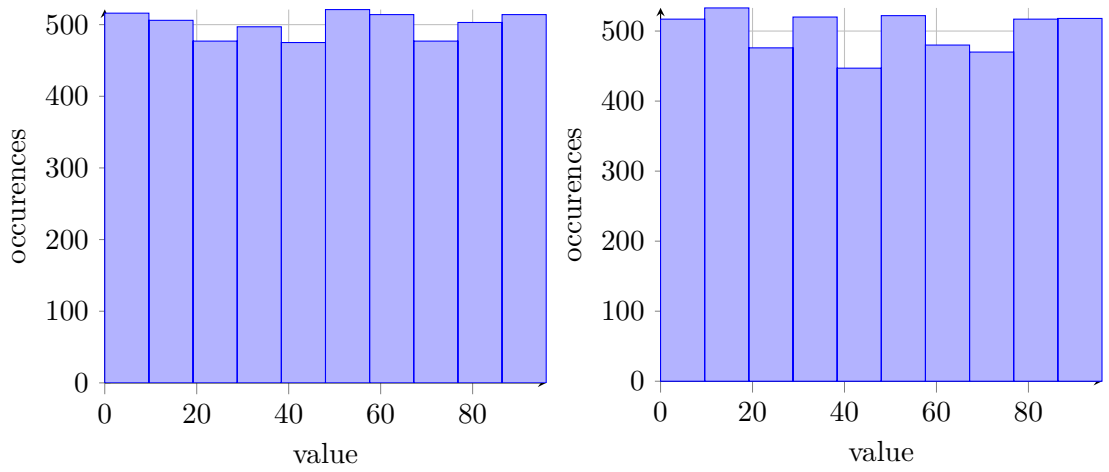
To show that the protocol is secure, all that is needed is to see that making the values d and e known to all parties does not reveal information about a or b. That the rest of the protocol is secure, can be seen from the proof of protocol **Protocol 2.10**.

Since α is uniformly random, so is $-\alpha$ and using the fact that adding a constant to a uniformly random variable still is a uniformly random variable, it is easily seen that d is just a uniformly random variable. The same goes for e. Thus, broadcasting d and e reveals no information about a or b.

□

It should be noted that secure multiplication is possible for an additive secret sharing scheme, given that a Beaver's triplet is available and that it is shared using the additive scheme.

As seen in the preceding proof, theoretically the values e and d should reveal no information. This fact, is now demonstrated in a more practically manner. Recall that p is the cardinality of the finite field, n is the number of parties and t is the number of corrupted parties. Simulating the execution of **Protocol 2.12** 5000 times, with $p = 97, n = 10$ and $t = 4$, shows that e and d are uniformly distributed on $\mathbb{F}_{97}$. The histograms for d and e from the described simulation, is seen in **Figure 2.3**.



(a) Histogram of the values of d.

(b) Histogram of the values of e.

Figure 2.3: Histograms, showing that the values d and e are uniformly distributed on $\mathbb{F}_{97}$, when simulating **Protocol 2.12** 5000 times, with $p = 97, n = 10$ and $t = 4$.

Since the compact notation $[a; f]_t = [f(1), \ldots, f(n)]$ was introduced, a few protocols has been considered. It should be clear by now that the polynomial f is actually of no concern, since it is just a random polynomial. The concern is rather the secret a. Thus, the notation can be made even more compact by suppressing f from it. Hence, from this point let $[a]_t = [a; f_a]_t$.

Throughout this section, the focus has been on passive security and protocols that are passively secure. The techniques presented are among those used in practice. However, in practice it is often not sufficient to assume only passive adversaries, one must also consider active ones.

2.4 Actively Secure MPC Protocols

In this section a few suggestions on how to deal with an active adversary is given. As mentioned previously an active adversary may deviate from the protocol, thus an actively secure protocol must be able to either verify that transmitted messages are valid *or* to correct invalid ones. Both these options are considered in the following. First, an error-correcting algorithm is presented.

2.4.1 Error-Correcting Algortihm

The error-correcting algorithm can be used in reconstruction phases, to ensure that the correct value is reconstructed even if t parties have transmitted an erroneous share of the value. There is, however, an upper bound on t, namely $3t + 1 \leq n$.

Suppose that a party receives shares of a value, a, from all parties and wants to reconstruct a. Refer to the share received from party P_i as y_i. The points $(1, y_1), \ldots, (n, y_n)$ are then points on the polynomial f_a, which in zero evaluates to a. However, it is possible that $f_a(i) \neq y_i$ for up to t of the points, since at most t parties are corrupted by an active adversary. Thus, Lagrange interpolation is not a good option, since it is unknown which points are correct and which are not. Instead, the idea is to define two polynomials $e(x)$ and $h(x)$ such that $f_a(x)e(x) = h(x)$. Hence, if somehow $h(x)$ and $e(x)$ can be determined, then $f_a(x)$ can be calculated as well.

$e(x)$ is called an *error locator polynomial* and is a degree t polynomial that is defined as being zero for all i where y_i is incorrect. This is written formally as,

$$e(i) = 0 \quad \text{whenever } f_a(i) \neq y_i,\ 1 \leq i \leq n. \tag{2.60}$$

As stated, it is required that the degree of $e(i)$ is t, but recall that $f_a(i) \neq y_i$ for at most t points. For this reason $e(i)$ is allowed to be zero for other i than those where $f_a(i) \neq y_i$ as long as it is degree t.

An important property of $e(i)$ is that

$$y_i e(i) = f_a(i)e(i), \quad 1 \leq i \leq n. \tag{2.61}$$

Now, if $h(x)$ is defined as the $2t$ degree polynomial $h(x) = f_a(x)e(x)$, it is seen that

$$y_i e(i) = h(i), \quad 1 \leq i \leq n. \tag{2.62}$$

This means that the coefficients of $h(x)$ and $e(x)$ can be determined by solving the following n equations in $3t + 1$ unknowns

$$y_1 \overbrace{(e_0 + e_1 + \ldots + 1^t)}^{e(1)} = \overbrace{h_0 + h_1 + \ldots + h_{2t}1^{2t}}^{h(1)} \tag{2.63}$$

$$\vdots$$

$$y_n \overbrace{(e_0 + e_1 n + \ldots + n^t)}^{e(n)} = \overbrace{h_0 + h_1 n + \ldots + h_{2t}n^{2t}}^{h(n)}, \tag{2.64}$$

where $e_t = 1$. This system can be solved, since if it were so that $y_1, \ldots, y_t$ were erroneous, then $f_a(x)$ can be calculated based on interpolation of the correct points, $e(x)$ can be stated as

$$e(x) = (x-1)\cdots(x-t), \tag{2.65}$$

and $h(x)$ can be calculated from $f_a(x)$ and $e(x)$. In this way, two polynomials, $e(x)$ and $h(x)$, that satisfy **Equation (2.62)** exist. In a similar way, two polynomials, $e(x)$ and $h(x)$, must exist when it is unknown which t of the y_i values are erroneous.

In fact, there may be several $e(x)$ and $h(x)$ polynomials that satisfies **Equation (2.62)** and thus it cannot be guaranteed that $e(x)$ and $h(x)$ are unique. Nonetheless, it can be guaranteed that $f_a(x)$ is unique. This result is stated in **Lemma 2.2**.

Lemma 2.2
Let $(e_1(x)), h_1(x))$ and $(e_2(x), h_2(x))$ be two distinct pairs of polynomials that both satisfy **Equation (2.62)** for a known set of points $(x_1, y_1), \ldots, (x_n, y_n)$. Furthermore, $e_1(x)$ and $e_2(x)$ are of degree t, $h_1(x)$ and $h_2(x)$ are of degree $2t$ and $3t+1 \leq n$. Then it holds that

$$\frac{h_1(x)}{e_1(x)} = \frac{h_2(x)}{e_2(x)}. \tag{2.66}$$

Proof. In the sequel, it is shown that

$$h_1(x)e_2(x) = h_2(x)e_1(x), \tag{2.67}$$

holds. Note that, $h_1(x)e_2(x)$ and $h_2(x)e_1(x)$ are polynomials of degree at most $3t$. By subtracting these polynomials, a new polynomial of degree at most $3t$ is formed as

$$R(x) = h_1(x)e_2(x) - h_2(x)e_1(x). \tag{2.68}$$

Now, from **Equation (2.62)**, the following holds

$$y_i e_1(x_i) = h_1(x_i) \quad \text{and} \quad y_i e_2(x_i) = h_2(x_i), \quad \forall\, i = 1, \ldots, n. \tag{2.69}$$

Using this in **Equation (2.68)**, with $1 \leq i \leq n$, yields

$$\begin{aligned} R(x_i) &= (y_i e_1(x_i))\, e_2(x_i) - (y_i e_2(x_i))\, e_1(x) && (2.70)\\ &= 0 && (2.71) \end{aligned}$$

This shows that $R(x)$ has at least n roots. Since $3t < n$, $R(x)$ has more zeros than its degree and thus it must be the zero polynomial. This concludes the proof. □

The algorithm is known as the Berlekamp-Welch algorithm and is written formally in **Algorithm 2.1**.

Algorithm 2.1 (Berlekamp-Welch Algorithm)
Let $f(x)$ be the degree t polynomial, that is to be reconstructed using n known but possibly erroneous points. Denote the points as $(x_1, y_1), \ldots, (x_n, y_n)$ where $x_i \neq x_j$ for $i \neq j$, and furthermore let $f(x_i) \neq y_i$ for at most $t \leq \frac{n-1}{3}$ values of i. Note that previously it was assumed that $x_1 = 1, \ldots, x_n = n$, however, generally this does not have to be the case.

1. Define the vectors
$$\boldsymbol{x} = \begin{bmatrix} e_0 \\ e_1 \\ \vdots \\ e_{t-1} \\ h_0 \\ h_1 \\ \vdots \\ h_{2t} \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} -y_1 x_1^t \\ -y_2 x_2^t \\ \vdots \\ -y_n x_n^t \end{bmatrix}, \tag{2.72}$$
where $e_0, \ldots, e_{t-1}, h_0, \ldots, h_{2t}$ are unknowns, and the matrix
$$\boldsymbol{A} = \begin{bmatrix} y_1 & y_1 x_1 & y_1 x_1^2 & \cdots & y_1 x_1^{t-1} & -1 & -x_1 & -x_1^2 & \cdots & -x_1^{2t} \\ y_2 & y_2 x_2 & y_2 x_2^2 & \cdots & y_2 x_2^{t-1} & -1 & -x_2 & -x_2^2 & \cdots & -x_2^{2t} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ y_n & y_n x_n & y_n x_n^2 & \cdots & y_n x_n^{t-1} & -1 & -x_n & -x_n^2 & \cdots & -x_n^{2t} \end{bmatrix}. \tag{2.73}$$
2. Solve
$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{2.74}$$
for $\boldsymbol{x}$.
3. Construct $e(x)$ as the polynomial
$$e(x) = e_0 + e_1 x + \cdots + x^t, \tag{2.75}$$
and $h(x)$ as the polynomial
$$h(x) = h_0 + h_1 x + \cdots + h_{2t} x^{2t}. \tag{2.76}$$
4. $f(x)$ is given as
$$f(x) = \frac{h(x)}{e(x)}. \tag{2.77}$$

The algorithm is from [Smart, 2016, p. 411].

This section has provided an algorithm to reconstruct the correct secret in the output-reconstruct phase of a secure protocol, even if active adversaries has corrupted up to one third of the parties. Sometimes it is necessary to verify shares during protocol execution, without having to open secrets. The following section introduces a way to do this.

2.4.2 Verification of Shares

In this section, the method of verification of shares is introduced. To do this, the general protocol used in passively secure protocols, **Protocol 2.8**, is adjusted to included verification of shares whenever needed. To see the proofs of security of the protocols in this section refer to [Beerliová-Trubíniová and Hirt, 2008].

Recall that **Protocol 2.8** consists of an input-sharing phase, a computation phase, and an output-reconstruction phase. Here, a preprocessing phase is added in the beginning. The preprocessing phase is in [Beerliová-Trubíniová and Hirt, 2008] referred to as a preparation phase, thus this section will follow the terminology of the source and use the word preparation phase.

The following gives an overview of the phases and states where verification of messages is needed.

1. **Preparation phase:** This phase consists of the creation of all the Beaver's triplets needed for protocol execution. Recall that using Beaver's trick to create a Beaver's triplet, every party must choose a random number, which they distribute. Since corrupted parties can deviate from the protocol, it is necessary to ensure that the values are random and that they have been distributed correctly.

2. **Input-sharing phase:** In this phase each party holding an input distributes it. No message verification is needed in this phase.

3. **Computation phase:**
   Linear function: Each party applies the linear function on their shares. No messages are sent and therefore no verification is required.
   Multiplication: Use **Protocol 2.12**. The values d and e are opened, thus verification of shares is necessary.

4. **Output-reconstruction phase:** In this phase the parties broadcast the values needed to reconstruct the output value. It is thus necessary to verify the shares.

Assume throughout the section that $t < \frac{n}{3}$, where n is the number of participating parties and t is the number of corrupted parties. First, consider how the preparation phase can be constructed to ensure correct creation of Beaver's triplets.

Preparation phase

As described, the goal of the preparation phase is to create a number of Beaver's triplets. The first issue is to be able to create a random number, which is unknown to all parties.

Imagine that Beaver's trick is used. Since the adversary is assumed to be active, it is possible that corrupted parties have not chosen a random number and that they have not distributed the number with a polynomial of the correct degree. To overcome these two issues, it turns out that so-called *hyper-invertible matrices* can be of help.

Definition 2.11 (Hyper-invertible Matrices)
A $r \times c$ matrix $\boldsymbol{M}$, of which every square sub-matrix is invertible, is called a hyper-invertible matrix. Specifically, let $R \subseteq \{1, \ldots, r\}$ and $C \subseteq \{1, \ldots, c\}$ with $|R| = |C| > 0$ and let $\boldsymbol{M}_R$ be the matrix consisting of the rows $i \in R$ of $\boldsymbol{M}$ and $\boldsymbol{M}^C$ be the matrix consisting of the columns $j \in C$ of $\boldsymbol{M}$. $\boldsymbol{M}$ is hyper-invertible, if for any R, C as defined above the matrix $(\boldsymbol{M}_R)^C$ is invertible. [Beerliová-Trubíniová and Hirt, 2008].

It is not necessary to know how hyper-invertible matrices can be created in order to understand how they are used to generate shares of random numbers, however a way to do so is given in **Appendix A**.

Two useful properties of hyper-invertible matrices are given in the following lemmas.

Lemma 2.3
Let $\boldsymbol{M}$ be a $n \times n$ matrix, that is hyper-invertible and let $[y_1, \ldots, y_n] = \boldsymbol{M}[x_1, \ldots, x_n]$. For any $A, B \subset \{1, \ldots, n\}$ with $|A| + |B| = n$, there exist an invertible map $f : {\mathbb{F}_p}^n \to {\mathbb{F}_p}^n$, that maps the values $\{x_i\}_{i \in A}$, $\{y_i\}_{i \in B}$ to the values $\{x_i\}_{i \in \bar{A}}$, $\{y_i\}_{i \in \bar{B}}$. [Beerliová-Trubíniová and Hirt, 2008].

Proof. Let $\boldsymbol{y} = \boldsymbol{M}\boldsymbol{x}$ and $\boldsymbol{y}_B = \boldsymbol{M}_B\boldsymbol{x} = \boldsymbol{M}_B^A\boldsymbol{x}_A + M_B^{\bar{A}}\boldsymbol{x}_{\bar{A}}$. Since $\boldsymbol{M}$ is hyper-invertible, $\boldsymbol{x}_{\bar{A}} = (\boldsymbol{M}_B^{\bar{A}})^{-1}(\boldsymbol{y}_B - \boldsymbol{M}_B^A\boldsymbol{x}_A)$. $\boldsymbol{y}_{\bar{B}}$ is computed similarly.

□

Lemma 2.4
Let $\boldsymbol{M}$ be a $n \times n$ matrix, that is hyper-invertible and let $\phi : {\mathbb{F}_p}^n \to {\mathbb{F}_p}^n$ be the linear map induced by $\boldsymbol{M}$. If k of the input coordinates are fixed to arbitrary values, then the affine map ϕ' induced by ϕ from the remaining $n - k$ input coordinates to any $n - k$ output coordinates, is a bijection.

Proof. The $n - k$ input coordinates corresponds to a subset, C, of the columns of $\boldsymbol{M}$, and the $n - k$ output coordinates corresponds to a subset, R, of the rows of $\boldsymbol{M}$. Let

$\boldsymbol{z} \in {\mathbb{F}_p}^{n-k}$ be an input vector to ϕ' and consider $\boldsymbol{x} \in {\mathbb{F}_p}^n$, where $\boldsymbol{x}_C = \boldsymbol{z}$ and $\boldsymbol{x}_{\bar{C}}$ consists of the k fixed coordinates. Then

$$\phi'(\boldsymbol{z}) = \boldsymbol{M}_R \boldsymbol{x} = \boldsymbol{M}_R^C \boldsymbol{x}_C + \boldsymbol{M}_R^{\bar{C}} \boldsymbol{x}_{\bar{C}}. \tag{2.78}$$

It is easy to see that ϕ' is a bijection, since $\boldsymbol{M}_R^{\bar{C}} \boldsymbol{x}_{\bar{C}}$ is fixed and $\boldsymbol{M}_R^C$ is invertible. □

A hyper-invertible matrix is then used in the following way. First, all parties agree on a hyper-invertible matrix $\boldsymbol{M}$, which is available to all parties. Then, each party, P_i, generates and distributes a random value s_i. Note that it is not assumed that all parties are honest, thus some s_i values may be erroneous in some way. The properties of $\boldsymbol{M}$, allows the parties to check some shares for consistency, while keeping the remaining shares random and unknown. How a consistency-check is done, is stated in **Definition 2.12**.

Definition 2.12 (Consistency-check)
When a party, P_i, does a *consistency-check* of $[s]_t$, it means that P_i receives all shares of $[s]_t$ and checks that all shares lie on a degree t polynomial. If this is the case, $[s]_t$ is *consistent*, if not P_i declares failure.

Protocol 2.13 states the protocol for producing $n - 2t$ random numbers and afterwards it is shown that the protocol produces consistent sharings, and that the produced numbers are random and unknown to the adversary.

Protocol 2.13 (Generate Random Numbers)
Given a hyper-invertible matrix $\boldsymbol{M}$ that is publicly known.
Outputs $[r_1]_t, \ldots, [r_{n-2t}]_t$, where $r_1, \ldots, r_{n-2t}$ er random, unknown numbers.

1. Every party P_i, $i = 1, \ldots, n$, chooses a random value s_i and distributes $[s_i]_t$.
2. The players compute
$$([r_1]_t, \ldots, [r_n]_t) = \boldsymbol{M}([s_1]_t, \ldots, [s_n]_t). \tag{2.79}$$
3. For $i = n - 2t + 1, \ldots, n$, every party P_j, $j = 1, \ldots, n$, sends their share of $[r_i]_t$ to P_i. P_i then checks that $[r_i]_t$ is consistent, and if not, P_i declares failure.
4. If no party has declared failure, the $n - 2t$ sharings $[r_1]_t, \ldots, [r_{n-2t}]_t$ are output of the protocol.

The protocol is from [Beerliová-Trubíniová and Hirt, 2008].

Lemma 2.5
If $3t < n$, **Protocol 2.13** outputs $n - 2t$ consistent sharings when up to t out of n parties are corrupted by an active adversary, [Beerliová-Trubíniová and Hirt, 2008].

Proof. In step 3. of **Protocol 2.13**, $2t$ of the $[r_i]_t$ sharings are checked for consistency by $2t$ distinct parties. Note that, t of the parties checking a sharing can be corrupted. However, if all parties having checked a sharing for consistency, says "accept", then out of the $2n$ sharings $[s_1]_t, \ldots, [s_n]_t, [r_1]_t, \ldots, [r_n]_t$, at least n must be consistent. Namely, the $n - t$ sharings inputted by honest parties, and the t sharings checked by honest parties. In this way, according to **Lemma 2.3**, all $2n$ sharings must be consistent, since the remaining sharings can be computed from the consistent ones. □

Lemma 2.6
Even if an adversary has corrupted up to t out of n parties, where $3t < n$, **Protocol 2.13** outputs $n - 2t$ sharings of numbers which are random and unknown to the adversary, [Beerliová-Trubíniová and Hirt, 2008].

Proof. To see that the outputted sharings are unknown to the adversary, note that the adversary knows t of the input sharings, s_k, namely those given by corrupt players, and it also knows at most t output sharings, r_k, namely those reconstructed by corrupted players. According to **Lemma 2.3**, a total of n input/output values are needed to construct the remaining n input/output values. Since $2t \leq n$, the adversary cannot construct the input/output values that it does not know.

To see that the outputted sharings are random, note that according to **Lemma 2.4**, when fixing the $2t$ sharings known by the adversary, there exist a bijective mapping from any other $n - 2t$ sharings inputted by honest players to the outputted sharings. Since the honest players have chosen random numbers, the bijective mapping ensures that the outputted sharings are also random. □

The parties can now generate random numbers, and thus they can generate the values a and b of a Beaver's triplet. Consider now how the parties can compute the last value $c = ab$. As they did in the case of a passive adversary, the parties can use **Protocol 2.10**. However, this protocol involves that each party distributes a value, and thus this distribution must be validated. This can be done by using a reconstruction protocol, which includes verification of shares.

Protocol 2.14 (Reconstruction)
Given P_R, which is the party allowed to reconstruct $[s]_t$.
Outputs s only for P_R to know.

1. Every party P_i sends their share of $[s]_t$ to P_R.
2. In the case that there exists a degree t polynomial f such that all shares lie on it, P_R computes $s = f(0)$. Otherwise P_R declares failure.

The protocol is from [Beerliová-Trubíniová and Hirt, 2008].

Now a protocol for the generation of Beaver's triplets can be stated.

Protocol 2.15 (Generate Triplet)
Given $[a]_t, [b]_t, [r]_t, [r]_{2t}$, where a, b and r are random numbers and $2t < n$.
Outputs $([a]_t, [b]_t, [c]_t)$, where $c = ab$.

1. The parties compute $[c]_{2t} = [a]_t[b]_t$.
2. The parties compute $[q]_{2t} = [c]_{2t} - [r]_{2t}$.
3. Use **Protocol 2.14** towards all parties to reconstruct q publicly.
4. The parties compute $[c]_t = [r]_t + q$.
5. The output of the protocol is $([a]_t, [b]_t, [c]_t)$.

The protocol is from [Beerliová-Trubíniová and Hirt, 2008].

Remark, it must also be checked that the sharings $[r]_t$ and $[r]_{2t}$, indeed reconstructs the same value r. This can be achieved by modifying **Protocol 2.13**, like it is done in [Beerliová-Trubíniová and Hirt, 2008, p. 10]. Here, it is preferred to keep things as simple as possible, thus the modification is not implemented.

Yet, it has not been explained what to do when a party has declared failure in any of the protocols presented in this section. To this report, it is not of great importance *how* to handle this in practice, it is only important that there *is* a way to handle it. In [Beerliová-Trubíniová and Hirt, 2008], a technique called player-elimination is used to eliminate parties that are assumed to be corrupt. This method is here described concisely and the interested reader can read more in [Beerliová-Trubíniová and Hirt, 2008, p. 12].

Basically, when a party has declared failure, it must be determined which party caused the failure, and then this party is eliminated from any further participation. To keep of track of declared failures, each party is equipped with a bit that tells if the party is either *happy* or *unhappy*. For instance, a party declares failure by setting his happy-bit to unhappy. To find out whether any failures has happened, a so-called *consensus-protocol* is used. This protocol allows all parties, P_i, holding a value x_i, to reach an agreement on a value x if $x_i = x \; \forall \, i$.

The preparation-phase, as stated in [Beerliová-Trubíniová and Hirt, 2008, p.12], is given in **Protocol 2.16**.

Protocol 2.16 (Preparation)
Given L, which is the number of Beaver's triplets needed for the computation phase. Let a *segment* denote the generation of one Beaver's triplet.
Outputs L Beavers triplets.

1. For each segment $k = 1, \ldots, L$ do:
   (a) Every party sets their happy-bit to happy.
   (b) Triplet Generation: **Protocol 2.15** is invoked to generate a Beaver's triplet.
   (c) Fault Detection: The parties must reach agreement on whether or not at least one party is unhappy:
      i. Every party P_i broadcasts their happy-bit. If party P_i receives at least one unhappy-bit, P_i gets unhappy.
      ii. The parties run a consensus protocol on their happy-bits. If they are all happy, the generated triplet are correct and the segment is finished. Otherwise, proceed to the following step.
   (d) Fault Localization: Localize E as a subset of the parties, with $|E| = 2$ and at least one party being corrupt:
      i. Choose a party, P_r, as referee.
      ii. Every party, P_i, sends everything they have received and every random number they have chosen during the actual segment to P_r.
      iii. P_r reproduces every message, x, that should have been sent to P_i from P_j and checks if this is in agreement with the message, x', that P_i claims to have received from P_j. If $x \neq x'$, P_r broadcasts (l, i, j, x, x'), where l is the index of the message.
      iv. The accused parties broadcasts whether they agree with P_r. If P_i disagrees, $E = \{P_r, P_i\}$, if P_j disagrees, $E = \{P_r, P_j\}$. Otherwise, $E = \{P_i, P_j\}$
   (e) Player Elimination: Eliminate the parties in E from the protocol, set $n = n - 2$, $t = t - 1$, and repeat the segment.

The protocol is from [Beerliová-Trubíniová and Hirt, 2008].

The preparation-phase consists of many computations, and thus it may be time consuming. However, remark that this phase can be completed before the parties has even decided on an input to the actual computation.

Computation phase

Consider now the computation-phase. As seen in the beginning of this section, there is no verification of shares for an addition protocol. Thus, solely a protocol for multiplication is presented, see **Protocol 2.17**. This protocol is very similar to **Protocol 2.12**, the only difference being that broadcasted values needs to be verified in **Protocol 2.17**.

Protocol 2.17 (Computation phase: Multiplication)
Given that the parties hold $[a]_t$ and $[b]_t$ and a Beaver's triplet $([\alpha]_t, [\beta]_t, [\gamma]_t)$.
Outputs $[y]_t$, $y = ab$.

1. The parties computes $[d]_t = [a]_t - [\alpha]_t$ and $[e]_t = [b]_t - [\beta]_t$.
2. Invoke **Protocol 2.14** towards all parties to publicly reconstruct d and e.
3. The parties compute

$$[y]_t = de + d[\beta]_t + e[\alpha]_t + [\gamma]_t. \tag{2.80}$$

The protocol is from [Beerliová-Trubíniová and Hirt, 2008].

Output-reconstruction phase

The last phase to consider is output-reconstruction. However, the protocol to use has already been introduced as **Protocol 2.14**. It should be noted that when a value is opened, as y for instance is in the reconstruction phase, all parties can broadcast their share of y and each party uses **Protocol 2.14** on their received shares. This is more efficient than if each party secretly sends their share to each of the other parties.

2.5 Summary

In this chapter an introduction to secure MPC has been presented. As described, the objective of secure MPC is to create secure protocols that allow a set of parties to calculate a function of each of their individual input, without them having to reveal their input. Secure MPC also takes into account that adversaries may attempt to attack these protocols. There is distinguished between a passive adversary and an active one. The former follows the protocol, but attempts to obtain information any way he can, while the latter is assumed to also deviate from the protocol.

The building block of the secure protocols presented in this report is *secret sharing schemes*. A secret sharing scheme is a method for a party to share a secret value with other parties, without actually revealing what the secret is. There exists different schemes for this, in this report solely two has been presented. One is the additive secret sharing scheme, which creates shares of a secret in such a way that the sum of all

shares equals the secret. The second one is Shamir's secret sharing scheme, which uses a polynomial of degree t to create the shares. In this way only $t+1$ shares are necessary to reconstruct the secret, which can be done by using Lagrange interpolation.

Passively secure protocols for addition and multiplication were stated and two different approaches to deal with active adversaries was presented.

Part II

Application

Prerequisites

The remaining part of this thesis is concerned with using the gained knowledge about secure MPC to convert simple known algorithms to secure MPC protocols. The goal is to understand the implications and pitfalls of doing this conversion, and to compare the results gained from a secure implementation to a non-secure implementation of the algorithms.

For the protocols stated in this part of the report, it is assumed that Shamir's secret sharing scheme is used. However, it is actually not a must, since any secret sharing scheme with secure addition and multiplication can be used. To underline this fact, the notation of the sharings of a secret x, which is $[x]_t$, will from now be denoted as $[x]$.

The following still holds for two secrets a, b:

$$[a] + [b] = [a + b], \tag{2.81}$$

$$k[a] = [ka], \tag{2.82}$$

$$[a][b] = [ab]. \tag{2.83}$$

Unless otherwise stated, all arithmetic in the rest of the report is finite field arithmetic. Furthermore, the adversary in the protocols stated in the remaining chapters is assumed to be passive.

3

Gradient Descent as a Secure MPC Protocol

The objective in this chapter, is to get some experience in converting an algorithm to a secure protocol and learn what some of the challenges might be. Thus, accuracy of the protocol is not the main concern, even though the matter is discussed at the end. It is desired to keep things simple, such that it is not the algorithm itself that causes difficulties, rather the focus is on understanding the implications of the secure MPC setup. At the same time it is desired to investigate algorithms that is of interest to the field of control, thus the optimization algorithm *gradient descent* is chosen. For the sake of keeping things simple, a quadratic function is chosen as cost function for the optimization.

Section 3.1 provides a short introduction to the method of gradient descent. In **Section 3.2** it is discussed how the gradient descent method could be converted to a secure MPC protocol. **Section 3.3** provides simulations of both the gradient descent algorithm and the secure gradient descent protocol. Finally, **Section 3.4** provides a summery of the chapter.

The reader is assumed to be familiar with the method of gradient descent, thus the following method is meant as a recap.

3.1 Method of Gradient Descent

Suppose that it is desired to minimize some convex cost function, $f : \mathbb{R}^m \to \mathbb{R}$. By assuming the function is convex any local minimum is the global minimum. More precisely, there exist $\boldsymbol{x}^*$, for which it holds that

$$f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}) \, \forall \, \boldsymbol{x} \in \mathbb{R}^m. \tag{3.1}$$

The minimum can be found by an iterative process of updating a point $\boldsymbol{x}_k \in \mathbb{R}^m$, at each iteration k, by moving it a step in the direction of the negative gradient. At each iteration it holds that $f(\boldsymbol{x}_k) < f(\boldsymbol{x}_{k-1})$ until $\boldsymbol{x}_k = \boldsymbol{x}^*$. The size of the step can be either fixed or at each iteration obtained by *a line search method*. The interested reader is referred to [Boyd and Vandenberghe, 2004, pp. 464 - 466] to learn about line search methods and for a more thorough review on the method of gradient descent. Here it is sufficient to state the method formally, see **Algorithm 3.1**.

Algorithm 3.1 (Gradient Descent)
Input: A starting point $\boldsymbol{x}_0 \in \mathbb{R}^m$.

1. For $k = 0, \ldots,$ till stopping criterion is reached, do:
   (a) $\Delta\boldsymbol{x} = -\nabla f(\boldsymbol{x}_k)$,
   (b) Either use $\gamma = C > 1$, where $C \in \mathbb{R}$ is a fixed constant, or obtain γ through a line search method.
   (c) $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \frac{1}{\gamma}\Delta\boldsymbol{x}$,

[Boyd and Vandenberghe, 2004, p. 466].

The stopping criterion in **Algorithm 3.1** can either be that a fixed number of iterations is reached, that the difference between $\boldsymbol{x}_k$ and $\boldsymbol{x}_{k-1}$ is below some threshold, or that the gradient of $f(\boldsymbol{x}_k)$ is below some threshold.

One line search method, which is quite efficient, while at the same time being very simple is *backtracking line search*. It works in the way that it starts with a somewhat large estimate of the step size, and then reduces it iteratively until a satisfying reduction of the cost function is achieved. The backtracking line search is formally introduced in **Algorithm 3.2**.

Algorithm 3.2 (Backtracking Line Search)
Input: a descent direction Δx for $f(x)$, $\alpha \in (0, 0.5)$ and $\beta \in (0, 1)$.

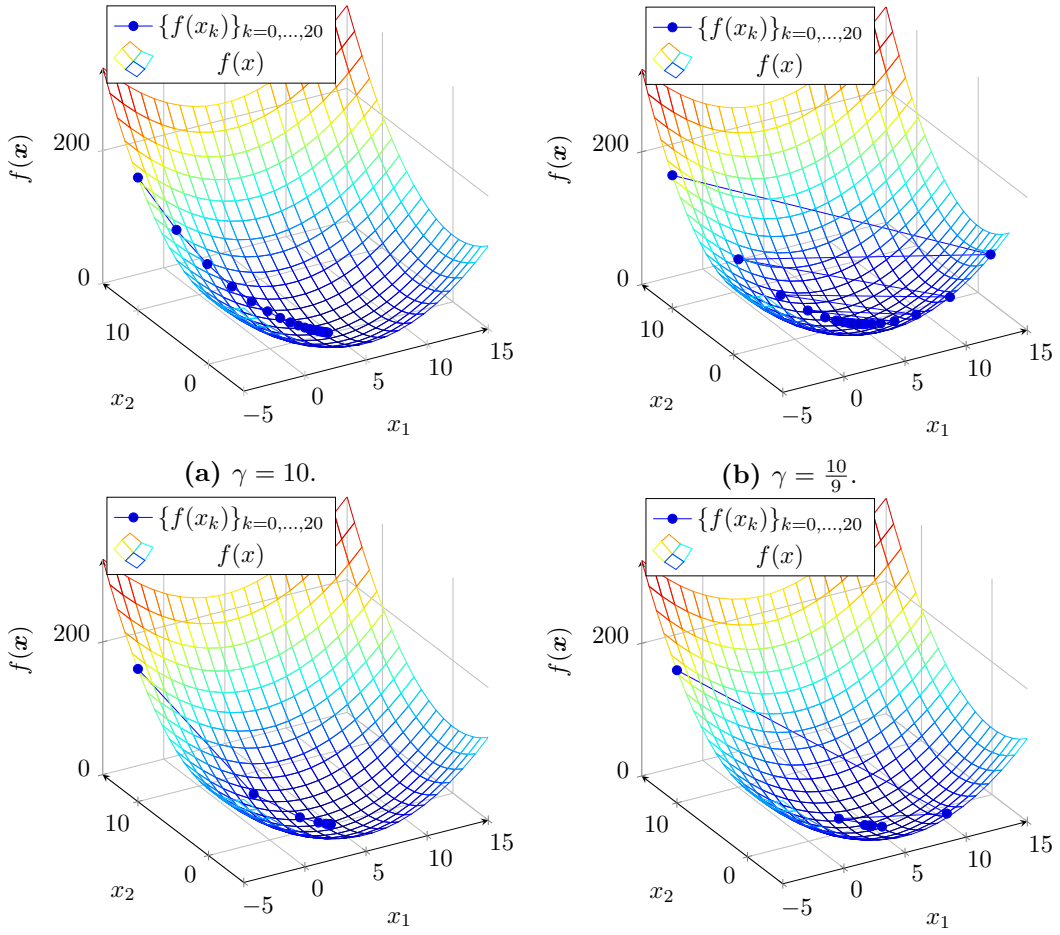
1. Initialize $\gamma = 1$.
2. while $f(\boldsymbol{x}) + \alpha\frac{1}{\gamma}\nabla f(\boldsymbol{x})^\top \Delta\boldsymbol{x} < f(\boldsymbol{x} + \frac{1}{\gamma}\Delta\boldsymbol{x})$:
   (a) $\gamma = \beta + \gamma$,

[Boyd and Vandenberghe, 2004, p. 464]

As an example, consider the cost function, $f : \mathbb{R}^2 \to \mathbb{R}$, given as

$$f(\boldsymbol{x}) \;=\; (\boldsymbol{x} + [-5, 0]^\top)^\top (\boldsymbol{x} \;+\; [-5, 0]^\top), \tag{3.2}$$

which has optimum value $\boldsymbol{x}^* = [5, 0]^\top$. **Figure 3.1 (a), (b), (c)** and **(d)** depicts $f(\boldsymbol{x})$, along with the decreasing sequence of $\{f(\boldsymbol{x}_k)\}_{k=0,\dots,20}$, where the sequence $\{\boldsymbol{x}_k\}_{k=0,\dots,20}$ is obtained using **Algorithm 3.1**. **Figure 3.1 (a)** and **(b)** use a fixed, but distinct γ, whereas **(c)** and **(d)** use backtracking line search both with $\alpha = 0.1$, but distinct β. For all figures $x_0 = 20$.



(a) $\gamma = 10$.

(b) $\gamma = \frac{10}{9}$.

(c) $\beta = 0.3$.

(d) $\beta = 0.7$.

Figure 3.1: Illustration of **Algorithm 3.1** for finding the minimum of $f(\boldsymbol{x}) = (\boldsymbol{x} + [-5, 0]^\top)^\top(\boldsymbol{x} + [-5, 0]^\top)$, which has optimum value $\boldsymbol{x}^* = [5, 0]^\top$. The values $x_0 = 20$ are chosen for all figures. **(a)** and **(b)** use a fixed γ, with different values, while **(c)** and **(d)** use backtracking line search both with $\alpha = 0.1$, but with different β.

Now, the challenge turns up, if it is desired to keep the cost function, $f(\boldsymbol{x})$, secret and calculate the minimum privately. The next section discusses how to convert **Algorithm 3.1** to a secure MPC protocol.

3.2 Converting the Gradient Descent Algorithm to a Secure MPC Protocol

In this section the goal is to create a secure MPC protocol for calculating the gradient descent solution to the problem of minimizing a secret function $f(\boldsymbol{x})$. To restrict the problem, it is assumed that $f(\boldsymbol{x})$ is a quadratic function. The first question that arises is how to keep a function secret. In the scope of this chapter, it is assumed that the form of the function is public, but coefficients are secret. This means that, assuming $f(\boldsymbol{x})$ is differentiable and that its gradient can be found analytically, the form of the gradient is public as well, which again in many cases entails that the parties can compute the optimal value, $\boldsymbol{x}^*$, from the secret coefficients of the function. However, as the purpose of this chapter is to gain knowledge on creating secure protocols, assume that the parties must estimate both the gradient and the optimal value, where the latter is achieved using a secure implementation of the gradient descent algorithm.

To convert **Algorithm 3.1** to a secure MPC protocol, the steps **(a),(b),(c)** of **Algorithm 3.1** are discussed in order to find a secure MPC equivalence. In the first step, **(a)**, the descent direction is assigned to be the negative gradient of $f(\boldsymbol{x})$ evaluated in the k'th estimate of $\boldsymbol{x}^*$. This step can be achieved securely, by approximating the gradient using the differential quotient, such that

$$\widehat{\nabla f(\boldsymbol{x}_k)} = \left[\frac{f(\boldsymbol{x} + [h, 0, \ldots, 0]^\top) - f(\boldsymbol{x})}{h}, \ldots, \frac{f(\boldsymbol{x} + [0, \ldots, 0, h]^\top) - f(\boldsymbol{x})}{h} \right]^\top, \tag{3.3}$$

where $h \in \mathbb{R}$ is strictly positive. The estimate of $\nabla f(x)$ improves in accuracy as $h \to 0$.

The obvious problem with this approach is that in the secure protocol $h \notin \mathbb{R}$, but rather $h \in \mathbb{F}_p$. Thus, h can only be as low as one. What impact this requirement will have on the final result will be discussed later.

The second step, **(b)**, involves either letting the step size, γ, be a fixed constant, C, or using a line search method to obtain it. By letting $C \in \mathbb{F}_p$ be a positive integer the former choice is easy to perform in a secure protocol. For the later choice, backtracking line search can be used. As seen in **Algorithm 3.2**, this method requires two operations not yet considered in this report, namely comparison and division.

The comparison operation is tricky to convert to a secure MPC protocol. The reason is that a finite field is not an ordered field. To see this, consider the following relations between field elements of a finite field $\mathbb{F}_p$:

$$0 < 1 + 1 < \cdots < \underbrace{1 + \ldots + 1}_{p \text{ times}} = 0. \tag{3.4}$$

However, what can also be seen from **Equation (3.4)**, is that it is possible to compare two elements of a finite field as long as none of the elements has made a so-called *wrap-around*. To see what is meant by a wrap-around, consider the example where a value $a \in \mathbb{F}_p$ is increased but the outcome is less than a. For instance, $(a + r) \mod p < a$ or $(ar) \mod p < a$ for $r \in \mathbb{F}_p$. This, is in the report referred to as a wrap-around p.

Similarly, a value $a \in \mathbb{F}_p$ has wrapped-around zero if $(a - r) \mod p > a$. How to handle these wrap-arounds, when it is desired to do comparison is discussed later.

Something similar is the case for division. First of all, it is important to note that division in the finite field $\mathbb{F}_p$ does not produce even nearly the same result as division in $\mathbb{R}$. To see this, consider the inverse of 5, which in $\mathbb{R}$ is 0.2. Considering the finite field $\mathbb{F}_7$, the inverse of 5 can be found by finding a field element, e, such that

$$5e \bmod 7 = 1. \tag{3.5}$$

In this way, it is seen that the inverse of $5 \in \mathbb{F}_7$ is 3. Similarly, considering $\mathbb{F}_{11}$, $5^{-1} \mod 11 = 9$. Hence, division in a finite field has little to do with division in $\mathbb{R}$. If results produced by a secure implementation of **Algorithm 3.1**, should be comparable to results produced by **Algorithm 3.1**, division cannot be carried out in the finite field. Neither can the division be carried out in $\mathbb{R}$, since this possibly will produce a non-integer result, which can then not be represented in $\mathbb{F}_p$ for further calculations. Here, this problem is solved by approximating the division with integer division, meaning that an integer result is guaranteed. In some cases it may be necessary to introduce scaling to keep the error, introduced by the approximation, from being too large. Note that, similarly to comparison, when doing integer division in a secure MPC protocol the result will not be as expected if a wrap-around has occurred for either the dividend or divisor. How to handle this is discussed later in the report. Finally, it should be noted that integer division by a public constant is much less complicated than division by a secret. For this reason, it is assumed that γ is publicly known.

The final step **(c)** also relies on division, but besides of that it does not need any protocol not already presented in this report. The following two sections discusses the creation of secure MPC protocols for comparison and integer division by a public constant, respectively.

3.2.1 Secure MPC: Comparison

The objective is now to create a secure MPC protocol for comparison, or more accurately a "less-than" protocol. The ideas in the work [Damgård et al., 2006] are used and the reader is referred to this work for the proofs of the protocols presented in this section. The protocol takes inputs $a, b \in \mathbb{F}_p$ and outputs 1 if $a < b$ and 0 otherwise. State-of-the-art secure MPC protocols for comparison relies on bit decomposition. To see why it is useful to compare bit representations, assume that $a \neq b$ and let $a_0, \ldots, a_{l-1}$ be such that $a = \sum_{i=0}^{l-1} a_i 2^i$ and let $b_0, \ldots, b_{l-1}$ be such that $b = \sum_{i=0}^{l-1} b_i 2^i$. Now, if i_0 is the largest index i where it holds that $a_i \neq b_i$, then $a < b$ if and only if $b_{i_0} = 1$.

To determine if $a_i \neq b_i$, the bitwise operations exclusive OR (XOR), which is summation modulo 2, and prefix-OR (PRE-OR), is used. The PRE-OR of a bit-sequence $\boldsymbol{k}$ defines a new bit-sequence $\boldsymbol{k'}$, such that

$$k'_i = \vee_{j=i}^{l-1} k_j. \tag{3.6}$$

As seen by **Equation (3.6)**, the PRE-OR can be computed by using the bitwise OR-operator multiple times.

For this reason, this section will beside from introducing a secure comparison protocol, introduce secure sub-protocols for the XOR and OR operations. Before this is done, an arithmetic approach to evaluating the truth of $a < b$, is stated in **Algorithm 3.3**.

Algorithm 3.3 (Less-Than)
Input: $a, b \in \mathbb{F}_p$.

Output: $y = \begin{cases} 1 & \text{if } a < b \\ 0 & \text{otherwise} \end{cases}$

1. Define $(a_0, \ldots, a_{l-1})$ and $(b_0, \ldots, b_{l-1})$, where $a_i, b_i \in \{0, 1\}$ for $i = 0 \ldots, l-1$, such that $a = \sum_{i=}^{l-1} a_i 2^i$ and $b = \sum_{i=}^{l-1} b_i 2^i$.
2. Define the bit-sequence $(e_0, \ldots, e_{l-1})$, where $e_i = a_i \oplus b_i$.
3. Define the bit-sequence $f_0, \ldots, f_{l-1}$, where $f_i = \vee_{j=0}^{i} e_j$.
4. Define the bit-sequence $g_0, \ldots, g_{l-1}$, where $g_i = f_i - f_{i+1}$ for $i = 0 \ldots, l-2$ and $g_{l-1} = f_{l-1}$.
5. Define the bit-sequence $h_1, \ldots, h_{l-1}$, where $h_i = g_i b_i$.
6. $y = \sum_{i=0}^{l-1} h_i$.

This algorithm is written with inspiration from [Damgård et al., 2006].

A few arguments to the correctness of **Algorithm 3.3** is given. Suppose that it is desired to determine whether $a < b$, where $a \neq b$ for two integers a, b. Defining $a_0, \ldots, a_{l-1}$ and $b_0, \ldots, b_{l-1}$ such that $a = \sum_{i=0}^{l-1} a_i 2^i$ and $b = \sum_{i=0}^{l-1} b_i 2^i$, respectively. Then, recall that if i_0 is the largest index i where it holds that $a_i \neq b_i$, then $a < b$ if and only if $b_{i_0} = 1$. Thus, by producing a bit-sequence, $\boldsymbol{g} = (g_0, \ldots, g_{l-1})$, that is zero everywhere except for the i_0'th bit, then $\sum_{i=0}^{l-1} b_i g_i$ produces a 1 if $a < b$ and 0 otherwise, which is done in steps **5.** and **6.** in the algorithm.

Now, consider the creation of $\boldsymbol{g}$. $\boldsymbol{g}$ can be produced by first taking the XOR between the bit representations of a and b. This will produce a bit-sequence, $\boldsymbol{e} = (e_0, \ldots, e_{l-1})$ which is 1 in e_{i_0} and 0 in e_i for $i > i_0$.

To make $e_i = 0$ for $i < i_0$, involves calculating the PRE-OR of $\boldsymbol{e}$. This creates a bit-sequence $\boldsymbol{f} = (f_0, \ldots, f_{l-1})$, where $f_i = \vee_{j=1}^{l-1} e_j$. This means that $f_i = 0$ for $i > i_0$ and $f_i = 1$ for $i \leq i_0$. Then $\boldsymbol{g}$ is created by $g_i = f_i - f_{i+1}$ and $g_{l-1} = f_{l-1}$. In this way, $\boldsymbol{g}$ is a bit sequence, where $g_{i_0} = 1$ and $g_i = 0$ for $\forall i \neq i_0$.

To convert **Algorithm 3.3** to a secure MPC protocol, it is necessary to introduce secure protocols for the XOR and PRE-OR operations. Computing the XOR of two bits securely is not complicated and thus this protocol is stated without further introduction.

Protocol 3.1 (XOR)
Given that the parties hold $[a_1]$ and $[b_1]$, where $a_1, b_1 \in \{0, 1\}$.
Outputs $[y]$, where $y = a_1 \oplus b_1$, where $\oplus$ denotes the XOR operator.

1. The parties compute $[d] = [a_1] - [b_1]$.
2. The parties compute $[y] = [d][d]$.

Proof. Security of **Protocol 3.1**, follows from the fact that it relies solely on addition and multiplication, which both previously have been proved to be secure. That the protocol returns the correct result is trivial. □

Computing the PRE-OR of a bit-sequence securely can be done by using a secure implementation of the OR-operator on an input bit-sequence. The OR-operation of a bit-sequence, $(a_0 \ldots, a_{l-1})$, is 0 if $a_i = 0, \forall i$ and 1 if $a_i = 1$ for at least one $i = 0, \ldots, l-1$. There is a simple formula for this computation which is stated in **Protocol 3.2**.

Protocol 3.2 (OR)
Given That the parties hold $[a_0], \ldots, [a_{l-1}]$, where $(a_0, \ldots, a_{l-1})$ is a bit-sequence of l bits.
Outputs $[y]$, where $y = \vee_{i=0}^{l-1} a_i$.

1. The parties invoke **Protocol 2.10** to compute $[y] = 1 - \prod_{i=0}^{l-1}(1 - [a_i])$.

Proof. Security of **Protocol 3.1**, follows from the fact that it relies solely on secure protocols. Correctness of the protocol is trivial. □

Finally, the secure "less-than" protocol can be stated, but before doing so notation for a bitwise shared secret is introduced.

Definition 3.1
Let $a \in \mathbb{F}_p$ and let $a_0, \ldots, a_{l-1}$ be the bit representation of a, such that $a = \sum_{i=0}^{l-1} a_i 2^i$ and let $P_1 \ldots, P_n$ be parties. Saying that a is bitwise shared, noted $[a]^B$, means that the parties hold $[a_0], \ldots, [a_{l-1}]$.

Protocol 3.3 (Less-Than)
Given that the parties hold $[a]^B$ and $[b]^B$.
Outputs $[y]$, where $y = \begin{cases} 1 & \text{if } a < b \\ 0 & \text{otherwise} \end{cases}$.

1. For $i = 0, \ldots, l-1$ the parties invoke **Protocol 3.1** to compute $[e_i] = [a_i] \bigoplus [b_i]$.
2. The parties invoke **Protocol 3.2** multiple times to define $([f_{l-1}], \ldots, [f_0])$, where $[f_i] = \vee_{j=0}^{(l-1)-i} e_j$.
3. The parties define $[g_{l-1}] = [f_{l-1}]$.
4. For $i = 0, \ldots, l-2$ the parties compute $[g_i] = [f_i] - [f_{i+1}]$.
5. For $i = 0, \ldots, l-1$ the parties invoke **Protocol 2.10** to compute $[h_i] = [g_i][b_i]$.
6. The parties compute $[y] = \sum_{i=0}^{l-1} [h_i]$.

The protocol is from [Damgård et al., 2006, p. 297].

The proof of **Protocol 3.3** can be found in [Damgård et al., 2006].

As discussed, to do comparison it is required that the inputs are represented by bits. This means that to compare two secrets, a secure protocol for bit decomposition is a necessity. The following section introduces this protocol.

3.2.2 Secure MPC: Bit Decomposition

Decomposing a shared secret into sharings of its bits can be done by adding a random number, r, to the secret, a, such that a value, v, which can be opened is obtained. By bit-decomposing v and afterwards bit-wise subtracting r, a secret bit-decomposition of a is obtained. The requirement is that the parties can obtain sharings of a uniformly random number, $[r]$, as well as sharings of its bit-representation, $[r]^B$. This can be done by the parties first obtaining $[r]^B$ and afterwards calculates $[r]$. How to obtain a bitwise shared random number is described in [Cramer et al., 2015, p. 190]. For the rest of the report, it is assumed that the parties can obtain sharings $[r]$ and $[r]^B$ in a preprocessing phase.

Before formally stating a secure protocol for bit-decomposition, consider a secure protocol for binary subtraction. A binary subtraction is done by iterating through the bits starting from the least significant bit and at each calculation keeping track of the carry bit. Without further introduction the secure protocol for binary subtraction is stated in **Protocol 3.4**.

Protocol 3.4 (Binary Subtraction)
Given that the parties hold $[a]^B$ and $[b]^B$.
Outputs $[y]^B$, where $y = a - b$.

1. The parties invoke **Protocol 3.1** to compute $[y_0] = [a_0] \oplus [b_0]$.
2. The parties compute $[c] = (1 - [a_0])[b_0]$.

3. For $i = 1, \ldots, l$ the parties do
   (a) The parties compute $[c'] = [b_i][c]$.
   (b) The parties invoke **Protocol 3.1** to compute $[e] = [b_i] \oplus [c]$.
   (c) The parties invoke **Protocol 3.1** to compute $[y_i] = [a_i] \oplus [e]$.
   (d) The parties compute $[c] = [c'] + (1 - [a_i])[e]$.

Proof. **Protocol 3.4** is seen to be secure, since it relies solely on secure protocols. Correctness of the protocol is most easily seen by constructing truth tables. First consider the bit y_0, for which the following truth table can be constructed:

a_0	b_0	y_0	c
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

In the truth table, the different possibilities for the bits a_0 and b_0 are seen, as well as the corresponding results for the bit y_0 and the carry bit. It is now easy too see that the formulas

$$y_0 = a_0 \oplus b_0, \tag{3.7}$$
$$c = (1 - a_0)b_0, \tag{3.8}$$

gives the desired results. For the subsequent bits y_i for $i = 1, \ldots, l - 1$, the following truth table can be constructed:

a_i	b_i	c_{prev}	y_i	c_{next}
0	0	1	1	1
0	0	0	0	0
0	1	1	0	1
0	1	0	1	1
1	0	1	0	0
1	0	0	1	0
1	1	1	1	1
1	1	0	0	0

In the truth table the possibilities for the bits a_i, b_i and the carry bit from the previous calculations, c_{prev}, are seen. Furthermore, the results for the bit y_i and the carry bit for the next calculation, c_{next}, are stated. Again, it is now easily verified that the formulas

$$y_i = a_0 \oplus (b_0 \oplus c_{prev}), \tag{3.9}$$
$$c_{next} = b_i c_{prev} + (1 - a_i)(b_i \oplus c_{prev}), \tag{3.10}$$

gives the desired results. □

Now it is straight forward to state the secure protocol for bit-decomposition, see **Protocol 3.5**.

Protocol 3.5 (Bit-decomposition)
Given that the parties hold $[a]$, and $[r], [r]^B$, where $r \in \mathbb{F}_p$ is a l-bit uniformly random number and $[r], [r]^B$ are obtained in a preprocessing phase.
Outputs $[a]^B$.

1. The parties compute $[d] = [a] + [r]$, and opens d.
2. The parties compute the bitwise decomposition of d and distributes $[d]^B$.
3. The parties compute the bitwise decomposition of $d' = d + p$ and distributes $[d']^B$.
4. The parties invoke **Protocol 3.3** to securely compute $[b] = \begin{cases} [1] & \text{if } d < r \\ [0] & \text{otherwise} \end{cases}$, using $[d]^B$ and $[r]^B$.
5. The parties define $[s]^B$, by computing $[s_i] = [b][d'_i] + (1 - [b])[d_i]$ for $i = 1, \ldots l+1$.
6. The parties invoke **Protocol 3.4** to compute $[y]^B = [s]^B - [r]^B$.

The protocol is from [Cramer et al., 2015, p. 189].

Refer to [Cramer et al., 2015, p. 189], for a proof of **Protocol 3.5**.

To improve readability later in the report, a protocol that combine **Protocol 3.5** and **Protocol 3.3**, is introduced in **Protocol 3.6**.

Protocol 3.6 (Integer Comparison)
Given that the parties hold a and b.
Outputs $[y]$, where $y = \begin{cases} 1 & \text{if } a < b \\ 0 & \text{otherwise.} \end{cases}$.

1. The parties invoke **Protocol 3.5** twice to compute both $[a]^B$ and $[b]^B$ from $[a]$ and $[b]$, respectively.
2. The parties invoke **Protocol 3.3** to securely compute $[y] = \begin{cases} [1] & \text{if } a < b \\ [0] & \text{otherwise.} \end{cases}$, using $[a]^B$ and $[b]^B$.

3.2.3 Secure MPC: Division By Public Constant

Protocol 3.7 gives the protocol to divide a secret by a publicly known constant. That the protocol gives the desired result is trivial and security follows from the fact that only local computation and secure protocols are used.

Protocol 3.7 (Division By Public Constant)
Given that that parties hold $[q]$ and that k is a publicly known integer.
Outputs $[y]$, where $y = \lfloor qk^{-1} \rfloor$.

1. Party P_1, chooses a random value r and computes $r_\top = \lfloor rk^{-1} \rfloor$.
2. Party P_1 distributes both $[r]$ and $[r_\top]$.
3. The parties compute $[z] = [q] + [r]$.
4. The parties open $[z]$ only for party P_2 to see.
5. Party P_2 computes $z_\top = \lfloor zk^{-1} \rfloor$ and distributes $[z_\top]$.
6. The parties compute $[y] = [z_\top] - [r_\top]$.

The protocol is from [Dahl et al., 2012, p. 12].

Refer to [Dahl et al., 2012, p. 12] for a formal proof of **Protocol 3.7**.

Finally, the secure protocol for the gradient descent algorithm can be stated in the following section.

3.2.4 Secure MPC: Gradient Descent

Using the protocols stated in the preceding sections, a secure MPC protocol for the method of gradient descent can be stated. To improve readability, the protocol is stated in the two-dimensional case, meaning that the input to the cost function is a scalar rather than a vector.

Before stating the protocol, there is some unfinished business to attend to regarding the comparison and division needed in the secure protocol. As described previously, when doing comparison and division it is critical to handle wrap-arounds appropriately.

Consider first the comparison operation in **Algorithm 3.2**, restated here for convenience.

$$f(\boldsymbol{x}) + \alpha\frac{1}{\gamma}\nabla f(\boldsymbol{x})^\top \Delta\boldsymbol{x} < f(\boldsymbol{x} + \frac{1}{\gamma}\Delta\boldsymbol{x}). \tag{3.11}$$

Assuming that the function $f(x)$ is quadratic, the rightmost term of the inequality is always positive, thus by choosing p large enough, a wrap-around cannot occur for this term. However, considering the leftmost term a subtraction is seen. This means that a wrap-around zero can happen for this term. Fortunately, this can be detected by

comparing the minuend and subtrahend, clearly, if the subtrahend is larger than the minuend a wrap-around zero occurs. Furthermore, if a wrap-around zero has occurred for the leftmost term of the inequality, the outcome of the comparison should be 1, since the rightmost term is always positive.

Concerning the division in the protocol, it can be seen that it is solely used to compute $\left\lfloor \frac{\Delta \boldsymbol{x}}{\gamma} \right\rfloor$, where $\Delta \boldsymbol{x}$ is the negative gradient of $f(\boldsymbol{x})$. However, in the secure protocol it is beneficial to convert some of the additions to subtractions and defining $\Delta \boldsymbol{x} = g$, where g is the derivative of $f(x)$. This means that, the division to worry about is $\left\lfloor \frac{g}{\gamma} \right\rfloor$. γ can be chosen in such a way that no wrap-arounds occur, so the only worry is g. Recall that g is approximated by the differential quotient, hence in the secure protocol

$$g = f(x+1) - f(x), \tag{3.12}$$

since $h = 1$. If a wrap-around zero has occurred to g, this can again be detected by comparing the minuend and subtrahend in **Equation (3.12)**. By noting that $\frac{-a}{b} = -\left(\frac{a}{b}\right)$, it suffices to "remove" the wrap-around from g before the division and adding it again after the division. This is all done securely in **Protocol 3.8**.

Previously, it was also noted that some times it can be beneficial to introduce scaling in connection to integer division, thus protocol **Protocol 3.8** is built such that it considers both the case where scaling is desired and the case where it is not. Later, results of both options are compared.

Protocol 3.8 (Gradient Descent)
Given that $f(x)$ is a convex function with secret, integer coefficients that can be evaluated using only addition and multiplication. The parties hold all coefficients of $f(x)$ and $[x_{start}]$. The scaling constant C, the step size γ, and the integer max_{iter} are public.
Outputs $[x_{max_{iter}}]$, where $x_{max_{iter}}$ is the estimate of x^* after max_{iter} iterations.

1. If scaling is desired: The parties compute $[x_0] = C[x_{start}]$. Otherwise: $[x_0] = [x_{start}]$.
2. For $k = 0, \ldots$ till $max_{iter} - 1$ is reached, do:
   (a) If scaling is desired: The parties invoke **Protocol 3.7** to compute $[x_k] = \left\lfloor \frac{[x_k]}{C} \right\rfloor$.
   (b) The parties compute $[g_1] = f(1 + [x_k])$.
   (c) The parties compute $[g_2] = f([x_k])$.
   (d) The parties compute $[g] = [g_1] - [g_2]$.
   (e) The parties invoke **Protocol 3.6** to securely compute $[H] = \begin{cases} [1] & \text{if } g_1 < g_2 \\ [0] & \text{otherwise} \end{cases}$, using $[g_1]$ and $[g_2]$.

(f) The parties compute $[S] = (-1)[H] + (1 - [H]) = 1 - 2[H]$.

(g) The parties either use $\gamma = C \in \mathbb{F}_p$, where $C > 1$ is a fixed constant, or obtain γ using backtracking line search as:

i. The parties set $G = 1$.

ii. While $G = 1$, the parties do:

A. The parties invoke **Protocol 3.7** to compute $[T] = [S] \left\lfloor \frac{[S][g]}{\gamma} \right\rfloor$.

B. The parties compute $[T'] = [g_2] - [T]$.

C. The parties invoke **Protocol 3.6** to securely compute $[K] = \begin{cases} [1] & \text{if } g_2 < T \\ [0] & \text{otherwise} \end{cases}$, using $[g_2]$ and $[T]$.

D. The parties compute $[H] = f([x_k] - [T])$.

E. The parties invoke **Protocol 3.6** to securely compute $[k] = \begin{cases} [1] & \text{if } T < H \\ [0] & \text{otherwise} \end{cases}$, using $[T]$ and $[H]$.

F. The parties invoke **Protocol 3.2** to compute $[G] = [k] \vee [K]$.

(h) If scaling is desired: The parties compute $[x_{k+1}] = [x_k] - \left\lfloor \frac{C}{\gamma} \right\rfloor [g]$. Otherwise, the parties compute $[x_{k+1}] = [x_k] - [T]$.

3. If scaling is desired: the protocol returns $[Cx_{k+1}]$.

Proof. Security follows since solely secure protocols are used in the calculations. A formal proof of correctness of the protocol is not given. However, that the protocol does output the desired result has been argued for immediately before stating the protocol and in the beginning of **Section 3.2**. □

3.3 Simulation

In this section **Protocol 3.8** is simulated in order to see if it works as intended. For comparison, also **Algorithm 3.1** is simulated. To see how the results from the simulations of the two methods compare, the squared error of the parameter estimate, denoted e_k^2, at each iteration, k, is computed. To be clear

$$e_k^2 = (x_k - x^*)^2 \quad \text{for } k = 0, 1, \ldots. \tag{3.13}$$

The following parameters are used for **Protocol 3.8** in the calculations:

- The number of parties, $n = 4$.
- The degree of the polynomial for Shamir's scheme, $t = 1$.
- The cardinality of $\mathbb{F}_p$, $p = 1125899839733759$.

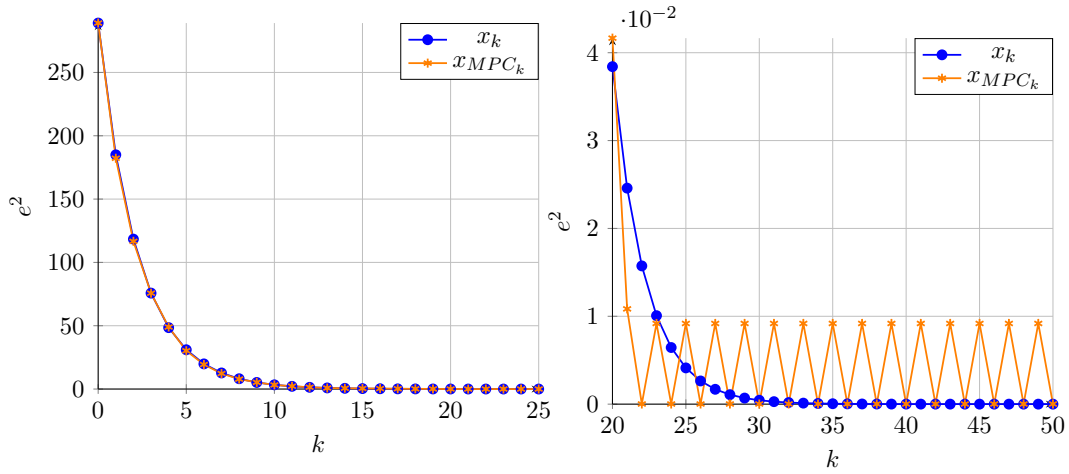
A cost function to minimize is chosen to be $f(x) = (x-[3])^2$, which clearly has $x^* = 3$. The parameter in common for all simulations are $x_0 = x_{start} = 20$. Furthermore, for **Protocol 3.8** the scaling constant is chosen to be $C = 2^{15}$. Refer to the estimate of x^* given by **Algorithm 3.1** to each iteration k as x_k, and by **Protocol 3.8** as x_{MPC_k}.

The rest of the section is divided into the following subsections:

1. Simulations of **Protocol 3.8** with fixed γ.
2. Simulations of **Protocol 3.8** with backtracking line search.

3.3.1 Simulation of Protocol 3.8 with Fixed γ

Figure 3.2 shows the squared error of approximating x^* with **Algorithm 3.1** and **Protocol 3.8**, where a fixed step size, $\gamma = 10$ is used. Furthermore, scaling is used in **Protocol 3.8**.



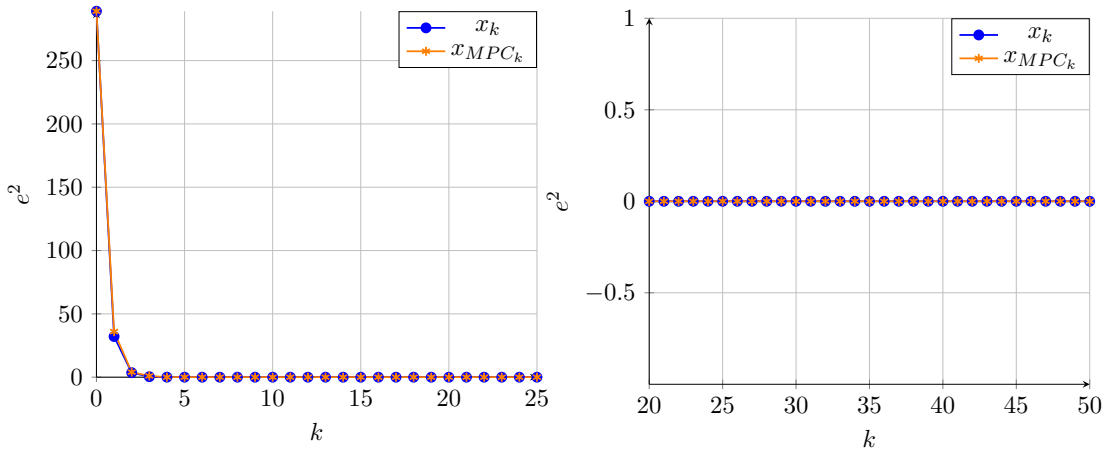
(a) The squared error for the first 25 iterations.

(b) The squared error for iteration number 20-50.

Figure 3.2: Simulation of **Algorithm 3.1** and **Protocol 3.8** for finding the minimum of $f(x) = (x-3)^2$. For both methods a fixed step size, $\gamma = 10$ is used. Regarding **Protocol 3.8** scaling is used. Figure **(a)** shows the squared error of each of the estimates for the first 25 iterations. Figure **(b)** shows a zoom on the squared error for 20-50 iterations.

Figure 3.2a shows that the estimates obtained with **Protocol 3.8** converges similarly to the estimates obtained with **Algorithm 3.1**. But, **Figure 3.2b** reviles that the estimates obtained with **Protocol 3.8** does in fact not converge, rather the estimates oscillates between two values after the first 20 iterations. This happens because of the poorly estimated derivative. It can be shown that when $x_{MPC_k} = 3$, then the derivative is estimated to be 1, meaning that $x_{MPC_{k+1}} = 3 + \gamma$. When x_{MPC_k} is below 3 the derivative is estimated to -1, which causes the oscillations. This behavior is actually avoided when not using scaling in **Protocol 3.8**.

Figure 3.3 shows the squared error of approximating x^* with **Algorithm 3.1** and **Protocol 3.8**, where a fixed step size, $\gamma = 3$ is used. Furthermore, scaling is not used in **Protocol 3.8**.



(a) The squared error for the first 25 iterations.

(b) The squared error for iteration number 20-50.

Figure 3.3: Simulation of **Algorithm 3.1** and **Protocol 3.8** for finding the minimum of $f(x) = (x-3)^2$. For both methods a fixed step size, $\gamma = 3$ is used. Regarding **Protocol 3.8** no scaling is used. Figure **(a)** shows the squared error of each of the estimates for the first 25 iterations. Figure **(b)** shows a zoom on the squared error for 20-50 iterations.

The reason that the oscillating behavior is avoided when not using scaling, is that even though the derivative is still approximated as 1 when $x_{MPC_k} = 3$, this entails that $\left\lfloor \frac{g}{\gamma} \right\rfloor = 0$, when $g = 1$ and $\gamma = 3$.

It should be noted that if the step size for instance was chosen as $\gamma = 10$, the estimate of x^* using **Protocol 3.8** without scaling, would converge to 7 because the derivative at this point would be estimated as $9 < 10$. Thus, the reason that **Protocol 3.8** performs so good in **Figure 3.3**, even though scaling is not used, is because γ is very low. This motives the use of backtracking line search to determine γ.

3.3.2 Simulation of Protocol 3.8 using Backtracking Line Search

Figure 3.4 shows the squared error of approximating x^* with **Algorithm 3.1** and **Protocol 3.8**, where a backtracking line search is used to determine the step size, γ, in each iteration. Furthermore, scaling is not used in **Protocol 3.8**. For **Algorithm 3.1**, the parameters $\alpha = 0.1$ and $\beta = 0.5$ are used.

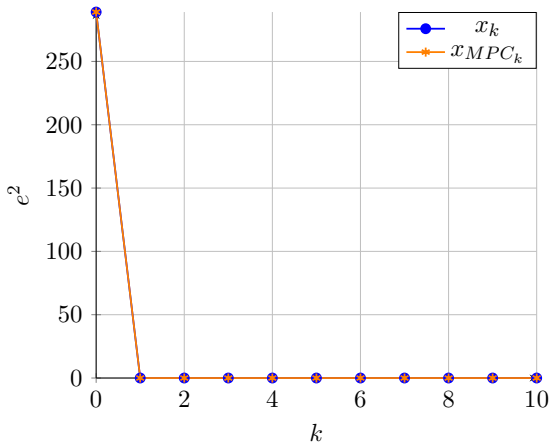


Figure 3.4: Simulation of **Algorithm 3.1** and **Protocol 3.8** for finding the minimum of $f(x) = (x-3)^2$. For both methods γ is determined at each iteration using backtracking line search, regarding **Algorithm 3.1**, the parameters $\alpha = 0.1$ and $\beta = 0.5$ are used. Regarding **Protocol 3.8** no scaling is used.

3.4 Summery

The focus of this chapter was not to create a perfect, secure MPC optimization protocol. Rather the goal was to learn about the challenges of converting a known algorithm to a secure protocol. What was discovered, is that not all operations are as simple to convert to a secure protocol as addition and multiplication. The reason that addition and multiplication are quite simple to convert, is that they can be computed directly on shares. The same cannot be said about operations such as comparison and division, since comparing shares does not correspond to comparing the secrets. Likewise, will dividing shares not correspond to division of the secrets.

It was however shown in this chapter that it is possible to create protocols that can compare secrets through their shares and that can approximately divide secrets through their shares. It was also pointed out that doing this means that care must be taken when wrap-arounds occur. For instance, if the operation $\frac{-2}{2}$ is carried out using finite field arithmetic on shares in $\mathbb{F}_{23}$, what will be carried out is the operation $\left\lfloor \frac{21}{2} \right\rfloor = 10 \neq -1$. A similar problem arises if it is desired to compare $-1 < 5$, which in $\mathbb{F}_{23}$ becomes $22 < 5$, which is false, opposite the expected result, which is true.

In **Section 3.3** it was shown that similar results are obtained when using a secure implementation of the gradient descent method and a non-secure implementation. However, simulations were solely carried out with a simple quadratic cost function such as $f(x) = (x-a)^2$, where $a \in \mathbb{F}_p$. It should be noted that there are still many limitations to the solution presented in this chapter. For instance, it is required that the cost function can be evaluated using addition and multiplication and it must have integer coefficients.

4

Pressure Control Algorithm as a Secure MPC Protocol

In this chapter a simple control algorithm is considered and it is investigated how it can be converted to a secure MPC protocol.

Section 4.1 gives an introduction to the control algorithm, while **Section 4.2** discusses how it can be converted into a secure MPC protocol. **Section 4.3** provides a simulation of the algorithm as well as a simulation of the secure MPC protocol. Finally, **Section 4.4** provides a summery of the chapter.

4.1 Introduction to the Control Algorithm

This section provides a short presentation of the problem that the control algorithm solves and finally the algorithm itself is stated.

Consider a water distribution network, which contains a pump, two valves, and two nodes where the water pressure can be measured. This system is seen at **Figure 4.1**.

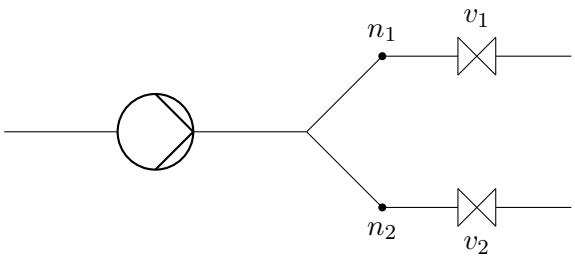


Figure 4.1: A water distribution network with (from the left) a pump, two nodes, and two valves.

Let $y_1(t)$ be the measured pressure at node n_1 at time t and $y_2(t)$ be the measured

pressure at node n_2 at time t. Assume, that in the system there is a minimum pressure requirement formulated as

$$y_i(t) \geq r, \quad \forall\, t, \text{ and } i = 1, 2, \tag{4.1}$$

where r is the desired pressure, which in this system is assumed to be constant in time and nodes. In the following, assume that $r = 8$ bar, and that the pump is running at a fixed speed, producing a pressure of 8 bar at the nodes when the valves are closed. Note that if the valves are opened the pressure in the network drops accordingly. Neglecting to account for any dynamics in the system, **Figure 4.2** shows a simulation of the pressure in the system, where $C_1(t)$ is the pressure drop over v_1 at time t and $C_2(t)$ is the pressure drop over v_2 at time t.

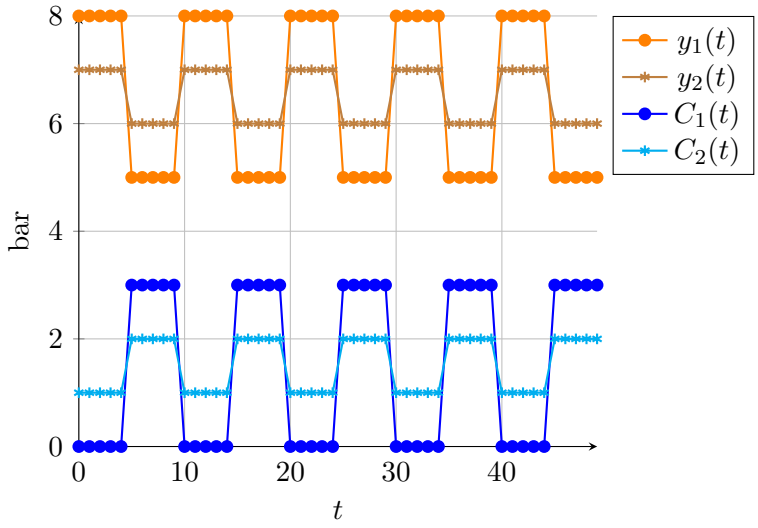


Figure 4.2: Illustration of water pressure and water consumption.

As seen in **Figure 4.2**, the water pressure at the nodes drops below r multiple times. This behavior can be accounted for by making the speed of the pump adapt to the water consumption. Therefore, consider now the distribution network in **Figure 4.3**, where the pump is controlled by a signal, $\kappa(t)$, instead of running at a fixed speed.

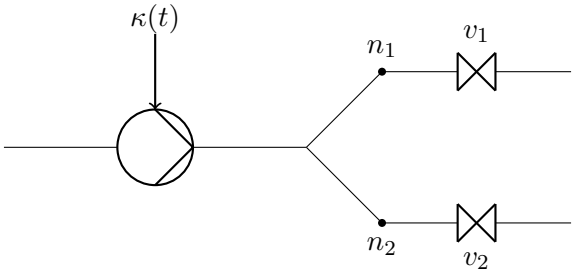


Figure 4.3: A water distribution network with (from the left) a pump, two nodes, and two valves.

To calculate the control signal, $\kappa(t)$, it is assumed that the total water demand in the network is periodic, as $C_1(t)$ and $C_2(t)$ shows in **Figure 4.2**, where the length of a period is seen to be 10. The water pressure is measured at the nodes w times within a period. The vector of measurements at node i for the k'th period is referred to as $\boldsymbol{y}_{i,k}$, and it is assumed that the desired pressure is constant in time and nodes. In this way, the control parameter vector $\boldsymbol{\kappa}_k = [\kappa(1+k), \ldots, \kappa(w+k)]$, is the pressure increases that the pump must deliver at the k'th period.

To determine $\boldsymbol{\kappa}_k$ the difference between desired and measured pressure is calculated. The error vector, $\boldsymbol{e}_k$, for the k'th period, is given as

$$\boldsymbol{e}_k = \max_i \{\boldsymbol{r} - \boldsymbol{y}_{i,k}\}, \quad i = 1, 2, \tag{4.2}$$

where $\boldsymbol{r} = r\mathbb{1}$ with $\mathbb{1}_w \in \mathbb{R}^w$ is the vector of ones and $\max\{\cdot\}$ is defined element-wise. As seen, the value at the j'th entry of $\boldsymbol{e}_k$ is the largest difference between measured and desired pressure amongst the nodes at time j, for $j = 1, \ldots, w$.

The following update law for the control parameter vector, $\boldsymbol{\kappa}_{k+1}$, for the $k+1$st period is proposed

$$\boldsymbol{\kappa}_{k+1} = \boldsymbol{\kappa}_k + K\boldsymbol{e}_k, \quad k = 1, 2, \ldots, \tag{4.3}$$

where $K \in \mathbb{R}$ is a control gain, which should be chosen such that $0 < K < 2$, to ensure convergence. The iterative calculation of $\boldsymbol{\kappa}_k$ is referred to as *the pressure control algorithm* and is stated in **Algorithm 4.1**.

Algorithm 4.1 (Pressure Control Algorithm)
Constants: r, K, where r is the desired pressure at the nodes and $K \in \mathbb{R}$ is a control gain.
Initialization: $\boldsymbol{\kappa}_1 = \boldsymbol{r}$, where $\boldsymbol{r} = r\mathbb{1}_w$.

1. For $k = 1, \ldots$ do:
   (a) For $i = 1, \ldots, n$, the pressure at node i is measured w times within the period k. The measurements are collected in a vector referred to as $\boldsymbol{y}_{i,k}$.
   (b) Calculate the error signal $\boldsymbol{e}_k$ for the k'th period as
   $$\boldsymbol{e}_k = \max_i \{\boldsymbol{r} - \boldsymbol{y}_{i,k}\}, \tag{4.4}$$
   where $\max\{\cdot\}$ is element-wise.
   (c) Calculate the control parameter vector for the next period, $k+1$, as
   $$\boldsymbol{\kappa}_{k+1} = \boldsymbol{\kappa}_k + K\boldsymbol{e}_k. \tag{4.5}$$

The algorithm is from [Jensen et al., 2017].

The interested reader is referred to [Jensen et al., 2017], for a more elaborate presentation of the algorithm.

Figure 4.4 shows a simulation of the system in **Figure 4.3** where **Algorithm 4.1** is used to calculate $\kappa(t)$.

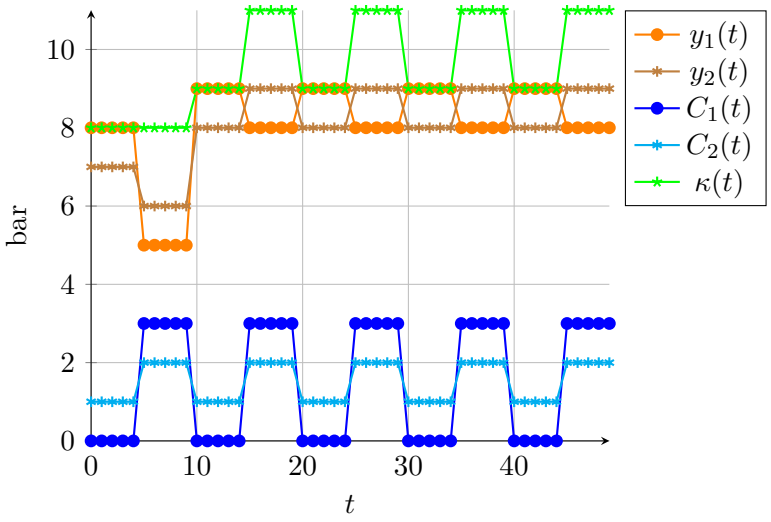


Figure 4.4: Simulation of the system in **Figure 4.3**, where **Algorithm 4.1** is used to calculate $\kappa(t)$ and $K = 1$.

Now, imagine that the nodes, where the pressure measurements in the network is made, is placed at the end-users. Then it would be preferred to keep these measurements secret. Thus, the next section discusses how **Algorithm 4.1** can be implemented in the setting of secure MPC.

4.2 Converting the Pressure Control Algorithm to a Secure MPC Protocol

By assuming that the desired pressure in the nodes, r, is public knowledge, it is seen from **Algorithm 4.1** that the operations needed in the algorithm is max and addition. The next section will present a secure MPC protocol for computing the maximum of a number of secret values.

4.2.1 Secure MPC: the max-function

This section investigates the implementation of a secure MPC maximum-function. To be clear, the goal is to determine shares of the largest secret-value out of a number of secret values.

Since a comparison has already been stated in **Protocol 3.3** it is relatively straight forward to create a protocol to compute the maximum value of a number of secrets.

This protocol is stated in **Protocol 4.1**.

Protocol 4.1 (Maximum Value)
Given that the parties hold $[x_1]^B, \ldots, [x_n]^B$.
Outputs $[y]^B$, where $y = \max\{x_1, x_2, \ldots, x_n\}$.

1. The parties define $[h]^B = [x_1]^B$.
2. For $i = 2, \ldots, n$ the parties
   (a) invoke **Protocol 3.3** to securely compute $[c] = \begin{cases} [1] & \text{if } h < x_i \\ [0] & \text{otherwise} \end{cases}$, using $[h]^B$ and $[x_i]^B$.
   (b) The parties compute $[h]^B = (1 - [c])[h]^B + [c][x_i]^B$.
3. The parties define $[y]^B = [h]^B$.

Proof. Since **Protocol 4.1** relies only on secure protocols, it is concluded that it is secure. Correctness of the protocol is trivial. □

4.2.2 Secure MPC: Pressure Control Algorithm

The secure protocol for **Algorithm 4.1** can be now stated. Before doing so, notation for a shared vector is defined.

Definition 4.1
When the parties hold shares of vector, $\boldsymbol{a} = [a_0, \ldots, a_n]$ it means that the parties hold shares of each of the entries in $\boldsymbol{a}$. Hence, if the parties hold $[\boldsymbol{a}]$, it means that the parties hold $[a_0], \ldots, [a_n]$.

Protocol 4.2 (Pressure Control Algorithm)
Given that party P_i for $i = 1, \ldots, n$, knows the vector of measurements $y_{i,k} \in {\mathbb{F}_p}^{w \times 1}$ for a period k, for $i = 1, \ldots, n$ and that r and K are public knowledge and $\boldsymbol{r} = r\mathbb{1}_w$. The parties hold $[\boldsymbol{\kappa}_k]$.
Outputs $[\boldsymbol{\kappa}_{k+1}]$, where κ_{k+1} is the control signal for the $k + 1'st$ period.

1. Each party, P_i, for $i = 1, \ldots, n$ computes $\boldsymbol{err}_i = [err_{i,1}, \ldots, err_{i,w}] = \boldsymbol{r} - \boldsymbol{y}_{i,k}$.
2. Each party, P_i, for $i = 1, \ldots, n$ calculates the bitwise representation of $err_{i,j}$ for $j = 1 \ldots, w$.

3. Each party, P_i, for $i = 1, \ldots, n$ distributes $[err_{i,j}]^B$ for $j = 1, \ldots, w$.
4. For $j = 1, \ldots, w$ the parties invoke **Protocol 4.1** to calculate $[e_j]^B = \max\{[err_{1,j}]^B, \ldots, [err_{n,j}]^B\}$.
5. The parties define the shared vector $[\boldsymbol{e}_k] = [\sum_{i=0}^{l-1}[e_{1_i}]2^i, \ldots, \sum_{i=0}^{l-1}[e_{w_i}]2^i]$.
6. The parties compute $[\boldsymbol{\kappa}_{k+1}] = [\boldsymbol{\kappa}_k] + K[\boldsymbol{e}_k]$.

Proof. Correctness follows from the fact that solely secure protocols are used. Correctness is not proven formally, however, it can be verified that the protocol follows the structure of **Algorithm 4.1**, which gives the desired output. Furthermore, the following section shows by simulation, that **Algorithm 4.1** and **Protocol 4.2** gives the same output. □

4.3 Simulations

In this section simulation results from **Algorithm 4.1** is compared to simulation results from **Protocol 4.2**. The system in **Figure 4.3** is simulated twice; first where the control signal, $\boldsymbol{\kappa}(t)$ is calculated using **Algorithm 4.1** and second where the control signal $\boldsymbol{\kappa}_{MPC}(t)$ is calculated using **Protocol 4.2**.

The following parameters are used in the simulations:

- The number of parties, $n = 4$.
- The degree of the polynomial for Shamir's scheme, $t = 1$.
- The cardinality of $\mathbb{F}_p$, $p = 1125899839733759$.
- The control gain, $K = 1$.
- The desired pressure, $r = 8$.
- The length of a period, $w = 10$.
- The measurement of water pressure, y_1 and y_2, are seen in **Figure 4.5**.

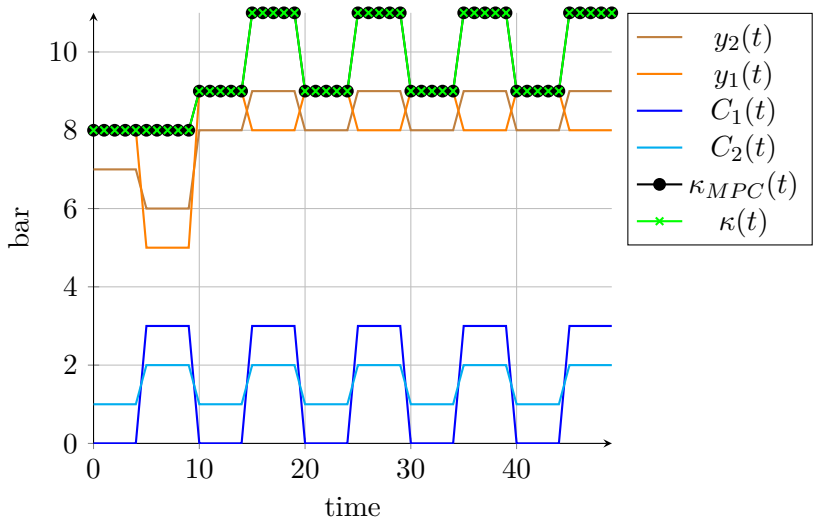


Figure 4.5: Comparison of the control signal obtained with **Algorithm ??** and **Protocol 4.2**, respectively.

As expected, **Figure 4.5** shows that simulating **Protocol 4.2** gives the exact same results as simulating **Algorithm 4.1**.

4.4 Summary

In this section a simple pressure control algorithm was introduced. The only new function that was converted into a secure MPC protocol was the max-function. The way that it was created was to use the comparison protocol presented in the previous chapter to compare every pair of secrets until the maximum amongst the secret was found. Using this, the pressure control algorithm was successfully converted to a secure MPC protocol.

At the end of the chapter a simulation of **Protocol 4.2** is compared to a simulation of **Algorithm 4.1**. Fortunately, the exact some result is obtained in both simulation, which was expected.

5

The Method of Least Squares as a Secure MPC Protocol

Least-squares methods are essential in estimation theory and regression analysis and moreover it is often used in adaptive control algorithms, [Nguyen, 2018, p.125]. Therefore, the method is considered to be of high importance to the field of control theory, and as such it is crucial that it can be implemented in the setting of secure MPC. The goal in this chapter is to study a simple estimation algorithm that uses the method of least squares, and investigate the possibilities of converting it to a secure MPC protocol.

In **Section 5.1** an introduction to recursive linear regression is provided and in **Section 5.2** it is investigated how to convert the method to a secure MPC protocol. **Section 5.3** provides simulations of the recursive least squares equations as well as simulations of the secure MPC protocol calculating the same equations. Finally, **Section 5.4** provides a summery of the chapter.

5.1 Introduction to Recursive Linear Regression using the Method of Least Squares

The problem of recursive linear regression is studied. This section serves as a reminder of the setup of this problem and its solution, in the form of the recursive least squares equations.

Suppose that an unknown linear system takes a vector of inputs and outputs a linear combination of these. Given m observations of inputs and outputs, it is desired to determine the parameters of the system. More precisely, the linear system is modeled as

$$\boldsymbol{y}_m = \boldsymbol{X}_m \boldsymbol{w}, \tag{5.1}$$

where $\boldsymbol{y}_m = [y_1, \dots, y_m]^\top$, $\boldsymbol{x}_i = [x_{1,i}, \dots, x_{q,i}]$ for $i = 1, \dots, m$, $\boldsymbol{X}_m = [\boldsymbol{x}_1^\top, \dots, \boldsymbol{x}_m^\top]^\top$ and

$\boldsymbol{w} = [w_1, \ldots, w_q]^\top$. y_i, $\boldsymbol{x}_i$ for $i = 1, \ldots, m$ are observed outputs and inputs, respectively. It is desired to determine the vector of parameters $\boldsymbol{w}$.

Assuming that $m > q$, **Equation (5.1)** describes an overdetermined system of equations. Since this system in general is inconsistent, the goal is to find the parameters that makes the best solution in a least squares sense. Thus, $\boldsymbol{w}$ is approximated by finding the parameters which minimizes the sum of squared residuals;

$$\hat{\boldsymbol{w}} = \min_{\boldsymbol{w}} ||\boldsymbol{y}_m - \boldsymbol{X}_m \boldsymbol{w}||^2. \tag{5.2}$$

The solution to this problem is the so-called *normal equations*

$$\hat{\boldsymbol{w}} = (\boldsymbol{X}_m^\top \boldsymbol{X}_m)^{-1} \boldsymbol{X}_m^\top \boldsymbol{y}_m, \tag{5.3}$$

where it is assumed that $\boldsymbol{X}_m$ has full column rank. For a more thorough review of the method of least squares and the derivation of the normal equations, see [Haykin, 2014, pp. 400-412].

Now, consider the case where all observations are not available at once but rather arrive one at a time. In this case it is desirable to update the estimate of $\boldsymbol{w}$ to account for the new data rather than solving **Equation (5.3)** from scratch. This is known as recursive linear regression. To see how it can be achieved, consider $\hat{\boldsymbol{w}}_N$, as the estimate of $\boldsymbol{w}$ at time $N \in \mathbb{N}$,

$$\hat{\boldsymbol{w}}_N = (\boldsymbol{X}_N^\top \boldsymbol{X}_N)^{-1} \boldsymbol{X}_N^\top \boldsymbol{y}_N. \tag{5.4}$$

Given a new observation of input and output, (x_{N+1}, y_{N+1}), the estimate $\hat{\boldsymbol{w}}_{N+1}$ can be written as

$$\hat{\boldsymbol{w}}_{N+1} = \left(\begin{bmatrix} \boldsymbol{X}_N \\ \boldsymbol{x}_{N+1} \end{bmatrix}^\top \begin{bmatrix} \boldsymbol{X}_N \\ \boldsymbol{x}_{N+1} \end{bmatrix} \right)^{-1} \begin{bmatrix} \boldsymbol{X}_N \\ \boldsymbol{x}_{N+1} \end{bmatrix}^\top \begin{bmatrix} \boldsymbol{y}_N \\ y_{N+1} \end{bmatrix}. \tag{5.5}$$

To make **Equation (5.5)** more tractable, define the matrix

$$\boldsymbol{S}_N = \boldsymbol{X}_N^\top \boldsymbol{X}_N, \tag{5.6}$$

and note that

$$\boldsymbol{S}_{N+1} = \begin{bmatrix} \boldsymbol{X}_N \\ \boldsymbol{x}_{N+1} \end{bmatrix}^\top \begin{bmatrix} \boldsymbol{X}_N \\ \boldsymbol{x}_{N+1} \end{bmatrix} \tag{5.7}$$

$$= \boldsymbol{S}_N + \boldsymbol{x}_{N+1}^\top \boldsymbol{x}_{N+1}. \tag{5.8}$$

Furthermore, since

$$\boldsymbol{S}_N \hat{\boldsymbol{w}}_N = (\boldsymbol{X}_N^\top \boldsymbol{X}_N)(\boldsymbol{X}_N^\top \boldsymbol{X}_N)^{-1} \boldsymbol{X}_N^\top \boldsymbol{y}_N \tag{5.9}$$

$$= \boldsymbol{X}_N^\top \boldsymbol{y}_N, \tag{5.10}$$

it is easy to see that

$$\begin{aligned} \boldsymbol{S}_{N+1}\hat{\boldsymbol{w}}_{N+1} &= \boldsymbol{S}_N\hat{\boldsymbol{w}}_N + \boldsymbol{x}_{N+1}y_{N+1} && (5.11)\\ &= \left(\boldsymbol{S}_{N+1} - \boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^\top\right)\hat{\boldsymbol{w}}_N + \boldsymbol{x}_N y_{N+1} && (5.12)\\ &= \boldsymbol{S}_{N+1}\hat{\boldsymbol{w}}_N - \boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^\top\hat{\boldsymbol{w}}_N + \boldsymbol{x}_{N+1}y_{N+1}, && (5.13)\end{aligned}$$

which again entails that

$$\hat{\boldsymbol{w}}_{N+1} = \hat{\boldsymbol{w}}_N + \boldsymbol{S}_{N+1}^{-1}\boldsymbol{x}_{N+1}\left(y_{N+1} - \boldsymbol{x}_{N+1}^\top\hat{\boldsymbol{w}}_N\right). \tag{5.14}$$

As seen, **Equation (5.14)** requires a matrix inversion, which is a computationally expensive operation. However, in this case it is possible to use *the matrix inversion lemma* [Haykin, 2014, p. 453], which offers a more convenient way to calculate the inverse of $\boldsymbol{S}_{N+1}$.

By letting $\boldsymbol{P}_N = \boldsymbol{S}_{N+1}^{-1}$ and using the matrix inversion lemma, $\boldsymbol{P}_{N+1}$ can be calculated as

$$\boldsymbol{P}_{N+1} = \boldsymbol{P}_N - \left(1 + \boldsymbol{x}_{N+1}^\top\boldsymbol{P}_N\boldsymbol{x}_{N+1}\right)^{-1}\boldsymbol{P}_N\boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^\top\boldsymbol{P}_N. \tag{5.15}$$

Now a recursive least squares algorithm can be stated, see **Algorithm 5.1**.

Algorithm 5.1 (Recursive Least Squares)
Initialization:

- $\boldsymbol{P}_0 = \boldsymbol{I}_q$.
- $\hat{\boldsymbol{w}}_0 = \boldsymbol{0}_q$, where $\boldsymbol{0} \in \mathbb{R}^{q\times 1}$ is the vector of zeros.

1. For $N = 1, \ldots$ do
   (a) Define the input to the system at time N as $\boldsymbol{x}_N \in \mathbb{R}^q$ and the observation of the output of the system at time N as $y_N \in \mathbb{R}$.
   (b) Make the following calculations

$$\begin{aligned} \boldsymbol{P}_N &= \boldsymbol{P}_{N-1} - \left(1 + \boldsymbol{x}_N^\top\boldsymbol{P}_{N-1}\boldsymbol{x}_N\right)^{-1}\boldsymbol{P}_{N-1}\boldsymbol{x}_N\boldsymbol{x}_N^\top\boldsymbol{P}_{N-1}, && (5.16)\\ \boldsymbol{g}_N &= \boldsymbol{P}_N\boldsymbol{x}_N, && (5.17)\\ e_N &= y_N - \boldsymbol{x}_N^\top\hat{\boldsymbol{w}}_{N-1}, && (5.18)\end{aligned}$$

   (c) The parameter estimate, $\hat{\boldsymbol{w}}_N$ at time N is

$$\hat{\boldsymbol{w}}_N = \hat{\boldsymbol{w}}_{N-1} + \boldsymbol{g}_N e_N. \tag{5.19}$$

The algorithm is from [Haykin, 2014, p. 454].

Now, if it is preferred to keep the observations, $\boldsymbol{x}_N$ and y_N, private, a secure implementation of **Algorithm 5.1** is needed.

5.2 Converting the Recursive Least Squares Equations to a Secure MPC Protocol

From **Algorithm 5.1** it is seen that the operations used are subtraction, division and multiplication. Subtraction can be done securely similarly to doing addition securely. Hence, the only operation to consider is division. If it was acceptable for the division to be carried out in the finite field $\mathbb{F}_p$, it would be easy to state a protocol for this operator. However, as pointed out in **Chapter 3**, to be able to compare a secure MPC estimate of $\boldsymbol{w}$, to a non-secure estimate of $\boldsymbol{w}$, the division cannot be carried out in the field. It is not straight forward how this division should then be carried out, thus in the following three possibilities are considered.

Idea One: The first idea is to perform the division after protocol execution, meaning that the protocol should output a vector $\tilde{\boldsymbol{w}}$ and a denominator, d, such that $\hat{\boldsymbol{w}} = \tilde{\boldsymbol{w}}./d$, where ./ denotes element-wise division. By choosing a finite field with cardinality large enough to ensure that no wrap-arounds occur, this technique works as intended. However, because of the recursions in the problem formulation, especially the denominator seems to grow without bound, requiring $|\mathbb{F}_p| \approx \infty$. To see this, consider **Equation (5.16)**, **(5.17)**, **(5.18)**, and **(5.19)**, where no division is performed, but instead the nominator and denominator for every expression is kept track of. Thus, rather than computing $\frac{n}{d}$, this is written as (n, d). To write the equations like this, it is necessary to add three equations to keep track of the different denominators. The term d_{N+1} refers to the calculation of $1+\boldsymbol{x}_{N+1}^\top \boldsymbol{P}_N \boldsymbol{x}_{N+1}$, which is the denominator in **Equation (5.16)**. D_N is the denominator for P_N and thus the denominator for P_{N+1} is $d_{n+1}D_N$. Furthermore, dw_N is the denominator for $\hat{w}$ at time N. Note that $d_0 = D_0 = dw_0 = 1$.

$$d_{N+1} = D_N \cdot 1 + \boldsymbol{x}_{N+1}^\top \boldsymbol{P}_N \boldsymbol{x}_{N+1} \tag{5.20}$$
$$D_{N+1} = D_N d_{N+1} \tag{5.21}$$
$$dw_{N+1} = dw_N \cdot D_{N+1} \tag{5.22}$$
$$\boldsymbol{P}_{N+1} = (d_{N+1}\boldsymbol{P}_N - \boldsymbol{P}_N \boldsymbol{x}_{N+1}\boldsymbol{x}_{N+1}^\top \boldsymbol{P}_N, D_{N+1}) \tag{5.23}$$
$$\boldsymbol{g}_{N+1} = (\boldsymbol{P}_{N+1}\boldsymbol{x}_{N+1}, D_{N+1}) \tag{5.24}$$
$$e_{N+1} = (dw_N y_{N+1} - \boldsymbol{x}_{N+1}^\top \hat{\boldsymbol{w}}_N, dw_N) \tag{5.25}$$
$$\hat{\boldsymbol{w}}_{N+1} = (D_{N+1}\hat{\boldsymbol{w}}_N + \boldsymbol{g}_{N+1}e, dw_{N+1}). \tag{5.26}$$

To see how the denominator of $\boldsymbol{P}_N, \boldsymbol{g}_N, e, \hat{\boldsymbol{w}}_N$ evolves for each recursion in terms of d_N, see **Table 5.1**.

Time	$\boldsymbol{P}$	$\boldsymbol{g}$	e	$\tilde{\boldsymbol{w}}$
1	d_1	d_1	1	d_1
2	$d_1 d_2$	$d_1 d_2$	d_1	$d_1^2 d_2$
3	$d_1 d_2 d_3$	$d_1 d_2 d_3$	$d_1^2 d_2$	$d_1^3 d_2^2 d_3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
m	$d_1 \cdots d_m$	$d_1 \cdots d_m$	$d_1^{m-1} d_2^{m-2} \cdots d_{m-1}$	$d_1^m d_2^{m-1} \cdots d_m$

Table 5.1: The denominator of $\boldsymbol{P}_N, \boldsymbol{g}_N, e, \tilde{\boldsymbol{w}}_N$ in **Equation (5.23)**, **(5.24)**, **(5.25)**, and **(5.26)** in terms of d_N from equation **Equation (5.20)**, corresponding to each round of recursion.

As seen from **Table 5.1**, dw_N grows exponentially, requiring the cardinality of $\mathbb{F}_p$ to be infinitely high. Therefore the idea cannot be used.

Idea Two: The second idea is to perform the division in $\mathbb{F}_p$ and at the end use *rational reconstruction* to recover $\hat{\boldsymbol{w}}$. Rational reconstruction is described in [Monagan, 2004], and according to this $p > 2|n|d$ when $n, d \in \mathbb{F}_p$ is the nominator and denominator, respectively. Again, since dw_n grows rapidly because of the recursions, this method will also require $\mathbb{F}_p$ to have an infinite cardinality, thus this idea will not work either.

Idea Three: The final idea is to approximate the division in $\mathbb{Q}$ by integer division. The downside of this idea is that, only an approximative result can be expected as outcome of a protocol that builds on approximations. However, in lack of a better idea, this option is chosen. The following section discusses how to modify the recursive least squares equations to use integer division.

5.2.1 Modifying the Recursive Least Squares Equations to use Integer Division

The first thing to be aware of when substituting division with integer division, is that when the divisor is larger than the dividend, the result becomes zero. This may be a very obvious observation, however it may not be so obvious, that this may entail convergence issues. For instance, using the identity matrix as initialization for $\boldsymbol{P}_0$, entails that $\boldsymbol{P}_i$ is the identity matrix for all $i = 1, \ldots$ and thus does not converge to the desired matrix. To see this, refer to the last term of **Equation (5.16)** as $\boldsymbol{M}_{N+1}$, such that

$$\boldsymbol{M}_{N+1} = \left(1 + \boldsymbol{x}_{N+1}^\top \boldsymbol{P}_N \boldsymbol{x}_{N+1}\right)^{-1} \boldsymbol{P}_N \boldsymbol{x}_{N+1} \boldsymbol{x}_{N+1}^\top \boldsymbol{P}_N. \tag{5.27}$$

Letting $\boldsymbol{P}_0 = \boldsymbol{I}$, $\boldsymbol{M}_1$ becomes

$$\boldsymbol{M}_1 = (1 + \boldsymbol{x}_{n+1}^\top \boldsymbol{x}_{n+1})^{-1} \boldsymbol{x}_{n+1} \boldsymbol{x}_{n+1}^\top. \tag{5.28}$$

From **Equation (5.28)**, it is seen that the divisor is one added to the inner product of $\boldsymbol{x}_{n+1}$. Since the inner product of a vector, can be proven to be larger than all indices of the outer product of the same vector, it is seen that $\boldsymbol{M}_1$ is the zero matrix. Then $\boldsymbol{P}_1$ is also the identity matrix, and hence $\boldsymbol{P}_i$ is the identity matrix for $i = 1, \ldots$. In this way

the algorithm does not converge. Fortunately, the problem can be solved by introducing scaling of $\boldsymbol{M}_N$ before the division. To state the recursive linear regression equations using scaling and integer division, the operation of integer division is formally defined.

Definition 5.1 (Integer Division)
Let $a, b, c, d, z \in \mathbb{F}_p$ be integers. The term *integer division*, refers to the operation of dividing two integers and afterwards truncating the result, such that the output is also an integer. The operation is denoted by the symbol $\backslash$, such that

$$a \backslash z = \left\lfloor \frac{a}{z} \right\rfloor. \tag{5.29}$$

Furthermore, let $.\backslash$ denote the element-wise integer division operation such that

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix} .\backslash z = \begin{bmatrix} \left\lfloor \frac{a}{z} \right\rfloor & \left\lfloor \frac{c}{z} \right\rfloor \\ \left\lfloor \frac{b}{z} \right\rfloor & \left\lfloor \frac{d}{z} \right\rfloor \end{bmatrix}. \tag{5.30}$$

The scaling is introduced by letting $\boldsymbol{P}_0$ be initialized as $C\boldsymbol{I}$, where $C \in \mathbb{F}_p$ is an appropriately chosen integer. The integer recursive least squares algorithm is stated in **Algorithm 5.2**.

Algorithm 5.2 (Integer Recursive Least Squares)
Initialization:

- Let $C \in \mathbb{F}_p$ be a positive, large enough integer.
- $\boldsymbol{P}_0 = C\boldsymbol{I}_q$.
- $\boldsymbol{w}_0 = \boldsymbol{0}_q$.

1. For $N = 1, \ldots$ do
   (a) Define the input to the system at time N as $\boldsymbol{x}_N \in {\mathbb{F}_p}^{q \times 1}$ and the observation of the output of the system at time N as $y_N \in \mathbb{F}_p$.
   (b) Make the following calculations

$$\boldsymbol{P}_N = \boldsymbol{P}_{N-1} - \left(\left(\boldsymbol{P}_{N-1} \boldsymbol{x}_N \boldsymbol{x}_N^\top \boldsymbol{P}_{N-1} \right) .\backslash \left(C + \boldsymbol{x}_N^\top \boldsymbol{P}_{N-1} \boldsymbol{x}_N \right) \right) \tag{5.31}$$

$$\boldsymbol{g}_N = \boldsymbol{P}_N \boldsymbol{x}_N, \tag{5.32}$$

$$e_N = y_N - \boldsymbol{x}_N^\top \hat{\boldsymbol{w}}_{N-1} \backslash C, \tag{5.33}$$

   (c) The parameter estimate, $\hat{\boldsymbol{w}}_N$ at time N is

$$\hat{\boldsymbol{w}}_N = \hat{\boldsymbol{w}}_{N-1} + \boldsymbol{g}_N e_N. \tag{5.34}$$

To implement **Protocol 5.2** in the setting of secure MPC, the operations needed are; subtraction, addition, multiplication, integer division, and division by public constant. It is solely integer division of secrets, which has not already been implemented securely in this report. Thus, the following section is devoted to introducing this operation.

5.2.2 Secure MPC: Integer Division

This section builds on the ideas from [Dahl et al., 2012]. Let $nom, d \in \mathbb{N}$ be two $l-$bit integers and $y = \lfloor \frac{nom}{d} \rfloor$ be the integer, it is desired to determine. Furthermore, assume that k is an appropriately large integer, that is publicly known. The idea is to compute a "k-shifted" approximation of $\frac{1}{d}$ by $\hat{a} = \lfloor \frac{2^k}{d} \rfloor$ and afterwards calculate the desired result $y = \frac{\hat{a} \cdot nom}{2^k}$. The reasoning behind this idea is that the division by 2^k is simpler since 2^k is public. The technique to compute $\hat{a}$ is inspired from the Taylor Series. To recap the Taylor Series, the following holds for $0 < \alpha < 1$,

$$\frac{1}{\alpha} = \sum_{i=0}^{\infty} (1-\alpha)^i = \sum_{i=0}^{w} (1-\alpha)^i + \epsilon_w, \tag{5.35}$$

where $\epsilon_w = \sum_{i=w+1}^{\infty} (1-\alpha)^i$. Thus, ϵ can be seen as an error term, which is dependent on w. To get some intuition about the size of ϵ_w, consider the case where $0 < (1-\alpha) < \frac{1}{2}$,

$$\epsilon_w = \sum_{i=w+1}^{\infty} (1-\alpha)^i = (1-\alpha)^{w+1} \sum_{i=0}^{\infty} (1-\alpha)^i \leq 2^{-w-1} \frac{1}{\alpha} \leq 2^{-w}. \tag{5.36}$$

Hence, for $\frac{1}{2} < \alpha < 1$, ϵ is bounded by 2^{-w}. By choosing integers k, l_d, and w the Taylor Series can be be used to compute an approximation of $\frac{2^k}{d}$ in the following way;

$$\begin{aligned}
\frac{2^k}{d} &= 2^{k-l_d} \frac{1}{d/2^{l_d}} && (5.37)\\
&= 2^{k-l_d} \left(\sum_{i=0}^{w} \left(1 - \frac{d}{2^{l_d}} \right)^i + \epsilon_w \right) && (5.38)\\
&= 2^{k-l_d(w+1)} \sum_{i=0}^{w} \left(1 - \frac{d}{2^{l_d}} \right)^i 2^{l_d w} + 2^{k-l_d} \epsilon_w && (5.39)\\
&= 2^{k-l_d(w+1)} \sum_{i=0}^{w} \left(1 - \frac{d}{2^{l_d}} \right)^i (2^{l_d})^i 2^{l_d(w-i)} + 2^{k-l_d} \epsilon_w && (5.40)\\
&= 2^{k-l_d(w+1)} \sum_{i=0}^{w} \left(2^{l_d} - d \right)^i 2^{l_d(w-i)} + 2^{k-l_d} \epsilon_w. && (5.41)
\end{aligned}$$

By discarding the last term of **Equation (5.41)**, the approximation $\hat{a} = \frac{2^k}{d}$ yields

$$\hat{a} = 2^{k-l_d(w+1)} \sum_{i=0}^{w} \left(2^{l_d} - d \right)^i 2^{l_d(w-i)}. \tag{5.42}$$

Letting $k = l^2 + l, l_d = \lfloor \log_2(d) + 1 \rfloor$ and $w = l$ entails that $k \geq l_d(w+1)$, ensuring that $\hat{a}$ is an integer as desired. Furthermore, by manipulating **Equation (5.42)** it can be seen that $\hat{a}$ can by computed as a product of 2^{k-l_d} and a polynomial with coefficients equal to one evaluated in $(2^{l_d} - d)2^{-l_d}$,

$$\hat{a} = 2^{k-l_d(w+1)} \sum_{i=0}^{w} \left(2^{l_d} - d\right)^i 2^{l_d(w-i)} \tag{5.43}$$

$$= 2^{k-l_d(w+1)} \sum_{i=0}^{w} \left(2^{l_d} - d\right)^i 2^{-l_d i} 2^{l_d w} \tag{5.44}$$

$$= 2^{k-l_d} \sum_{i=0}^{w} \left((2^{l_d} - d)2^{-l_d}\right)^i. \tag{5.45}$$

As a recap of the described method, to compute an approximation of $\lfloor \frac{nom}{d} \rfloor$ it is necessary to retrieve $[2^{l_d}]$, $[2^{-l_d}]$, and to divide a field element with 2^k.

Secure calculation of 2^{l_d}

$[2^{l_d}]$ can be calculated using the prefix-OR of $[d]$, hereafter denoted by $[D]$, since it holds that $[2^{l_d}] = [D] + 1$. Therefore, the idea is to calculate the prefix-OR of $[d]$. In **Section 3.2.1** a protocol for securely calculating the OR-operation was given, namely **Protocol 3.2**. This means that the prefix-OR can be calculated after decomposing $[d]$ into bits, which can be achieved with **Protocol 3.5**. The protocol is stated in **Protocol 5.1**.

Protocol 5.1 (PRE-OR of an integer)
Given that the parties hold $[d]$.
Outputs $[y]$, where $y = 2^{l_d}$, for $l_d = \lfloor \log_2(d) + 1 \rfloor$.

1. The parties invoke **Protocol 3.5** to achieve $[d]^B$.
2. The parties invoke **Protocol 3.2** to define the bit sequence $[u]^B$, where $[u_i] = \vee_{j=i}^{l-1}[d_j]$, for $i = 0, \ldots, l-1$.
3. The parties compute $[y] = 1 + \sum_{i=0}^{l-1}[u_i]2^i$.

The protocol is from [Dahl et al., 2012, p. 11].

Dealing directly with bits in a secure MPC protocol induces a vast amount of communication, thus it is preferred to avoid bit decomposition. The work [Dahl et al., 2012, pp. 19-23] introduces a secure way of obtaining $[D]$, without resorting to bit-decomposition. Since it is a long an not very readable protocol, it is stated in **Appendix B** and the report settles with **Protocol 5.1**.

In the next section, it is considered how to compute $[2^{-l_d}]$.

Secure calculation of 2^{-l_d}

Now that 2^{l_d} can be calculated securely, it is simple to compute its inverse $[2^{-l_d}]$. The reason for this, is that 2^{-l_d} can be seen as the inverse field element of 2^{l_d} and not as a division. The protocol for computing the inverse of a field element is given in **Protocol 5.2**.

Protocol 5.2 (Inverse Field element)
Given that the parties hold $[x]$ and $[r]$, where r is a random number.
Outputs $[y]$, where $y = x^{-1}$.

1. The parties invoke **Protocol 2.10** to compute $[w] = [x][r]$ and opens w.
2. The parties compute $[y] = w^{-1}[r]$.

Proof. Security follows from the fact that solely secure operations are used, as well as the fact that w is a uniformly random number, thus opening it leaks no information. Correctness is trivial. □

5.2.3 Secure MPC: Integer Division Protocol

Finally, the integer division protocol is stated in **Protocol 5.3**.

Protocol 5.3 (Integer Division)
Given that the parties hold $[nom]$ and $[d]$, where nom and d are two $l-$bit integers.
Outputs $[y]$, where $y = nom \backslash d$.

1. The parties invoke **Protocol 5.1** to compute $[2^{l_d}] = [D] + 1$.
2. The parties invoke **Protocol 5.2** to compute $[2^{-l_d}]$.
3. The parties compute $[d'] = ([2^{l_d}] - [d])[2^{-l_d}]$.
4. The parties compute $[\hat{a}] = 2^k[2^{-l_d}] \sum_{i=0}^{w} [d']^i$.
5. The parties compute $[q] = [nom][\tilde{a}]$.
6. The parties invoke **Protocol 3.7** to compute $y = [q]2^{-k}$.

The protocol is from [Dahl et al., 2012].

Refer to [Dahl et al., 2012] for a proof of **Protocol 5.3**.

5.2.4 Secure MPC: Detecting wrap-around zero

Now, there is still one issue to consider, namely that when performing integer division in $\mathbb{F}_p$, the desired result is ensured only if no wrap-arounds has occured for both the dividend and divisor. To remind the reader of the problem, suppose that it is desired to calculate

$$d = (-a \mod p) \backslash a, \tag{5.46}$$

in $\mathbb{F}_p$ using **Protocol 5.3**. The desired result is $-1 \mod p$, since -1 is the result achieved if the calculation was done in $\mathbb{R}$. However, to see that the desired result would not be achieved, consider $a = 2$ and $p = 23$, the result of **Equation (5.46)** is

$$d = 21 \backslash 2 = 10 \neq -1 \mod 23 = 22. \tag{5.47}$$

Hence, if either or both of the nominator and denominator has wrapped-around zero, the desired result will not be achieved. However, by noting that

$$\frac{-a}{b} = -\left(\frac{a}{b}\right), \tag{5.48}$$

the desired result to **Equation (5.46)**, can be computed in three steps. The first step is to "remove" the wrap-around zero by computing a new nominator, nom_{new}:

$$\begin{aligned} nom_{new} &= (-1 \mod p) \cdot (-a \mod p) && (5.49)\\ &= ((p-1) \cdot (p-a)) \mod p && (5.50)\\ &= (p^2 - ap - p + a) \mod p && (5.51)\\ &= a. && (5.52) \end{aligned}$$

The second step is to compute **Equation (5.46)** using the new nominator, to obtain $\hat{d}$,

$$\hat{d} = nom_{new} \backslash a = a \backslash a = 1. \tag{5.53}$$

The third step is to "add" the wrap-around again to obtain d,

$$d = ((-1 \mod p)\hat{d} \mod p) = (-\hat{d}) \mod p = (p-1). \tag{5.54}$$

This example focuses on wrap-around zero. Similarly, the desired result would not be achieved, if a wrap-around p has occurred for either or both of the nominator and denominator. However, that no wrap-around p can occur, can be achieved by choosing p large enough.

Ensuring that no wrap-around zero occurs is more challenging. First, consider where this problem may happen in the integer least squares algorithm, for convenience the equations **(5.31) - (5.34)** are restated.

$$\begin{aligned} \boldsymbol{P}_N &= \boldsymbol{P}_{N-1} - \left(\left(\boldsymbol{P}_{N-1}\boldsymbol{x}_N\boldsymbol{x}_N^\top\boldsymbol{P}_{N-1}\right) .\backslash \left(C + \boldsymbol{x}_N^\top\boldsymbol{P}_{N-1}\boldsymbol{x}_N\right)\right) && (5.55)\\ \boldsymbol{g}_N &= \boldsymbol{P}_N\boldsymbol{x}_N, && (5.56)\\ e_N &= y_N - \boldsymbol{x}_N^\top\hat{\boldsymbol{w}}_{N-1} \backslash C, && (5.57)\\ \hat{\boldsymbol{w}}_N &= \hat{\boldsymbol{w}}_{N-1} + \boldsymbol{g}_N e_N. && (5.58) \end{aligned}$$

The only equations that involves integer division, is **Equation (5.55)** and **Equation (5.57)**. In both equations the divisor is always positive, which is easy to see by noting that $\boldsymbol{P}_N$ is a positive definite matrix and C is a positive integer. Furthermore, the dividend in **Equation (5.57)** is also positive, when assuming that $w_1, w_2 > 0$ and that $x_i \geq 0 \, \forall \, i$. However, the dividends in **Equation (5.55)** may or may not have wrapped around zero. To detect whether or not a wrap-around zero has occurred to the dividends in **Equation (5.55)**, recall the requirements to integer division made in **Section 5.2.2**. To remind the reader of the requirements, consider the computation of,

$$\left\lfloor \frac{a}{b} \right\rfloor, \tag{5.59}$$

for two secrets $a, b \in \mathbb{F}_p$, where a, b er two $l-$bit integers. Recall that the idea is to compute $\frac{2^k}{d}$, where it is required that $k = l^2 + l < p$. Thus, $p \geq 2^{l^2+l}$. Using this, it is seen that $a, b < \frac{p}{2}$. This means that if $a > \frac{p}{2}$ a wrap-around zero has occurred. This fact is formally written in **Proposition 5.1**.

Proposition 5.1
Let $a < \frac{p}{2}$ be an integer in $\mathbb{F}_p$, then it holds that

$$(-1 \bmod p)a \bmod p > \frac{p}{2}. \tag{5.60}$$

Proof. To see that **Equation (5.60)** holds, note that

$$\begin{aligned}
(-1 \bmod p)a \bmod p &= (p-1)a \bmod p && (5.61)\\
&= ap - a \bmod p && (5.62)\\
&= p - a && (5.63)\\
&> \frac{p}{2}, && (5.64)
\end{aligned}$$

where the inequality is true, given the assumption that $a < \frac{p}{2}$. □

As seen, in this case a wrap-around zero can be detected using comparison.

Since a secure protocol for bit-decomposition is available, comparison of two secrets can be achieved through **Protocol 3.5** and **Protocol 3.3**. This means that a secure protocol for detecting a wrap-around zero can be stated.

Protocol 5.4 (Wrap-around zero detection)
Given that the parties hold $[a]$, where $a < \frac{p}{2}$.
Outputs $[y]$, where $y = \begin{cases} 1, & \text{if } a \text{ has wrapped around zero,} \\ 0, & \text{otherwise.} \end{cases}$.

1. The parties invoke **Protocol 3.5** to achieve $[a]^B$.

2. The parties decompose $m = \frac{p}{2}$ into bits and create $[m]^B$.
3. The parties invoke **Protocol 3.3** to compute $[y] = \begin{cases} [1] & \text{if } m < a \\ [0] & \text{otherwise,} \end{cases}$, using $[m]^B$ and $[a]^B$.

Proof. Security follows since only secure protocols are used. Correctness follows from **Proposition 5.1**. □

5.2.5 Secure MPC: Recursive Least Squares Equations

Based on protocols previously stated, a protocol for the secure recursive least squares equations can now be presented. The protocol is a secure implementation of **Algorithm 5.1**, and can be seen in **Protocol 5.5**.

Protocol 5.5 (Recursive Least Squares)
Given that for each time $N = 1, \ldots$ the parties get to hold $[y_N]$ and $[\boldsymbol{x}_N]$, where $y_N \in \mathbb{F}_p$ and $\boldsymbol{x}_N \in {\mathbb{F}_p}^{q \times 1}$ are observations of output and input, respectively. Furthermore, $C \in \mathbb{F}_p$ is a public integer, $\boldsymbol{P}_0$ is the $q \times q$ identity matrix, and let $\hat{\boldsymbol{w}}_0 = \boldsymbol{0}_q$ be the $q \times 1$ vector of zeros. The parties hold $[\boldsymbol{P}_0]$, $[\hat{\boldsymbol{w}}_0]$.
Outputs $[C\hat{\boldsymbol{w}_N}]$, where $\hat{\boldsymbol{w}_N}$ is the parameter estimate at time N.

1. For $N = 1, \ldots$ do:
   (a) The parties compute $[d] = C + [\boldsymbol{x}_N]^\top [\boldsymbol{P}_{N-1}][\boldsymbol{x}_N]$.
   (b) The parties compute $[\boldsymbol{K}] = [\boldsymbol{P}_{N-1}][\boldsymbol{x}_N][\boldsymbol{x}_N]^\top [\boldsymbol{P}_{N-1}]$.
   (c) For each entry $[K_{i,j}]$ in $[\boldsymbol{K}]$, the parties invoke **Protocol 5.4** to compute $[H_{i,j}] = \begin{cases} [1], & \text{if } [K_{i,j}] \text{ has wrapped around zero} \\ [0], & \text{otherwise} \end{cases}$.
   (d) For each entry $[H_{i,j}]$ in $[\boldsymbol{H}]$, the parties compute $[G_{i,j}] = [H_{i,j}]\,(-[K_{i,j}]) + (1 - [H_{i,j}])\,[K_{i,j}] = [K_{i,j}] - 2[H_{i,j}]\,[K_{i,j}]$.
   (e) The parties invoke **Protocol 5.3** to compute $[\boldsymbol{Q}] = [\boldsymbol{G}].\backslash[d]$.
   (f) The parties compute $[\boldsymbol{P}_N] = [\boldsymbol{P}_{N-1}] - [\boldsymbol{Q}]$.
   (g) The parties compute $[\boldsymbol{g}_N] = [\boldsymbol{P}_N][\boldsymbol{x}_N]$.
   (h) The parties invoke **Protocol 3.7** to compute $[\boldsymbol{w'}] = ([\boldsymbol{x}_N]^\top [\hat{\boldsymbol{w}}_{N-1}])\backslash C$.
   (i) The parties compute $[e_N] = [y_N] - [\boldsymbol{w'}]$.
   (j) The parties compute $[\hat{\boldsymbol{w}}_N] = ([\hat{\boldsymbol{w}}_{N-1}] + [e][\boldsymbol{g}])$.
2. The protocol outputs $[C\hat{\boldsymbol{w}_N}]$.

Proof. Security follows since solely secure protocols are used. A formal proof of correctness is not provided. However, throughout the chapter arguments for the correctness has been given. □

5.3 Simulations

In this section **Protocol 5.5** is simulated and the results are compared to simulations of **Algorithm 5.1**. To compare results from both methods, the mean squared error (MSE) is used. The MSE of the parameter estimate at time N is defined as

$$e_{MSE_N} = \frac{1}{q}\sum_{i=1}^{q}(w_i - \hat{w}_{i_N})^2, \quad \text{for } N = 1, 2, \ldots. \tag{5.65}$$

Regarding **Protocol 5.5**, the following parameters are used in all simulations:

- The number of parties, $n = 4$.
- The degree of the polynomial for Shamir's scheme, $t = 1$.
- The cardinality of $\mathbb{F}_p$,
  $p = 136300555243466607821742128462127993362710278088105335847 3$.
- The scaling constant, $C = 2^7$.
- The allowed bit length of the nominator and denominator in **Equation (5.31)** and **(5.33)**, $l = 13$.

A minor discussion of the parameter choices is in order. The first thing that sticks out is how huge p is. The reason for its size, is that l and p depends on each other. Since it is desired to have l large, p must be large. To see why a large l makes p so enormous, recall that secure integer division works by calculating a k-"shifted" approximation of the denominator. To achieve accuracy, it is required that $k \geq l^2 + l$. Putting $k = l^2 + l$, means that $k = 182$, when $l = 13$. Furthermore, it is required to both multiply and divide by $2^k = 2^{182} = 612998216346355543343338810860123673447495648873440870 4$, thus this number must be represented in the finite field, meaning that p must be larger than this number.

Also the scaling constant C should be remarked on. Intuitively, C should be large to obtain as much accuracy from the integer division as possible. However, with the bound that the denominator and nominator can at most be $l = 13$ bits, C multiplied with the input data may not exceed 13 bits. In this way there is also a dependence between C and the input data. Considering the first iteration, where $\boldsymbol{P}_0$ is the identity matrix, the nominator, which contains C^2, is $2^{14}\boldsymbol{x}\boldsymbol{x}^\top$. Assuming the values in $\boldsymbol{x}$ are small, the denominator is $13 + 1$ bits, which in the simulation has proved to work. This fact also explains why the data in the following simulations have been chosen the way they have.

It should be pointed out that the bit-decomposition protocol, **Protocol 3.5**, and the protocol to generate bitwise sharings of a random number, has not been implemented

in the simulation software. Assuming that they would return the correct result once implemented, this should not affect the results in this section.

Refer to the estimate of $\boldsymbol{w}$ at time N obtained from **Algorithm 5.1** as $\hat{w}_N$ and from **Protocol 5.5** as $\hat{w}_{MPC_N}$.

The rest of this section is divided into the following three subsections;

1. Estimation of parameters of a one dimensional system.
2. Estimation of parameters of a two dimensional system.
3. Estimation of parameters of a three dimensional system.

5.3.1 Estimation of Parameters of a One Dimensional System

This section simulates the use of **Protocol 5.5** and **Algorithm 5.1** by estimating the parameters of a one dimensional linear system. The data displayed in **Figure 5.1** are used as test data.

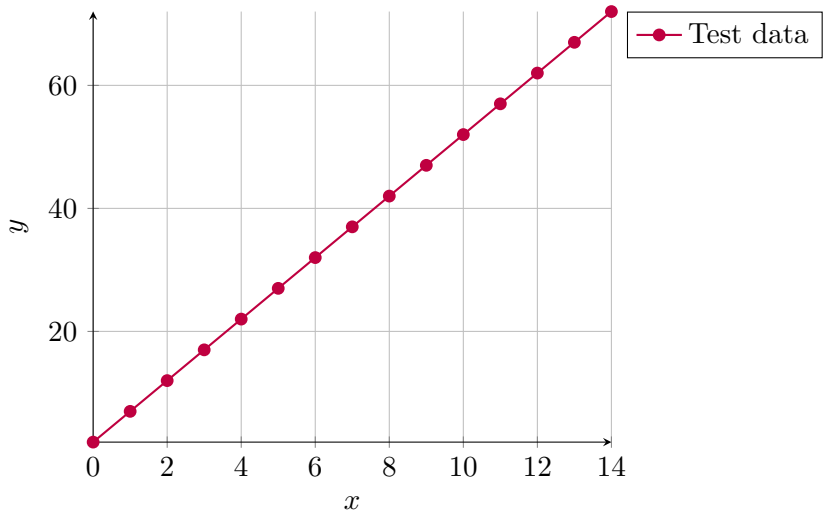


Figure 5.1: Test data, described by the equation $y = 2 + 5x$.

The test data are a line given by the equation

$$y = 2 + 5x. \tag{5.66}$$

Thus, the true parameter vector is $\boldsymbol{w} = [2, 5]^\top$. The observations of input are $x = 0, \ldots, 14$ and the observations of output are calculated using **Equation (5.66)**.

Estimating $\boldsymbol{w}$ using **Protocol 5.5** and **Algorithm 5.1** and the described observations of input and output, gives the e_{MSE_N} for both estimates respectively, as seen in **Figure 5.2**.

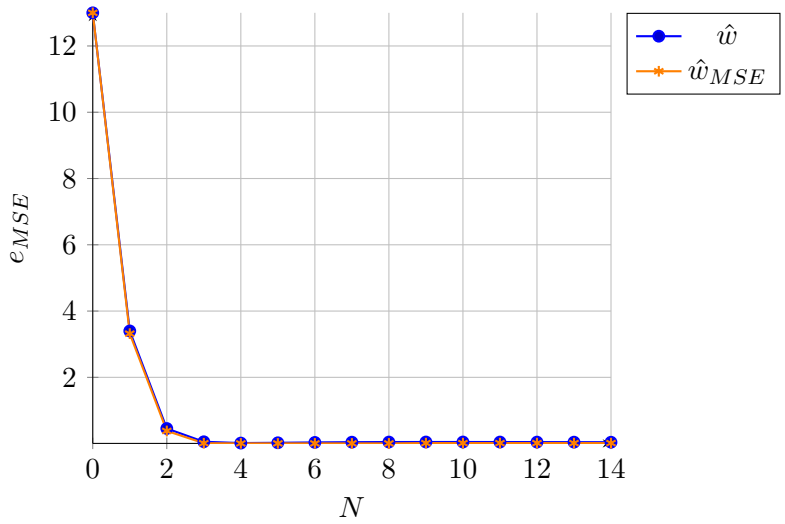


Figure 5.2: Estimating $\boldsymbol{w} = [2, 5]$, using the observations seen in **Figure 5.1**. $\hat{\boldsymbol{w}}$ is the estimate obtained with **Algorithm 5.1**, and $\hat{\boldsymbol{w}}_{MSE}$ is the estimate obtained with **Protocol 5.5**.

As seen in **Figure 5.2**, the results from **Protocol 5.5** are similar to the results obtained from **Algorithm 5.1**.

It is also interesting to test whether **Protocol 5.5** is robust to disturbances. Therefore the observations of x are now changed to being uniformly random integers on the interval $[0, 9]$. For each observation of y a random number is drawn from a Gaussian distribution with mean value zero and variance two, and this is added to the corresponding observation of y. In **Figure 5.3** the observations are marked with a red $*$.

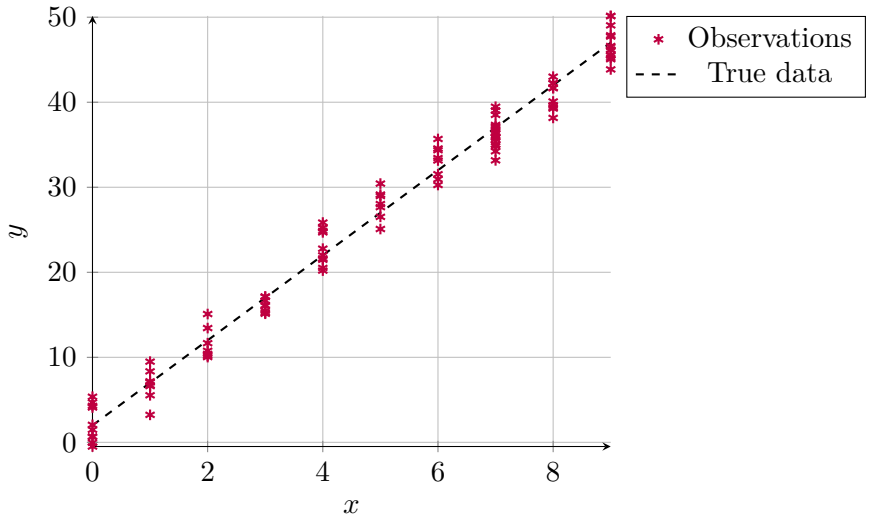


Figure 5.3: Test data, where the observations of y are corrupted by Gaussian disturbances. The data is described by the equation $y = 2 + 5x$ and the disturbances on y are drawn from a Gaussian distribution with mean value zero and variance two.

The estimates of $\boldsymbol{w}$ obtained with **Algorithm 5.1** and **Protocol 5.5** using the observations given by **Figure 5.3**, gives the e_{MSE_N} for both estimates respectively, as seen in **Figure 5.4**.

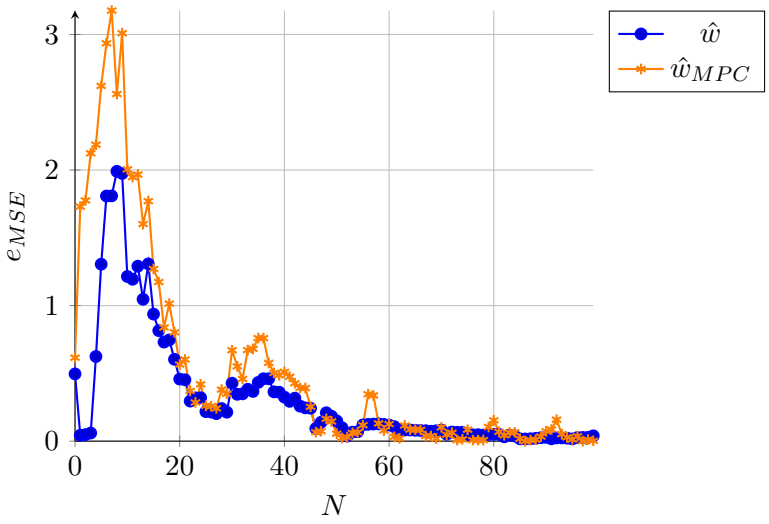


Figure 5.4: Estimating $\boldsymbol{w} = [2, 5]$, using the observations seen in **Figure 5.3**. $\hat{\boldsymbol{w}}$ is the estimate obtained with **Algorithm 5.1**, and $\hat{\boldsymbol{w}}_{MSE}$ is the estimate obtained with **Protocol 5.5**.

As seen in **Figure 5.4**, it seems that **Protocol 5.5** is slightly more sensitive to noise compared to **Algorithm 5.1**. However, the same tendencies are seen for both convergence curves, thus the differences are assumed to be caused by the approximations caused by integer division for **Protocol 5.5**.

5.3.2 Estimation of Parameters of a Two Dimensional System

This section simulates the use of **Protocol 5.5** and **Algorithm 5.1** by estimating the parameters of a two dimensional linear system. The test data is now described by the equation

$$y = 2x_1 + 5x_2, \tag{5.67}$$

thus the true parameter vector is $\boldsymbol{w} = [2, 5]$. As observations of x_1 the numbers $0, \ldots, 19$ are chosen. As observations of x_2 uniformly distributed random numbers on the interval $[0, 9]$ are chosen. The test data are seen in **Figure 5.5**.

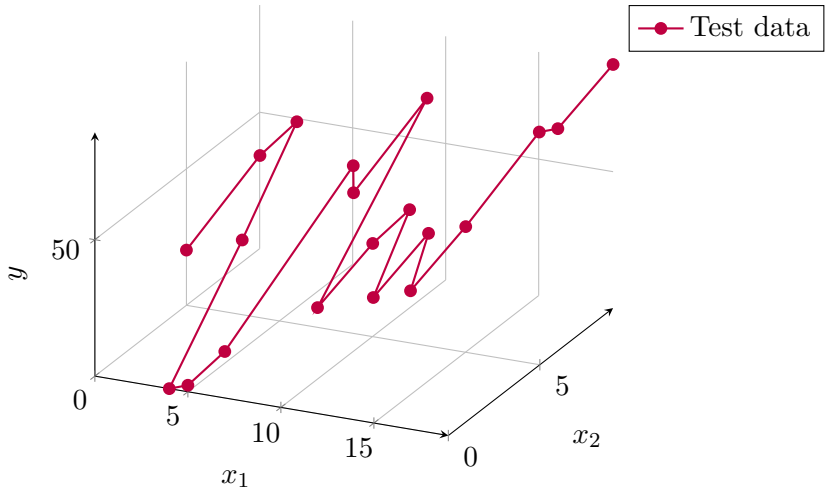


Figure 5.5: Test data described by the equation $y = 2x_1 + 5x_2$.

The estimates of $\boldsymbol{w}$ obtained with **Algorithm 5.1** and **Protocol 5.5** using the observations given by **Figure 5.5**, gives the e_{MSE_N} for both estimates respectively, as seen in **Figure 5.6**.

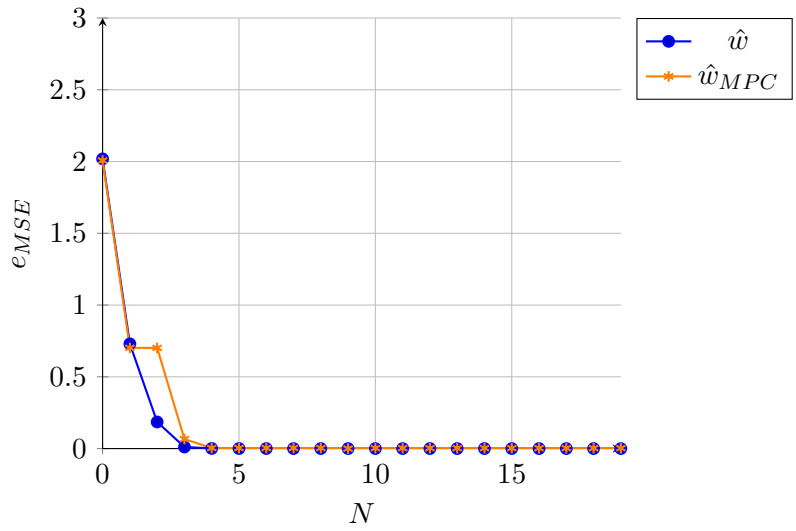


Figure 5.6: Estimating $\boldsymbol{w} = [2, 5]$, using the observations seen in **Figure 5.5**. $\hat{\boldsymbol{w}}$ is the estimate obtained with **Algorithm 5.1**, and $\hat{\boldsymbol{w}}_{MSE}$ is the estimate obtained with **Protocol 5.5**.

As seen in **Figure 5.6**, the obtained results with **Protocol 5.5** are very similar to those obtained with **Algorithm 5.1**. Only the estimate $\hat{w}_{MPC_2}$ sticks out. The reason for this is the truncation happening in **Equation (5.57)**. What concretely happens is that

$$y_2 - (\boldsymbol{x}_2^\top \hat{\boldsymbol{w}}_{MPC_1}) \backslash 2^7 = 0 \tag{5.68}$$

This means that the estimate of the parameter vector is not updated at time $N = 2$, causing the same mean square error for the estimate two times in a row.

It it also interesting to see what happens when the output observations of the two dimensional system is subject to disturbances. The observations of x_1 are now uniformly random integers on the interval $[0, 9]$ and the observations of x_2 are uniformly random integers on the interval $[0, 4]$. The observations of y are given by **Equation (5.67)**. For each observation of y, a random number is drawn from a Gaussian distribution with mean value zero and variance two, which is added to the corresponding observation of y.

The estimates of $\boldsymbol{w}$ obtained with **Algorithm 5.1** and **Protocol 5.5**, giving the described observations, gives the e_{MSE_N} for both estimates respectively, as seen in **Figure 5.7**.

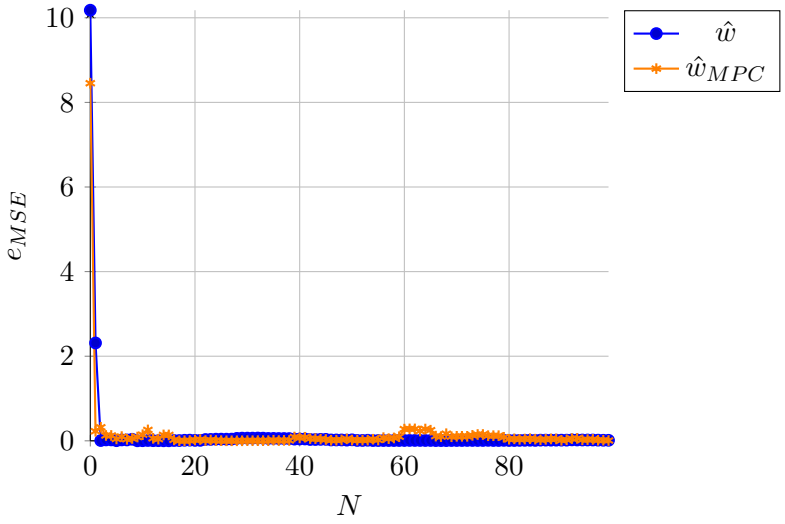


Figure 5.7: Estimating $\boldsymbol{w} = [2, 5]$, using the described observations. $\hat{\boldsymbol{w}}$ is the estimate obtained with **Algorithm 5.1**, and $\hat{\boldsymbol{w}}_{MSE}$ is the estimate obtained with **Protocol 5.5**.

5.3.3 Estimation of Parameters of a Three Dimensional System

This section simulates the use of **Protocol 5.5** and **Algorithm 5.1** by estimating the parameters of a three dimensional linear system. The test data is now described by the equation

$$y = 2x_1 + 5x_2 + 7x_{3_i}, \tag{5.69}$$

thus the true parameter vector is $\boldsymbol{w} = [2, 5, 7]$. The observations for both x_1, x_2 and x_3 are chosen to be uniformly random integers on the interval $[0, 4]$. The observations of y are given by **Equation (5.69)**.

The estimates of $\boldsymbol{w}$ obtained with **Algorithm 5.1** and **Protocol 5.5**, gives the e_{MSE_N} for both estimates respectively, as seen in **Figure 5.8**.

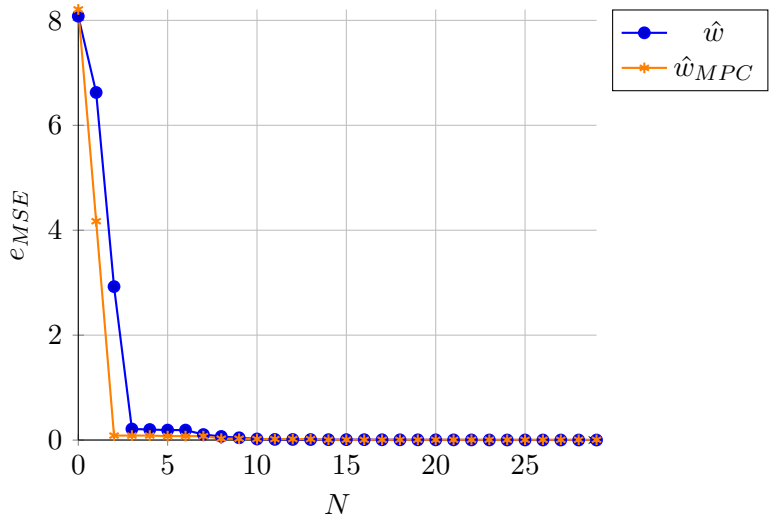


Figure 5.8: Estimating $\boldsymbol{w} = [2, 5, 7]$, using the observations described by **Equation (5.69)**. $\hat{\boldsymbol{w}}$ is the estimate obtained with **Algorithm 5.1**, and $\hat{\boldsymbol{w}}_{MSE}$ is the estimate obtained with **Protocol 5.5**.

As seen in **Figure 5.8**, it seems that the convergence rate for **Protocol 5.5** is slightly faster than for **Algorithm 5.1**. This is considered to be coincidental.

It it also interesting to see what happens when the output observations of the three dimensional system is subject to disturbances. The same observations for x_1, x_2 and x_3 are used. The observations for y are still given by **Equation (5.69)**, however, for each observation of y, a random number is drawn from a Gaussian distribution with mean value zero and variance two, which is added to the corresponding observation of y. The estimates of $\boldsymbol{w}$ obtained with **Algorithm 5.1** and **Protocol 5.5**, gives the e_{MSE_N} for both estimates respectively, as seen in **Figure 5.9**.

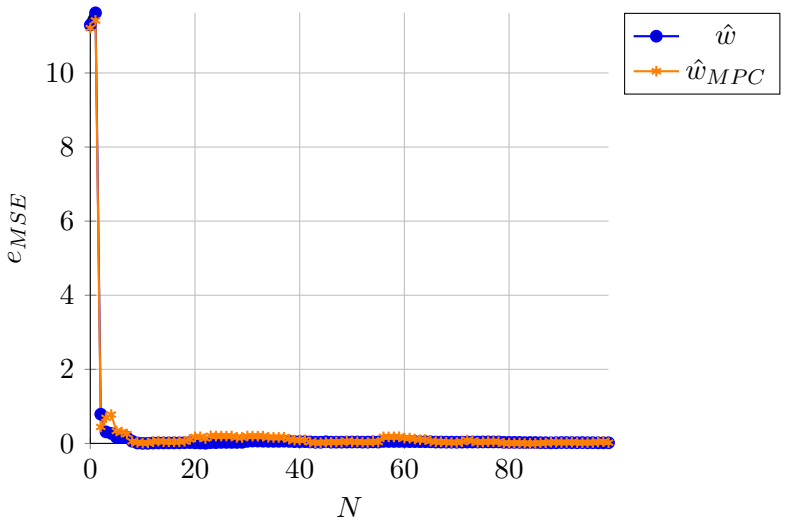


Figure 5.9: Estimating $\boldsymbol{w} = [2, 5, 7]$, using the observations described. $\hat{\boldsymbol{w}}$ is the estimate obtained with **Algorithm 5.1**, and $\hat{\boldsymbol{w}}_{MSE}$ is the estimate obtained with **Protocol 5.5**.

It is concluded that adding more inputs to the linear system, such that more parameters needs to be estimated, does not decrease the accuracy of the parameter estimates.

5.4 Summary

In this chapter a secure MPC protocol for calculating the recursive least squares equation has been proposed. The protocol is a naive conversion of the recursive least squares algorithm. It was found that division of two secrets was needed, and thus a secure protocol for approximating this operation was stated.

The problem that was encountered, was that wrap-arounds has a huge impact when integer division is performed in the protocol. A way to detect these wrap-arounds was found, given that some assumptions for the data holds. Thus, for this particular problem the solutions works, but it is not a general solution. Another problem was that for the the division not to give zero, which would result in divergence of the algorithm, scaling had to be introduced. As was discussed in the chapter, there are some requirements regarding the size of the scaling constant. Obviously, it cannot be too small, since accuracy would be lost, but neither can it be to big since there are restrictions on the size of the nominator and denominator in the secure integer division.

A solution was proposed, and simulations shows that for the test data chosen, the expected results were obtained. However, choosing arbitrarily large data does not work, both due to the scaling constant and the size of the field. Hence, to use this protocol in general there are a few issues that needs attention.

6

Conclusion and Future Work

In this thesis the aim has been to examine the potential in using results from the field of secure multiparty computation to reach privacy preserving control algorithms. In particular, the objective of the thesis has been to answer the following:

- *How can results from secure multiparty computation be used to create privacy preserving control algorithms?*

In the concern of secure MPC computation, security is not only that a secret must remain hidden, but also that malicious behavior cannot corrupt protocol execution. However, generally passive security is considered separately from active security. In this report, the main focus has been on passive security and thus no conclusions can be made regarding active security.

It has been established that secure protocols for various operations such as integer division, bit decomposition, and comparison already exist. Thus, the question has more specifically been how to use the ideas in these existing protocols to put together a secure protocol for entire algorithms, and furthermore whether iterations in the algorithms would effect the result.

In the thesis, three iterative algorithms, namely the gradient descent method, a water pressure control algorithm, and the recursive least squares equations, has been converted into secure protocols. For the creation of these protocols, important topics such as communication and computation complexities has not been considered. Furthermore, there are still some unanswered questions regarding the protocols, which will be discussed in the following section.

The secure version of all three algorithms have been written in software in order to simulate their behavior. The results of the simulations were compared to simulations from a non-secure implementation of the corresponding algorithms. For all three algorithms similar results were obtained from the secure and non-secure implementations.

The main conclusion is that there is a potential for using secure multiparty computation methods to create privacy preserving algorithms. The proposed secure solutions for the three algorithms, stated in this thesis, should not be seen as finished protocols, rather they serve as a proof of concept. In this way, the thesis has shown that there are grounds for further research within this topic.

6.1 Future Work

This section will point out some of the subjects, that has not been attended to in this thesis. The section can in this way work as inspiration to future work within this line of research.

6.1.1 Comparison

Three algorithms are converted to a secure protocol in this report. All three of them uses, in some way or the other, comparison. For this reason, it has become clear that the operation of comparison is used frequently, in all kinds of algorithms. State-of-the-art solutions to a secure comparison protocol, involves bit-decomposition. This imply that instead of each party having two shares (one for each of the secrets), they must each have $2l$ shares, if each secret is l-bit long. Thus, the communication and computation complexity rises significantly, when secrets are bitwise shared. A suggestion for future work is therefore to look into the possibilities of creating a secure comparison protocol, that does not rely on bit-decomposition.

6.1.2 Scaling in Connection With Secure Integer Division

In **Chapter 5**, the recursive least squares equations were converted to a secure MPC protocol. To do this, secure integer division was needed. As was discussed, scaling had to be introduced, because in one equation the denominator was larger than the nominator, which would result in zero, entailing that the algorithm would not converge. There was, however, the problem that the scaling constant can, for accuracy, not be too low, but neither can it be too large because the product between the constant and the data sample has too be below a certain limit. Thus, if the data is secret and nothing can be assumed about the size of the data, how one picks a suitable scaling constant is an open question.

6.1.3 Other Secret Sharing Schemes or Encryptions

In this report all focus has been put on Shamir's secret sharing scheme. It could be interesting to investigate other schemes or encryptions for their properties, to see if there are other schemes that fits the purpose better. For instance, performing comparison using Shamir's scheme to hide the secrets, seems to be almost impossible to do without bit-decomposition. Maybe some other scheme would be better, at least for comparison.

6.1.4 Designing Algorithms for the Purpose of Privacy Preservation

In this thesis, the approach has been to naively convert known algorithms into secure MPC protocols. As was clear in more than one occasion, the explored algorithms was not exactly designed to be computed using finite field arithmetic. It could be interesting to see if other results are achieved if the algorithm is designed to be a secure MPC protocol.

Bibliography

Mussab Alaa, A.A. Zaidan, B.B. Zaidan, Mohammed Talal, and M.L.M. Kiah. A review of smart home applications based on internet of things. *Journal of Network and Computer Applications*, 97(Supplement C):48 – 65, 2017. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2017.08.017. URL http://www.sciencedirect.com/science/article/pii/S1084804517302801.

Gilad Asharov and Yehuda Lindell. A full proof of the bgw protocol for perfectly secure multiparty computation. *Journal of Cryptology*, 30:58–151, 2011.

Zuzana Beerliová-Trubíniová and Martin Hirt. *Perfectly-Secure MPC with Linear Communication Complexity*, pages 213–230. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-78524-8. doi: 10.1007/978-3-540-78524-8_13. URL https://doi.org/10.1007/978-3-540-78524-8_13.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004. ISBN 0521833787.

Ronald Cramer, Ivan Bjerre Damgaard, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 1 edition, 2015. ISBN 978-1-107-04305-3.

Morten Dahl, Chao Ning, and Tomas Toft. On secure two-party integer division. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, pages 164–178, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32946-3.

Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, pages 285–304, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32732-5.

David Fischer and Hatef Madani. On heat pumps in smart grids: A review. *Renewable and Sustainable Energy Reviews*, 70(Supplement C):342 – 357, 2017. ISSN 1364-0321. doi: https://doi.org/10.1016/j.rser.2016.11.182. URL http://www.sciencedirect.com/science/article/pii/S1364032116309418.

Simon Haykin. *Adaptive Filter Theory*. Pearson, 5 edition, 2014. ISBN 978-0-273-76408-3.

Tom Nørgaard Jensen, Carsten Skovmose Kaalesøe, Jan Dimon Bendtsen, and Rafal Wisniewski. Iterative learning pressure control in water distribution networks. 2017.

A. R. Khattak, S. A. Mahmud, and G. M. Khan. The power to deliver: Trends in smart grid solutions. *IEEE Power and Energy Magazine*, 10(4):56–64, July 2012. ISSN 1540-7977. doi: 10.1109/MPE.2012.2196336.

Michael Monagan. Maximal quotient rational reconstruction: An almost optimal algorithm for rational reconstruction. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 243–249, New York, NY, USA, 2004. ACM. ISBN 1-58113-827-X. doi: 10.1145/1005285.1005321. URL http://doi.acm.org.zorac.aub.aau.dk/10.1145/1005285.1005321.

N.T. Nguyen. *Model-Reference Adaptive Control*. Springer, 1 edition, 2018. ISBN 978-3-319-56392-3.

Claudio Orlandi. Is multiparty computation any good in practice? *I E E E International Conference on Acoustics, Speech and Signal Processing. Proceedings*, pages 5848–5851, 2011. ISSN 1520-6149. doi: 10.1109/ICASSP.2011.5947691. Title of the vol.: Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on ISBN: 978-1-4577-0537-3 ISBN: 978-1-4577-0538-0.

Anand Paul, Naveen Chilamkurti, Alfred Daniel, and Seungmin Rho. Chapter 2 - intelligent transportation systems. In Anand Paul, Naveen Chilamkurti, Alfred Daniel, and Seungmin Rho, editors, *Intelligent Vehicular Networks and Communications*, pages 21 – 41. Elsevier, 2017. ISBN 978-0-12-809266-8. doi: https://doi.org/10.1016/B978-0-12-809266-8.00002-8. URL https://www.sciencedirect.com/science/article/pii/B9780128092668000028.

A. M. Pereira, H. Anany, O. P***r***ibyl, and J. P***r***ikryl. Automated vehicles in smart urban environment: A review. In *2017 Smart City Symposium Prague (SCSP)*, pages 1–8, May 2017. doi: 10.1109/SCSP.2017.7973864.

M.M. Prabhakaran and A. Sahai. *Secure Multi-Party Computation*, volume 10 of *Cryptology and Information Security Series*. IOS Press, January 2013.

Nigel P. Smart. *Secret Sharing Schemes*, pages 403–416. Springer International Publishing, Cham, 2016. ISBN 978-3-319-21936-3. doi: 10.1007/978-3-319-21936-3_19. URL https://doi.org/10.1007/978-3-319-21936-3_19.

A

Creating Hyper-invertible Matrices

A hyper-invertible $n \times n$ matrix over a finite field $\mathbb{F}_p$ with $|\mathbb{F}_p| > 2n$, can be constructed by exploiting the linearity of Lagrange interpolation. The method is given in

Construction A.1 (Hyper-invertible Matrix)
Let $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$ be fixed points in $\mathbb{F}_p$. Let $f : {\mathbb{F}_p}^n \to {\mathbb{F}_p}^n$ be a function that maps $(x_1, \ldots, x_n)$ to $(y_1, \ldots, y_n)$, such that $(\beta_1, y_1), \ldots, (\beta_n, y_n)$ are points of the polynomial g of degree $n-1$. The polynomial g is defined by the points $(\alpha_1, x_1), \ldots, (\alpha_n, x_n)$. The matrix $\boldsymbol{M} = \lambda_{i,j=1,\ldots,n}$, where $\lambda_{i,j} = \prod_{k=1,k\neq j}^{n} \frac{\beta_i - \alpha_j}{\alpha_j - \alpha_k}$, that expresses the linear function f, is hyper-invertible.

B

PRE-OR of Integer Without Bit-Decomposition

The work [Dahl et al., 2012, pp. 19-23] introduces a secure way of obtaining the PRE-OR, $[D]$, of an integer, without resorting to bit-decomposition. This protocol is presented here. The protocol consists of many computations, for this reason it is divided into four steps each with a number of substeps. The four steps are;

1. **Add randomness**: Random numbers are added to the input, to retrieve a value, which can be opened without leakage of information.
2. **Create the sequence $\boldsymbol{H} = 1 \ldots, 1, H_{ld}, 0, \ldots, 0$**: Create a bit sequence, $H_{l+1}, \ldots, H_0$, where $H_i = 1$ for $i > l_d$ and $H_i = 0$ for $i < l_d$. By the calculations used to create $\boldsymbol{H}$, it is unknown whether H_{l_d} is 0 or 1.
3. **Calculate the value of H_{l_d}**: By determining H_{l_d}, the desired prefix-OR can be computed from $\boldsymbol{H}$.
4. **Transform $\boldsymbol{H}$ into $\boldsymbol{D}$**: If $H_{l_d} = 1$, $\boldsymbol{H}$ is bit-shifted once to the left and afterwards subtracted from the bit sequence of 1's. If $H_{l_d} = 0$, D is obtained by subtracting $\boldsymbol{H}$ from the bit sequence of 1's.

The substeps of the four steps are seen in **Protocol 5.1**. The derivation of the steps and substeps as well as the proof of security of the whole protocol can be found in [Dahl et al., 2012, pp. 19-23].

Protocol B.1 (PRE-OR of an integer)
Given that the parties hold $[d]$, $[r]$ and $[r]^B$, where d is a $l-$bit integer and r is a $l-$bit unknown random number. $[r]$ and $[r]^B$ are obtained in a preprocessing-phase.
Outputs $[y]$, where $y = \sum_{i=0}^{l_d+1} D_i 2^i$, where $\boldsymbol{D} = [d_{l_d+1}, \ldots, D_0]$ is the prefix-OR of d.

1. **Add randomness**
   (a) Each party, P_i for $i = 1, \ldots, n$, chooses a κ-bit, random value $r_{\top_i}$ and distributes it.
   (b) The parties compute
   $$[e] = [d] + [r] + \sum_{i=1}^{n} \left([r_{\top i}] 2^l \right), \tag{B.1}$$
   and afterwards the parties open $[e]$.
   (c) The parties compute $\bar{e} = e \mod 2^l$.
   (d) The parties invoke **Protocol 3.3** to compute $[B] = \begin{cases} 1 & \text{if } [r]^B < \bar{e} \\ 0 & \text{otherwise.} \end{cases}$.
   (e) The parties compute $[\tilde{e}]^B = B\bar{e} + (1 - B)(\bar{e} + 2^l)$.
2. **Create the sequence $\boldsymbol{H} = 1 \ldots, 1, H_{ld}, 0, \ldots, 0$**
   (a) The parties invoke **Protocol 3.1** to compute $[e']^B = [\tilde{e}]^B \oplus [r]^B$.
   (b) The parties invoke **Protocol 3.2** to define $[E]^B$, where $[E_i] = \vee_{j=i}^{l-1} [e'_j]$.
   (c) The parties compute $[h]^B$ by computing $[h_i] = [r_i] - [r_i][\tilde{e}_i]$ for $i = 0 \ldots, l$.
   (d) Let $\mathbf{1}_l$ denote the bit string of l 1's. The parties compute $[h']^B = [h]^B + \mathbf{1}_{l+1} - [\overrightarrow{E}^1]^N$, where $\overrightarrow{a}^x$ means that a is bit-shifted x times to the right.
   (e) The parties invoke PRE-AND to compute $[H]^B = \text{PRE}_\wedge [h']^B$.
3. **Calculate the value of H_{l_d}**
   (a) The parties invoke **Protocol 3.1** to compute $[H']^B = [H]^B \oplus \mathbf{1}_{l+1}$.
   (b) The parties compute $[\tilde{e0}]^B = [H']^B \wedge [\tilde{e}]^B$, by setting $[\tilde{e0}_i] = [H'_i][\tilde{e}_i]$.
   (c) The parties compute $[r0]^B = [H']^B \wedge [r]^B$, by setting $[r0_i] = [H'_i][r_i]$.
   (d) The parties invoke **Protocol 3.3** to compute $[B1] = \begin{cases} 1 & \text{if } [\tilde{e0}]^B < [r0]^B \\ 0 & \text{otherwise.} \end{cases}$
   (e) The parties compute $[H_{aim}] = 1 - [B1]$.
4. **Transform $\boldsymbol{H}$ into $\boldsymbol{D}$**
   (a) The parties compute $[\bar{D}]^B = [H_{aim}][\overleftarrow{H}^1]^B + (1 - [H_{aim}])[H]^B$.
   (b) The parties compute $[D]^B = \mathbf{1}_{l+1} - [\bar{D}]^B$.
   (c) The parties compute $[y] = \sum_{i=0}^{l+1} [D_i] 2^i$.

The protocol is from [Dahl et al., 2012, pp. 19-23].