Implementation of a Secure Multiparty Computation Protocol

M.Sc.(Tech.) Thesis
University of Turku
Department of Future Technologies
Networked Systems Security
October 2017
Tahsin Civan Mert Dönmez

Supervisors:
Antti Hakkala M.Sc.(Tech.)
Nanda Kumar Thanigaivelan M.Sc.(Tech.)

UNIVERSITY OF TURKU
Department of Future Technologies

Tahsin Civan Mert Dönmez: Implementation of a Secure Multiparty Computation Protocol

M.Sc.(Tech.) Thesis, 122 p., 16 app. p.
Networked Systems Security
October 2017

Secure multiparty computation (SMC) allows a set of parties to jointly compute a function on private inputs such that, they learn only the output of the function, and the correctness of the output is guaranteed even when a subset of the parties is controlled by an adversary. SMC allows data to be kept in an uncompromisable form and still be useful, and it also gives new meaning to data ownership, allowing data to be shared in a useful way while retaining its privacy. Thus, applications of SMC hold promise for addressing some of the security issues information-driven societies struggle with.

In this thesis, we implement two SMC protocols. Our primary objective is to gain a solid understanding of the basic concepts related to SMC. We present a brief survey of the field, with focus on SMC based on secret sharing. In addition to the protocol implementations, we implement circuit randomization, a common technique for efficiency improvement. The implemented protocols are run on a simulator to securely evaluate some simple arithmetic functions, and the round complexities of the implemented protocols are compared. Finally, we attempt to extend the implementation to support more general computations.

Keywords: MPC, SMC, secure multiparty computation, implementation, secret sharing

Acknowledgements

I would like to thank Antti Hakkala, M.Sc.(Tech.), and Nanda Kumar Thanigaivelan, M.Sc.(Tech.), for being my supervisors. Many thanks go to Josefine Wegert for being a good friend, and for proofreading the thesis. Last but not least I thank my family for their support and encouragement.

Contents

List of Figures

List of Tables

List of Abbreviations

APT	**A**dvanced **P**ersistent **T**hreat
AS	**A**utonomous **S**ystem
BGP	**B**order **G**ateway **P**rotocol
CDCF	**C**ross-**D**omain **C**ooperative **F**irewall
CEAS	**C**ircuit **E**valuation (with) **A**ctive **S**ecurity
CEPS	**C**ircuit **E**valuation (with) **P**assive **S**ecurity
CR	**C**ircuit **R**andomization
FHE	**F**ully **H**omomorphic **E**ncryption
IP	**I**nternet **P**rotocol
ISP	**I**nternet **S**ervice **P**rovider
MAC	**M**assage **A**uthentication **C**ode
MPC	**M**ulti**p**arty **C**omputation
OT	**O**blivious **T**ransfer
P2P	**P**eer-**t**o-**P**eer
PRSS	**P**seudo**r**andom **S**ecret **S**haring
PRZS	**P**seudo**r**andom **Z**ero **S**haring
PoK	**P**roof **o**f **K**nowledge
SFE	**S**ecure **F**unction **E**valuation
SHE	**S**omewhat **H**omomorphic **E**ncryption

SMC	**S**ecure **M**ultiparty **C**omputation
VPN	**V**irtual **P**rivate **N**etwork
VSS	**V**erifiable **S**ecret **S**haring
ZKP	**Z**ero-**K**nowledge **P**roof
ZKPoK	**Z**ero-**K**nowledge **P**roof **o**f **K**nowledge
ZKPoPK	**Z**ero-**K**nowledge **P**roof **o**f **P**laintext **K**nowledge

Chapter 1

Introduction

We live in the Information Age; an era marked by the central role of information in most societies. Information is created, transfered, stored, and processed at an ever increasing rate. An estimated capacity of more than 10^{20} bytes of storage and more than 10^{21} bytes of transmission per second has been reached almost a decade ago, taking into account only general-purpose computers [66]. A 2013 article [98] reports that 90 percent of world's data had been generated within the previous two years. It is estimated that, a handful of big organizations each process more than 10^{16} bytes of data per day, and the amount of data stored by hoarders such as Google[1] and NSA[2] is estimated to be on the scale of 10^{19} bytes [68].

Information with varying degrees of confidentiality have become assets for various kinds of organizations. Organizations use the available data to improve their services and to make better decisions. However, these benefits do not come without a cost. Information-driven societies have seen the emergence of new problems such as large scale data breaches [95], mass surveillance [90], and even mass manipulation [55].

A blog post titled '*Crypto is Dead; Long Live Crypto!*' [99] points out that, one cannot

[1]https://www.google.com/intl/en/about/

[2]https://www.nsa.gov/

obtain an ultimate solution to such problems through traditional uses of cryptography, such as encryption and authentication. The very reason behind the collection of data is that, it will be processed at some point. As we do not know yet how to efficiently operate on encrypted data, whatever we encrypt to protect, will have to be decrypted before it can be of any use. Moreover, with the emergence of advanced persistent threats (APTs), it is not unreasonable to assume that some attacker is already inside the security parameter, possibly holding the decryption key. Secure multiparty computation (SMC, or secure MPC) is an alternative way of using cryptography, that can address these issues. SMC allows a set of parties to jointly compute a function on private inputs such that, they learn only the output of the function (i.e. the privacy of the inputs are preserved) and correctness of the outputs is guaranteed, even when a subset of the parties is controlled by an adversary. In typical use cases of SMC, parties involved are mutually distrustful, but one can also imagine the case of multiple machines owned by a single party, performing SMC to collectively decrypt and process confidential data. No single machine would have the key, and no single machine would see the plaintext. Now it would not be enough for the APT to compromise a single machine holding the decryption key, but every single one of the machines would have to be compromised.[3]

In addition to solving the dilemma of keeping data in a useful form or keeping data in an uncompromisable form, SMC also gives new meaning to data ownership and data sharing. With regard to data sharing, we are faced with another dilemma: On the one hand, people would like to make their data available in order to benefit from customized and/or improved services, or in order to contribute to advancing research or improving administrative decisions,[4] but on the other hand, people often want to keep their data

[3]This is true, assuming that the SMC protocol used can handle a dishonest majority. Otherwise compromising a smaller subset of the machines might suffice.

[4]Consider, for example, a citizen making her health care data available to a medical research company.

private and under their control. SMC allows input providers to make their data available as input to computations without ever revealing them. As a consequence, this way of sharing data has the beneficial properties of reversibility and controllability. For example, the access control system proposed in [112] leverages these two properties, and allows the input provider to control how her data is used, and block access at a later time if she wishes so. In short, the input provider retains the ownership of her data.

While the promises of SMC might look too good to be true at first sight, its theoretical foundations are solid. SMC have been rigorously studied since the early 80s. Keller et al. [75] note that, ever more efficient SMC protocols have been discovered in the last decade, changing the status of SMC from a purely theoretical study, to a research field with practical applications. Following figures demonstrate this rapid move: Performing a single AES block encryption (via SMC) took 60 seconds in 2009, only hundredth of a second in 2013 [99], and a 2016 article [75] reports a 100 to 1000 fold throughput increase compared to what was possible in 2013. Moreover, these promising results are for protocols that work with strict and realistic adversary models: As long as one of the participants is honest, any number of players that deviate from the protocol in arbitrary ways can be tolerated.

1.1 Thesis Structure

In this thesis, we implement two SMC protocols: P_{CEPS} [41, p.38] and P_{CEAS} [41, p.117].[5] For P_{CEAS}, we also implement circuit randomization [13], which is a common technique for efficiency improvement. Our main objective is to gain an understanding of

[5] P_{CEPS} and P_{CEAS} stand for *Protocol (for) Circuit Evaluation (with) Passive Security* and *Protocol (for) Circuit Evaluation (with) Active Security*, respectively. Formal definitions of these protocols are given in Chapter 4, after the preliminaries.

the basic ideas and concepts related to SMC, and to solidify that understanding by means of the implementations. Additionally, we aim to gain insight into the inner workings of state-of-the-art SMC protocols. Finally, we aim to gain familiarity with the past and current state of the research field.

The rest of the thesis is structured as follows. Chapter 2 briefly covers some concepts, techniques, and cryptographic primitives that are needed for the presentation of the following chapters. Chapter 3 presents a short survey of the research field, with focus on SMC based on secret sharing. Some important theoretical results, and a few selected SMC implementations and applications are mentioned. Chapter 4 provides an overview of the implemented protocols P_{CEPS} and P_{CEAS}, and summarizes the results of formal security analysis of these protocols. Chapter 5 describes in detail the protocol implementations and the implementation of the simulator, which provides an environment where the protocols can be executed. In Chapter 6, we first securely evaluate a few simple arithmetic functions to observe and to compare the round complexities of the implemented protocols. Next, the implementation is extended to support more general computations. In particular, the computing parties are provided with the ability to remember shares from previous computations, and a custom arithmetic circuit is built for a specific computation, namely, secure comparison of integers. In Chapter 7, we conclude the thesis with closing remarks.

Chapter 2

Preliminaries

This chapter briefly covers the concepts, techniques, and cryptographic primitives that are needed for the presentation of following chapters. The presentation of Section 2.1 is mostly based on the related sections from [41, 17]. The presentations of Sections 2.2 - 2.6 follow closely [41].

2.1 Secure Multiparty Computation (SMC)

Secure multiparty computation [109, 108] allows a set of possibly mutually distrusting parties to perform computations on private inputs, such that the following two properties are satisfied even when a subset of the players are dishonest[1]:

- *Correctness:* Computation yields the correct result.
- *Privacy:* No new information is released other than the result of the computation.

In this section, we make this definition more concrete.

[1]Some alternative definitions require an additional property: fairness or robustness. These properties are defined in Section 2.1.4.

2.1.1 Players and Other Parties

The parties who participate in the computation will be referred to as *players* (or *computing parties*). Other parties of interest are *input providers* and *data users*. An input provider provides one or more of the private inputs used in the computation. Computation result is opened to the data users. Input providers and data users will be collectively referred to as *users*.

2.1.2 Network Model

Existence of secure peer-to-peer channels between every pair of players, and between every player-user pair is assumed. Additional assumptions might be made, such as the existence of a consensus broadcast channel (See Section 2.2).

Synchronous Networks

Entities on the network have synchronized clocks,[2] which allow collective definition of *communication rounds*. A communication round is defined by its start time and duration. Communication proceeds in rounds and there is an upper bound on message delivery time. The upper bound on message delivery time constrains the definitions of the communication rounds, such that the following condition always holds: any honest player, who tries to deliver a message during round r, will always have it delivered at the beginning of round $r + 1$. This constraint allows to distinguish an honest player, who experiences network delay, from a dishonest player, who does not send her messages at the time specified by the protocol. Communication on synchronous networks is inherently inefficient in the sense that, each round has to last at least as long as the worst-case delivery time.

[2] In the current context, entities on the network are players, input providers, and data users.

Asynchronous Networks

Messages may be delivered out of order, but any message that is sent will eventually be delivered.

2.1.3 Adversary Model

A concrete adversary model addresses the following considerations about an adversary.

Active Adversary vs. Passive Adversary

Dishonest behaviour is modeled by assuming an adversary *corrupting* a subset of the players. The degree of control the adversary has over a corrupted player defines the corruption. Adversaries can be classified according to the corruption they cause. A *passive* adversary can read the internal states of corrupted players. An *active* adversary can, in addition to reading internal states, make the corrupted parties deviate arbitrarily from the protocol. A passive adversary is sometimes referred to as a *semihonest* adversary or *honest but curious* adversary in the literature. An active adversary is sometimes referred to as a *malicious* adversary in the literature.

Static Adversary vs. Adaptive Adversary

An *adaptive* adversary is allowed to choose the players to corrupt during protocol execution, based on information gathered up to that point. A static adversary cannot adapt her choice (of players to corrupt) to information gathered during the computation, and is assumed to make the choice before protocol execution starts.

Computational Power

An adversary may be, for example, *unbounded* or *polynomially bounded* in terms of computational power.

Message Scheduling Capabilities

Rushing is an adversary behaviour defined in the context of synchronous networks, where an adversary may be able to delay the sending of messages *within a round*. A rushing adversary is allowed to prepare the messages to be sent by corrupted players at the end of round $r+1$, based on information contained in all the messages sent to corrupted players at the end of round r.

In asynchronous networks, an adversary may be able to delay the sending of messages *arbitrarily*. This allows the adversary to use the information read from honest players' messages, in the process of constructing her own messages.

Corruption Capabilities and Adversary Structures

In *threshold-t* model, an adversary can corrupt up to t players. It is sometimes useful to consider more powerful models, which take into account other factors, such as the differences between players in terms of susceptibility to corruption. For example, on a network with N machines $\mathbf{M_i}$, it might take twice as much resources to compromise one of $\mathbf{M_1}$ or $\mathbf{M_2}$, compared to any one of the others.[3] It could be useful to be able to express that, an adversary is capable of compromising any three of the machines $\mathbf{M_3}, ..., \mathbf{M_N}$, or one of $\mathbf{M_1}$, $\mathbf{M_2}$ and one other weaker machine, but not $\mathbf{M_1}$ and $\mathbf{M_2}$ together.

Threshold-t model can be generalized by considering a set $\mathcal{A}$ of a subset of players instead of a single threshold value [41]. Set $\mathcal{A}$ contains all possible subsets of players

[3]Perhaps, $\mathbf{M_1}$ and $\mathbf{M_2}$ are better protected because they hold pieces of a master key.

that can be corrupted by the adversary. The special case of all subsets of size less than t corresponds to the threshold-t model. A feasibility result regarding SMC, which will be mentioned in Section 3.2.2, demonstrates the greater power of this more general adversary model. When expressed in the threshold-t model, the feasibility result tells that, for a particular number of players n_0, it is not possible to have active, information theoretic security for more than t_0 corruptions. On the other hand, when expressed in terms of the more general adversary structure, it is possible to have active security even when subsets of size greater than t_0 are corrupted. The reason why greater than t_0 corruptions can be tolerated is that, the general adversary model, with its superior expressive power, allows one to express that certain players cannot be simultaneously corrupted.

2.1.4 Definition of Security

It is possible to describe security in terms of a set of desirable properties. Two properties, *correctness* and *privacy*, are defined in Section 2.1. Two additional properties are defined below:

- *Robustness:* The adversary cannot prevent honest parties from receiving the computation result (for example, by aborting prematurely).
- *Fairness:* Once the adversary receives information about the computation result, she can no longer prevent honest parties from also receiving it. In other words, fairness is a weaker version of robustness, which requires the robustness property to hold only in cases where the adversary learns the computation result.

This is not a full list of properties one can come up with. With property-based security definitions, there are concerns such as the security definition being application dependent and the possibility of missing some properties [82].

Simulation-Based Security

The proof of a cryptographic protocol often takes place in a *game-based* setting, and in case of computational security, it involves a reduction to an assumed hard problem. Formal analysis of SMC protocols, on the other hand, are often formalized in a *simulation-based* setting [61, 35]. The notion of simulation-based security definition is also known as the *real/ideal paradigm*.

In the ideal model, input providers send their private inputs to a trusted third-party, who performs the computation and shares the result with the data users. In the real model, players run the SMC protocol to perform the computation. Security of an SMC protocol π is defined with respect to an *ideal functionality* (or *intended functionality*) $\mathbf{F}$, which should be regarded as the specification of π. This specification determines the behaviour of the trusted third-party. Proving the security of π is equivalent to finding, for every possible adversary, a simulator $\mathbf{S}$ interacting with $\mathbf{F}$, such that the output of $\mathbf{S}$ is indistinguishable from the output of π.[4] Existence of such simulators can be put into words in the following alternative ways:

- If an attack exists in the real model, it also exists against the trusted third-party (represented by $\mathbf{F}$) in the ideal model. Since no attacks exist against the trusted third-party (by definition), no attack exists against π.
- The adversary can simulate the real model, given the information she obtains in the ideal model. Hence, against π in the real model, the adversary cannot achieve more than what she could achieve against the trusted third-party in the ideal model.
- π is proved to be secure with respect to $\mathbf{F}$.

[4]Simulator $\mathbf{S}$ can be thought of as the *ideal model adversary*.

- π is at least as secure as $\mathbf{F}$.
- π securely implements $\mathbf{F}$.

Tolerable Adversary

Simulation-based definition of security is based on the indistinguishability of the real model from the ideal model. Consequently, computational power of the tolerable adversary must be specified for a precise definition of security. The following definitions are concerned with the computational power of the tolerable adversary [17]:

A protocol is *information-theoretically secure* if it tolerates a computationally-unbounded adversary. An information-theoretically secure protocol is

- *perfectly secure*, in case of perfect indistinguishability.
- *statistically secure*, if a negligible probability of error is allowed.

A protocol is *computationally secure* (or *cryptographically secure*), if it can only tolerate a computationally-bounded adversary.

The following definitions are concerned with other capabilities of the tolerable adversary [17]:

- A protocol is *actively secure* if it tolerates an active adversary, and *passively secure* if it can only tolerate a passive adversary.
- A protocol is *adaptively secure* if it tolerates an adaptive adversary, and *statically secure* if it can only tolerate a static adversary.

2.2 Consensus Broadcast

Let's assume that, at some point during its execution, a protocol expects the participating parties to broadcast a message. If party P_i is corrupted by an active adversary, in general, there is no guarantee that all other parties will receive the same broadcast message from P_i. A stronger version of broadcast, called *consensus broadcast* (or *Byzantine agreement*), guarantees that all honest parties receive the same message even when the sender is actively corrupted. Cramer et al. [41] note that, consensus broadcast can be considered as a special case of secure function evaluation, where a single player provides the input m, and every other player receives m as the computation result.

2.3 Circuit Evaluation

Most SMC protocols perform secure evaluation of either Boolean circuits, or arithmetic circuits, over a finite ring or field [75]. A circuit is a collection of gates and connecting wires. A circuit can be thought of as an acyclic directed graph [41], where gates correspond to nodes, and wires correspond to edges. A circuit can have any number of incoming wires, onto which the inputs are assigned prior to evaluation. A circuit can have any number of outgoing wires, onto which the evaluation results are assigned at the end of evaluation. Each gate of the circuit has a specific number of incoming wires which depend on the type of gate, and any number of outgoing wires. When the computation of a gate is complete, result is assigned to all of its outgoing wires. Secure evaluation of a circuit is specified by an SMC protocol.

The implemented protocols P_{CEPS} and P_{CEAS} evaluate arithmetic circuits.

2.3.1 Boolean Circuits

A set of Boolean functions is functionally complete, if every Boolean function can be expressed in terms of its elements. For example, each one of the sets $\{\wedge, \oplus\}$, $\{\wedge, \neg\}$, and $\{|\}$[5] are functionally complete.

It is advantageous to use a single type of gate ($\mathbf{NAND}$ gate) for building hardware circuits. However, in the context of SMC, Boolean circuits are often built out of $\mathbf{AND}$ and $\mathbf{XOR}$ gates, as this makes optimizations such as *Free XOR* possible [78].

It follows from functional completeness that, any function that is feasible to compute, can be specified as a polynomial-size Boolean circuit [41]. Hence, the ability to securely evaluate Boolean circuits, implies the ability to securely evaluate functions. In the context of SMC, the ideal functionality corresponding to this ability is referred to as *secure function evaluation* (SFE).

General Secure Computing

It is possible to extend the ability to securely evaluate functions, to more general secure computations. A relevant result from the field of computability theory is that, while a Boolean circuit is only capable of computing a single Boolean function on a fixed number of inputs, a (uniform) family of circuits is capable of computing the unbounded set of functions computed by a Turing machine [94].

Cramer et al. [41] note that the SFE ideal functionality can be extended to a more general *reactive functionality*, as follows. Reactive functionality would use the ideal functionality SFE to securely evaluate functions depending on both the internal state and inputs from players. In addition to delivering the outputs to the players, it would also update the internal state.

[5] $|$ is the *Sheffer stroke*, or $\mathbf{NAND}$.

While theoretically possible, attaining the ability to perform general secure computations involves practical challenges. For example, because of the particular way correctness of an evaluation is verified, SPDZ[6] does not allow the use of private secret shared data in subsequent function evaluations. Consequently, SPDZ does not support reactive functionalities. An SMC protocol based on SPDZ, whose online phase supports reactive functionalities, is proposed in [49]. Another example of such practical considerations comes from Zyskind et al. [112], who report that their Turing-complete SMC interpreter handles conditional statements depending on secret values by evaluating both branches.

2.3.2 Arithmetic Circuit

It is argued in the previous section that, general computations can be achieved through evaluation of Boolean circuits. In order to make the same claim for arithmetic circuits, it is enough to notice that any Boolean circuit can be simulated by arithmetic operations in the underlying field [41]. Boolean values *true* and *false* can be encoded as 1 and 0, respectively. Then, the negation of a bit b ($\neg b$) can be simulated by $1 - b$,[7] and the logical conjunction of two bits b_1 and b_2 ($b_1 \wedge b_2$) can be simulated by $b_1 \cdot b_2$. Because the set $\{\wedge, \neg\}$ is functionally complete, any Boolean function can be simulated.

Arithmetic circuits are made up of addition ($\mathbf{ADD}$), multiply-by-constant ($\mathbf{CMUL}$), and multiplication ($\mathbf{MUL}$) gates. While $\mathbf{ADD}$ and $\mathbf{MUL}$ gates have two input wires, a $\mathbf{CMUL}$ gate have a single input wire and the gate itself is labeled by a constant $\alpha \in \mathbb{F}_p$, where $\mathbb{F}_p$ is the underlying field and α is used as one of the multiplicands.

A large part of the cost associated with the evaluation of arithmetic circuits comes from computation of $\mathbf{MUL}$ gates. For SMC protocols that work in the honest majority

[6]SPDZ is an SMC protocol, which is discussed in Section 3.2.3.

[7]$1 + b \cdot (-1)$

setting, such as P_{CEPS} and P_{CEAS}, this cost is mostly in the form of network communications, whereas for protocols that work in the *dishonest majority* setting,[8] cost of local computations associated with the use of expensive public-key cryptography tend to dominate. An extreme case on this end is the trivial SMC protocol based on fully homomorphic encryption, which is outlined in Section 3.1.3.

2.3.3 Boolean Circuits vs. Arithmetic Circuits

SMC comes in two flavours: SMC based on secret sharing and SMC based on garbled circuits.[9] Usually the former evaluates arithmetic circuits, and the latter evaluates binary circuits.[10] Keller et al. [75] note that, protocols for secure evaluation of Boolean circuits can usually do with only symmetric cryptography, whereas protocols evaluating arithmetic circuits usually require more expensive public key cryptography techniques, except in the honest majority setting. However, SMC protocols based on secret sharing, which evaluate arithmetic circuits, have the desirable property that computation of $\mathbf{ADD}$ and $\mathbf{CMUL}$ gates come almost for free. Especially when the computation heavily involves arithmetics over a large field, which is often the case for application areas such as benchmarking and auctions [21], arithmetic circuit is the natural choice. Of course, there are also cases where Boolean circuits are the most compact way to express the desired computation [50]. For arithmetic circuits, by using a large enough field, in other words by choosing a p that is large compared to the inputs, one can avoid modular reductions. As a result, additions and multiplications in $\mathbb{F}_p$ directly correspond to integer addition and multiplication.

[8] In the dishonest majority setting, up to $n-1$ out of n of the players may be corrupted by an adversary.

[9] More is said about this categorization in Section 3.2.

[10] An exception is the arithmetic variant of Yao's construction introduced in [6].

2.4 Secret Sharing

Secret sharing was first introduced back in 1979 [97, 25], and since then it has become a fundamental cryptographic primitive [41]. A (t, n)-threshold secret sharing scheme divides a secret into n pieces called *shares*, such that, less than t shares reveal no information about the secret, but with t or more shares, the secret can be reconstructed.

2.4.1 Shamir's Secret Sharing Scheme

Shamir's secret sharing scheme [97] is based on polynomials over a finite field $\mathbb{F}$. In order to share a secret $s \in \mathbb{F}$, one first chooses a random polynomial $f_s(x) \in \mathbb{F}[x]$ of degree at most d such that $f_s(0) = s$. Then, to every other player P_j, the share $s_j = f_s(j)$ is privately sent. In accordance with the notation in [41], the set of shares will be denoted by $[s; f_s]_d$. Let $t = d + 1$ be the threshold and n be the total number of players. The scheme is a (t, n)-threshold scheme. It follows from Lagrange interpolation that, any set of d or fewer shares contains no information on secret s, and s can be reconstructed from any t or more shares [41]. For an SMC protocol based on Shamir's secret sharing, such as P_{CEPS} and P_{CEAS}, the former yields an upper bound on the number of corrupted players, and the latter yields a lower bound on the number of honest players (See Section 5.2.3).

Perfect security of P_{CEPS} and P_{CEAS} stems from the fact that, the secret sharing scheme perfectly hides a secret. Communication-free computation of $\mathbf{ADD}$ and $\mathbf{CMUL}$ gates stems from the linearity of the secret sharing scheme. These are examples that demonstrate the intimacy between an SMC protocol (based on secret sharing) and the underlying secret sharing scheme. SMC based on secret sharing is sometimes referred to as *share computing* in the literature [30].

Lagrange Interpolation

If $f(x)$ is a polynomial of degree at most d over a field $\mathbb{F}$, and if C is a subset of $\mathbb{F}$ with $|C| = d + 1$, then

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

where

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

are the Lagrange basis polynomials.

Recombination Vector

It follows from Lagrange interpolation that, for all polynomials $f(x)$ of degree at most d and for $n < |\mathbb{F}|$, there exists a vector $\mathbf{r} = (r_1, ..., r_n)$ such that

$$f(0) = \sum_{i=1}^{d+1} r_i f(i)$$

where $r_i \in \mathbb{F}$ is given by

$$r_i = \delta_i(0)$$

This vector $\mathbf{r}$ is referred to as the recombination vector [41]. In relation to the SMC implementations presented in this work, recombination vector depends only on the protocol parameters p, (t, n), and on the subset C, which corresponds to the set of players whose shares are combined in order to recover the secret $s = f(0)$. Hence, all players can independently calculate the recombination vector.[11]

[11]It is not a requirement that the players agree on the set of shares to combine, and hence they could in theory use different recombination vectors. As far as the implementations presented in this work are concerned, all honest players agree on the set of shares to combine.

2.4.2 General Linear Secret Sharing Schemes

Cramer et al. [41] note that, basing SMC protocols on more general linear secret sharing schemes (than Shamir's Secret Sharing Scheme) might bring the following benefits:

- handling general adversary structures
- removing the dependency between the number of players and the size of the field underlying the secret sharing scheme[12]

Shamir's secret sharing scheme, being a threshold scheme, can handle the threshold-t model naturally, but it cannot handle more general adversary structures as it is. Some level of generalization can be achieved externally, for example, by generating more shares than there are players, and distributing different number of shares to different players.

2.5 Verifiable Secret Sharing

The cryptographic primitive Verifiable Secret Sharing (VSS) was first introduced in [38]. VSS adds the following property on top of the secret sharing primitive: Each player can verify that, every player received a valid share of the secret,[13] while possessing no knowledge about the secret.

In context of protocol P_{CEAS}, VSS is used to prevent corrupted players from distributing inconsistent shares. VSS is implemented as suggested in [41]. As with ordinary secret sharing, a player who wants to share a secret s chooses a random polynomial. But as an additional step, the player commits to each coefficient of this polynomial. Linearity allows construction of commitments to the shares generated from the polynomial. Finally,

[12]Shamir's secret sharing scheme imposes the constraint $n < |\mathbb{F}|$.

[13]In other words, the distributed set of shares are consistent, and the received share is an element of that set.

instead of privately sending shares, commitments to shares are transferred by running a secure protocol. These steps are explained in detail in Section 5.4.4. The point to make here is that, in this particular construction, VSS is built on top of commitment.

2.5.1 Commitment via Redundant Sharing

In protocol P_{CEAS}, commitment to a value $a \in \mathbb{F}$ is achieved by secret sharing a *along with* redundant information to be used in consistency checks [41]. Instead of a univariate polynomial as in Shamir's secret sharing, a symmetric bivariate polynomial

$$f_a(x, y) = \sum_{i,j} \alpha_{i,j} x^i y^j$$

is sampled, where

$$f_a(0, 0) = \alpha_{0,0} = a$$

For the commitment, a player P_k receives not an ordinary share, but a whole univariate polynomial

$$f_k(x) = f_a(x, k) = \sum_i (\sum_j \alpha_{i,j} k^j) x^i$$

which is referred to as a *verifiable share* in the following chapters. In accordance with the notation in [41], a set of verifiable shares will be denoted by $[[s; f_s]]_d$.

Protocol P_{CEAS} leverages the symmetric property of the sampled polynomials $f_a(x, y)$ to allow consistency checks (See Section 5.4.4).

2.6 Preprocessing Model

An SMC protocol in the preprocessing model assumes a *trusted dealer*, who knows nothing about the function to be evaluated and the private inputs to be used in the evaluation [50]. Trusted dealer supplies *raw material* for the computation before it starts. An SMC protocol is run in the *online phase* to carry out the secure computation, making use of the supplied raw material. Trusted dealer is implemented by another secure protocol, which is run in the *preprocessing phase* (or *offline phase*).[14] The purpose of this is to push part of the work, that is costly in terms of local computations or network communications, to the preprocessing phase. Having an efficient online phase increases the practical value of an SMC system. One practical concern is that one would not want players come and go during actual function evaluation. Such circumstances might be more tolerable during a preprocessing phase. Knowing the list of participants and the time of computation (i.e. time of running the online phase) well in advance, can make a preprocessing phase more feasible, and this is the case for many application scenarios of SMC.

2.6.1 Circuit Randomization

Circuit randomization is a technique for efficiency improvement, introduced by Beaver in [13]. It involves secure computation of multiplication triples with randomly picked multiplicands in the preprocessing phase, and a trick to make use of those triples for more efficient computation of MUL gates in the online phase.[15] The trick involves the use of the identity

$$a \cdot b = x \cdot y + e \cdot b + d \cdot a - e \cdot d$$

[14]The reason it is called an offline phase is not because it does not require network communication, but because it can be carried out off the peek hours [102].

[15]Multiplication triples are sometimes referred to as Beaver triples.

where (x, y) are multiplicands of a triple generated in preprocessing, (a, b) are the inputs of a multiplication gate being computed during online phase, $e = a - x$, and $d = b - y$. Linearity of the commitment or encryption scheme is leveraged in order to make multiplication triples $(a, b, a \cdot b)$, out of multiplication triples generated in the preprocessing phase.[16]

Section 5.4.6 explains how P_{CEAS} uses this technique, and in Section 6.1.2, the efficiency increase of the online phase is observed and quantified. However, circuit randomization is not just about moving cost from online phase to preprocessing phase [41]. It also opens the way for an overall efficiency increase, as it is possible to come up with more efficient ways of generating a batch of multiplication triples with random multiplicands, compared to the efficiency of generating them individually at each $\mathbf{MUL}$ gate. For example, Hyperinvertible Matrices [18, 17] and Pseudorandom Secret Sharing (PRSS) [42, 47] can be used for efficient generation of *random sharings*,[17] and Packed Secret Sharing [56] can be used for generation of multiplication triples in a parallel fashion. An implementation of a preprocessing phase using PRSS and Hyperinvertible Matrices can be found in project VIFF (See Section 3.2.3).

The preprocessing phase implementation in this thesis work does not use any techniques for efficient generation of multiplication triples. In this case, moving cost from online phase to preprocessing phase is indeed the only benefit of using circuit randomization. Basically, for each triple, committed shares to two random sharings are generated, and these are multiplied in a way similar to what is done for computation of multiplication gates in a run without circuit randomization. Since generating random sharings require

[16]Linearity follows from the scheme having the proper homomorphic properties. The scheme could be, for example, a homomorphic commitment scheme as in the case of protocol P_{CEAS}, or a SHE scheme as in the case of SPDZ [50].

[17]Generation of random sharings basically means generation of consistent shares for secret random values [17].

interaction, overall cost actually increases compared to a run without circuit randomization. While this triple generation process is not efficient, it still allows us to accurately compare the online phases in terms of efficiency.

Another way of generating multiplication triples, namely, generation of multiplication triples using oblivious transfer, is discussed in Section 2.7.1. An implementation of this approach can be found in project SPDZ.

2.7 Oblivious Transfer

Oblivious Transfer (OT) was introduced as a cryptographic primitive in 1981 [93].[18] Since then, a few variants have been studied in addition to the original one. Beaver [12] lists three OT variants:

- OT (Rabin's original protocol): Alice sends bit b. Bob receives either $(0, 0)$ *("failed")* or $(1, b)$ *("received b")* uniformly at random. Alice does not know which of the two happened. [93]

- $\frac{1}{2}$ OT (1-out-of-2 OT): Alice has input bits b_0 and b_1. Bob receives (c, b_c) for a random c outside his control. Alice does not learn c. [53]

- $\binom{2}{1}$ OT (Chosen 1-out-of-2 OT)[19]: Alice has input bits b_0 and b_1. Bob chooses $c \in \{0, 1\}$ and obtains b_c. Alice does not learn c.

For each variant, several protocols exist with different security properties and efficiency.

[18]According to [39], the idea appeared earlier in another context in [106].

[19]In some sources, chosen 1-out-of-2 OT is referred to as 1-out-of-2 OT, and 1-out-of-2 OT is referred to as random OT.

Equivalence between OT and $\frac{1}{2}$ OT is shown in [44]. Beaver [14] demonstrates that, other variants can be obtained from a number of precomputed 1-out-of-2 or chosen 1-out-of-2 OTs. Generalizations of OT variants also exist [34, 71]. For example, a generalization of $\binom{2}{1}$ OT is $\binom{n}{1}$ OT [34].

OT creates an asymmetry in knowledge, between the participants. Beaver [12] notes that, this asymmetry makes OT a natural basis for achieving security in a wide variety of interactive protocols, including SMC. Killian [77] has shown that, secure two-party computation can be based on OT, i.e. given a protocol P_{OT} implementing OT, there exists a protocol P_{SFE} which implements two-party secure function evaluation *with unconditional security*. This result is extended to SMC in [45], by showing that, given OT, bit commitment, and a consensus broadcast channel, MPC *with unconditional security* and *fairness* is possible, even when any number of players may arbitrarily deviate from the protocol.

P_{CEPS} and P_{CEAS} do not use the OT primitive. However, OT is mentioned in Chapter 3, as it plays an important role in some state-of-the-art SMC protocols.

2.7.1 Generating Multiplication Triples Using OT

This section explains how two players P_1 and P_2 can generate a multiplication triple $c = ab$ over $\mathbb{F}_2$ using OT. Before going any further, it is useful to note that, when written in terms of the shares, a triple has the form

$$c_1 \oplus c_2 = (a_1 \oplus a_2)(b_1 \oplus b_2) = (a_1b_1) \oplus (a_1b_2) \oplus (a_2b_1) \oplus (a_2b_2) \qquad (2.1)$$

where, c_i is player P_i's share of the product.

Using $\binom{2}{1}$ OT

Gilboa [59] proposes a protocol for generation of multiplication triples in $\mathbb{F}_p$ using $\binom{2}{1}$ OTs. A description of the protocol, for the special case $\mathbb{F}_2$, is given in [40]:

- Each player P_i chooses at random $(a_i, b_i) \in (\mathbb{F}_2)^2$ as her shares for the multiplicands.
- P_1 chooses at random $r_1 \in \mathbb{F}_2$ and runs a $\binom{2}{1}$ OT with inputs $(r_1, a_1 \oplus r_1)$, acting as the sender. P_2 uses b_2 as the selection bit. Hence, P_2 learns r_1 if b_2 is 0, $a_1 \oplus r_1$ otherwise. Therefore, once the OT is complete, P_2 will know $a_1 b_2 \oplus r_1$.
- Similarly, P_2 chooses at random r_2 and runs a $\binom{2}{1}$ OT with inputs $(r_2, a_2 \oplus r_2)$, acting as the sender. P_1 uses b_1 as the selection bit to learn $a_2 b_1 \oplus r_2$.
- P_1 calculates $c_1 = r_1 \oplus a_1 b_1 \oplus a_2 b_1 \oplus r_2$ and P_2 calculates $c_2 = r_2 \oplus a_2 b_2 \oplus a_1 b_2 \oplus r_1$.

To see that c_1 and c_2 are indeed shares of the product, we note that $c_1 \oplus c_2$ has the form given in Equation 2.1.

Using $\frac{1}{2}$ OT

- Each player P_i chooses at random $(r_{i1}, r_{i2}) \in (\mathbb{F}_2)^2$ to be used as inputs to OTs.
- P_1 runs a $\frac{1}{2}$ OT with inputs (r_{11}, r_{12}), acting as the sender. P_2 receives r_{1x}, which she uses as her share for the first multiplicand a. P_1 computes $r_{11} \oplus r_{12}$, which she uses as her share for the second multiplicand b.
- Similarly, P_2 runs a $\frac{1}{2}$ OT with inputs (r_{21}, r_{22}), acting as the sender. P_1 receives r_{2y}, which she uses as her share for the first multiplicand a. P_2 computes $r_{21} \oplus r_{22}$, which she uses as her share for the second multiplicand b.

- Following the OTs, P_1 has her shares for the multiplicands set to

  $$a_1 = r_{2y} \qquad b_1 = r_{11} \oplus r_{12}$$

  and P_2 has her shares for the multiplicands set to

  $$a_2 = r_{1x} \qquad b_2 = r_{21} \oplus r_{22}$$

  where $r_{2y} \in \{r_{21}, r_{22}\}$ and $r_{1x} \in \{r_{11}, r_{12}\}$.

  Finally, P_1 calculates

  $$c_1 = a_1 b_1 \oplus a_1 \oplus r_{11} r_{12}$$

  and P_2 calculates

  $$c_2 = a_2 b_2 \oplus a_2 \oplus r_{21} r_{22}$$

To see that c_1 and c_2 are indeed shares of the product, we first note that

$$a_1 \oplus r_{21} r_{22} = r_{2y} r_{21} \oplus r_{2y} r_{22} = a_1 b_2$$

which is easy to verify considering the two possible values of r_{2y}. Similarly,

$$a_2 \oplus r_{11} r_{12} = a_2 b_1$$

Hence $c_1 \oplus c_2$ has the form given in Equation 2.1, and c_1 and c_2 are indeed shares of the product.

2.7.2 OT-Extension

All known OT protocols require public-key cryptography [75]. Beaver [12] has shown that, it is possible to extend OTs in the sense that, many OTs can be generated from a few *seed* OTs, using symmetric primitives.[20] By doing so, cost of many OTs can be reduced to the cost of a few public-key operations, plus the cost of relatively cheap symmetric-key operations. Following this feasibility result, several more efficient OT-extension schemes have been proposed with both passive [70] and active [74] security.

Chou and Orlandi [39] point to the analogy between OT-extension and hybrid encryption, where a symmetric key is encapsulated using a public-key cryptosystem, and then used to encrypt large amounts of data. Clearly, OT-extension can provide a substantial performance boost when large number of OT's are needed, and one such scenario is the generation of multiplication triples in the preprocessing phase.

2.8 Homomorphic Encryption Schemes

This section provides brief descriptions of four types of homomorphic encryption schemes, which are relevant in the context of SMC. P_{CEPS} and P_{CEAS} do not use any (homomorphic) encryption schemes, but these schemes are mentioned in Chapter 3, as they are used in some other SMC protocols.

2.8.1 Fully Homomorphic Encryption

A fully homomorphic encryption (FHE) scheme is homomorphic with respect to both addition and multiplication, and hence allow arbitrary computations on ciphertexts. First

[20]Extension can be regarded as going from k bits to $m > k$ bits, or alternatively, as going from k OTs to $m > k$ OTs [110].

realization of a FHE scheme is introduced in [57].

2.8.2 Additively Homomorphic Encryption

An additively homomorphic encryption scheme allows computation of linear combinations of ciphertexts.

2.8.3 Semi-Homomorphic Encryption

A semi-homomorphic encryption scheme [21] is a weaker version of an additively homomorphic encryption scheme, where additively homomorphic behaviour is allowed to be broken if the plaintext value corresponding to the input ciphertext is not sufficiently small.

2.8.4 Somewhat Homomorphic Encryption

A somewhat homomorphic encryption (SHE) scheme [51, 107] is a weaker version of a FHE scheme. Unlike a FHE scheme that allows an arbitrary number of operations, a SHE scheme limits the number of operations that can be carried out on the ciphertexts. In the context of SMC, this would mean that, only circuits of a limited depth can be calculated. SHE schemes are sometimes also called leveled homomorphic encryption schemes.

2.9 Zero-knowledge Proofs

A *zero-knowledge proof* (ZKP) can be informally described as a proof that does not reveal anything other than the validity of the assertion [60].[21] The notion of a ZKP was first

[21] This is similar to how SMC is defined informally: The computation does not reveal anything other than the result.

introduced in [63], which also introduces the notion of an *interactive proof*, as a multi-round randomized two-party protocol between a *prover* and a *verifier*. More formally, a ZKP has the following properties [60]:

- *Completeness*: Prover is able to convince the verifier of the validity of any true assertion with high probability.
- *Soundness*: Prover is not able to convince the verifier of the validity of a false assertion, except with small probability.[22]
- *Zero-knowledge*: Anything that is feasibly computable from the proof is also feasibly computable from the (proved) assertion itself.

First two properties, completeness and soundness, define an interactive proof system.

In the context of SMC, zero-knowledge proofs play an important role in enforcing conformance to a protocol [60]. A generic way of transforming a passively secure protocol into an actively secure protocol is outlined in [60, 61]:

- *Step 1:* Each player makes commitments to all her inputs.
- *Step 2:* Each player proves that she has knowledge of her own inputs using zero-knowledge proofs-of-knowledge (See Section 2.9.1). This guarantees that the commitments to inputs can be opened. This step has to be carried out in such a way that, a player cannot fix her inputs in a way that depends on the inputs of other players.
- *Step 3:* Players jointly generate a sequence of random bits for each player, to be used as inputs to the passively secure protocol.[23] The generation is carried out

[22]This probability is called the soundness error.

[23]For example, part of the generated random bits might later be used as the coefficients of a sampled polynomial, during the emulation of the passively secure protocol.

such that, a player only knows her own sequence of random bits, but every player possesses a commitment to each generated sequence.

- *Step 4: (This step is necessary (and sufficient) for robustness, and can be omitted otherwise.)* Each player shares her inputs and random bits with other players via a VSS protocol. Now, if any dishonest player P_k leaves prematurely, remaining players can run the protocol on behalf of P_k as well, using the shares P_k left behind, and hence, prevent premature termination of the protocol.

- *Step 5:* Finally, the passively secure protocol is emulated. At each step of the emulated protocol, players are expected to provide a ZKP for each message they send, to prove that the message was indeed formed in accordance with the emulated protocol. This is possible, because every message m of a player P running the emulated protocol depends only on the inputs and random bits of P, and the messages P has received so far. Inputs and random bits are determined by the commitments from *Steps* 1-3, and previous messages have been sent over a broadcast channel. Thus, the next message m is a function of publicly known strings. By the result obtained in [62], it is possible to prove in zero-knowledge that m is formed in accordance with the emulated protocol.

This process does not necessarily yield an efficient protocol, but it leads to the important feasibility result in [61], for the case of actively secure MPC. In several efficient SMC protocols with active security, variations of ZKPs and related notions, such as proofs of knowledge and non-interactive zero-knowledge proofs, play similar roles. In particular, Section 2.9.2 outlines how they can be used to achieve an actively secure preprocessing phase. P_{CEPS} and P_{CEAS} do not use ZKPs. However, some cases of their use in other protocols are mentioned in Chapter 3.

2.9.1 Proofs of Knowledge

Proofs of knowledge (PoK) are interactive proofs in which the prover asserts *knowledge* of a particular object, and not merely its existence [60]. A zero-knowledge proof-of-knowledge (ZKPoK) additionally possesses the zero-knowledge property.

In particular, a zero-knowledge proof of plaintext knowledge (ZKPoPK) [8, 73] proves knowledge of a plaintext m of some ciphertext $\mathcal{C} = \mathcal{E}(m)$ in a given public encryption scheme, without revealing anything about m.

As we shall also see for the example protocol in Section 2.9.2, a ZKPoK requires input(s) from the verifier in the form of a challenge. If this was not the case, following an accepted proof, the verifier could assume the role of a prover and *replay* the proof to another verifier, having it accepted. But this would mean that either the zero-knowledge property was violated in the first run of the protocol or the soundness property was violated in the second run.

2.9.2 Zero-Knowledge Verification of Multiplication Triples

An example use case of ZKPs in the context of SMC involves proving the correctness of multiplication triples generated in a preprocessing phase. Two specific examples are the protocols in [21] and [50]. Preprocessing phases of both protocols are based on homomorphic encryption schemes, and use ZKPoKs to prove that ciphertexts of generated multiplication triples satisfy a multiplicative relation. These protocols are relatively complex, and this makes the basic idea harder to notice. Instead, a simpler protocol [41, 43] will be considered.[24] This protocol ensures correctness of a multiplication triple in zero-knowledge, in the context of information theoretically secure MPC. In contrast to the

[24] See *Commitment Multiplication Protocol* in [43] and Section 12.5.3 in [41].

protocols in [21] and [50], in this setting one deals with commitments instead of ciphertexts, and the requirement for a homomorphic encryption scheme becomes a requirement for a homomorphic commitment scheme. The outline of the protocol is given below:

- *Step 1:* Prover P makes three commitments $[a], [b], [c]$ to values $a, b, c \in \mathbb{F}$, respectively. P asserts that, $c = ab$. P chooses random β and makes commitments $[\beta]$ and $[\beta b]$.
- *Step 2:* Verifier V generates and publishes a random challenge r.
- *Step 3:* P opens the commitment $[r_1] = r[a] + [\beta]$, revealing r_1. P also opens the commitment $[z] = r_1[b] - [\beta b] - r[c]$, to reveal z.
- *Step 4:* If a commitment opening fails or $z \neq 0$, then V rejects the proof.
- *Step 5: Steps 2-4* are repeated until soundness error is small enough.

Clearly, if P is honest, r_1 is a random value, and z being 0 tells nothing but the validity of the assertion. Note that, the smaller $|\mathbb{F}|$ is, the more repetitions will be needed to achieve a desired soundness error, and the less efficient the protocol will become.

Chapter 3

A Brief Survey of SMC: Theory and Applications

This chapter presents a brief survey of the research field, with focus on SMC based on secret sharing. Some important theoretical results, and a few selected SMC implementations and applications are mentioned.

3.1 Other Approaches to Privacy-Preserving Computations

Before focusing on SMC, it might be useful to take a step back and look at some other approaches.

3.1.1 Data anonymization

Data anonymization methods can be considered relevant, in the sense that, often the purpose of application is to make data available for computation while preserving privacy. Privacy preservation has a more restricted meaning here, as the real goal is to prevent identification of data with a person or a legal entity. Data anonymization is regarded as the standard way of protecting medical records [4]. While the basic idea is easy to understand, application turns out to be quite challenging. This is demonstrated by cases such as the deanonymizations of the AOL and Netflix datasets [10, 88].

3.1.2 Randomized Response Techniques

One approach that was originally proposed as a survey technique in [104], is randomized response techniques. The main idea can be captured by considering an example in the original setting, a survey interview. The respondent privately tosses a coin. She responds 'yes' to the question if the toss result is 'heads', and responds with her actual answer, otherwise. Individual answers are hidden by the randomness of the coin toss, but a collection of answers can still be useful to the interviewer, because half of the 'no' answers are converted to a 'yes', and hence, the real ratio of 'no' answers is double the observed ratio. This basic idea has been improved and generalized over the years. Some applications in the area of privacy-preserving data mining can be found in [2, 3, 54]. Relevant concepts are *statistical databases* and *differential privacy* [52].

3.1.3 Homomorphic Encryption Schemes

Another approach is to deploy special encryption schemes which allow computations on ciphertexts. With a FHE scheme at hand, SMC becomes a trivial task, and can be achieved as follows [41]:

- First, parties collaborate to generate a key pair. At the end of this phase, parties agree on a single public key, and the corresponding private key is secret shared among them.

- Next, input providing parties encrypt their inputs using the public key and publish the ciphertexts. Because the scheme is fully homomorphic, each party is able to compute a result ciphertext using the input ciphertexts, without any need for interaction.

- Finally, parties collaborate to decrypt the computed result.

Obviously, this is a *constant-round* solution; the amount of interaction required is independent of the computation performed. This makes an interesting contrast with information theoretically secure MPC, which usually suffers from high communication overhead, and local computations are less of a concern. While FHE allows a constant-round solution, the computational overhead is too high to allow practical applications. For example, Gentry et al. [58] reports an evaluation time of 36 hours for a single AES encryption operation. Weaker, but faster versions of homomorphic encryption schemes, such as SHE schemes and semi-homomorphic encryption schemes, play a role in some SMC systems.

3.2 Secure Multiparty Computation

SMC protocols can often be categorized into one of two classes: SMC protocols based on secret sharing and SMC protocols based on garbled circuits. SMC protocols implemented for this thesis work are based on secret sharing, hence circuit garbling is only briefly touched upon. The following subsection outlines the garbling technique by Yao, and provides references to a few selected publications that enhanced the original idea over the years. The rest of the chapter focuses on work related to SMC based on secret sharing.

3.2.1 Garbled Circuits

Emergence of SMC as a research field is often attributed to the work of Yao [108].[1] Yao's protocol is a 2-party protocol based on a symmetric encryption scheme and OT. The protocol allows any function to be securely evaluated by two players. It is historically associated with the *Millionaires' Problem* [109]: Two millionaires want to know which of them is richer, but neither of them want to tell the other how much he or she owns. The protocol does not treat the players symmetrically. One player is the *garbler*, and the other is the *evaluator*. As the name suggests, the *garbler* garbles the circuit, which involves garbling each gate of the circuit. For each gate, the *garbler* maps the binary values in the truth table to encryption keys,[2] and then encrypts each mapped output in a row of the truth table with the keys corresponding to the inputs in that row. Only the *evaluator* evaluates the (garbled) circuit. The circuit for this specific problem would be a comparator circuit accepting as inputs the bits in the binary representation of the wealth of each millionaire. Obviously, the involvement of the symmetric encryption scheme is to prevent the *evaluator* from learning the inputs of the *garbler*. The OT scheme, on the other hand, is there to enable the *evaluator* to compute the garbled gates. The protocol proceeds as follows:

- The *evaluator* receives from the *garbler* a garbled table for each gate, and also the garbled inputs of the *garbler*.
- The final missing piece of information needed by the *evaluator* to evaluate the circuit is her own garbled inputs, as the un-garbled inputs she possesses is no good

[1] Yao [108] does not explain the protocol, but according to [41, 19], it is mentioned in oral presentations of this work.

[2] As the actual inputs are not known, garbling a gate necessarily involves preparing a garbled table for all possible input values.

for evaluating the garbled circuit. The *evaluator* receives garbled versions of her inputs from the *garbler* by running a chosen 1-out-of-2 OT protocol, once for each input bit. This way, the *garbler* does not learn the input bits of the *evaluator*, and the *evaluator* learns the garbled versions of her input bits.

- As a consequence of the way garbling is carried out, garbled inputs can be used as decryption keys to remove the double encryption on gate outputs. The *evaluator* is able to open (exactly) one row from each gate's garbled table, which is enough to compute all gates and reveal the *mapped* circuit output.
- Finally, the *garbler* and the *evaluator* cooperate once more to combine the mapped output held by the *evaluator*, with the mapping information known to the *garbler*, to reveal the computation result.

A more detailed description of the protocol can be found in [100]. A formal description and a proof of security are given in [83].

Generalization

Until 1990, the statement that, more complex functions (i.e. functions requiring more complex circuits to express them) require more interaction (or communication rounds) to securely evaluate in the multiparty setting, had been a sensible hypothesis [16]. Beaver et al. [16] proved this hypothesis wrong by introducing the constant-round *BMR protocol*.[3] SMC protocols based on secret sharing mostly follow the GMW-paradigm [61].[4] As it generalizes Yao's protocol to the multiparty setting, the GMW-paradigm takes a gate-by-gate approach, i.e. many two-party computations take place at each gate, in the process of

[3]The BMR protocol is named after the authors of [16].

[4]The GMW-paradigm is named after the authors of [61].

obtaining shares of a gate output from shares of gate inputs. Beaver at al. [16] break from the GMW-paradigm, and take another approach to generalizing Yao's 2-party protocol. In this approach, garbled circuit and garbled inputs are public. Unlike Yao's protocol, there is no distinction between the players, such as *garbler* and *evaluator*; players interact to construct the garbled circuit and the garbled inputs collectively. Each player evaluates the garbled circuit on its own without any need for interaction, so the constant-round complexity of the 2-party protocol is preserved.

Lindell et al. [84] have shown that the BMR protocol [16] can be combined with an SMC protocol based on secret sharing (such as the SPDZ protocol [50]) for the preprocessing phase, to obtain a protocol which is both efficient and constant-round. Even though most of communication overhead can be pushed into a preprocessing phase in protocols based on secret sharing, it is not possible to totally eliminate communication overhead from the online phase.[5] The online phase of the proposed protocol, on the other hand, consists of each party locally evaluating a garbled circuit, and therefore it is fast even when the network is slow.

Bellare et al. [19] provide a formalization of several concepts related to garbled circuits, and introduce garbling schemes, which are meant to be used as primitives.

Optimization

Two of the several optimizations developed for garbled circuits are free-xor [78] and row-reduction [92]. A list of works related to optimization of garbled circuits can be found in [9].

[5]This is also the case for P_{CEAS}. See Section 5.4.6.

Implementation

Early notable implementations of SMC based on garbled circuits are Fairplay and FairplayMP (See Section 3.2.3). Two more recent implementations are presented in [67, 79].

3.2.2 Feasibility Results

Results concerning the feasibility of SMC were established back in 1980s. One such result concerns *Byzantine Generals* and *Consensus* problems. Bracha [33] has shown that, with n parties, t of which might be corrupted (or Byzantine), solutions exist with a round complexity of $\mathcal{O}(\log n)$ for $t = \dfrac{n}{(2+\delta)}$, if cryptographic methods are allowed, and for $t = \dfrac{n}{(3+\delta)}$, otherwise. A completeness theorem for protocols with honest majority is given in [61]. Another completeness theorem which concerns information theoretic SMC, and is more relevant to the implemented protocols, is given in [23]. Ben-Or et al. [23] have shown that, in the absence of Byzantine parties, no set of size $t < \frac{n}{2}$ of players gets any information other than the evaluation result. If Byzantine behaviour is allowed, corresponding bound is $t < \frac{n}{3}$. In the same work, it is also proven that these bounds are tight. For example, in the non-Byzantine case, $\frac{n}{2}$ passive dishonest parties will always be able to collaborate to break privacy. P_{CEPS} and P_{CEAS} are similar to the passively secure and actively secure protocols proposed in [23].[6] Given the existence of authenticated private channels between each pair of parties, any general multiparty protocol can be solved when $t < \frac{n}{3}$, such that the secrecy of inputs are unconditional [36]. This result follows from the work of Lamport et al. [80], which has shown that Byzantine

[6]One difference is that, multiplication gates are computed differently. Another difference (between the actively secure protocol in [23] and P_{CEAS}) is that, P_{CEAS} performs VSS in a relatively indirect and less efficient way, but can be generalized more easily to work with other linear secret sharing schemes [41].

Generals problem is solvable with perfect security when pairs are connected by secure point-to-point channels and $t < \frac{n}{3}$.

The results above concern SMC over synchronous networks. SMC over asynchronous networks is studied, for example, in [22, 24].

Following the feasibility results, complexity of SMC has been studied extensively. A list of related work is given in [48].

3.2.3 SMC Frameworks and Other Implementations

Fairplay and FairplayMP

Both systems allow the computation to be expressed in a high-level language, and convert it to a Boolean circuit. Fairplay system [85] allows secure computation between two parties, and evaluates the generated circuit using Yao's protocol. FairplayMP [20] is an extension of Fairplay to the multiparty setting. It evaluates the generated circuit using a protocol, which is based on the BMR protocol.

Sharemind

Sharemind [29] is one of the early attempts at realizing large scale SMC. The system is based on the earlier theoretical results given in [23, 36, 15], and SMC is based on secret sharing. Sharemind provides information theoretic security. Number of players are fixed at three, and only a single corruption by a passive adversary can be tolerated.

Virtual Ideal Functionality Framework (VIFF)

VIFF [103] is an SMC framework. Damgård et al. [47] propose an SMC protocol based on secret sharing, and report on its implementation using the VIFF framework. Proposed protocol is secure against an adaptive and active adversary corrupting less than $\frac{n}{3}$ players. The protocol provides information theoretic security if secure point-to-point channels and Byzantine agreement is given. However, for efficiency reasons, point-to-point channels are implemented using standard encryption tools and broadcast channel implementation uses public-key signatures. Therefore, the implementation of the protocol (on VIFF framework) provides computational security. The protocol uses circuit randomization, and the user is allowed to choose from two different preprocessing phase implementations: one based on Pseudorandom Secret Sharing and the other based on Hyperinvertible Matrices.

SPDZ

SPDZ [76] is an SMC software system that combines various techniques and protocols developed over the years.[7] Following three papers constitute the theoretical background for SPDZ.

Bendlin et al. [21] identify a number of semi-homomorphic encryption schemes, and show that they can be used in the construction of an efficient preprocessing phase. Proposed protocol is often referred to as BDOZ.[8] Its online phase uses information theoretic MACs of shares to achieve information theoretic security against an active adversary. Its preprocessing phase can be based on one of several semi-homomorphic encryption schemes. Players use ZKPoPKs to prove that their ciphertexts are correctly computed. ZKPoKs are used to prove that ciphertexts of generated multiplication triples satisfy a multiplicative relation (See Section 2.9.2). Homomorphic property is leveraged in the construction of zero-knowledge proofs. While the ZKPs used in the preprocessing phase

[7]At the time of writing, SPDZ software system has three major versions: SPDZ, SPDZ2, and MASCOT [105].

[8]Protocol BDOZ is named after the authors of [21].

work with any field $\mathbb{F}_p$, they become less efficient for small values of p, for the same reason given at the end of Section 2.9.2. We have seen in Section 3.2.2 that, unconditionally secure protocols cannot exist in the dishonest majority setting. BDOZ protocol works in the dishonest majority setting, and its preprocessing phase has both a statistical security parameter and a computational security parameter associated with it.

Damgård et al. [50] introduce a more efficient online phase (compared to that proposed in [21]), which is statistically secure against an active and adaptive adversary corrupting majority of the players, when synchronous communication and secure point-to-point channels are given. The work required for running the proposed online phase is larger by only a small constant factor, compared to the work required for evaluating the circuit in the clear.[9] The increase in efficiency comes from using the MAC of the secret itself, instead of MACs of its shares, which is the case for the protocol proposed in [21]. Instead of each player having her own MAC key, a single global key is secret shared, so that checking of MACs is possible, while forging is not. Preprocessing phase is based on a SHE scheme. It adapts the ZKPoPK proposed in [21]. Generation of multiplication triples involve parallel evaluation of several small circuits with a single multiplication gate.[10]

Nielsen et al. [89] introduce TinyOT, an efficient, actively secure two-party computation protocol based on OT. It is claimed that the introduced actively secure OT-extension is nearly equivalent to the passively secure OT-extension [70] in terms of efficiency.

At the time of writing, codebase for SPDZ software system [101] includes implementation of the online phase from [50], and implementation of the preprocessing phase from

[9]When the circuit is evaluated in the clear, players first reveal their secrets, and then evaluate the circuit on their own.

[10]Recall that, homomorphism of a SHE scheme would break for large circuits. Also note that, these circuits are independent of the circuit to be evaluated in the online phase.

MASCOT [75].[11] MASCOT is a protocol based on OT, which provides active security in the dishonest majority setting, same as the above mentioned protocols, but brings significant efficiency improvements to the preprocessing phase compared to each one of them. Large part of the efficiency improvement comes from the actively secure OT-extension introduced in [74], which is even more efficient compared to the OT-extension of TinyOT. Generation of multiplication triples in MASCOT preprocessing phase is based on Gilboa's method [59], which was presented in Section 2.7.1 for the simplest case.

SPDZ software system allows the user to express the computation via a python-based front-end, and supports floating point and fixed point operations.

3.3 Applications of SMC

Keller et al. [75] note that, there has been a growing interest in applications of SMC, as several efficient protocols with realistic adversary assumptions, and implementations capable of handling complex computations have emerged in the last decade. In this section, some application scenarios and applications that appeared in the literature [41, 75, 91] are discussed briefly.

3.3.1 Auctions

An auction can be carried out in several ways. Perhaps the most familiar type of auction involves an item on sale and multiple bidders, where the highest bid wins. A bidder Alice does not want to reveal the maximum price she is willing to pay, as otherwise an auctioneer could collaborate with another bidder to raise the price just below Alice's maximum price.

[11]MASCOT is short for 'Malicious Arithmetic Secure Computation with Oblivious Transfer'.

A Double Auction in Denmark

Bogetoft et al. [31] report a nationwide double auction that took place in Denmark in 2008. In the auction, contracts entitling farmers to sugar-beet production have been exchanged. The parties involved were Danisco, the buyer of sugar-beet, and DKS representing the farmers. Both parties had reasons not to have the other party act as the auctioneer. Hiring a consultancy house to act as a trusted third-party was considered too expensive a solution. SMC was deployed instead, and the role of the auctioneer was played by a multiparty computation. 25000 tons of production rights changed owner in the auction.

3.3.2 Procurement

In a typical procurement, a public institution asks companies to bid for a contract. Contract specifies the job public institution needs done, and the lowest bid wins. Bidders are often competing companies. A bidder Alice does not want to reveal her bid to another bidder Bob, otherwise Bob could win the contract by making an offer that is only slightly lower.

3.3.3 Benchmarking

In a typical benchmark analysis, multiple companies in the same line of business provide inputs to a trusted third-party to see how well they perform compared to the others. All participants are interested in learning the result, but none of them want their inputs leaked to other participants.

Benchmarking for Danish Banks

Damgård et al. [46] report a confidential benchmarking that involved several Danish banks and a consultancy house,[12] where the participating banks jointly evaluated the risks associated with their customers using the SPDZ protocol. Use of SMC allowed the banks and the consultancy house to retain the privacy of their customer data.

3.3.4 Privacy Preserving Data Mining

A typical use case for data mining involves one or more databases, such as those kept within health care systems or tax systems of countries. Coordinated access to such databases can be very valuable for the purposes of research and administration. On the other hand, such access might be abused for compiling complete dossiers on individuals and there is a growing privacy concern.

Tax Fraud Detection in Estonia

In 2013, Estonian parliament proposed a legislation that requires companies to declare their purchase and sales invoices to be used in risk analysis and fraud detection, but it was vetoed on the grounds of confidentiality breach and other concerns. Bogdanov et al. [27] report on the Estonian Tax and Customs Board's evaluation of a tax fraud detection system, which uses Sharemind SMC framework[13] to address the confidentiality concern.

A Statistical Study in Estonia

Bogdanov et al. [28] report on a statistical study conducted by the Estonian Center of Applied Research in 2015. The study linked the database of individual tax payments

[12]The consultancy house was involved as an additional input provider, not as a trusted third-party.

[13]See Section 3.2.3.

from the Estonian Tax and Customs Board and the database of higher education events from the Ministry of Education and Research, to look for correlations between working during university studies and failing to graduate in time. Over ten million tax records and half a million education records were analyzed using Sharemind SMC framework.

3.3.5 Electronic Voting

Secure evaluation of a function as simple as

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$$

can be interpreted as voting on a *yes/no* decision. A player P_i would provide input $x_i = 1$ to vote *yes* and $x_i = 0$ to vote *no*. Computation result would give the number of *yes* votes. Ballot secrecy seems to come naturally with SMC, but a voting scheme often has to satisfy additional requirements, such as *public verifiability*.

Low-Latency Voting from SMC

Baum et al. [11] extend the SPDZ protocol with the additional property of public verifiability. The standard network model is enhanced with an idealization of a public append-only bulletin board, the ideal functionality $F_{Bulletin}$, on which the transcript of the protocol is published. Anyone with access to the transcript of the protocol can verify the correctness of the computation; whether or not she participated in the computation, and even if all the participants were corrupted. As an example application scenario for the protocol, Baum et al. [11] suggest replacing mix-nets used in electronic voting schemes such as Helios [1], with an SMC-based implementation of shuffling.

3.3.6 Set Intersection

Set intersection problem involves parties, who have their own sets of items, and want to find out whether or not an item appears in all of the sets. For example, an investigation agency might have a list of suspects, an airline might have a list of passengers, and they could run an SMC protocol to find out whether any of the suspects are among the passengers, while retaining the privacy of their lists.

Cross-Domain Cooperative Firewall (CDCF)

A CDCF allows two collaborative networks F and H to enforce each other's firewall rules [37]. In a typical scenario, a roaming user makes a VPN connection from a foreign network F to her home network H, in order to preserve the privacy of her communications. Firewall of network F ($\mathbf{F}_F$) wants the traffic going through the encrypted tunnel inspected, to ensure that it obeys the rules defined for F, but it does not want to reveal its firewall rules to the firewall of network H ($\mathbf{F}_H$), which is capable of performing the inspection. Cheng et al. [37] propose a design for a privacy-preserving CDCF, which involves an oblivious membership verification algorithm. The private inputs are the firewall rules enforced by $\mathbf{F}_F$, and the connection descriptors of any connections made by the roaming user. The former input is provided by $\mathbf{F}_F$, and is a collection of 4-tuples[14] and associated verdicts. The verdict can be, for example, *accept* or *deny*, for a particular 4-tuple. The latter input is provided by $\mathbf{F}_H$, and is a 4-tuple that describes a connection. Oblivious membership verification has to be run only once at the time of connection setup, and packets are dropped depending on the precomputed result.

[14] A 4-tuple includes two IP address-port pairs, one for the source and one for the destination.

3.3.7 Other Applications

Satellite Collision Analysis

Countries do not want to reveal orbital information about their strategic satellites to other countries, but they also want to avoid collisions. Kamm and Willemson [72] demonstrate the feasibility of computing the probability of a collision between satellites using SMC, where private inputs are the orbital information provided by satellite operators. They used Sharemind SMC framework to implement the collision analysis.

Inter-Domain Routing

The Internet is made up of administrative domains called Autonomous Systems (ASes), and the routes between them are computed by the Border Gateway Protocol (BGP). ASes want to keep their routing policies private, because they can leak information about their business relationships with other ASes. While BGP does not require ASes to explicitly reveal their routing policies, these policies are susceptible to inference attacks. Gupta et al. [64] suggest computation of routes using SMC, providing ASes with provable privacy guarantees. Asharov et al. [7] report on an application of SMC to inter-domain routing. They convert *neighbor relation* and *neighbor preference* BGP algorithms into circuits and evaluate them using the GMW protocol [61].

Inter-Domain Network Monitoring

Iacovazzi et al. [69] report on an SMC-based implementation for inter-domain network monitoring, where providers of private inputs are individual ISPs.

3.3.8 Synergy with Blockchains

A typical blockchain is a data structure made up of timestamped blocks linked by hash pointers to form a tamper-evident chain [87], and is managed by a peer-to-peer network of peers running a specific protocol, such as the Bitcoin [86] protocol. This section discusses the synergy which seems to exist between SMC and cryptocurrencies, and more generally, between SMC and blockchain technologies.

Andrychowics et al. [5] note that the standard definition of SMC guarantees only the emulation of a trusted third-party. According to this definition, SMC does not guarantee correctness of inputs, and if an action is to be taken based on the output of the computation, it does not guarantee that the output will be respected. Andrychowics et al. [5] claim that it is possible to address these issues by linking the inputs and outputs with Bitcoin transactions, for example in the form of security deposits. Cryptocurrencies offer a variety of ways to enforce distributed business logic, from the relatively simple *timed commitments* of Bitcoin, to Ethereum [26] *smart contracts* written in a Turing-complete language. A notion that becomes relevant in this aspect is covert security. This relaxed security model does not require that the probability of getting away with cheating is negligible. However, it is still required that deviations from the protocol are detected with a high probability. A covertly secure variant of the SPDZ protocol is proposed in [49]. Relaxation of security requirements results in a more efficient protocol, as expected. With the addition of a mechanism that can enforce punishment on cheaters (and/or a mechanism that rewards repeated honest behaviour), covert security might suffice for some practical applications. Andrychowics et al. [5] also note that, in addition to addressing the issues of incorrect inputs and disrespected outputs, fees and security deposits can be used to enhance robustness and fairness. For example, players might refrain from repeatedly joining computations just to abort prematurely, in order to avoid the financial cost associated with the loss of security deposits.

Security of a typical blockchain comes from a combination of cryptography and incentive engineering [87]. While the immutability property of a blockchain comes with cryptographic guarantees, achieving decentralized consensus strongly relies on having properly incentivized participants. Not having some sort of currency associated with the blockchain might make the incentive engineering part difficult, if not impractical. Having said that, there are also possible benefits to an SMC system, originating purely from having access to a blockchain materializing a public immutable ledger, irrespective of whether or not a currency is associated with it. Zyskind et al. [111] note that, in the case of a blockchain managing access to private data, laws and regulations concerning access to private data (and also any rules that may be set by the data owner) can be programmed into the blockchain and enforced automatically.[15] They also note that, even when the enforcement fails or was not possible to begin with, the immutable ledger could act as a temper-proof access log, which can be used as legal evidence. Note that regulations and rules enforced on the computations might as well be of technical nature. For example, consider a rule such as *"No query is answered with respect to less than six records."*. This is a real-life example of a rule used for the purpose of inference control [4].[16] With a blockchain assuming the role of a manager, one can go beyond access control based on identities. Contents of computations can be managed and logged, and inference control mechanisms can be implemented. The standard definition of SMC, based on emulation of a trusted third-party, clearly provides no guarantees with respect to inference control. A trusted third-party, provided with secret inputs x_1 and x_2, will evaluate the function

[15]This can be achieved, for example, via smart contracts.

[16]Even when the query is prepared with the intention of revealing a statistical result about a group of people, if the criteria are not chosen carefully, it might end up revealing information about a specific individual.

$f(x_1, x_2) = x_1$ without complaint, and make the output public. Hence this is another area, where a blockchain can complement the security provided by SMC.

Enigma

Zyskind et al. [112] describe Enigma as a decentralized computation platform, which utilizes SMC in order to *"overcome the public nature of blockchains"*. Shares of private data are held in a distributed hashtable, and an external blockchain is used to manage access control and identities. External blockchain additionally has the critical role of implementing the $F_{Bulletin}$ ideal functionality defined in [11] (See Section 3.3.5). The SMC protocol used by the Enigma platform is the publicly verifiable version of SPDZ proposed in [11].

Chapter 4

Specification of Implemented Protocols

The two main SMC protocols implemented for this thesis work are P_{CEPS} and P_{CEAS}. A third implemented protocol is $P_{CEAS,CR}$, which is a version of P_{CEAS} that uses circuit randomization. $P_{CEAS,CR}$ is composed of $P_{CEAS,P}$ and $P_{CEAS,O}$, the protocols for preprocessing and online phases, respectively. This chapter provides brief overviews of P_{CEPS} and P_{CEAS}, and summarizes the results of formal security analysis of P_{CEPS}, P_{CEAS}, and $P_{CEAS,CR}$.

4.1 Overview of the Protocols

The starting point for all the implementations in this thesis work are the protocol descriptions given in [41]. These descriptions will be expanded into a few dozen pages of text, when the protocols and their implementations are described in Chapter 5. However, the original brief descriptions serve better for getting a quick overview of the protocols. Such descriptions for P_{CEPS} and P_{CEAS} are presented below.[1] Similar descriptions for the other protocols can be found in [41].[2]

[1]Both descriptions are adapted from [41]. See page 38 for P_{CEPS} and page 117 for P_{CEAS}.

[2]See pages 111, 113, 127, 128 for subprotocols used in P_{CEAS}, and page 170 for $P_{CEAS,O}$.

4.1.1 Protocol CEPS (Circuit Evaluation with Passive Security)

P_{CEPS}

- **Input Sharing Phase:** For each private input x_i, corresponding *input provider* distributes the set of shares $[x_i; f_{x_i}]_d$.
- **Computation Phase:** As long as the circuit has gates waiting to be processed, players pick the next gate to be processed and does one of the following depending on the type of gate:
  - **Addition Gate (ADD):** Players hold $[a; f_a]_d$ and $[b; f_b]_d$ for the gate inputs a and b, respectively. Players compute $[a; f_a]_d + [b; f_b]_d = [a + b; f_a + f_b]_d$.
  - **Multiply-by-constant Gate (CMUL):** Players hold $[a; f_a]_d$ and $[b; f_b]_d$ for the gate inputs a and b, respectively. Players compute $\alpha[a; f_a]_d = [\alpha a; \alpha f_a]_d$.
  - **Multiplication Gate (MUL):** Players hold $[a; f_a]_d$ and $[b; f_b]_d$ for the gate inputs a and b, respectively. Players compute $[a; f_a]_d * [b; f_b]_d = [ab; f_a f_b]_{2d}$. Each player P_i distributes the set of shares $[h(i); f_i]_d$, where $h = f_a f_b$, by definition. Players compute
  $$\sum_i r_i[h(i); f_i]_d = \sum_i [h(0); r_i f_i]_d = \sum_i [ab; r_i f_i]_d$$
  where $\mathbf{r}$ is the recombination vector defined in Section 2.4.1.

  If there are no more gates to compute, players move on to the output reconstruction phase.
- **Output Reconstruction Phase:** For an output wire of the circuit that has value y assigned to it, players hold $[y; f_y]_d$. Each player P_i securely sends $f_y(i)$ to the *data users*, who use Lagrange interpolation to obtain the result $y = f_y(0)$.

4.1.2 Protocol CEAS (Circuit Evaluation with Active Security)

P_{CEAS}

- **Input Sharing Phase:** For each private input x_i, corresponding *input provider* distributes the set of shares $[[x_i; f_{x_i}]]_d$ by running the *VSS* protocol (See Section 5.4.4). If a *VSS* fails, honest players learn that the corresponding input provider is corrupt, and use a default input instead of x_i.

- **Computation Phase:** As long as the circuit has gates waiting to be processed, players pick the next gate to be processed and does one of the following depending on the type of gate:

  - **Addition Gate (ADD):** Players hold $[[a; f_a]]_d$ and $[[b; f_b]]_d$ for the gate inputs a and b, respectively. Players compute $[[a; f_a]]_d + [[b; f_b]]_d = [[a + b; f_a + f_b]]_d$.
  - **Multiply-by-constant Gate (CMUL):** Players hold $[[a; f_a]]_d$ and $[[b; f_b]]_d$ for the gate inputs a and b, respectively. Players compute $\alpha[[a; f_a]]_d = [[\alpha a; \alpha f_a]]_d$.
  - **Multiplication Gate (MUL):** Players hold $[[a; f_a]]_d$ and $[[b; f_b]]_d$ for the gate inputs a and b, respectively. Players compute $[[a; f_a]]_d * [[b; f_b]]_d = [[ab; f_a f_b]]_{2d}$. This involves each player running the *commitment multiplication* protocol (See Section 5.4.4) to prove that their multiplications were carried out correctly, and may fail for up to d players. Each player P_i distributes the set of shares $[[h(i); f_i]]_d$ by running the *VSS* protocol, where $h = f_a f_b$, by definition. These *VSS* runs may also fail for up to d players. Let C be the indices of the players for which the previous *commitment multiplication* and *VSS* runs did not fail. Players compute

$$\sum_{i \in C} r_i [[h(i); f_i]]_d = \sum_{i \in C} [[h(0); r_i f_i]]_d = \sum_{i \in C} [[ab; r_i f_i]]_d$$

    where $\mathbf{r_C}$ is the recombination vector defined in Section 2.4.1.

If there are no more gates to compute, players move on to the output reconstruction phase.

- **Output Reconstruction Phase:** For an output wire of the circuit that has value y assigned to it, players hold $[[y; f_y]]_d$. Each player P_i opens its committed share for y to the *data users* by running the *designated open* protocol (See Section 5.4.4). *Data users* use Lagrange interpolation to obtain the result $y = f_y(0)$.

4.2 Security of the Protocols

Rigorous security analysis of a protocol requires an exact problem definition and usually takes the form of a mathematical proof, as previously mentioned in Section 2.1.4. Problem definition includes network model, adversary model, and a clarification of what is meant by security. In this section, only the problem definitions and the results are provided. The proofs can be found in [41].

4.2.1 Network Model

It is assumed that the parties are on a synchronous network, and secure peer-to-peer channels exist between every pair of players, and between every player-user pair.

4.2.2 Adversary Model

A threshold-t adversary is assumed.

4.2.3 Security

Let n be the number of players. If the assumptions about the network and the adversary given above hold, then

- P_{CEPS} is perfectly, adaptively, and passively secure, when $t < \frac{n}{2}$.
- P_{CEAS} and $P_{CEAS,CR}$ are perfectly, adaptively, and actively secure, when $t < \frac{n}{3}$.

Chapter 5

Implementation

This chapter describes in detail the protocol implementations and the implementation of the simulator, which provides an environment where the protocols can be executed. Please refer to Appendix A for information regarding the software project, including source code availability.

5.1 A Motivating Example

This section presents a motivating example, and the details of the implementations are presented in the following sections. An imaginary scenario involving an application of SMC is considered. An attempt is made to realize this scenario using the implementations that are presented in this work. The scenario is depicted in Figure 5.1.

Actors involved are a population of potential input providers, a pool of computing parties (or players), an entity called the *manager*, and a data user that will be referred to as *Data.org*. The *manager* and the pool of computing parties together make up *Compute.org*.

Input providers are users of a service provided by *Compute.org*. By using this service, they grant *Compute.org* permission to use their data in certain computations, possibly in exchange for some sort of compensation. Thanks to the involvement of SMC, they never

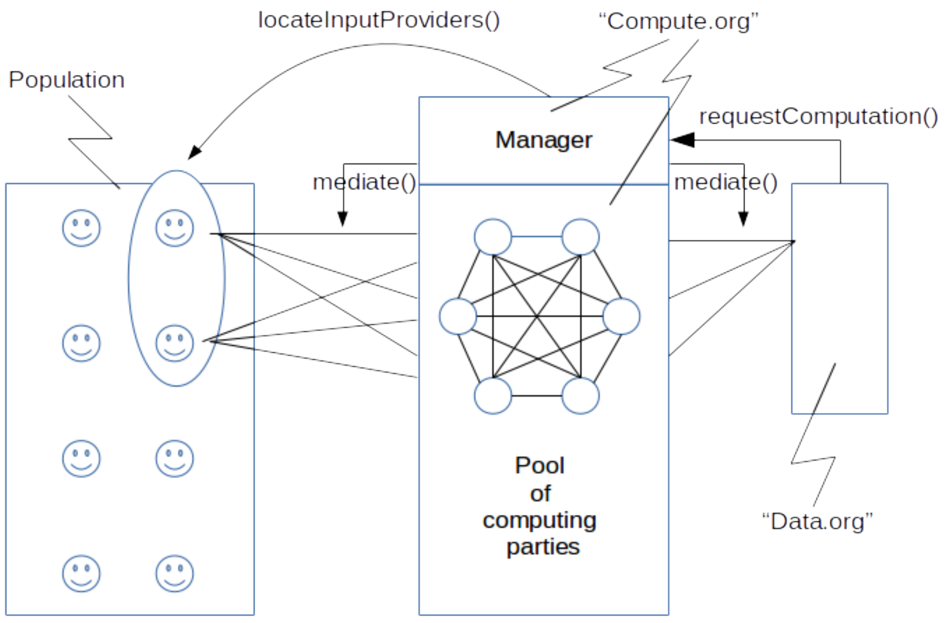


FIGURE 5.1: A depiction of the imaginary scenario.

have to reveal their private inputs. They can make their data (i.e. their inputs for any previous calculations) unavailable to *Compute.org* at any later time, if they wish so.

Data.org might be, for example, a market research company. It calls *Compute.org* and tells them that it requires the result of a certain computation. Once sufficiently many input providers agree to provide data for this computation, *Compute.org* performs the computation and reveals the result to *Data.org*, again possibly in exchange for some sort of compensation. *Data.org* receives the result it needs without the costs and risks associated with operating on and storing confidential data. Furthermore, more people might want to provide them with data, if they know that their data will not be revealed to any other party, and the people who participate might provide data more honestly.

The part of *Compute.org* that is of interest in the current context is the pool of computing parties. The other part, *manager*, does not get involved in the computation. It is tasked with processing computation requests, locating input providers, and mediating connections. It mediates the connections between the input providers and computing parties, and also between the computing parties and data users such as *Data.org*. Computing parties might be, for example, on a regular P2P network, possibly slightly modified to match the needs of the particular SMC protocol (See Section 2.1.2). Computing parties perform the computations, and reveal the results to the data users. SMC will allow *Compute.org* to remain oblivious to both the inputs, and the computation result.

The chosen SMC protocol must be compatible with the realities of this particular application domain. In other words, the assumptions of the protocol about the network and the adversary must hold, and the security provided by the protocol must be sufficient. In order to quantify and manage incentives, and limit corruption, game theory and incentive engineering could be utilized in the system's design. In the event of success, the realities of the application domain would be transformed in a favorable way, possibly allowing the use of a less demanding, more efficient SMC protocol. Furthermore, an SMC protocol is indifferent to the correctness of provided inputs. Eventually, one has to assume that the input providers provide correct inputs. Again, incentive engineering could be utilized in the system's design to make that assumption more realistic. These matters related to the overall system design will not be discussed further. Instead, focus will be on the realization of the secure computations.

Data.org makes the two computation requests shown in Table 5.1, and ten input providers accept to provide inputs for these computations. It is no coincidence that the chosen inputs, *monthly income* and *drug use history*,[1] are usually regarded as highly confidential information. Assumed private inputs of the participants are as shown in Table 5.2.[2] Circuit description strings[3] for mean value (μ) and standard deviation (σ) computations

[1]Let's assume that the input *drug use history* corresponds to the answer of a question such as "Have you ever used an illegal drug?".

[2]As the values are arbitrarily chosen, the unit of currency does not matter and is omitted.

TABLE 5.1: Computation requests made by *Data.org*.

Computation	**Inputs Required**
Mean value of incomes of people with a history of drug use (μ)	Monthly income (Numeric) History of drug use (Boolean)
Standard deviation of incomes of people with a history of drug use (σ)	Monthly income (Numeric) History of drug use (Boolean)

TABLE 5.2: Private inputs of the input providers.

	Income	**Drug use**		**Income**	**Drug use**
Input provider 1	800	NO	Input provider 6	600	NO
Input provider 2	400	YES	Input provider 7	200	YES
Input provider 3	600	YES	Input provider 8	300	NO
Input provider 4	0	YES	Input provider 9	500	YES
Input provider 5	500	YES	Input provider 10	700	YES

are

$$C_\mu = \sum_{i=1}^{10} drug_i * income_i$$
$$C_\sigma = \sum_{i=1}^{10} drug_i * ((income_i + (\mu \cdot -1)) * (income_i + (\mu \cdot -1)))$$

where $drug_i$ is 1 if the i^{th} input provider has ever used an illegal drug, 0 otherwise. $income_i$ is the monthly income of i^{th} input provider.

Note that, evaluation of these circuits will not directly yield the mean value and standard deviation. To obtain those, it is necessary to know the number of people with a

[3]A circuit description string is exactly what its name suggests. It is discussed further in Section 5.3.

history of drug use. Hence, computing parties will also evaluate

$$C_D = \sum_{i=1}^{10} drug_i$$

In order to avoid finding multiplicative inverses and taking square roots in this example, it is assumed that *Compute.org* reveals the results of all three evaluations to *Data.org*. Then, *Data.org* calculates

$$\mu = \frac{result(C_\mu)}{result(C_D)} \tag{5.1}$$

$$\sigma = \sqrt{\frac{result(C_\sigma)}{result(C_D)}} \tag{5.2}$$

In general, one has to be careful about revealing intermediary results when multiple computations are carried out. In this case, *compute.org* should only reveal $result(C_D)$, if the agreement with the input providers included all three computations.

Another complication arises from the fact that, C_σ contains μ, which is equal to evaluation result of C_μ times a constant factor. Computing parties should either hold on to their shares for $result(C_\mu)$ until the computation of C_σ, or they should somehow receive μ as input, if they want to avoid performing the same mean value computation multiple times. Both options are easy to implement. The first option is discussed in Section 6.2.1. The problem with this approach in this particular case is that, at the end of first computation, computing parties do not hold shares of the mean value, but rather they hold shares of a constant multiple of it. Since the second computation is not linear in μ, this approach cannot be used without computing multiplicative inverses. Luckily, the second option is readily applicable to our case. Using the results $result(C_\mu)$ and $result(C_D)$ revealed to

TABLE 5.3: Computation results revealed to *Data.org*.

$\mathbf{result(C_\mu)}$	$\mathbf{result(C_\sigma)}$	$\mathbf{result(C_D)}$	μ	σ
2900	348572	7	414	223

it, data user will calculate μ as in Equation 5.2, and secret share it for the second computation. Hence, the second computation will be run with inputs secret shared by both the input providers and the data user *Data.org*.

This scenario is run with the inputs given in Table 5.2. Evaluation is done by executing protocol P_{CEAS}. Results of secure evaluations revealed to *Data.org*, and μ and σ values calculated from these, are shown in Table 5.3. Note that, restricting ourselves to integers have cost us in terms of accuracy. The real values up to two significant digits are

$$C_\mu = 414.29$$

$$C_\sigma = 241.03$$

This is not a weakness associated with SMC in general. For example, the application of SMC to satellite collision analysis, mentioned in Section 3.3.7, makes heavy use of floating-point arithmetic.

5.2 Simulator

The most practical way of executing the implemented protocols, requires the simulation of the environment in which they are executed. While the protocol implementations are the main focus of this chapter, in this section, the simulation part of the project is presented. An overview of the simulator is given in Figure 5.2. Main responsibilities of the simulator are

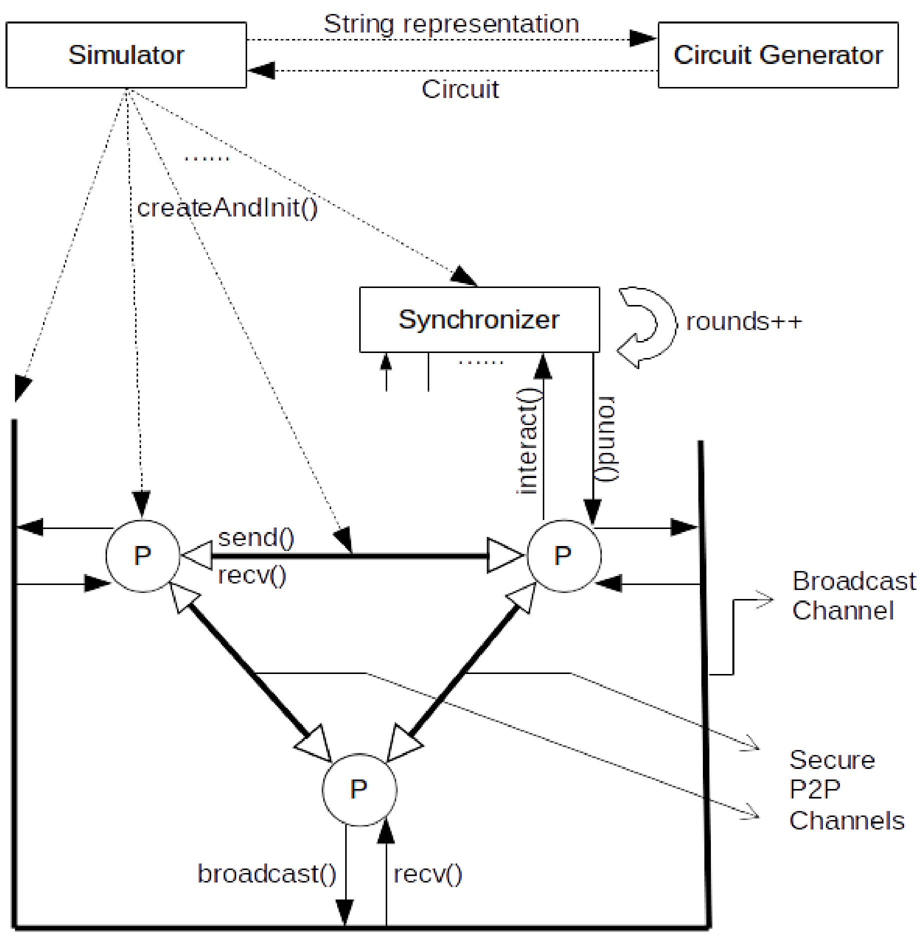


FIGURE 5.2: Overview of the simulator.

- reading the options file (See Section 5.2.2)
- creating and initializing `Party` objects
- simulating the network (See Section 5.2.1)
- simulating an active adversary (See Section 5.2.3)[4]

[4]Simulation of an active adversary is not applicable to executions of P_{CEPS}.

Simulator gets the circuit generator (See Section 5.3) to generate a circuit from the circuit description string it reads from the options file. It then provides each player with its own copy of the generated circuit.

Simulator logs the contents of all the sent messages, and can optionally[5] log internal states[6] of the parties at specified times.

5.2.1 Simulating the Network

Implemented protocols require as communication resource secure peer-to-peer channels,[7] which are usually implemented using cryptography. It is mentioned in Section 3.2.2 that the *Byzantine Generals* problem is solvable with perfect security within the setting assumed for P_{CEAS}. Hence, even though P_{CEAS} requires consensus broadcast, it is not required to make a separate assumption about the existence of a broadcast channel. Indeed, it is stated in Section 4.2.1 that, the protocols require only synchronization and existence of secure peer-to-peer channels from the network. However, in this work, consensus broadcast is not implemented as a subprotocol,[8] and has to be simulated, in addition to these two.

The implemented protocols are executed on a simulated network: both the secure peer-to-peer channels and the broadcast channel are simulated.

[5]See `VERBOSE` in `Pceas.h`.

[6]An example of internal state information is the state of the table of commitment records, previously mentioned in Section 5.4.3.

[7]A secure channel provides authenticity, integrity, and confidentiality guarantees for the messages sent over it.

[8]An implementation of consensus broadcast could be built on top of the simulated secure peer-to-peer channels.

Synchronization

Simulation of synchronous communication is achieved via a synchronizer thread, which keeps the computing threads (players) in step with each other by means of *locks* and *condition variables*. As seen in Figure 5.2, players notify the synchronizer when they require interaction to continue with the execution of the protocol, and the synchronizer notifies the players when a communication round has taken place, so that the players can continue executing the protocol. Transmission of messages takes place after all players become interactive, and is simulated by swapping pointers to the containers holding the message objects. Within a single communication round, each player sends

- at most one private message to each of the other players via the corresponding secure peer-to-peer channel
- at most one broadcast message via the consensus broadcast channel

Messages on the simulated network are delivered instantly. We do not impose a specific timeout to distinguish between delayed and maliciously omitted messages. Case of a maliciously omitted message can be simulated by a call to `interact()` when the outgoing message buffer is empty.[9] Note that calls to `interact()` are part of network simulation and are exempt from malicious behaviour (See Section 5.2.3).

5.2.2 Simulator Options

When the simulator is run, it first reads the options file `opt` under `options` folder. The following can be configured by editing this file:

- Protocol parameters

[9]Message buffers reside in `SecureChannel` and `ConsensusBroadcast` classes.

- The protocol to execute
- Input providers and their private inputs
- Data user
- Parties that are corrupted by an (active) adversary (See Section 5.2.3)
- Circuit description string (See Section 5.3)
- Sequential run flag and related parameters (See Section 6.2.1)
- Comparator options (See Section 6.2.2)

Protocol Parameters

Protocol parameters are

- N: Number of players
- T: Threshold for the secret sharing scheme
- p: Field prime

The (Shamir) secret sharing scheme underlying the implemented protocols uses polynomials over the finite field $\mathbb{Z}/p\mathbb{Z}$. Prime p is referred to as the *field prime*.[10] Implemented protocols do not allow 2-party secure computation, hence it is required that $N > 2$. As mentioned in Section 2.3.3, p must be chosen large enough in order to avoid modular reductions and be able to add and multiply integers optimally. Finally, there is the constraint $N < p$ from Section 2.4.2.

[10]$\mathbb{Z}/m\mathbb{Z}$ is a field if and only if m is a prime.

Specifying the Protocol to Execute

User chooses one of the three implemented protocols (P_{CEPS}, P_{CEAS}, $P_{CEAS,CR}$) to execute.

Players and Users

Parties run on their own threads and communicate with each other by sending and receiving message objects through the simulated channels.[11] Players (computing parties), input providers, and data users are all modeled by `Party` class. In order to simplify the implementation, the following assumptions are made:

- input providers and data users are both subsets of the players
- there is a single data user

Simulator assigns parties unique identifiers called `partyID`, which range from 1 to N. Players who have the additional role of an input provider or a data user are specified in the options file. For input providers, in addition to `partyID`s, the private input values and their *labels* (See Section 5.3) are also specified.

Computing parties work on the shares of private inputs provided to them by the input providers. Recall that VSS guarantees consistency of shares. Only constraint for being an input provider (for a P_{CEAS} run) is to be able to run VSS with the N computing parties. Similarly, only constraint for being a data user (for a P_{CEAS} run) is to be able to participate in the opening of shares as receivers, and to be able to combine the shares to recover the result. Upon recovering the computation result, the data user simply writes it to the standard output stream.

[11] `Message` and other classes can be found inside `message` folder.

5.2.3 Simulating an Active Adversary

An inspection of the implementation of P_{CEAS} reveals that, most of the code is there to handle cases of deviation from the protocol, and several code blocks gets activated only in case of malicious behaviour. While there is not much that can be done in terms of testing, for observing the effects of semihonest behaviour, malicious behaviour can be simulated to test whether the implementation behaves as expected. This is what we did for the implementation of P_{CEAS}. Certain *cases of malicious behaviour* (See Section 5.4.5) are hard-coded into the implementation. Any player, who is marked as corrupted by an active adversary in the options file, executes these cases.[12] Many more cases of malicious behaviour could be included, but the existing cases are enough to achieve almost full coverage of the previously mentioned code blocks. Cases of malicious behaviour can be enabled individually or collectively by modifying the macro definitions `TEST_CASE_X` and `TEST_ALL` in `Pceas.h`.

Determining C_{max}

N and T together determine the maximum number of corrupted players that can be tolerated, C_{max}. The degree of polynomials used for secret sharing will be denoted by D, where $D = T - 1$. It is required that the honest players are able to interpolate, and T shares are needed to do that. Therefore, we have

$$N - C_{max} > D$$

[12]Note that, the contents of the options file are decided before the simulation starts, and as a result, only a static adversary can be simulated.

TABLE 5.4: Values of C_{max} for the given T and N.

$(\mathbf{T}, \mathbf{N})$	$\mathbf{C_{max}}$
(2,3)	0
(2,4)	1
(2,7)	1
(3,7)	2
(3,10)	2
(4,10)	3

However, during computation of multiplication gates, one has to deal with polynomials of degree $2D$.[13] Hence, the actual constraint is

$$N - C_{max} > 2D \tag{5.3}$$

It is also required that the corrupted parties are not able to gather their shares and interpolate. This yields the second constraint

$$C_{max} \leq D \tag{5.4}$$

In this case, there is no need to consider computation of multiplication gates, as doing so yields a less restrictive constraint. C_{max} for a given (T, N) pair can be found using Equation 5.3 and Equation 5.4 together. Table 5.4 shows values of C_{max} for a few selected (T, N) pairs. As expected, combining equations 5.3 and 5.4 to eliminate D yields

$$N - C_{max} > 2C_{max}$$

[13]Two polynomials of degree D get multiplied.

or

$$C_{max} < \frac{N}{3}$$

5.3 Circuit Generator

We provide a minimalistic circuit generator, which allows avoiding the tedious work of hard-coding arithmetic circuits. A simple recursive descent parser parses a given *circuit description string*, generates the necessary `Gate` and `Wire` objects, and assembles them into a `Circuit` object. An example circuit description string is `(a+b)*(b.2)`. The symbols '+', '`.`', and '$\star$' represent **ADD**, **CMUL**, and **MUL** gates, respectively. `a` and `b` are *labels*. A label can be a single alphabetic character as in the example, or a string of characters starting with an alphabetic character and possibly containing numeric characters. Due to left recursion, symbols are consumed from left to right. Parenthesis can be used to affect the order of **MUL** and **CMUL** gates within the circuit. For example, `(a+b)*(b.2)` and `(a+b)*b.2` would yield different circuits.[14] Other than being part of labels, numeric characters are allowed only as multiplier values associated with **CMUL** gates. In this case, they always follow a '`.`', possibly with a preceding '$-$' for a negative value.

Labels allow the players to match private inputs of input providers with the input wires of their circuits. During the input sharing phase, each player will expect to receive shares for each label specified in the circuit description string, that was used to build the circuit it is computing.

If a label is used more than once in the circuit description string, multiple input wires are assigned the input value associated with that label. Again considering the circuit

[14]Though, both circuits would yield the same result.

generated from `(a+b)*(b.2)` as an example, label `b` will be assigned both to one of the input wires of the **ADD** gate, and to the single input wire of the **CMUL** gate. By the end of input sharing phase, each player will have assigned its share of the input associated with label `b`, to all input wires labeled with `b`.

5.4 Protocols

This section describes the implementations of P_{CEPS}, P_{CEAS} and $P_{CEAS,CR}$. It also serves as a detailed presentation of these protocols' inner workings.

5.4.1 $\mathrm{P_{CEPS}}$

P_{CEPS} is a very simple protocol, whose implementation comes almost for free if P_{CEAS} is implemented. P_{CEPS} is included as a separate protocol mostly because it serves as a stepping stone in describing P_{CEAS}. Furthermore, having both protocols separately allows us to appreciate the increase in complexity during the transition from passive security to active security.

An overview of P_{CEPS} is given in Section 4.1.1. Below, we go through each phase of the protocol for a simple circuit. To keep things manageable, we consider the minimum possible number of computing parties, which is 3. Party P_1 and party P_2 are given the additional role of an input provider, and party P_3 is given the role of a data user. Description string for the circuit to evaluate is `(a+b)*(b.2)`. The circuit is chosen so that, it has one gate of each type and it is as simple as possible. P_1 and P_2 will provide their inputs with labels `a` and `b`, respectively. P_1's private input is 2, and P_2's private input is 3. With these inputs, the expected evaluation result is 30. The field prime p is chosen to be 31. T is chosen to be 2, which is the minimum non-trivial value. With these choices, P_{CEPS}

tolerates a passive adversary corrupting a single computing party. Choices for protocol parameters and private inputs are summarized below.

$$(T, N) = (2, 3) \qquad p = 31 \qquad a = 2 \qquad b = 3$$

Computing parties start execution of the protocol by running some sanity checks on protocol parameters, and then they calculate the recombination vector as described in Section 2.4.1 to obtain

$$\mathbf{r} = (3, 28, 1)$$

Input Sharing Phase

Data providers P_1 and P_2 each sample a polynomial of degree D and distribute shares of their secrets to each computing party (or player). We assume that P_1 sampled

$$f(x) = 2 + 5x$$

and P_2 sampled

$$g(x) = 3 + 17x$$

Constant terms are the values of the secrets to be secret shared, and randomly chosen coefficients of x^1 terms will hide the values of the secrets. Players distribute the shares:

$$P_1 : f(1) = 7(label : \texttt{a}) \longrightarrow P_1 \qquad P_2 : g(1) = 20(label : \texttt{b}) \longrightarrow P_1$$

$$P_1 : f(2) = 12(label : \texttt{a}) \longrightarrow P_2 \qquad P_2 : g(2) = 6(label : \texttt{b}) \longrightarrow P_2$$

$$P_1 : f(3) = 17(label : \texttt{a}) \longrightarrow P_3 \qquad P_2 : g(3) = 23(label : \texttt{b}) \longrightarrow P_3$$

Note that, P_1 and P_2 send shares to themselves, as they are both input providers and players. Circuit of each player at the end of input sharing phase is shown in Figure 5.3.

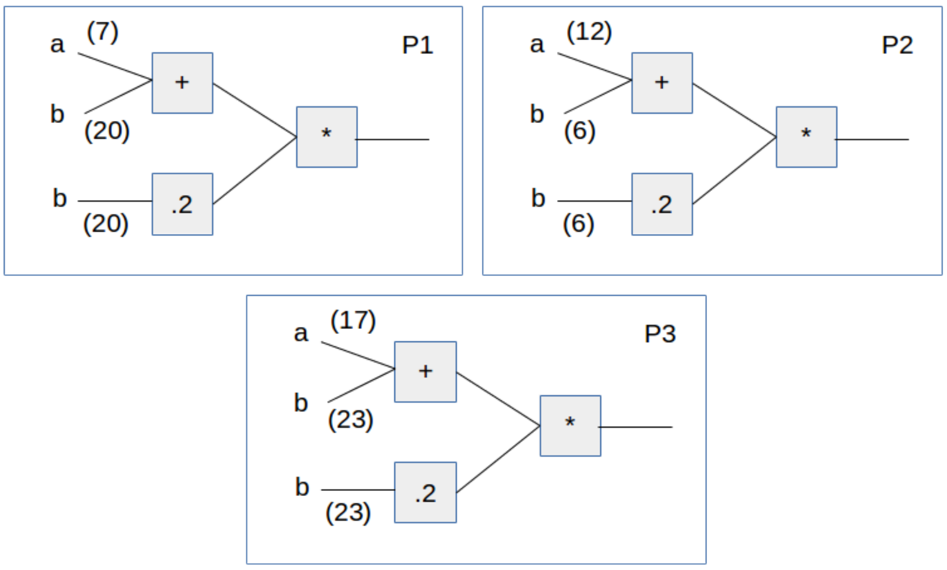


FIGURE 5.3: Circuit of each player at the end of input sharing phase.

Computation Phase

Computation order of the gates has no effect on the result, but players have to be able to agree on the order. The implementations presented in this work achieve this by numbering the gates during circuit generation. All players compute the gate that has the smallest gate number, and is *computable*, meaning it has not yet been computed and all its input wires are assigned values. When there are no more computable gates, computation phase ends.

ADD and **CMUL** gates are computed locally. Circuit of each player after computation of first two gates is shown in Figure 5.4.

Computation of multiplication gates requires interaction between the players. They first calculate a product locally by multiplying the values assigned to the input wires.

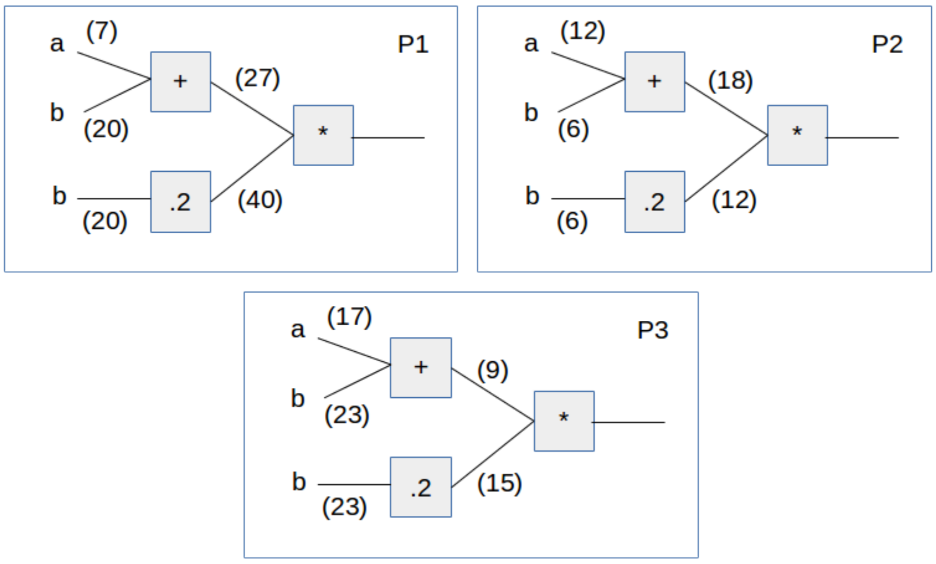


FIGURE 5.4: Circuit of each player after computation of **ADD** and **CMUL** gates.

This product, however, is not a T-*share*: Two polynomials of degree, $D = T - 1$, get multiplied to yield a polynomial of degree $2D$. To obtain T-shares, they first distribute shares of their local products, in the same way input providers shared their secrets in input sharing phase. We assume that P_1, P_2 and P_3 sampled $h_1(x) = 26 + 29x, h_2(x) = 30 + 7x, h_3(x) = 11 + 18x$, respectively, where the constant terms are the local products of the players, and coefficients of x^1 terms are randomly chosen. Players distribute the shares:

$$P_1 : h_1(1) = 24 \longrightarrow P_1 \qquad P_2 : h_2(1) = 6 \longrightarrow P_1 \qquad P_3 : h_3(1) = 29 \longrightarrow P_1$$

$$P_1 : h_1(2) = 22 \longrightarrow P_2 \qquad P_2 : h_2(2) = 13 \longrightarrow P_2 \qquad P_3 : h_3(2) = 16 \longrightarrow P_2$$

$$P_1 : h_1(3) = 20 \longrightarrow P_3 \qquad P_2 : h_2(3) = 20 \longrightarrow P_3 \qquad P_3 : h_3(3) = 3 \longrightarrow P_3$$

Next, player P_i produces T-share s_i by arranging the received shares into a vector and performing a dot product with the recombination vector $\mathbf{r}$.

$$P_1 : s_1 = (24, 6, 29) \cdot \mathbf{r} = 21$$
$$P_2 : s_2 = (22, 13, 16) \cdot \mathbf{r} = 12$$
$$P_3 : s_3 = (20, 20, 3) \cdot \mathbf{r} = 3$$

This process of obtaining shares of lower degree from shares of higher degree is called *degree reduction*. Figure 5.5 visualizes the degree reduction that took place above. Local products of the players can be seen lying on a polynomial of degree 2, whereas the shares s_i lie on a polynomial of degree 1. Player P_i assigns T-share s_i to the output wire of its

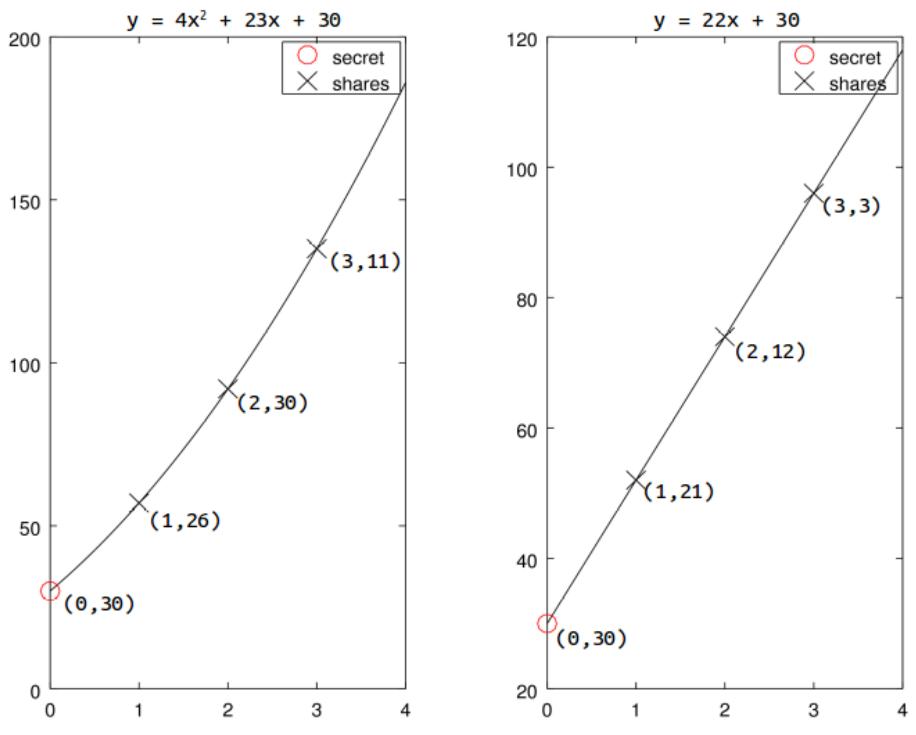


FIGURE 5.5: Visualization of the degree reduction process.

multiplication gate. As there are no more gates to compute, computation phase is over. Circuit of each player at the end of computation phase is shown in Figure 5.6.

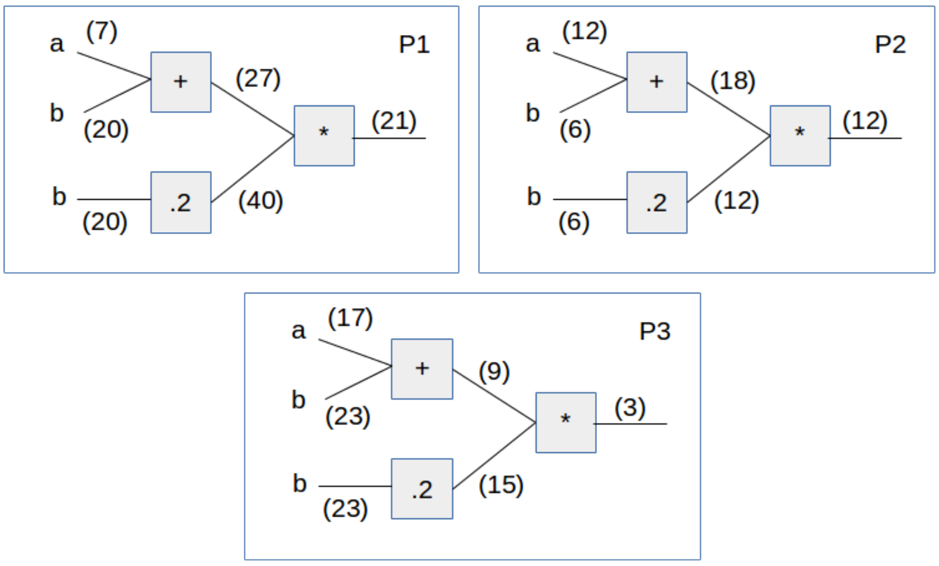


FIGURE 5.6: Circuit of each player at the end of computation phase.

Output Reconstruction Phase

Each player privately sends her share of the evaluation result to the data user (in this case P_3) via the corresponding secure peer-to-peer channel. Data user combines the shares by performing a dot product with the recombination vector to obtain

$$(21, 12, 3) \cdot \mathbf{r} = 30$$

which is the expected result.

5.4.2 $\mathrm{P_{CEAS}}$

Due to the complexity of P_{CEAS}, it is not possible to present a tractable example to explain the protocol, as it was done for P_{CEPS}. As P_{CEAS} is described, focus will be on things that has to be done differently in order to achieve active security. From an implementation point of view, the most significant difference from P_{CEPS} is that, commitments are used instead of elements of the underlying field. For example, wires are assigned commitments, and operations that take place during gate computations are carried out on commitments. This requires each player to keep and manage a table of all commitments (See Section 5.4.3), both their own and those belonging to other players, in such a way that all honest will agree at all times on existing commitments. A player who makes a commitment will be referred to as the *owner* of that commitment. In order to be able to keep a consistent view of existing commitment records, players perform local operations on other players' commitments, in addition to performing them on their own. In such cases, we will say that players perform the operation *locally and consistently*. Unlike with ordinary shares in P_{CEPS}, a player can perform operations on committed shares owned by other players. For example, a player can meaningfully add two commitment records, even when she is not the owner of the commitments.[15] As the protocol evaluates arithmetic circuits, it is necessary to be able to add commitments, multiply commitments with a constant, and multiply commitments. In similarity with P_{CEPS}, multiplication of commitments cannot be done locally, unlike addition and multiplication-with-constant. Addition and multiplication-with-constant are needed not just for gate computations, but also for creating linear combinations of existing commitments as part of some of the subprotocols, which are described later in this chapter. For circuit randomization, it is also necessary to be able to add commitments with a constant and subtract commitments from one another.

[15]Though, the commitments used as operands must have the same owner.

These operations can be expressed in terms of addition and multiplication-with-constant, and are discussed further in Section 5.4.6.

A particular *naming scheme* is used in order to provide commitment records with unique *names* (or *identifiers*). The naming scheme allows players to agree on a unique name without the need to communicate, and it plays an important role in *local and consistent* computations. Obviously, players do not have access to the circuits of other players, but the naming scheme allows a player to work out the name of the corresponding commitment owned by any other player, by inspecting the name of the commitment in her own circuit. Another benefit of using this naming scheme is that, it prevents actively corrupted players from crafting names that will cause name conflicts. If a player P broadcasts a name that does not fit the naming scheme, for some commitment she intends to make, all honest players can agree that P is corrupted. The details of the naming scheme are omitted.

Execution of some parts of the protocols result in creation of numerous commitment records that are only temporarily useful. For example, we shall see in Section 5.4.4 that, in step 3 of protocol *Commitment Transfer*, each player ends up creating up to $4 \cdot D \cdot N$ new commitment records.[16] Only $2N$ of these records are used in the following steps. Moreover, *Commitment Transfer* itself is called consecutively N times during a single *VSS*. In order to prevent inflation of the tables holding the commitment records, a mechanism to remove the unwanted commitment records is deployed. All the commitment records, except those which are marked as *permanent*, are removed at certain points during the protocol execution. This purging mechanism is referred to as *cleanup of records* in the rest of this work.

[16] D records are created as result of multiplication-with-constant operations. D records are created as result of addition operations. With 2 polynomials per transfer, and assuming none of the N transfers are erroneous, total is $4 \cdot D \cdot N$ records per transfer.

P_{CEAS} is designed such that malicious behaviour will be noticed by honest players. Each honest player keeps a set of players who were detected to be maliciously corrupted. All honest players will agree on this set of corrupted players at all times.

Preserving the agreement between honest players on matters such as existence of commitment records and set of corrupted players, is crucial for the protocols' functioning. It is possible to have cases where a single honest player is certain that some player behaved dishonestly but there is no way all honest can agree on this at that particular step of the protocol. In such cases, protocol will leave the handling of dishonest behaviour to future rounds. The implementations presented in this work respect this principle, and update commitment records and set of corrupted players only when an agreement between honest players is possible. Often, an agreement is possible when evidence of dishonest behaviour arrives from the broadcast channel. Players run sanity checks on information contained in both private and broadcast messages. For example, if some player includes in a message its intention to run a protocol on a particular commitment, the receiver checks from its table of records whether the commitment exists and whether the sender of the message is actually the owner of the commitment.[17] If the message is a broadcast message and the check fails, sender can be (and will be) marked as corrupt right away, as all honest players agree at all times on which commitments exist and who their owners are, which in turn allow them to agree that the sender is corrupted. As a second example, absence of a broadcast message will also get a player marked as corrupt, if the protocol required the message to be sent, but the player did not send it. As the protocols are detailed in the following sections, a check performed on received messages is mentioned only if it is a core part of the protocol, and the kinds of checks mentioned above are omitted.

[17]These conditions hold for all the subprotocols used in P_{CEAS}.

In P_{CEPS}, computing the recombination vector once at the very beginning is sufficient. In P_{CEAS}, on the other hand, each time a player is marked as corrupt, recombination vector has to be recomputed.[18] Furthermore, each time a player is marked as corrupt, honest players check whether the number of corrupted players C exceeds C_{max} (See Section 5.2.3). If $C > C_{max}$, then the assumption about the corruption capability of the adversary was wrong, and all honest players abort execution.

P_{CEAS} uses *VSS*, *Commitment Multipication*, and *Designated Open Commitment* as subprotocols. These protocols are described in Section 5.4.4.

An overview of P_{CEAS} is given in Section 4.1.2. Players start execution of the protocol by running some sanity checks on protocol parameters. If sanity checks are passed, input sharing phase starts. Following sections describe each phase of the protocol.

Input Sharing Phase

Input providers run protocol VSS consecutively, as many times as needed, until every honest player has all the input wires of her circuit assigned. Players who are not input providers, and input providers who ran out of secrets to secret share, passively participate. To prevent a malicious input provider from withdrawing an input and causing other players to be stuck in an infinite loop, honest players note the number of distinct labels in their circuits, N_L, and run VSS at most N_L times. Note that, the maximum number of iterations required is equal to N_L, and it is observed when all the secrets are provided by a single party. If an input is withheld, all honest players complain that an input is missing and stop execution. Once all inputs have been secret shared, each honest player locates

[18]Shares of players marked as corrupt are effectively excluded. Obviously, when a share is excluded, a different recombination vector is needed to obtain the same result.

the received shares within her table of commitments, and assigns them to input wires with matching labels. Finally, players do a *cleanup of records*.

Computation Phase

Until there are no more computable gates, honest players pick the next computable gate and compute it. $\mathbf{ADD}$ and $\mathbf{CMUL}$ gates are computed *locally and consistently*. For an $\mathbf{ADD}$ gate, a player retrieves the two commitments assigned to the input wires and adds them. For a $\mathbf{CMUL}$ gate, there is a single input wire, and the commitment on that wire is multiplied with the multiplier of the gate, which is a value in $\mathbb{Z}/p\mathbb{Z}$.

A $\mathbf{MUL}$ gate cannot be computed locally. A player retrieves the two commitments assigned to the input wires of the gate, and runs the protocol *Commitment Multiplication* with these commitments, creating a commitment to a local product in the process. As is the case with P_{CEPS}, these local products of T-shares are not T-shares of the product and a degree reduction is needed. Each player distributes committed shares of its local product by running the protocol *VSS*. Both *Commitment Multiplication* and *VSS* might cause some players to be marked as corrupt, and more than $2D$ shares are needed to uniquely determine a polynomial of degree $2D$. If honest players are left with enough shares when those sent by players marked as corrupt are excluded, each honest player combines the remaining committed shares of local products using the recombination vector. In this case, degree reduction requires more effort than performing a dot product. Players *locally and consistently* multiply their committed shares with the corresponding elements of the recombination vector, and take a sum of these commitments, again *locally and consistently*, to obtain committed T-shares.

After each gate computation, players do a *cleanup of records*. When there are no more gates to compute, players enter the output reconstruction phase of the protocol.

Output Reconstruction Phase

Players locate their shares of the computation result and open it to the data user by running the protocol *Designated Open Commitment*. A party can be the target of a single designated open operation at a time, and all shares have to be opened to the same party. Hence, players are required to take turns for opening their shares: When one player opens its share to the data user, other players passively participate. Once the players are done opening their shares, if more than $D\,T$-shares have been received from players not marked as corrupt, the data user forms a vector from the *opened values* of these committed shares. The vector is formed by ordering the opened values with respect to their openers. The data user calculates the dot product of this vector with the recombination vector. Reducing the result of the dot product modulo field prime p yields the computation result.

5.4.3 Local Operations on Commitment Records

Describing the local operations on commitment records requires knowledge about the structure of a commitment record. In this section, first the *table of commitment records* and the relevant fields of a commitment record are introduced, and then the local operations are described.

Table of Commitment Records

A *table of commitment records* is part of the internal state of a player and holds a collection of commitment records. Each commitment record in the table contains several pieces of information. In addition to information describing the corresponding commitment, a commitment record also holds information related to management of the record, and state information with regard to protocols that are being executed (or were executed) on the corresponding commitment. Only the fields that describe a commitment are mentioned here.

Figure 5.7 shows a simplified view of a *table of commitment records*. We assume that there are only three players: P_1, P_2 and P_3. Further, we assume that `p1c1`, `p2c1`, `p3c1` are the names of the only commitments made up to that point, whose owners are P_1, P_2 and P_3, respectively. Figure 5.7 shows only *names*, *verifiable shares* $f_i(x, j)$, and univariate polynomials $F_0(x)$. Two other fields of interest are *share* and *opened value*. *Share* corresponds to an ordinary share, similar to the ones mentioned in the P_{CEPS} example presented in Section 5.4.1, and is equal to the *verifiable share* (a univariate polynomial over $\mathbb{Z}/p\mathbb{Z}$) evaluated at 0. *Opened value* is the value originally committed to, and also the value revealed when the commitment is opened. *Opened value* is equal to $F_0(x)$ (also a univariate polynomial over $\mathbb{Z}/p\mathbb{Z}$) evaluated at 0. These fields are further discussed in Section 5.4.4.

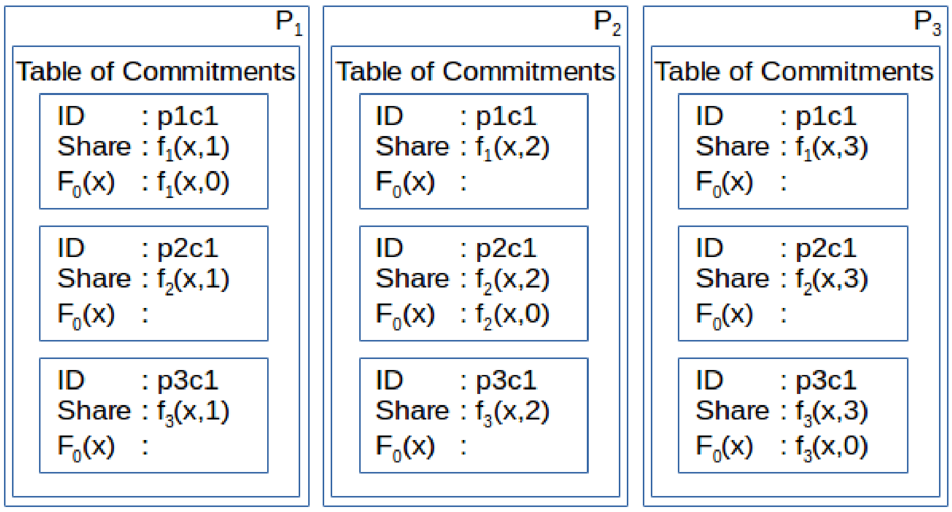


FIGURE 5.7: A simplified view of a table of commitment records. (One for each player.)

Addition of Commitments

Addition of two commitments is carried out as follows. The player P performing the operation creates a new commitment record to hold the result of the operation. *Share* field of the new record is set to the sum of *share* fields of the operands. If P is the owner of the operands, two additional fields of the commitment record must be set. Field $F_0(x)$ is set to the sum of the corresponding fields of the operands, via an addition of polynomials over $\mathbb{Z}/p\mathbb{Z}$. Field *opened value* is set to the value obtained by evaluating the $F_0(x)$ field of the new commitment record at 0. Finally, the new record is marked as *done*, which indicates that it is not a record belonging to a commitment in progress.[19]

Multiplication of a Commitment With a Constant

Multiplication of a commitment with a constant c is carried out in a way similar to addition of commitments. The player P performing the operation creates a new commitment record to hold the result of the operation. *Share* field of the new record is set to the product of the *share* field of the operand and the constant c, which are both in $\mathbb{Z}/p\mathbb{Z}$. If the owner of the operand is the player performing the operation, field $F_0(x)$ of the new commitment record is set to $F_0(x)$ of the operand multiplied by c, via multiplication of a polynomial over $\mathbb{Z}/p\mathbb{Z}$ by a constant. Field *opened value* is set to the value obtained by evaluating the $F_0(x)$ field of the new commitment record at 0. Finally, the new record is marked as *done*.

[19] If a record is not marked as *done*, it means that the protocol *Commitment* is still being run, and the corresponding commitment has not been made yet.

5.4.4 Subprotocols Used by P_{CEAS}

In this section, the protocols that are used within P_{CEAS} are described. Each of these subprotocols are artificially split into several *steps*, in order to make their description easier.[20]

Commitment

By running protocol *Commitment*, a player commits to a value $a \in \mathbb{Z}/p\mathbb{Z}$. Recall from Section 2.5.1 that, commitment is achieved via redundant sharing. After successful completion of the protocol, commitment owner cannot *open* the commitment (by running either *Open Commitment* or *Designated Open Commitment*) with a value different than a (binding property), and a is hidden in the sense that it is secret shared among N parties. Commitments happen in parallel, in the sense that, while one honest player is committing to some value, all other honest players are simultaneously committing to some values of their own.

In step 1 of the protocol, commitment owner samples a symmetric bivariate polynomial $f(x, y)$ of degree D, such that the zero coefficient is equal to a. The first communication round takes place, and the commitment owner privately sends to each other player P_j the verifiable share $f(x, j)$. Players update their table of commitments from the broadcast commitment intents. While other players store only the name and their verifiable shares for this commitment, the owner also stores the univariate polynomial $F_0(x) = f(x, 0)$, which will later allow her to open the commitment (Figure 5.7). At this stage, commitment records are not yet marked as *done*, and they remain this way until the final stage of this protocol.

[20]Splitting was not done based on communication rounds: A single step might span any number of rounds.

In step 2, for each ongoing commitment, each player calculates points on received verifiable shares and privately sends them to every other player. These values will be referred to as *verifiers*. To give an example, we consider the scenario described in Section 5.4.3. For commitment `p3c1`, P_1 will evaluate its verifiable share $f_3(x, 1)$ at $x = 2$ and send it to P_2, and P_2 will evaluate its own verifiable share for the same commitment $f_3(x, 2)$ at $x = 1$, and send it to P_1. An actively corrupted player might send an invalid verifier to one or more parties, for one or more of the commitments. See *Case 1* in Section 5.4.5.

In step 3, players perform consistency checks on the verifiers. If the verifiable shares were indeed generated from a *symmetric* bivariate polynomial, players should observe that the verifiers they received from other players are consistent with the polynomial they received as their own verifiable share. Continuing with the example given above, one of the checks performed by P_1 and P_2 for commitment `p3c1` is to compare $f_3(1, 2)$ and $f_3(2, 1)$. In case of an inconsistency, a *dispute* is broadcast. For example, if P_2 observes that the values do not match, she broadcasts a message which says *'P_2 is disputing the verifier sent by P_1 for the commitment `p3c1`'*.

In step 4, a player takes a note of every dispute broadcast (in its table of commitment records) for future use, and for disputes concerning her own commitment, she broadcasts every disputed value. Continuing with the example given above, for the dispute received from P_2 concerning the verifier sent by P_1, P_3 would evaluate $f_3(1, 2)$ using the symmetric bivariate polynomial it sampled at the beginning, and broadcast it. An actively corrupted owner might refuse to broadcast for one or more disputes. See *Case 2* in Section 5.4.5.

In step 5, honest players expect to see that owners have broadcast values for all disputes concerning their commitments, and that every value they have broadcast is consistent with the previously received verifiable share. Unless both conditions hold, an honest player will broadcast a message, saying that it *accuses* the commitment owner. Players store the broadcast values in their tables of commitment records for future use. An actively corrupted player might make false accusations at this stage. See *Case 3* in Section 5.4.5.

In step 6, players first update their tables of commitment records with the broadcast accusations. If an owner is accused, she broadcasts the verifiable share for the accusing party, which is supposed to be the same as that privately sent to the accusing party during the very first communication round. An accused owner who is actively corrupted might refuse to broadcast a verifiable share for one or more accusations. See *Case 4* in Section 5.4.5.

In step 7, honest players check for each commitment, whether verifiable shares of all accusing players were broadcast by the owners, and whether every verifiable share broadcast is consistent with both the points broadcast previously in Step 4 and with the verifiable shares received in Step 1. If a commitment owner failed to broadcast a verifiable share, or if the broadcast verifiable share is inconsistent with previous messages, an honest player will accuse the commitment owner.

So far, players may have observed signs of malicious behaviour, but no judgment has been passed. The idea behind the protocol is to force corrupted players broadcast information, so that inconsistencies will eventually get them caught, while ensuring that all that broadcast information does not break the hiding property or hurt privacy. To be able to keep that balance, one relies on the assumption about the corruption capability of the adversary. More specifically, in the eighth and last step, where the honest players will finally pass judgment, the number of accusations will be weighed against the maximum number of corrupted players.

In step 8, players first update their table of commitments with accusations made in step

7. If a commitment owner broadcast inconsistent information or if more than D players accused the commitment owner (meaning, at least one honest player accused),[21] then all honest players will agree that the commitment owner is corrupted and the intended commitment will fail. Otherwise, the commitment will succeed, and each player will set as its share for the commitment, the received verifiable share evaluated at 0. Basically, at this moment, the verifiable shares have completed their task, and players can perform local operations on successful commitments by operating on their shares. Whether the commitment succeeded or failed, all honest players mark the commitment record as *done*. If the owner's broadcasts were consistent, and the number of accusations is nonzero, but less than or equal to D, then a tolerable number of corrupted players tried to sabotage the commitment. In this case, honest players will behave the same way they would if the number of accusations were 0. If a commitment fails, all honest players mark the owner as corrupt, and force a *public commitment* to 0. Honest players force a *public commitment* to a particular value a, by locally updating the record corresponding to the commitment, as if the value committed to by the owner were a. By forcing a *public commitment*, honest players can continue executing the protocol, and safely use this commitment record if it is required at a later stage. Computation result will not be affected by the forced commitment, as the owner is now marked as corrupt and will be excluded from recombination. In case of success, there is one tricky case that has to be taken care of. An actively corrupted owner may behave maliciously at the start, but later behave honestly and have her commitment accepted. Assume that player P_k accuses the owner, and have the owner broadcast a new verifiable share in step 6, which is different from the one P_k (privately) received previously. Then, in step 8, P_k should update the record corresponding to this commitment such that it uses the broadcast verifiable share, as it is this broadcast one

[21] Recall that $C_{max} \leq D$ (See Section 5.2.3).

which passed the consistency checks in step 7. P_k will know that the commitment owner is corrupted, but there is no way all honest can agree on this, because inconsistency involves the verifiable share sent to P_k privately. The owner will not be marked as corrupt by any player. The best protocol *Commitment* can do in this case is to (eventually) force honest behaviour, if the commitment is to succeed.

Open Commitment

By running protocol *Open Commitment*, a player opens a commitment it has previously made. While one honest player is opening a commitment, other honest players might be simultaneously opening commitments of their own. However, there are cases where an honest player will just passively participate in the openings of other players, without opening a commitment herself.

It was mentioned in the previous section that, in step 1 of protocol *Commitment*, commitment owner stores $F_0(x) = f(x, 0)$ to be used when opening the commitment. In step 1 of *Open Commitment*, $F_0(x)$ is broadcast by the player opening the commitment, along with the commitment identifier (or name). By broadcasting a polynomial different than that used during the commitment, a corrupted player might try to open her commitment with a different value. See *Case 5* in Section 5.4.5.

In step 2, players update corresponding commitment records with the broadcast polynomials. Protocol *Open Commitment* guarantees that, if the opening succeeds, by evaluating this polynomial at 0, players will get the originally committed value. Further interaction is required to decide success. For each commitment being opened, each player broadcasts her share, which was stored in step 8 of protocol *Commitment*. An actively corrupted player might broadcast a different share to sabotage an opening. See *Case 6* in Section 5.4.5.

In step 3, for each commitment being opened, each player counts the number of shares broadcast, that are consistent with the polynomial broadcast in step 1. For example, P_1 evaluates $F_0(x)$ at $x = 2$ and if it is equal to the share broadcast by P_2 in step 2, P_1 increments the number of consistent shares by one. As the total number of consistent shares is solely determined by information from consensus broadcast, all honest players will agree on it. If total number of consistent shares is greater than $2D$,[22] then the opening succeeds and the commitment record is marked as *opened*. Otherwise, the opening fails, and all honest players mark the opening player as corrupt.

Designated Open Commitment

Protocol *Designated Open Commitment* uses *Open Commitment* as a subprotocol. By running *Designated Open Commitment*, a player opens a commitment it has previously made, to a single player, which will be referred to as the *target*. While one honest player is running *Designated Open* for a commitment, other players might be running *Designated Open* for a commitment of their own, but with a *different* target. There are cases where an honest player will passively participate in the openings of other parties, without doing a designated open herself. Targets are determined by the *target selection scheme*. The target selection scheme guarantees that a player is targeted by at most one honest player at a time, and it is also used in *Transfer Commitment* and *VSS* protocols. Before going further, we briefly explain this scheme.

The target selection scheme allows us to keep the homogeneity of roles among the players, and allows the players to run the same protocol simultaneously while receiving no more than one private message from any single player in a single communication round. Figure 5.8 depicts the target selection scheme for $N = 4$. With four players, three

[22]Recall that $N - C_{max} > 2D$ (See Section 5.2.3).

iterations are needed so that each player gets a chance to select every other player as target (exactly) once.[23]

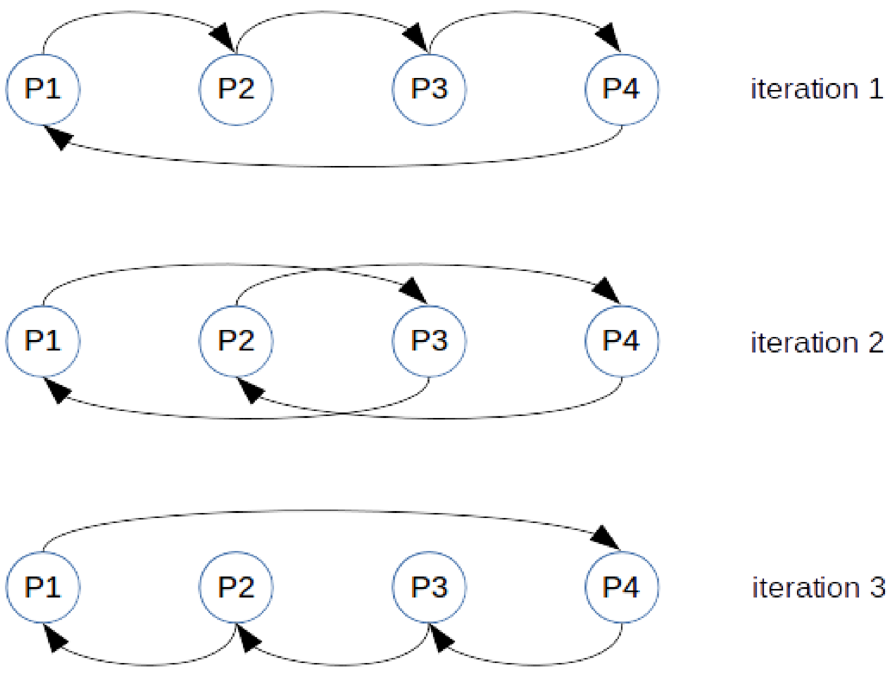


FIGURE 5.8: Target selection scheme for $N = 4$.

Step 1 of *Designated Open Commitment* is similar to that of *Open Commitment*. One difference is that, $F_0(x)$ is privately sent to the target party instead of being broadcast. Commitment identifier (or name) and identifier of the target (`partyID`) is broadcast separately, so that all players know about ongoing designated open runs. Thanks to the target selection scheme, each player knows which opening player is allowed to target her at any round, hence honest players will notice when an invalid designated open intention

[23]For none of the protocols that uses this scheme (*Designated Open Commitment*, *Transfer Commitment*, *VSS*), it makes sense for a party to select itself as the target.

is broadcast. By privately sending a polynomial different than that used in the commitment ($F_0(x)$), a corrupted player might try to (designated) open her commitment with a different value. See *Case 7* in Section 5.4.5.

In step 2, players who are selected as targets of a designated open, update their commitment records with the privately sent polynomial. Protocol *Designated Open Commitment* guarantees that, if the opening succeeds, by evaluating this polynomial at 0, players will get the value originally committed to. However, further interaction is required to decide success. For each commitment being designated opened, each player privately sends her share to the corresponding target. An actively corrupted player might privately send a different share to sabotage an opening. See *Case 8* in Section 5.4.5.

In step 3, for the commitment being designated opened to her, a player counts the number of privately sent shares, that are consistent with the polynomial privately sent to her in step 1. If total number of consistent shares is not greater than $2D$, an honest target will broadcast a message, saying that she *rejects* the designated open.

In step 4, an honest commitment owner whose designated open got rejected in step 3, runs *Open Commitment* protocol with the same commitment. Players who don't have to open a commitment, passively participate in openings of other players. For each designated open that was rejected, if the subsequent *Open Commitment* fails, opening player is marked as corrupt and designated open also fails. Otherwise, designated open succeeds. If the designated open was not rejected in step 3, honest players mark their records for this commitment as *designated opened* to the corresponding target. An actively corrupted player whose designated open got rejected in step 3, might refuse to run *Open Commitment*. See *Case 9* in Section 5.4.5.

Transfer Commitment

Protocol *Transfer Commitment* uses *Commitment*, *Designated Open Commitment*, and *Open Commitment* protocols as subprotocols. By running *Transfer Commitment*, a player (who will be referred to as the *source of transfer*, or *source*) transfers a commitment she previously made, to a single player (who will be referred to as the *transfer target*, or *target*). After successful completion of the protocol, transfer target will end up being the owner of a new commitment, such that the committed value in the new commitment is guaranteed to be equal to the value committed to by the source in the original transferred commitment. Furthermore, players other than the transfer target will learn nothing about this value.

While one honest player is running *Transfer Commitment*, every other honest player simultaneously runs *Transfer Commitment* to transfer a commitment of their own. The target selection scheme, which was previously described in Section 5.4.4, guarantees that a player is never the target of more than one commitment transfer (with an honest transfer source).

In step 1, players broadcast identifiers of the commitments they intend to transfer, and identifiers of the transfer targets. Following a communication round, players read the broadcast messages to enforce the target selection scheme and to initialize the list of transfers they will keep internally. Each honest player runs *Designated Open* with the commitment they intend to transfer, where the open target is the target of transfer. In the steps that follow, some of the transfers will be marked as *erroneous* if malicious behaviour is observed from either the source or the target. In the rest of this section, transfers which have not (yet) been marked as erroneous, will be referred to as *ongoing transfers*.

In step 2, players update their list of transfers according to the results of the *Designated Open* runs from step 1. If a commitment to be transferred was not opened to the transfer target, transfer is marked as erroneous. Otherwise, an honest target makes a commitment to the value opened to it, by running protocol *Commitment*. Players, who are targets of erroneous transfers passively participate in the commitments of others. An actively corrupted player might make a commitment to a value different than the one opened to her by the source. See *Case 10* in Section 5.4.5.

In step 3, players update their list of transfers according to the results of the *Commitment* runs. If a commitment failed, corresponding transfer is marked as erroneous. In the rest of this step, transfer target and transfer source will have to work together to create a piece of *evidence* to convince other players that the commitment being transferred, and the new commitment made by the transfer target are to the same value. Each source of an ongoing transfer will sample a univariate polynomial of degree D, whose constant term is equal to the opened value of the commitment being transferred, and commit to every coefficient by running protocol *Commitment*. Sources of erroneous transfers will passively participate in commitments of others. If an honest source samples a polynomial in this step, she privately sends the coefficients to the transfer target, so that the target can do its part in creation of the necessary evidence. An actively corrupted transfer source might send a wrong value, i.e. a value different than the committed value, for one or more of the coefficients. See *Case 11* in Section 5.4.5.

Following a communication round, each target who received coefficients, commits to each of the received coefficients, by running the protocol *Commitment* D times. No more commitments will occur in the rest of the protocol. Each player goes over her list of transfers and marks a transfer as erroneous if one or more of the commitments associated with the transfer failed. Next, for each ongoing transfer, each player will *locally and consistently* create committed shares, once from the source's version of the sampled polynomial, and once from the target's version of the sampled polynomial. As an example, we consider a single transfer, where the commitment being transferred is `commit_source`, the commitment made by the transfer target in step 2 is `commit_target`, commitments made by the source to the coefficients are $\texttt{coeff\_source}_i$, and commitments made by the target to the privately received coefficients are $\texttt{coeff\_target}_i$. For this transfer, committed shares created from source's version of the polynomial are

$$\texttt{share\_source}_k = \texttt{commit\_source} + \sum_{i=1}^{D} k^i \cdot \texttt{coeff\_source}_i$$

and committed shares created from target's version of the polynomial are

$$\texttt{share\_target}_k = \texttt{commit\_target} + \sum_{i=1}^{D} k^i \cdot \texttt{coeff\_target}_i$$

Once all honest players carry out the local operations described above, sources and targets are able to distribute the evidence mentioned earlier. Note that each player is the source of (exactly) one transfer, and target of (exactly) one other transfer. First, each player will assume the role of a transfer source, and if the corresponding transfer has not been marked as erroneous, run *Designated Open* $N - 1$ times to open the committed share $\texttt{share\_source}_k$ to every other player P_k.[24] Next, each player will assume the role of a transfer target, and if the corresponding transfer has not been marked as erroneous, run *Designated Open* $N - 1$ times to open the committed share $\texttt{share\_target}_k$ to every other player P_k.

In step 4, for each ongoing transfer, each player will use the provided evidence to determine whether or not the commitment being transferred, and the new commitment made by the transfer target are to the same value. For each ongoing transfer, a player

[24]Target for the consecutive *Designated Open* runs at each iteration and the transfer target are both determined by the target selection scheme. Consequently, we have a target selection scheme running within a target selection scheme.

P_k checks whether $\texttt{share\_source}_k$ and $\texttt{share\_target}_k$ have been opened to her, and whether the opened values are equal. If not, she rejects that transfer. In the end, P_k broadcasts a message, listing all the transfers she rejects. An actively corrupted player might reject one or more transfers even though the opened values are equal. See *Case 12* in Section 5.4.5.

Following a round of communication, each player takes note of the broadcast rejections. If a player provided evidence in step 3 (either as the source or as the target), for each rejection, she opens again the shares of the rejecting player, but this time to all players by running protocol *Open Commitment*. What we have here is the execution of a common strategy, which can also be observed in other parts of protocol P_{CEAS}: A portion of shares are revealed to resolve claims of malicious behaviour, while the upper bound on the number of corrupted players ensures that privacy is not breached.

In order to minimize the number of times protocol *Open Commitment* is run, players go over all rejections to determine the maximum number of openings a single player has to make. Each player opens the commitments it has to open, and then passively participates in the rest of the openings.

Next, players go over the list of transfers once again. If a source or target player failed to open a commitment for any of the rejections, she is marked as corrupt and the transfer is marked as erroneous. For each rejection associated with an ongoing transfer, each player checks whether or not the opened values are equal. If the opened values turned out to be same, honest players will know that the rejector lied and can mark her as corrupt.[25] Otherwise, the transfer is marked as erroneous. At this point, the protocol is done with transfers having honest targets and sources.

In step 5, players handle the transfers which have been marked as erroneous at some

[25]Marking dishonest rejectors is optional. Omitting it does not affect the security of the protocol.

point during the execution of the protocol. For each erroneous transfer, transfer source opens the commitment being transferred, by running the *Open Commitment* protocol. Note that, at this point it is known that either the source or the target is corrupted, hence no information is being revealed that is not already known to the adversary. Here, an actively corrupted transfer source might refuse to open her commitment. See *Case 13* in Section 5.4.5.

Following the openings, each player goes over the list of erroneous transfers. If a source failed to open her commitment, she is marked as corrupt, and nothing else needs to be done for this transfer. As we shall see later when the *VSS* protocol is described, honest players will simply assume 0 for the values of shares they receive from corrupted players. If the source did open her commitment, honest players force the transfer target to do a *public commitment* to the opened value: every honest player updates the commitment record corresponding to the commitment made by the transfer target in step 2, such that the committed value is equal to the value opened by the source. Note that the protocol does not in general allow honest players to pinpoint the corrupted player. What honest players know is that, one or both of the source-target pair is corrupted for an erroneous transfer.

Commitment Multiplication

Protocol *Commitment Multiplication* uses *Commitment*, *Designated Open Commitment*, and *Open Commitment* protocols as subprotocols. By running *Commitment Multiplication*, a player creates a new commitment (which is referred to as the *committed product*) from two existing commitments (which are referred to as the *committed multiplicands*). If the player is honest, opened value of the committed product is the product of opened values of the two committed multiplicands. If the protocol succeeds, all players will know that the value committed to is indeed the product, but they will learn nothing about the opened values of the committed multiplicands and the committed product. While one honest player is running *Commitment Multiplication*, every other honest player simultaneously runs *Commitment Multiplication* with different committed multiplicands. A player can multiply commitments only if she is the owner of both of them. The player who carries out the multiplication is referred to as the *multiplication owner*.

In step 1, players commit to the product of opened values of the multiplicands by running protocol *Commitment*, creating the committed product in the process. An actively corrupted multiplication owner might make a commitment to a value different than the product. See *Case 14* in Section 5.4.5. Following the commitments, each player broadcasts a message, including identifiers of the committed multiplicands and identifier of the committed product.

In step 2, each player reads the broadcast messages and stores a list of multiplications internally. In the following steps, if malicious behaviour is observed from any multiplication owner, the corresponding multiplication will be marked as *erroneous*. In the rest of this section, multiplications which have not (yet) been marked as erroneous, will be referred to as *ongoing multiplications*. Each multiplication owner samples two univariate polynomials of degree D, such that the constant terms are the opened values of the committed multiplicands. These polynomials will be referred to as f and g. Multiplication owner then multiplies f and g to obtain h, which is a polynomial of degree $2D$. Note that, if the multiplication is done honestly, constant term of h will be equal to the opened value of the committed product. Next, each multiplication owner commits to each coefficient of these three polynomials, by consecutively running protocol *Commitment*. Following the commitments, each player goes over its list of multiplications, and if one or more commitments associated with a multiplication failed, the multiplication is marked as erroneous.

In step 3, purpose of a multiplication owner is to provide other players with evidence that the committed multiplicands and the committed product form a multiplication triple. For each ongoing multiplication, each player *locally and consistently* creates committed shares. As an example, we consider a single multiplication, where the committed multiplicands are `mult1` and `mult2`, the committed product is `prod`, and commitments made to coefficients of f, g, h are $\texttt{coeff\_f}_i$, $\texttt{coeff\_g}_i$, $\texttt{coeff\_h}_j$, respectively. Committed shares created for this multiplication are:

$$\begin{aligned}\texttt{share\_f}_k &= \texttt{mult1} + \sum_{i=1}^{D} k^i \cdot \texttt{coeff\_f}_i \\ \texttt{share\_g}_k &= \texttt{mult2} + \sum_{i=1}^{D} k^i \cdot \texttt{coeff\_g}_i \\ \texttt{share\_h}_k &= \texttt{prod} + \sum_{j=1}^{2D} k^j \cdot \texttt{coeff\_h}_j\end{aligned}$$

Once all honest players carry out the local operations as described above, multiplication owners are able to distribute the evidence mentioned earlier. An honest multiplication owner runs *Designated Open* $N-1$ times for each one of f, g and k, to open the committed shares $\texttt{share\_f}_k$, $\texttt{share\_g}_k$ and $\texttt{share\_h}_k$ to every other player P_k. At each iteration, target for designated open is determined by the target selection scheme.

In step 4, players use the shares provided by multiplication owners to determine whether or not the multiplications are honestly performed. For each ongoing multiplication, a player P_k checks whether $\texttt{f}_k$, $\texttt{g}_k$ and $\texttt{h}_k$ have been opened to her, and whether the opened values form a multiplication triple, with opened value of $\texttt{h}_k$ as the product. If not, she rejects that multiplication. In the end, P_k broadcasts a message, listing all the multiplications she rejects. An actively corrupted player might reject one or more multiplications even though the opened values form a multiplication triple. See *Case 15* in Section 5.4.5.

In step 5, players take note of the broadcast rejections. If a player distributed shares in step 3, for each rejection, she opens again the shares of the rejecting player, but this time to all players by running protocol *Open Commitment*. In order to minimize the number of times protocol *Open Commitment* is run, players go over all rejections to determine the maximum number of openings a single player has to make. Each player opens the commitments it has to open, and then passively participates in the rest of the openings.

Next, players go over the list of multiplications in order to handle rejections. If a multiplication owner failed to open a committed share for any of the rejections, or if the opened values are not the operands and product of a multiplication, multiplication owner is marked as corrupt. Otherwise, honest players will know that the rejector lied and can mark her as corrupt.[26]

Verifiable Secret Sharing (VSS)

Protocol *VSS* uses *Commitment* and *Transfer Commitment* protocols as subprotocols. *VSS* is called within P_{CEAS} during the input sharing phase when the input providers secret share their private inputs, and during the computation of multiplication gates when the players secret share their local products prior to degree reduction. By running the *VSS* protocol, a player distributes committed shares of a secret. The player, who distributes the committed shares, is referred to as the *distributing player*. A VSS run is referred to as an *ongoing VSS*, if the distributing player has not yet been marked as corrupt. After completion of the protocol, either all honest players hold consistent shares of the secret,

[26]Marking dishonest rejectors is optional. Omitting it does not affect the security of the protocol.

or all honest players agree that the distributing player is corrupted. While one honest player is running VSS, every other honest player simultaneously runs VSS for a secret of their own. If the distributing player does not already own a commitment to the secret to be secret shared, first she commits to it by running protocol *Commitment*.

In step 1, each distributing player samples a univariate polynomial of degree D, such that the constant term is equal to the secret, and commits to each coefficient of the polynomial, by consecutively running protocol *Commitment* D times. Each distributing player broadcasts the identifier of the commitment to her secret. If VSS is being run during input sharing phase, sender also includes in the message the label associated with the private input.

If one or more of the commitments in step 1 failed, corresponding distributing players are already marked as corrupt by all honest players by the time step 2 begins. In step 2, each player goes over the broadcast messages. For each ongoing VSS, each player will *locally and consistently* create committed shares. As an example, we consider a single VSS, where the commitment to the secret is `secret`, and commitments made to the coefficients are $\texttt{coeff}_i$. Committed shares created for this VSS are

$$\texttt{share}_k = \texttt{secret} + \sum_{i=1}^{D} k^i \cdot \texttt{coeff}_i$$

At this stage, honest players have stored commitment records for every share of every player. If VSS is being run during input sharing phase, players also store the labels read from the messages, without breaking their association with the shares.

In step 3, each honest distributing player transfers the shares $\texttt{share}_k$ to every other player by calling *Transfer Commitment* consecutively $N-1$ times, where the transfer target at each iteration is determined by the target selection scheme. During these transfers, some players might be marked as corrupt. However, even when the source of malicious behaviour cannot be pinpointed, consistency of shares is guaranteed via the forced public commitments, which take place in the final step of the *Transfer Commitment* protocol. Following the transfers, each player checks -not only for the shares transferred to her, but for all the shares- whether the distributing player is marked as corrupt.[27] If a distributing player P_i is corrupt, honest players will do a *public commitment* to 0, i.e. they will locally create a committed share of 0 with P_i as its owner, and use this instead of the shares received from P_i.[28] Finally, if VSS is being run during the input sharing phase, committed share is updated with the label stored in step 2.

5.4.5 Cases of Malicious Behaviour

This section describes the cases of malicious behaviour, which can be simulated for protocol P_{CEAS}. Some of the cases have player identifiers hard-coded into them, so care should be taken when choosing the corrupted players. When stating the effect of malicious behaviour for a particular case, it is assumed that all other cases are deactivated.

Case 1

In step 2 of protocol *Commitment*, corrupted player(s) sends a defective verifier to player P_1 for all commitments. As a result, P_1 disputes all commitments. At Step 4, owners broadcast values for all disputes. Broadcast values are accepted. No player gets accused, and all commitments succeed.

[27]Following a transfer, `owner` of a committed share is the transfer target. Hence, identifier of the distributing player is stored separately within the commitment record.

[28]The choice of value 0 is arbitrary and have no effect on the computation result, as the corrupted player will be excluded from recombination.

Case 2

In step 4 of protocol *Commitment*, corrupted player(s) refuses to broadcast in response to disputes concerning her commitment. As a result, her commitment fails and she is marked as corrupt.

Case 3

In step 5 of protocol *Commitment*, corrupted player(s) makes a false accusation directed at player P_1. P_1 broadcasts the verifiable share of the accuser. Eventually, commitment of (wrongfully) accused P_1 succeeds. Furthermore, accusing player updates its record with the newly broadcast verifiable share.[29]

Case 4

In step 6 of protocol *Commitment*, corrupted player(s) refuses to broadcast verifiable shares in response to accusations concerning her commitment. As a result, her commitment fails and she is marked as corrupt.

Case 5

In step 1 of protocol *Open Commitment*, corrupted player(s) broadcasts the negative of the polynomial used in commitment, hence tries to open her commitment as negative of the value originally committed to. As a result, her opening does not succeed and she is marked as corrupt by all honest players. Openings of other players are not affected.

[29]Though, this is not the case where this code piece realizes its true purpose. It is meant for handling a commitment owner who starts maliciously, but later behaves honestly to have her commitment accepted. This case was mentioned during the description of step 8 of protocol *Commitment*.

Case 6

In step 2 of protocol *Open Commitment*, instead of broadcasting her share, a corrupted player(s) increments her share by 1 and broadcasts that value, for all ongoing commitment openings. Despite the sabotage attempt, openings of honest players succeed.

Case 7

In step 1 of *Designated Open Commitment* protocol, corrupted player(s) privately sends to the target, the negative of the polynomial used in commitment, hence tries to open her commitment with the negative of the value originally committed to. As a result, her designated open gets rejected and she is forced to open her commitment by running *Open Commitment*.

Case 8

In step 2 of *Designated Open Commitment* protocol, instead of (privately) sending her share, a corrupted player(s) increments her share by 1 and sends that value, for all ongoing designated open runs. Despite the sabotage attempt, designated open runs of honest players succeed.

Case 9

In step 4 of *Designated Open Commitment* protocol, corrupted player(s) refuses to open her commitment (by running *Open Commitment*), after her designated open gets rejected by the target. As a result, she is marked as corrupt by all honest players. Designated open fails and her commitment remains unopened.

Case 10

In step 2 of *Transfer Commitment* protocol, corrupted transfer target(s) increments the value opened to her by transfer source P_3 by 1, and makes a commitment to the incremented value instead. As a result, the transfer from P_3 to the corrupted target is rejected by the honest players. P_3 opens her commitment in step 5, and the commitment made by the corrupted target is overridden by the forced public commitment to the value opened by P_3.

Case 11

In step 3 of *Transfer Commitment* protocol, corrupted transfer source(s) privately sends a coefficient value (for the first coefficient) to transfer target P_3, where the value sent is the value committed to incremented by 1. As a result, transfer gets rejected by the honest players. Corrupted source opens her commitment in step 5, and the commitment made by P_3 is overridden by the forced public commitment to the value opened by the corrupted source.

Case 12

In step 4 of *Transfer Commitment* protocol, corrupted player(s) rejects the transfer from player P_3 to player P_1, where P_1 and P_3 are both set as honest players. As a result, P_1 and P_3 are forced to open the committed shares associated with the rejection, which they previously opened to the corrupted player via *Designated Open*. In step 5, each honest player checks the opened values. They match, and the transfer from P_3 to P_1 is not marked as erroneous.

Case 13

In step 5 of *Transfer Commitment* protocol, corrupted transfer source(s), whose transfer is marked as erroneous, refuses to open her commitment. As a result, she is marked as corrupt by all honest players.

Case 14

In step 1 of *Commitment Multiplication* protocol, corrupted player(s) increments the value she calculated as the product by 1, and makes a commitment to the incremented value instead. As a result, multiplication is rejected by all honest players. Corrupted multiplication owner is forced to open her commitments in step 5, and is marked as corrupt by all honest players, as the opened values are not the operands and product of a multiplication. Other multiplications are not affected.

Case 15

In step 4 of *Commitment Multiplication* protocol, corrupted player(s) rejects the multiplication of honest player P_3. As a result, P_3 is forced to open the committed shares associated with the rejection, which she previously opened to the corrupted player via *Designated Open*. In step 5, each player observes that the opened values are the operands and product of a multiplication. P_3 is not marked as corrupt. Rejector is marked as corrupt by all honest players.

5.4.6 Implementation of Circuit Randomization

It was mentioned in Section 2.6 that a protocol in the preprocessing model runs in two distinct phases: *preprocessing phase* and *online phase*. When the simulator is set to run with protocol $P_{CEAS,CR}$, the protocol for the preprocessing phase ($P_{CEAS,P}$) is run right before the protocol for the online phase ($P_{CEAS,O}$), as we saw no reason to temporally separate their executions for simulations. Recall that, our purpose in using circuit randomization is to decrease the interactivity required for computation of multiplication gates. Hence, it is no surprise that, computation of multiplication gates is different in $P_{CEAS,O}$, compared to P_{CEAS}. First, the implementation of the preprocessing phase is described.

$\mathbf{P_{CEAS,P}}$ - Preprocessing Phase

In preprocessing phase, purpose of the players is to generate and store sufficiently many multiplication triples. Normally, preprocessing phase is independent of the circuit to be evaluated, but for the sake of convenience, instead of making a guess for *sufficiently many*, we peek at the circuit and count multiplication gates. Minimum number of triples needed is equal to the number of multiplication gates, because using a triple twice will leak information. Lets assume M triples are needed. An internal data structure is initialized to hold M multiplication triples. The generation of a single triple is described below.

Each player P_i randomly chooses two values $r_{1i}, r_{2i} \in \mathbb{Z}/p\mathbb{Z}$, each of which will be used for one of the multiplicands. For the first multiplicand, each P_i runs protocol *VSS* to distribute T-shares of r_{1i}. Next, using homomorphism of the commitment scheme, all the received committed shares are added up to form the random sharing:

$$[[r]] = \sum_i [[r_{1i}]]$$

Value r is not known to any of the players. Note that, even though the shares are summed up, any linear combination with nonzero coefficients would do. As long as a single player is honest, r will be a random value. $[[r]]$ is used as the committed share for the first multiplicand. Committed share for the second multiplicand is computed in a similar way, using r_{2i} instead of r_{1i}:

$$[[r\prime]] = \sum_i [[r_{2i}]]$$

Next, each player runs protocol *Commitment Multiplication* with the commitments to the multiplicands, creating in the process a commitment to their product. Each party runs protocol *VSS* again, this time to distribute committed shares of the product. Note that these shares (of the product) are not T-shares, so a degree reduction will be needed before a triple of T-shares is obtained. Players store their shares for the operands and the product of this multiplication triple, in the data structure that holds the triples.

$\mathbf{P_{CEAS,O}}$ - Online Phase

Only the computation of multiplication gates will be described, because the rest of $P_{CEAS,O}$ is the same as P_{CEAS}. Computation of multiplication gates in $P_{CEAS,O}$ requires two new kinds of local operations: *addition with a constant* and *subtraction*.

Adding a constant c to a commitment `comm` is done by first creating a commitment record with c as the committed value, and owner of `comm` as the owner.[30] Then the two commitment records are added, as described in Section 5.4.3.

Subtraction of commitments can be expressed in terms of *addition* and *multiplication with a constant*. The commitment being subtracted is multiplied by the constant -1, as described in Section 5.4.3, and then added to the other commitment.

Players start computation of a multiplication gate by locating the multiplication triple they generated in preprocessing phase for this particular gate. The two commitments assigned to the input wires of the gate are retrieved. The multiplicands of the triple will

[30]Note that, owner of `comm` may be different than the player performing the operation.

be denoted by x and y, and the gate inputs will be denoted by a and b. Players *locally and consistently* create the following new commitments via *subtraction* of commitments

$$e = a - x$$

$$d = b - y$$

and then each player opens e and d, by running protocol *Open Commitment* twice.

Next, players perform degree reduction on the shares of the product, which they received and stored during the preprocessing phase. They remove the shares distributed by players marked as corrupt, and if they are left with more than $2D$ shares, they combine these using the recombination vector to obtain T-shares. The T-share of the product will be denoted by $x \cdot y$.

Using the triple (x, y, $x.y$), inputs a and b, and the identity from Section 2.6.1

$$a \cdot b = x \cdot y + e \cdot b + d \cdot a - e \cdot d$$

players will form a committed T-share for the product of a and b, which will be denoted by $a \cdot b$. Note that e and d were opened to all players and will be treated as constant values. The usual local operations are carried out on commitments, except that *addition with a constant* is used for adding $-e \cdot d$. All local operations are performed *locally and consistently*. Players assign their committed T-share $a \cdot b$ to all output wires of the gate. Gate computation is completed.

The only part of $P_{CEAS,O}$ that requires interaction is the part where e and d are opened by running protocol *Open Commitment* twice. For P_{CEAS} on the other hand, computation of a multiplication gate requires running *Commitment Multiplication* and *VSS*, both of which have relatively high round complexity. With circuit randomization, the cost of running these protocols are pushed to the preprocessing phase. In Section 6.1.2, the number of communication rounds required in computation phases of $P_{CEAS,O}$ and P_{CEAS} are compared.

Chapter 6

Round Complexities and General Computing

In this chapter, we first securely evaluate a few simple arithmetic functions to observe and to compare the round complexities of the implemented protocols. Next, we attempt to extend the implementation to support more general computations. In particular, the computing parties are provided with the ability to remember shares from previous computations, and a custom arithmetic circuit is built for a specific computation, namely, secure comparison of integers.

6.1 Round Complexities

In this section, the implemented protocols are executed in order to observe and to compare their performance in terms of the number of required communication rounds.

TABLE 6.1: Total number of communication rounds required by the protocols.

$(\mathbf{T},\mathbf{N})$	♯ **Corrupted**	$\mathbf{P_{CEPS}}$ $\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{C_3}$	$\mathbf{P_{CEAS}}$ $\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{C_3}$
$(2,4)$	0	4	5	6	475	741	1007
$(3,7)$	0	4	5	6	1405	2164	2923
$(3,7)$	1	-	-	-	1423	2194	2965
$(3,7)$	2	-	-	-	1441	2224	3007
$(4,10)$	0	4	5	6	2815	4307	5799

6.1.1 $\mathbf{P_{CEPS}}$ vs. $\mathbf{P_{CEAS}}$

We start with a circuit containing a single multiplication gate, then we double and triple the amount. Description strings for the circuits to be evaluated are

$$C_1 : x_1 * x_2$$

$$C_2 : x_1 * x_1 + x_1 * x_2.2$$

$$C_3 : x_1 * x_1 + x_1 * x_2.2 + x_2 * x_2$$

In addition to the number of multiplication gates (M), protocol parameters T and N, and the number of actively corrupted players (only for P_{CEAS}) are varied. All cases of malicious behaviour are enabled during the runs, so the actively corrupted players specified in the options file run every case of malicious behaviour described in Section 5.4.5. Table 6.1 shows the parameters chosen, and the total number of communication rounds required for evaluating the circuits ($Total_{round}$). For all runs, we have $p = 4973$, $x_1 = 10$, $x_2 = 11$.[1]

For protocol P_{CEPS}, while $Total_{round}$ increases with the number of multiplication

[1]Options files used for these runs are `round_complexity_c1`, `round_complexity_c2` and `round_complexity_c3`. They can be found under `options` folder.

gates M,[2] it is independent of the number of players N. However, as N increases, each player has to send a greater number of messages per round, and eventually longer rounds might be required.

For protocol P_{CEAS}, $Total_{round}$ increases with both M and N. Inspection of the protocol descriptions given in Section 4.1.1 and Section 4.1.2 suggests that both protocols require $\mathcal{O}(M)$ rounds, and the values observed for $Total_{round}$, given in Table 6.1, support this claim. Understanding how exactly $Total_{round}$ increases with N is not as easy as it was for M. However, one can easily see why the increase occurs, just by looking at what it takes to share a secret via the VSS protocol: a player has to execute a multi-round protocol for each additional player. The results on the second, third and fourth rows of Table 6.1 shows that, malicious behaviour has an effect on $Total_{round}$: an active adversary can increase the runtime of the protocol, but not indefinitely. The increase in $Total_{round}$ may stem from several different cases. For example, during *Transfer Commitment*, a false rejection from a corrupted player can force additional executions of protocol *Open Commitment*, as mentioned in *Case 12* in Section 5.4.5.

As one goes for more meaningful computations, total number of rounds required might become unmanageably large. Table 6.2 shows the total number of rounds required by P_{CEAS} for the mean value and standard deviation calculations carried out in Section 5.1. There we had $p = 100000007$, and the private inputs were given in Table 5.2.[3]

[2]As the total number of rounds required depends on the evaluated circuit, by definition, P_{CEPS} is *not* a constant-round protocol.

[3]Options files used for these runs are `motiv_mean_times_sum`, `motiv_stdev_sq_times_sum` and `motiv_sum_bdrug`. They can be found under `options` folder.

TABLE 6.2: Total number of communication rounds required by P_{CEAS} for the computations in Section 5.1.

$(\mathbf{T}, \mathbf{N})$	$\sharp$ **Corrupted**	$\mathbf{C_D}$	$\mathbf{C_\mu}$	$\mathbf{C_\sigma}$
$(4, 10)$	0	1323	17528	33733

TABLE 6.3: Number of communication rounds required by the protocols in computation phase.

	$\mathbf{P_{CEAS}}$			$\mathbf{P_{CEAS,CR}}$		
$(\mathbf{T}, \mathbf{N})$	$\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{C_3}$	$\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{C_3}$
$(2, 4)$	267	533	799	7	13	19
$(3, 7)$	760	1519	2278	7	13	19
$(4, 10)$	1493	2985	4477	7	13	19

6.1.2 $\mathbf{P_{CEAS}}$ vs. $\mathbf{P_{CEAS,CR}}$

We use the same three circuits C_1, C_2, and C_3 that were used in Section 6.1.1. Table 6.3 shows the number of communication rounds required ($Comp_{round}$) in computation phases of P_{CEAS} and $P_{CEAS,CR}$.[4] For all runs, we have $p = 4973$, $x_1 = 10$, $x_2 = 11$. Number of actively corrupted players is 0. It was noted in Section 5.4.6 that, all the interaction that takes place in computation phase of $P_{CEAS,CR}$ is due to a constant number of *Open Commitment* runs. Therefore, it is no surprise that

- a drastic reduction in $Comp_{round}$ is observed, compared to P_{CEAS}
- $Comp_{round}$ does not depend on N

[4]Gate computation in $P_{CEAS,CR}$ occurs during the online phase, so it could also be said that the comparison is between computation phases of P_{CEAS} and $P_{CEAS,O}$.

6.2 Performing General Computations

Secure computations performed in the previous sections involved only addition and multiplication of integers. This section presents two extensions to the simulator, which enable more general computations. In Section 6.2.1, players are given the ability to remember the result of a previous computation, effectively giving them a restricted version of an internal state. In Section 6.2.2, the simulator is given the ability to generate and evaluate a family of circuits custom-made for secure comparison of integers.

6.2.1 Remembering Previous Results

It was mentioned in Section 2.3.1 that, SFE can be extended to a reactive functionality by keeping and maintaining internal state. Later, during the computation of standard deviation in Section 5.1, we were faced with a situation, where an efficient solution required the players to keep their shares from the mean value calculation. The simulator is extended, so that the shares from previous evaluations are remembered, when it is run with the *sequential run* option.

Sequential Run

If a computation repeats within a circuit description string, circuit generator will not attempt any optimizations. For example, `(a*b)` occurs twice in `(a*b)+(a*b)` and the generated circuit will have two **MUL** gates when one would be enough.[5] A way to avoid this undesirable situation is to do a *sequential run*.[6] When sequential run is enabled via

[5]If we hard-coded this circuit, we could add two output wires coming out of the single **MUL** gate and feed them as inputs to the **ADD** gate.

[6]Sequential run is implemented for P_{CEAS} only.

the options file,[7] players hold on to their shares of the computation result from the first computation, and assign them to appropriate wires in the circuit for the second computation. For the example given above, first run would evaluate `(a*b)`, and second run would evaluate `c+c`, where players would assign their shares for the result of first computation to wires labeled with `c`. This feature takes us one step closer to a reactive functionality.

6.2.2 Building a Circuit For Comparison

Protocol Compare [41, p.192] is a protocol for secure comparison of private inputs. The protocol suggests a particular way of comparing integers, which is conservative in terms of multiplication operations.[8] The circuit generator is extended with the ability to generate arithmetic circuits, which mimic the way *Protocol Compare* compares its inputs. The generated circuits are referred to as *comparator circuits*. If a comparator circuit can compare secrets with at most l_{max} bits in their binary representations, we will say that l_{max} is the *size* of the comparator circuit. A flag in the options file, when set to `true`, tells the simulator that the circuit generator is to build a comparator circuit of a given size. Size of the circuit is also read from the options file.[9]

Let the private inputs be $a, b \in \mathbb{Z}/p\mathbb{Z}$. Let binary representations of a and b be $a_l \dots a_1 a_0$ and $b_l \dots b_1 b_0$, respectively. Indices run from 0 to l, where $l = l_{max} - 1 = \lfloor \log_2(\max(a, b)) \rfloor$. Result r will be 1, if $a > b$, and 0 otherwise.[10] An overview of the circuit is shown in Figure 6.1. We look at the internals of the named boxes shown in the figure, in the following subsections.

[7]A sample options file for a sequential run is `example_sequential_run`, and it can be found under `options` folder.

[8]This is how the SMC application mentioned in Section 3.3.1 handles comparisons.

[9]A sample options file for secure comparison is `example_comparator`, and it can be found under `options` folder.

[10]The asymmetry between inputs of the circuit can be observed in inputs to $C_{\Sigma XY}$. See Figure 6.1.

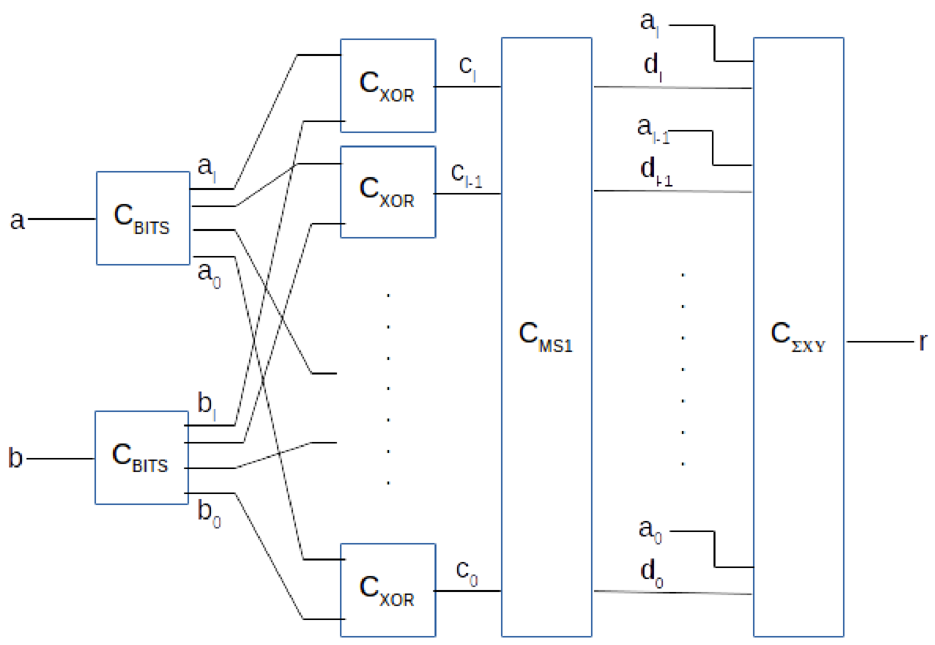


FIGURE 6.1: Overview of the comparator circuit.

C_{BITS} Box

For secure comparison of two committed shares, committed shares to each bit in their binary representations are needed. Figure 6.1 shows C_{BITS} boxes decomposing shares into individual bits of their binary representations. Cramer et al. [41] present a protocol for this functionality, which could be converted into a C_{BITS} circuit. But for the sake of brevity, C_{BITS} boxes are omitted, and instead we start with the binary representations of the inputs. Hence, an input provider, who wants her secret compared to another value, secret shares each bit in the secret's binary representation separately.

C_{XOR} Box

A C_{XOR} box XOR's the two bits provided as inputs. XOR of two bits a_i and b_i can be expressed as $a_i + b_i - 2a_ib_i$. Corresponding arithmetic circuit is shown in Figure 6.2.

Within the comparator circuit, the $l+1$ C_{XOR} boxes serve to identify the indices at which

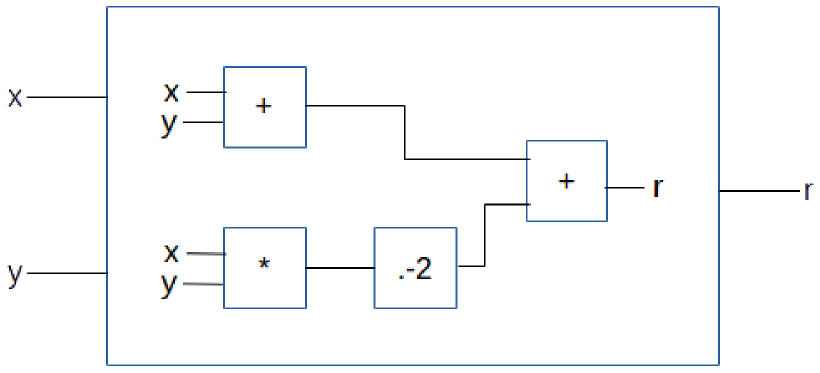


FIGURE 6.2: Internals of C_{XOR}.

binary representations of the inputs differ.

$\mathbf{C_{MS1}}$ Box

This box finds the index of the most significant 1 in the binary representation provided as input. Let the index of most significant 1 be k for input $c_l \ldots c_1 c_0$. Then, this box outputs $d_l \ldots d_1 d_0$, where $d_k = 1$ and $d_j = 0$ for all $j \neq k$. Hence, when the whole comparator circuit is considered, k is the index of the most significant bit such that $a_i \neq b_i$. Internals of C_{MS1} are shown in Figure 6.3. When the circuit is evaluated with P_{CEAS}, internal input wires labeled with '1' are to be assigned a committed share to value 1.[11] C_{SUB} boxes used within C_{MS1} subtract second input from the first. Internals of a C_{SUB} box are shown in Figure 6.4.

[11]See `@labelOne` in the sample options file `example_comparator`.

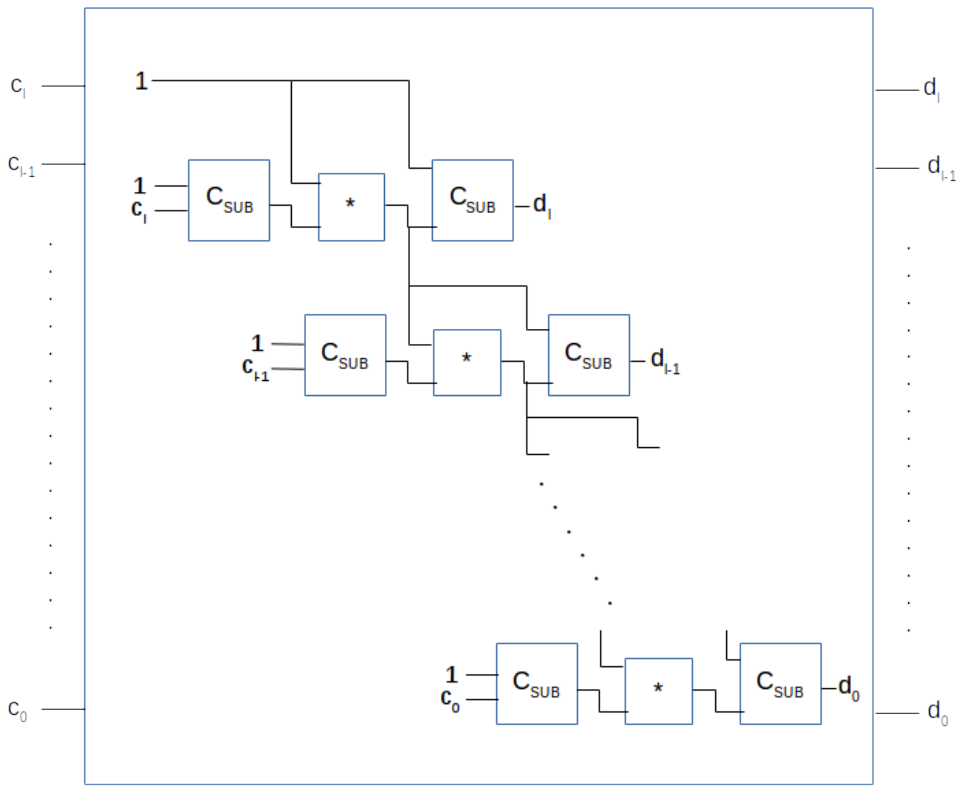


FIGURE 6.3: Internals of C_{MS1}.

$\mathbf{C_{\Sigma XY}}$ Box

$C_{\Sigma XY}$ box computes the sum $\sum_i a_i d_i$ for the two bit vectors a_i and d_i provided as inputs. This sum is also the result output from the comparator circuit. Considering the output of C_{MS1}, $d_i = 1$ only at index k, and because $a_k \neq b_k$, $a_k = 1$ only when $b_k = 0$. Hence, the sum is 1 if and only if $a > b$, and we have the desired behaviour. Internals of $C_{\Sigma XY}$ are shown in Figure 6.5.

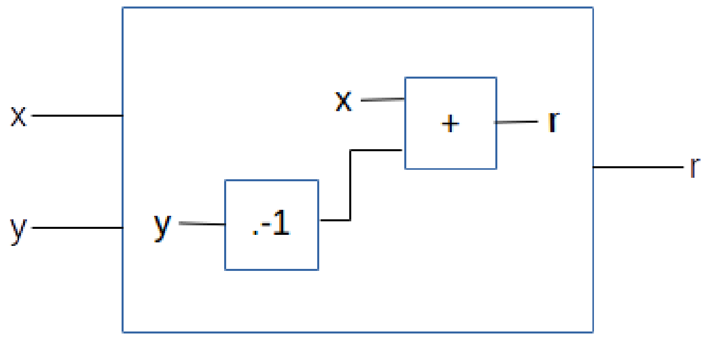


FIGURE 6.4: Internals of C_{SUB}.

TABLE 6.4: Summary of the secure comparisons.

Comparison	**Result**	**Size**	**Rounds**
`6 > 5`	1	3	9337
`6 > 6`	0	3	9337
`6 > 7`	0	3	9337
`15 > 10`	1	4	12234
`31 > 10`	1	5	15131

Secure Comparisons

This section summarizes the results of a few sample secure comparisons. Comparator circuits of varying sizes are evaluated using protocol P_{CEAS}. Table 6.4 shows the inputs, result, and the number of communication rounds required ($Total_{round}$). For all runs, we have $(T, N) = (3, 7)$, $p = 4973$. Number of actively corrupted players is 0.

It was noted in Section 6.1.1 that, round complexity of P_{CEAS} is $\mathcal{O}(M)$, where M is the number of multiplication gates in the circuit. Inspecting the internals of a comparator circuit reveals that, number of multiplication gates within a comparator circuit of size l_{max} is $3 \cdot l_{max}$, hence we expect $\mathcal{O}(S)$ round complexity, where S is the size of the comparator circuit. The results in Table 6.4 suggest that, $(Total_{round}, S)$ pairs fit on a line of slope 3, as expected.

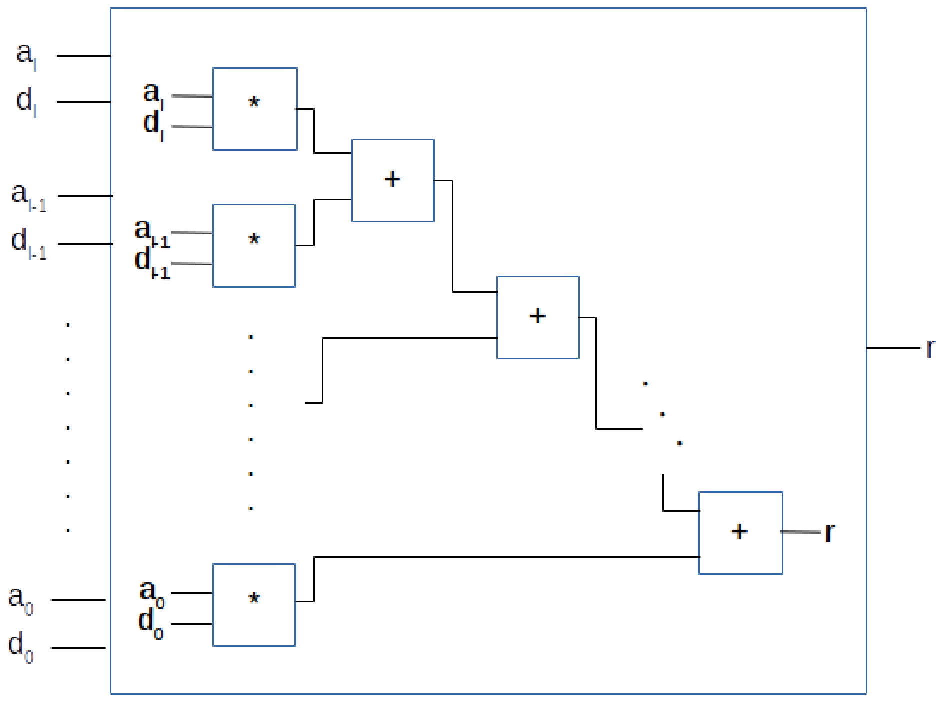


FIGURE 6.5: Internals of $C_{\Sigma XY}$.

Chapter 7

Concluding Remarks

We implemented two SMC protocols, P_{CEPS} and P_{CEAS}. We also implemented $P_{CEAS,CR}$, a version of P_{CEAS} which uses circuit randomization. However, we did not implement an efficient preprocessing phase. Doing so would increase the practical value of the $P_{CEAS,CR}$ implementation. While the honest majority assumptions of the implemented protocols may seem restrictive, there exists application scenarios where honest majority assumption makes sense, and some examples were mentioned in Section 3.3. Chapter 6 can be regarded as a general -in the sense that, no specific application scenario is considered- evaluation of the presented implementations' practical value. It was observed that, the number of communication rounds required for performing a computation becomes a concern, when a high number of computing parties are involved, and/or when the computation is complex and a high number of multiplication gates have to be computed. It was also observed that, additional effort is required in order to go beyond secure evaluation of simple arithmetic functions.

Creating an implementation with practical value is not one of the objectives stated in Section 1.1. Frameworks, implementations (See Section 3.2.3), and libraries [96, 81] with superior integrity and efficiency are readily available. However, for achieving our primary objective, namely, gaining a solid understanding of the basic concepts related to

SMC, we believe that reinventing the wheel was the better choice, and we are positive that the presented implementations have served their purpose in this respect.

Bibliography

[1] Ben Adida. “Helios: Web-based Open-audit Voting”. In: *Proceedings of the 17th Conference on Security Symposium*. SS’08. San Jose, CA: USENIX Association, 2008, pp. 335–348. URL: http://dl.acm.org/citation.cfm?id=1496711.1496734.

[2] Dakshi Agrawal and Charu C. Aggarwal. “On the Design and Quantification of Privacy Preserving Data Mining Algorithms”. In: *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’01. Santa Barbara, California, USA: ACM, 2001, pp. 247–255. ISBN: 1-58113-361-8. DOI: 10.1145/375551.375602. URL: http://doi.acm.org/10.1145/375551.375602.

[3] Rakesh Agrawal and Ramakrishnan Srikant. “Privacy-preserving Data Mining”. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’00. Dallas, Texas, USA: ACM, 2000, pp. 439–450. ISBN: 1-58113-217-4. DOI: 10.1145/342009.335438. URL: http://doi.acm.org/10.1145/342009.335438.

[4] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 2nd ed. Wiley Publishing, 2008. ISBN: 9780470068526.

[5] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. "Secure Multiparty Computations on Bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. May 2014, pp. 443–458. DOI: 10.1109/SP.2014.35.

[6] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. "How to Garble Arithmetic Circuits". In: *Proceedings of the 2011 IEEE 52Nd Annual Symposium on Foundations of Computer Science*. FOCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 120–129. ISBN: 978-0-7695-4571-4. DOI: 10.1109/FOCS.2011.40. URL: http://dx.doi.org/10.1109/FOCS.2011.40.

[7] Gilad Asharov, Daniel Demmler, Michael Schapira, Thomas Schneider, Gil Segev, Scott Shenker, and Michael Zohner. "Privacy-Preserving Interdomain Routing at Internet Scale". In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2017.3 (2017). To appear, pp. 143–163. URL: http://thomaschneider.de/papers/ADSSSSZ17.pdf.

[8] Yonatan Aumann and Michael O. Rabin. *A Proof of Plaintext Knowledge Protocol and Applications*. Manuscript. June 2001.

[9] Marshall Ball, Tal Malkin, and Mike Rosulek. "Garbling Gadgets for Boolean and Arithmetic Circuits". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 565–577. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978410. URL: http://doi.acm.org/10.1145/2976749.2978410.

[10] Michael Barbaro and Tom Zeller. "A Face Is Exposed for AOL Searcher No. 4417749". In: *New York Times* (2006). URL: http://query.nytimes.co

m/gst/fullpage.html?res=9E0CE3DD1F3FF93AA3575BC0A9609C8B63.

[11] Carsten Baum, Ivan Damgård, and Claudio Orlandi. "Publicly Auditable Secure Multi-Party Computation". In: *Security and Cryptography for Networks: 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*. Ed. by Michel Abdalla and Roberto De Prisco. Cham: Springer International Publishing, 2014, pp. 175–196. ISBN: 978-3-319-10879-7. DOI: 10.1007/978-3-319-10879-7_11. URL: http://dx.doi.org/10.1007/978-3-319-10879-7_11.

[12] Donald Beaver. "Correlated Pseudorandomness and the Complexity of Private Computations". In: *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 479–488. ISBN: 0-89791-785-5. DOI: 10.1145/237814.237996. URL: http://doi.acm.org/10.1145/237814.237996.

[13] Donald Beaver. "Efficient Multiparty Protocols Using Circuit Randomization". In: *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '91. London, UK, UK: Springer-Verlag, 1992, pp. 420–432. ISBN: 3-540-55188-3. URL: http://dl.acm.org/citation.cfm?id=646756.705383.

[14] Donald Beaver. "Precomputing Oblivious Transfer". In: *Advances in Cryptology — CRYPTO' 95: 15th Annual International Cryptology Conference Santa Barbara, California, USA, August 27–31, 1995 Proceedings*. Ed. by Don Coppersmith. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 97–109. ISBN:

978-3-540-44750-4. DOI: 10.1007/3-540-44750-4_8. URL: http://dx.doi.org/10.1007/3-540-44750-4_8.

[15] Donald Beaver. “Secure Multiparty Protocols and Zero-knowledge Proof Systems Tolerating a Faulty Minority”. In: *J. Cryptol.* 4.2 (Jan. 1991), pp. 75–122. ISSN: 0933-2790. DOI: 10.1007/BF00196771. URL: http://dx.doi.org/10.1007/BF00196771.

[16] Donald Beaver, Silvio M. Micali, and Phillip Rogaway. “The Round Complexity of Secure Protocols”. In: *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*. STOC ’90. Baltimore, Maryland, USA: ACM, 1990, pp. 503–513. ISBN: 0-89791-361-2. DOI: 10.1145/100216.100287. URL: http://doi.acm.org/10.1145/100216.100287.

[17] Zuzana Beerliová-Trubíniová. “Efficient Multi-Party Computation with Information-Theoretic Security”. PhD dissertation. ETH ZURICH, 2008. URL: http://www.crypto.ethz.ch/alumni/trubini/Beerli08.pdf.

[18] Zuzana Beerliová-Trubíniová and Martin Hirt. “Perfectly-Secure MPC with Linear Communication Complexity”. In: *Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings*. Ed. by Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 213–230. ISBN: 978-3-540-78524-8. DOI: 10.1007/978-3-540-78524-8_13. URL: http://dx.doi.org/10.1007/978-3-540-78524-8_13.

[19] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. “Foundations of Garbled Circuits”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012,

pp. 784–796. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382279. URL: http://doi.acm.org/10.1145/2382196.2382279.

[20] Assaf Ben-David, Noam Nisan, and Benny Pinkas. “FairplayMP: A System for Secure Multi-party Computation”. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS ’08. Alexandria, Virginia, USA: ACM, 2008, pp. 257–266. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455804. URL: http://doi.acm.org/10.1145/1455770.1455804.

[21] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. “Semi-homomorphic Encryption and Multiparty Computation”. In: *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology*. EUROCRYPT’11. Tallinn, Estonia: Springer-Verlag, 2011, pp. 169–188. ISBN: 978-3-642-20464-7. URL: http://dl.acm.org/citation.cfm?id=2008684.2008699.

[22] Michael Ben-Or, Ran Canetti, and Oded Goldreich. “Asynchronous Secure Computation”. In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*. STOC ’93. San Diego, California, USA: ACM, 1993, pp. 52–61. ISBN: 0-89791-591-7. DOI: 10.1145/167088.167109. URL: http://doi.acm.org/10.1145/167088.167109.

[23] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. “Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation”. In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*.

STOC '88. Chicago, Illinois, USA: ACM, 1988, pp. 1–10. ISBN: 0-89791-264-0. DOI: 10.1145/62212.62213. URL: http://doi.acm.org/10.1145/62212.62213.

[24] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. "Asynchronous Secure Computations with Optimal Resilience (Extended Abstract)". In: *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '94. Los Angeles, California, USA: ACM, 1994, pp. 183–192. ISBN: 0-89791-654-9. DOI: 10.1145/197917.198088. URL: http://doi.acm.org/10.1145/197917.198088.

[25] George R. Blakley. "Safeguarding cryptographic keys". In: *Proceedings of the 1979 AFIPS National Computer Conference*. Monval, NJ, USA: AFIPS Press, 1979, pp. 313–317.

[26] *Blockchain App Platform*. URL: https://www.ethereum.org/.

[27] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. "How the Estonian Tax and Customs Board Evaluated a Tax Fraud Detection System Based on Secure Multi-party Computation". In: *Financial Cryptography and Data Security: 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*. Ed. by Rainer Böhme and Tatsuaki Okamoto. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 227–234. ISBN: 978-3-662-47854-7. DOI: 10.1007/978-3-662-47854-7_14. URL: http://dx.doi.org/10.1007/978-3-662-47854-7_14.

[28] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. *Students and Taxes: a Privacy-Preserving Social Study Using Secure*

Computation. Cryptology ePrint Archive, Report 2015/1159. http://eprint.iacr.org/2015/1159. 2015.

[29] Dan Bogdanov, Sven Laur, and Jan Willemson. “Sharemind: A Framework for Fast Privacy-Preserving Computations”. In: *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*. ESORICS ’08. Malaga, Spain: Springer-Verlag, 2008, pp. 192–206. ISBN: 978-3-540-88312-8. DOI: 10.1007/978-3-540-88313-5_13. URL: http://dx.doi.org/10.1007/978-3-540-88313-5_13.

[30] Dan Bogdanov, Riivo Talviste, and Jan Willemson. “Deploying Secure Multi-Party Computation for Financial Data Analysis”. In: *Financial Cryptography and Data Security: 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*. Ed. by Angelos D. Keromytis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 57–64. ISBN: 978-3-642-32946-3. DOI: 10.1007/978-3-642-32946-3_5. URL: http://dx.doi.org/10.1007/978-3-642-32946-3_5.

[31] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. “Financial Cryptography and Data Security”. In: ed. by Roger Dingledine and Philippe Golle. Berlin, Heidelberg: Springer-Verlag, 2009. Chap. Secure Multiparty Computation Goes Live, pp. 325–343. ISBN: 978-3-642-03548-7. DOI: 10.1007/978-3-642-03549-4_20. URL: http://dx.doi.org/10.1007/978-3-642-03549-4_20.

[32] *Boost C++ Libraries*. URL: http://www.boost.org/.

[33] Gabriel Bracha. “An O(Lg N) Expected Rounds Randomized Byzantine Generals Protocol”. In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. STOC ’85. Providence, Rhode Island, USA: ACM, 1985, pp. 316–326. ISBN: 0-89791-151-2. DOI: 10.1145/22145.22180. URL: http://doi.acm.org/10.1145/22145.22180.

[34] Gilles Brassard, Claude Crepeau, and Jean-Marc Robert. “All-or-Nothing Disclosure of Secrets”. In: *Advances in Cryptology — CRYPTO’ 86: Proceedings*. Ed. by Andrew M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 234–238. ISBN: 978-3-540-47721-1. DOI: 10.1007/3-540-47721-7_17. URL: http://dx.doi.org/10.1007/3-540-47721-7_17.

[35] Ran Canetti. “Universally Composable Security: A New Paradigm for Cryptographic Protocols”. In: *Proceedings of the 42Nd IEEE Symposium on Foundations of Computer Science*. FOCS ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 136–. ISBN: 0-7695-1390-5. URL: http://dl.acm.org/citation.cfm?id=874063.875553.

[36] David Chaum, Claude Crépeau, and Ivan Damgard. “Multiparty Unconditionally Secure Protocols”. In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC ’88. Chicago, Illinois, USA: ACM, 1988, pp. 11–19. ISBN: 0-89791-264-0. DOI: 10.1145/62212.62214. URL: http://doi.acm.org/10.1145/62212.62214.

[37] Jerry Cheng, Hao Yang, Starsky Wong, Petros Zerfos, and Songwu Lu. “Design and Implementation of Cross-Domain Cooperative Firewall”. In: *2007 IEEE International Conference on Network Protocols*. Oct. 2007, pp. 284–293. DOI: 10.1109/ICNP.2007.4375859.

[38] Benny Chor, Shafi Goldwasser, Silvio M. Micali, and Baruch Awerbuch. "Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults". In: *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*. SFCS '85. Washington, DC, USA: IEEE Computer Society, 1985, pp. 383–395. ISBN: 0-8186-0844-4. DOI: 10.1109/SFCS.1985.64. URL: https://doi.org/10.1109/SFCS.1985.64.

[39] Tung Chou and Claudio Orlandi. "The Simplest Protocol for Oblivious Transfer". In: *Proceedings of the 4th International Conference on Progress in Cryptology – LATINCRYPT 2015 - Volume 9230*. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 40–58. ISBN: 978-3-319-22173-1. DOI: 10.1007/978-3-319-22174-8_3. URL: http://dx.doi.org/10.1007/978-3-319-22174-8_3.

[40] Geoffroy Couteau. *What are the ways to generate Beaver triples for multiplication gate?* Nov. 19, 2016. URL: https://crypto.stackexchange.com/questions/41651/what-are-the-ways-to-generate-beaver-triples-for-multiplication-gate/41660#41660 (visited on 2017).

[41] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. 1st. New York, NY, USA: Cambridge University Press, 2015. ISBN: 1107043050, 9781107043053.

[42] Ronald Cramer, Ivan Damgård, and Yuval Ishai. "Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation". In: *Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings*. Ed. by Joe Kilian. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2005, pp. 342–362. ISBN: 978-3-540-30576-7. DOI: 10.1007/978-3-540-30576-7_19. URL: http://dx.doi.org/10.1007/978-3-540-30576-7_19.

[43] Ronald Cramer, Ivan Damgård, and Ueli Maurer. "General Secure Multi-party Computation from Any Linear Secret-sharing Scheme". In: *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'00. Bruges, Belgium: Springer-Verlag, 2000, pp. 316–334. ISBN: 3-540-67517-5. URL: http://dl.acm.org/citation.cfm?id=1756169.1756200.

[44] Claude Crépeau. "Equivalence Between Two Flavours of Oblivious Transfers". In: *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*. CRYPTO '87. London, UK, UK: Springer-Verlag, 1988, pp. 350–354. ISBN: 3-540-18796-0. URL: http://dl.acm.org/citation.cfm?id=646752.704744.

[45] Claude Crépeau, Jeroen van de Graaf, and Alain Tapp. "Committed Oblivious Transfer and Private Multi-Party Computation". In: *Advances in Cryptology — CRYPT0' 95: 15th Annual International Cryptology Conference Santa Barbara, California, USA, August 27–31, 1995 Proceedings*. Ed. by Don Coppersmith. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 110–123. ISBN: 978-3-540-44750-4. DOI: 10.1007/3-540-44750-4_9. URL: http://dx.doi.org/10.1007/3-540-44750-4_9.

[46] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. *Confidential Benchmarking based on Multiparty Computation*. Cryptology ePrint Archive, Report 2015/1006. http://eprint.iacr.org/2015/1006. 2015.

[47] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. "Asynchronous Multiparty Computation: Theory and Implementation". In: *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09*. Irvine. CA: Springer-Verlag, 2009, pp. 160–179. ISBN: 978-3-642-00467-4. DOI: 10.1007/978-3-642-00468-1_10. URL: http://dx.doi.org/10.1007/978-3-642-00468-1_10.

[48] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. "Scalable Multiparty Computation with Nearly Optimal Work and Resilience". In: *Proceedings of the 28th Annual Conference on Cryptology: Advances in Cryptology*. CRYPTO 2008. Santa Barbara, CA, USA: Springer-Verlag, 2008, pp. 241–261. ISBN: 978-3-540-85173-8. DOI: 10.1007/978-3-540-85174-5_14. URL: http://dx.doi.org/10.1007/978-3-540-85174-5_14.

[49] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. "Practical Covertly Secure MPC for Dishonest Majority – Or: Breaking the SPDZ Limits". In: *Computer Security – ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*. Ed. by Jason Crampton, Sushil Jajodia, and Keith Mayes.

Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–18. ISBN: 978-3-642-40203-6. DOI: 10.1007/978-3-642-40203-6_1. URL: http://dx.doi.org/10.1007/978-3-642-40203-6_1.

[50] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. "Multiparty Computation from Somewhat Homomorphic Encryption". In: *Proceedings of the 32Nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 643–662. ISBN: 978-3-642-32008-8. DOI: 10.1007/978-3-642-32009-5_38. URL: http://dx.doi.org/10.1007/978-3-642-32009-5_38.

[51] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. "Fully Homomorphic Encryption over the Integers". In: *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT'10. French Riviera, France: Springer-Verlag, 2010, pp. 24–43. ISBN: 3-642-13189-1, 978-3-642-13189-9. DOI: 10.1007/978-3-642-13190-5_2. URL: http://dx.doi.org/10.1007/978-3-642-13190-5_2.

[52] Cynthia Dwork. "Differential Privacy: A Survey of Results". In: *Proceedings of the 5th International Conference on Theory and Applications of Models of Computation*. TAMC'08. Xi'an, China: Springer-Verlag, 2008, pp. 1–19. ISBN: 3-540-79227-9, 978-3-540-79227-7. URL: http://dl.acm.org/citation.cfm?id=1791834.1791836.

[53] Shimon Even, Oded Goldreich, and Abraham Lempel. "A Randomized Protocol for Signing Contracts". In: *Commun. ACM* 28.6 (June 1985), pp. 637–647. ISSN:

0001-0782. DOI: 10.1145/3812.3818. URL: http://doi.acm.org/10.1145/3812.3818.

[54] Alexandre Evfimievski, Ramakrishnan Srikant, Rakesh Agrawal, and Johannes Gehrke. "Privacy Preserving Mining of Association Rules". In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '02. Edmonton, Alberta, Canada: ACM, 2002, pp. 217–228. ISBN: 1-58113-567-X. DOI: 10.1145/775047.775080. URL: http://doi.acm.org/10.1145/775047.775080.

[55] *Facebook reveals news feed experiment to control emotions*. Guardian News and Media Limited. June 29, 2014. URL: https://www.theguardian.com/technology/2014/jun/29/facebook-users-emotions-news-feeds (visited on 2017).

[56] Matthew Franklin and Moti Yung. "Communication Complexity of Secure Computation (Extended Abstract)". In: *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*. STOC '92. Victoria, British Columbia, Canada: ACM, 1992, pp. 699–710. ISBN: 0-89791-511-9. DOI: 10.1145/129712.129780. URL: http://doi.acm.org/10.1145/129712.129780.

[57] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*. STOC '09. Bethesda, MD, USA: ACM, 2009, pp. 169–178. ISBN: 978-1-60558-506-2. DOI: 10.1145/1536414.1536440. URL: http://doi.acm.org/10.1145/1536414.1536440.

[58] Craig Gentry, Shai Halevi, and Nigel P. Smart. "Homomorphic Evaluation of the AES Circuit". In: *Proceedings of the 32Nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 850–867. ISBN: 978-3-642-32008-8. DOI: `10.1007/978-3-642-32009-5_49`. URL: `http://dx.doi.org/10.1007/978-3-642-32009-5_49`.

[59] Niv Gilboa. "Two Party RSA Key Generation". In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '99. London, UK, UK: Springer-Verlag, 1999, pp. 116–129. ISBN: 3-540-66347-9. URL: `http://dl.acm.org/citation.cfm?id=646764.703977`.

[60] Oded Goldreich. *Foundations of Cryptography: Volume 1*. New York, NY, USA: Cambridge University Press, 2006. ISBN: 0521035368.

[61] Oded Goldreich, Silvio M. Micali, and Avi Wigderson. "How to Play ANY Mental Game". In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: ACM, 1987, pp. 218–229. ISBN: 0-89791-221-7. DOI: `10.1145/28395.28420`. URL: `http://doi.acm.org/10.1145/28395.28420`.

[62] Oded Goldreich, Silvio M. Micali, and Avi Wigderson. "Proofs That Yield Nothing but Their Validity or All Languages in NP Have Zero-knowledge Proof Systems". In: *J. ACM* 38.3 (July 1991), pp. 690–728. ISSN: 0004-5411. DOI: `10.1145/116825.116852`. URL: `http://doi.acm.org/10.1145/116825.116852`.

[63] Shafi Goldwasser, Silvio M. Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof-systems". In: *Proceedings of the Seventeenth Annual*

ACM Symposium on Theory of Computing. STOC '85. Providence, Rhode Island, USA: ACM, 1985, pp. 291–304. ISBN: 0-89791-151-2. DOI: 10.1145/22145.22178. URL: http://doi.acm.org/10.1145/22145.22178.

[64] Debayan Gupta, Aaron Segal, Aurojit Panda, Gil Segev, Michael Schapira, Joan Feigenbaum, Jenifer Rexford, and Scott Shenker. "A New Approach to Interdomain Routing Based on Secure Multi-party Computation". In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. HotNets-XI. Redmond, Washington: ACM, 2012, pp. 37–42. ISBN: 978-1-4503-1776-4. DOI: 10.1145/2390231.2390238. URL: http://doi.acm.org/10.1145/2390231.2390238.

[65] William Hart, Fredrik Johansson, and Sebastian Pancratz. *FLINT: Fast Library for Number Theory*. 2015. URL: http://www.flintlib.org/.

[66] Martin Hilbert and Priscila López. "The World's Technological Capacity to Store, Communicate, and Compute Information". In: *Science* 332.6025 (2011), pp. 60–65. ISSN: 0036-8075. DOI: 10.1126/science.1200970. eprint: http://science.sciencemag.org/content/332/6025/60.full.pdf. URL: http://science.sciencemag.org/content/332/6025/60.

[67] Yan Huang, Chih-hao Shen, David Evans, Jonathan Katz, and Abhi Shelat. "Efficient Secure Computation with Garbled Circuits". In: *Information Systems Security: 7th International Conference, ICISS 2011, Kolkata, India, December 15-19, 2011, Procedings*. Ed. by Sushil Jajodia and Chandan Mazumdar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 28–48. ISBN: 978-3-642-25560-1. DOI: 10.1007/978-3-642-25560-1_2. URL: http://dx.doi.org/10.1007/978-3-642-25560-1_2.

[68] Mikael Huss and Joel Westerberg. *Data size estimates*. June 24, 2014. URL: https://followthedata.wordpress.com/2014/06/24/data-size-estimates/ (visited on 2017).

[69] Alfonso Iacovazzi, Alessandro D'Alconzo, Fabio Ricciato, and Martin Burkhart. "Elementary Secure-multiparty Computation for Massive-scale Collaborative Network Monitoring". In: *Comput. Netw.* 57.17 (Dec. 2013), pp. 3728–3742. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2013.08.017. URL: http://dx.doi.org/10.1016/j.comnet.2013.08.017.

[70] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. "Extending Oblivious Transfers Efficiently". In: *Advances in Cryptology - CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings*. Ed. by Dan Boneh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 145–161. ISBN: 978-3-540-45146-4. DOI: 10.1007/978-3-540-45146-4_9. URL: http://dx.doi.org/10.1007/978-3-540-45146-4_9.

[71] Yuval Ishai and Eyal Kushilevitz. "Private simultaneous messages protocols with applications". In: *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*. June 1997, pp. 174–183. DOI: 10.1109/ISTCS.1997.595170.

[72] Liina Kamm and Jan Willemson. "Secure Floating Point Arithmetic and Private Satellite Collision Analysis". In: *Int. J. Inf. Secur.* 14.6 (Nov. 2015), pp. 531–548. ISSN: 1615-5262. DOI: 10.1007/s10207-014-0271-8. URL: http://dx.doi.org/10.1007/s10207-014-0271-8.

[73] Jonathan Katz. “Efficient and Non-malleable Proofs of Plaintext Knowledge and Applications”. In: *Proceedings of the 22Nd International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT’03. Warsaw, Poland: Springer-Verlag, 2003, pp. 211–228. ISBN: 3-540-14039-5. URL: http://dl.acm.org/citation.cfm?id=1766171.1766189.

[74] Marcel Keller, Emmanuela Orsini, and Peter Scholl. “Actively Secure OT Extension with Optimal Overhead”. In: *Advances in Cryptology – CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. Ed. by Rosario Gennaro and Matthew Robshaw. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 724–741. ISBN: 978-3-662-47989-6. DOI: 10.1007/978-3-662-47989-6_35. URL: http://dx.doi.org/10.1007/978-3-662-47989-6_35.

[75] Marcel Keller, Emmanuela Orsini, and Peter Scholl. “MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pp. 830–842. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978357. URL: http://doi.acm.org/10.1145/2976749.2978357.

[76] Marcel Keller, Dragos Rotaru, Peter Scholl, and Nigel Smart. *SPDZ Software*. University of Bristol. 2016. URL: https://www.cs.bris.ac.uk/Research/CryptographySecurity/SPDZ/ (visited on 2017).

[77] Joe Kilian. “Founding Cryptography on Oblivious Transfer”. In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC ’88. Chicago, Illinois, USA: ACM, 1988, pp. 20–31. ISBN: 0-89791-264-0. DOI: 10.

1145/62212.62215. URL: http://doi.acm.org/10.1145/62212.62215.

[78] Vladimir Kolesnikov and Thomas Schneider. “Improved Garbled Circuit: Free XOR Gates and Applications”. In: *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*. ICALP ’08. Reykjavik, Iceland: Springer-Verlag, 2008, pp. 486–498. ISBN: 978-3-540-70582-6. DOI: 10.1007/978-3-540-70583-3_40. URL: http://dx.doi.org/10.1007/978-3-540-70583-3_40.

[79] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. “Billion-gate Secure Computation with Malicious Adversaries”. In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Security’12. Bellevue, WA: USENIX Association, 2012, pp. 14–14. URL: http://dl.acm.org/citation.cfm?id=2362793.2362807.

[80] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176. URL: http://doi.acm.org/10.1145/357172.357176.

[81] *LIBSCAPI - The Secure Computation API*. URL: https://github.com/cryptobiu/libscapi.

[82] Yehuda Lindell. *Tutorial on Secure Multi-Party Computation*. IBM T.J.Watson. URL: http://u.cs.biu.ac.il/~lindell/research-statements/tutorial-secure-computation.ppt (visited on 2017).

[83] Yehuda Lindell and Benny Pinkas. “A Proof of Security of Yao’s Protocol for Two-Party Computation”. In: *J. Cryptol.* 22.2 (Apr. 2009), pp. 161–188. ISSN:

0933-2790. DOI: 10.1007/s00145-008-9036-8. URL: http://dx.doi.org/10.1007/s00145-008-9036-8.

[84] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. “Efficient Constant Round Multi-party Computation Combining BMR and SPDZ”. In: *Advances in Cryptology – CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*. Ed. by Rosario Gennaro and Matthew Robshaw. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 319–338. ISBN: 978-3-662-48000-7. DOI: 10.1007/978-3-662-48000-7_16. URL: http://dx.doi.org/10.1007/978-3-662-48000-7_16.

[85] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. “Fairplay—a Secure Two-party Computation System”. In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM’04. San Diego, CA: USENIX Association, 2004, pp. 20–20. URL: http://dl.acm.org/citation.cfm?id=1251375.1251395.

[86] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[87] Arvind Narayanan. “Bitcoin and Cryptocurrency Technologies”. University Lecture. 2017. URL: https://www.coursera.org/learn/cryptocurrency/lecture/EYEAo/hash-pointers-and-data-structures (visited on 2017).

[88] Arvind Narayanan and Vitaly Shmatikov. *Shmatikov How To Break Anonymity of the Netflix Prize Dataset. arxiv cs/0610105*. 2006.

[89] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. "A New Approach to Practical Active-Secure Two-Party Computation". In: *Proceedings of the 32Nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 681–700. ISBN: 978-3-642-32008-8. DOI: 10.1007/978-3-642-32009-5_40. URL: http://dx.doi.org/10.1007/978-3-642-32009-5_40.

[90] *NSA Spying Timeline*. Electronic Frontier Foundation. 2017. URL: https://www.eff.org/nsa-spying/timeline# (visited on 2017).

[91] Valerio Pastro. "Zero-Knowledge Protocols and Multiparty Computation". PhD dissertation. Aarhus University, 2013. URL: http://www.cs.yale.edu/homes/pastro-valerio/au/thesis.pdf.

[92] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. "Secure Two-Party Computation Is Practical". In: *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*. ASIACRYPT '09. Tokyo, Japan: Springer-Verlag, 2009, pp. 250–267. ISBN: 978-3-642-10365-0. DOI: 10.1007/978-3-642-10366-7_15. URL: http://dx.doi.org/10.1007/978-3-642-10366-7_15.

[93] Michael O. Rabin. *How To Exchange Secrets with Oblivious Transfer*. Harvard University Technical Report 81 talr@watson.ibm.com 12955 received 21 Jun 2005. 2005. URL: http://eprint.iacr.org/2005/187.

[94] John E. Savage. *Models of Computation: Exploring the Power of Computing*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, p. 372. ISBN: 0201895390. URL: http://cs.brown.edu/~jes/book/.

[95] Risk Based Security. *2016 Data Breach Trends*. Year In Review. Risk Based Security, Jan. 1, 2017. URL: https://pages.riskbasedsecurity.com/hubfs/Reports/2016%20Year%20End%20Data%20Breach%20QuickView%20Report.pdf?utm_campaign=2016+Year+End+Data+Breach+QuickView+Report&utm_source=hs_automation&utm_medium=email&utm_content=41076100&_hsenc=p2ANqtz-9YSFTnUDy2n3azbjo5khYqM1kprhERFPK6le6JE3DsBfqVSB5oLIlqqsPdst0oTqFYV7jWcPiXrG12iNfaCjTySbT4_w&_hsmi=41076100 (visited on 2017).

[96] *SEPIA: Security through Private Information Aggregation*. URL: http://sepia.ee.ethz.ch/index.html.

[97] Adi Shamir. "How to Share a Secret". In: *Commun. ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: http://doi.acm.org/10.1145/359168.359176.

[98] SINTEF. *Big Data, for better or worse: 90% of world's data generated over last two years.* ScienceDaily. May 22, 2013. URL: www.sciencedaily.com/releases/2013/05/130522085217.htm (visited on 2017).

[99] Nigel Smart. *Crypto is Dead; Long Live Crypto!* Mar. 4, 2013. URL: https://mpclounge.wordpress.com/2013/03/04/crypto-is-dead-long-live-crypto/ (visited on 2017).

[100] Pete Snyder. *Yao's Garbled Circuits*. URL: https://www.cs.uic.edu/pub/Bits/PeterSnyder/Yaos_Garbled_Circuits_-_Recent_Directions_and_Implementations_slides.pdf (visited on 2017).

[101] *SPDZ-2*. University of Bristol. 2016. URL: https://github.com/bristolcrypto/SPDZ-2 (visited on 2017).

[102] VIFF Development Team. *Preprocessing*. 2007. URL: http://viff.dk/doc/preprocessing.html (visited on 2017).

[103] VIFF Development Team. *VIFF, the Virtual Ideal Functionality Framework*. 2007. URL: http://viff.dk/ (visited on 2017).

[104] Stanley L. Warner. "Randomized Response: A Survey Technique for Eliminating Evasive Answer Bias". In: *Journal of the American Statistical Association* 60.309 (1965). PMID: 12261830, pp. 63–69. DOI: 10.1080/01621459.1965.10480775. eprint: http://www.tandfonline.com/doi/pdf/10.1080/01621459.1965.10480775. URL: http://www.tandfonline.com/doi/abs/10.1080/01621459.1965.10480775.

[105] *What is SPDZ? Part 3: SPDZ specifics*. University of Bristol. Nov. 4, 2016. URL: https://bristolcrypto.blogspot.fi/2016/11/what-is-spdz-part-3-spdz-specifics.html (visited on 2017).

[106] Stephen Wiesner. "Conjugate Coding". In: *SIGACT News* 15.1 (Jan. 1983), pp. 78–88. ISSN: 0163-5700. DOI: 10.1145/1008908.1008920. URL: http://doi.acm.org/10.1145/1008908.1008920.

[107] David Wu. *Somewhat Practical Homomorphic Encryption*. crypto.stanford.edu. 2014. URL: https://crypto.stanford.edu/~dwu4/talks/SecurityLunch0214.pdf (visited on 2017).

[108] Andrew C. Yao. “How to generate and exchange secrets”. In: *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. Oct. 1986, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

[109] Andrew C. Yao. “Protocols for Secure Computations”. In: *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*. SFCS ’82. Washington, DC, USA: IEEE Computer Society, 1982, pp. 160–164. DOI: 10.1109/SFCS.1982.88. URL: http://dx.doi.org/10.1109/SFCS.1982.88.

[110] Michael Zohner. *More Effiicient Oblivious Transfer*. Oct. 29, 2013. URL: http://www.cs.cornell.edu/~asharov/slides/ALSZ13.pdf (visited on 2017).

[111] Guy Zyskind, Oz Nathan, and Alex Pentland. “Decentralizing Privacy: Using Blockchain to Protect Personal Data”. In: *Proceedings of the 2015 IEEE Security and Privacy Workshops*. SPW ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 180–184. ISBN: 978-1-4799-9933-0. DOI: 10.1109/SPW.2015.27. URL: http://dx.doi.org/10.1109/SPW.2015.27.

[112] Guy Zyskind, Oz Nathan, and Alex Pentland. “Enigma: Decentralized Computation Platform with Guaranteed Privacy”. In: *CoRR* abs/1506.03471 (2015). URL: http://arxiv.org/abs/1506.03471.

Appendix A

Appendix

A.1 Source Code Availability

Source code of the project can be accessed either at

`https://github.com/mertdnmz/pceas/tree/master/Pceas/src`

or at

`https://sourceforge.net/p/pceas/code/ci/master/tree/Pceas/src/`.

A.2 Dependencies

The program is implemented with C++ and uses C++11 features.

A.2.1 FLINT: Fast Library for Number Theory

We used FLINT Library [65] (version 2.5.2) for number theoretic operations. In particular, we used

- `fmpz` and `fmpz_vec` classes for operations involving integers

- `fmpz_mod_poly` class for operations involving polynomials over $\mathbb{Z}/p\mathbb{Z}$

We also rely on FLINT Library for random number generation. FLINT internally uses a linear congruential generator. When players want to generate random numbers as they run the protocol, they use their `partyID` and the randomness they get from `std::random_device` as seeds. Hence, players pick different randoms in consecutive runs.[1]

Symmetric bivariate polynomials over $\mathbb{Z}/p\mathbb{Z}$ (class `SymmBivariatePoly`) are implemented on top of FLINT's `fmpz_mod_poly` class.

A.2.2 Boost C++ Libraries

We used `/tokenizer.hpp` and `/algorithm/string.hpp` classes from Boost Library [32], for parsing the options file.

A.3 Building

The project was built and run on a machine running 64-bit Ubuntu (16.04). Linux GCC toolchain (5.4.0 20160609 Ubuntu 5.4.0-6ubuntu1 16.04.4) was used to build the project. Compiler option `-std=c++0x` (ISO C++ 11 Language Standard), and linker flags `-lmpfr -lgmp -pthread` are required. One needs FLINT [65] and its dependencies installed on the system. Please refer to FLINT's documentation for these installations.

[1]It might be useful to change this behaviour for debugging. See `NO_RANDOM` in `Pceas.h`.

A.4 Running

An options file template and sample options files (including those used for the runs mentioned in Chapter 5 and Chapter 6) can be found along with the source code, under `options` folder. In order to run the simulator, make sure that the `options` folder (containing the options file `opt`) is in the same directory with the binary :

```
/.../Release$ ls
options Pceas
/.../Release$ ./Pceas
```

A.5 Project Structure

Project structure and source files included in the project are shown in Figure A.1.

A.6 Sample Code

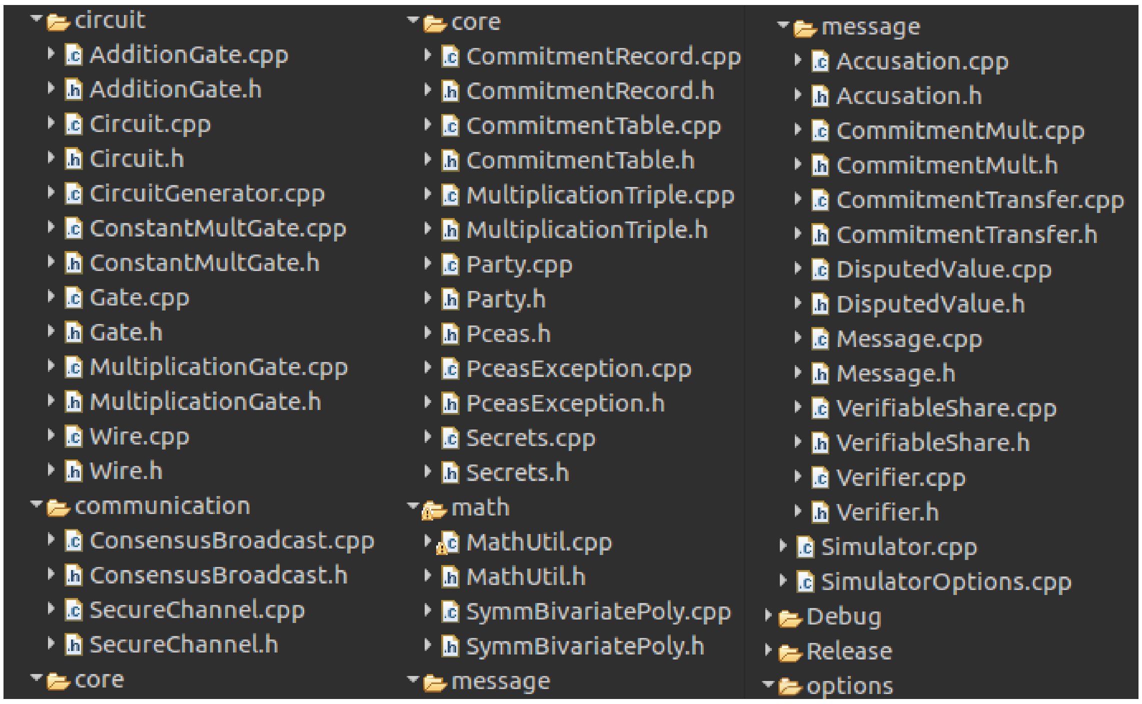
Complete source code is too large to include in printed text. Please refer to Appendix A.1 if you need source code of the project.

This section includes the source code for two selected methods, which roughly correspond to the outlines given in Section 4.1.1 and Section 4.1.2 for protocols P_{CEPS} and P_{CEAS}, respectively. Both methods are from class `Party`.

A.6.1 Method `runPceps()`

```

/**
 * Protocol 'CEPS' (Circuit Evaluation with Passive Security)
 */
```



```
circuit
  AdditionGate.cpp
  AdditionGate.h
  Circuit.cpp
  Circuit.h
  CircuitGenerator.cpp
  ConstantMultGate.cpp
  ConstantMultGate.h
  Gate.cpp
  Gate.h
  MultiplicationGate.cpp
  MultiplicationGate.h
  Wire.cpp
  Wire.h
communication
  ConsensusBroadcast.cpp
  ConsensusBroadcast.h
  SecureChannel.cpp
  SecureChannel.h
core
  CommitmentRecord.cpp
  CommitmentRecord.h
  CommitmentTable.cpp
  CommitmentTable.h
  MultiplicationTriple.cpp
  MultiplicationTriple.h
  Party.cpp
  Party.h
  Pceas.h
  PceasException.cpp
  PceasException.h
  Secrets.cpp
  Secrets.h
math
  MathUtil.cpp
  MathUtil.h
  SymmBivariatePoly.cpp
  SymmBivariatePoly.h
message
  Accusation.cpp
  Accusation.h
  CommitmentMult.cpp
  CommitmentMult.h
  CommitmentTransfer.cpp
  CommitmentTransfer.h
  DisputedValue.cpp
  DisputedValue.h
  Message.cpp
  Message.h
  VerifiableShare.cpp
  VerifiableShare.h
  Verifier.cpp
  Verifier.h
Simulator.cpp
SimulatorOptions.cpp
Debug
Release
options
```

FIGURE A.1: Project Structure.

```
void Party::runPceps() {

  sanityChecks();
  setRecombinationVector();//calculate recombination vector

  // Step 1 of 3 :input sharing
  const unsigned long CIRCUIT_INPUT_NUM = circuit->getInputCount();
  auto const& secretsMap = secrets->getSecrets();
  auto it = secretsMap.begin();
  vector<MessagePtr> messages;
  while (messages.size() < CIRCUIT_INPUT_NUM) {
    if (it != secretsMap.end()) {//if still has secrets to share
      fmpz_set_ui(value, it->second);
      distributeShares(value, it->first);
      it++;
    }

    interact();

    for (ulong i = 0; i < N; ++i) {//receive shares sent
      //"outward clocking"
      if (channels[i]->hasMsg()) {//only data providers are expected to send messages
        messages.push_back(channels[i]->recv());
      }
    }
  }
  for (auto const& m : messages) {
    circuit->assignInput(m->getShare(), m->getInputLabel());
  }

  // Step 2 of 3 : computation
  Gate* g;
  while ((g = circuit->getNext()) != nullptr) {
    switch (g->getType()) {
    case ADD:
    case CONST_MULT:
      g->localCompute();
      fmpz_mod(g->getLocalResult(), g->getLocalResult(), FIELD_PRIME);//reduce
      g->assignResult(g->getLocalResult());
      break;
    case MULT:
      g->localCompute();
      fmpz_mod(g->getLocalResult(), g->getLocalResult(), FIELD_PRIME);//reduce
      distributeShares(g->getLocalResult());

      interact();

      _fmpz_vec_zero(shares, N);
      for (ulong i = 0; i < N; ++i) {//receive shares sent by other parties
        //"outward clocking"
        if (!channels[i]->hasMsg()) {//Even if the protocol could handle some missing
            shares, we stop here because our assumption(no active cheaters) is violated.
          throw PceasException("A␣party␣fails␣to␣participate.");
        }
        fmpz_set(shares+i, channels[i]->recv()->getShare());
      }
      // We produce a degree D Shamir share, via degree reduction, by recombining local
          shares for a degree 2D polynomial
      _fmpz_vec_dot(g->getLocalResult(), recombinationVector, shares, N);
      fmpz_mod(g->getLocalResult(), g->getLocalResult(), FIELD_PRIME);//reduce
      g->assignResult(g->getLocalResult());
      break;
    }
  }

  // Step 3 of 3 : output reconstruction

  // find output gate's output and send it privately to the data user
  MessagePtr m = newMsg();
  m->setShare(circuit->retrieveOutput());
  channels[dataUser-1]->send(m);

  interact();

  if (pid == dataUser) {// data user performs interpolation to find f(0) and prints it
    ulong receivedShareCount = 0;
    _fmpz_vec_zero(shares, N);
    for (ulong i = 0; i < N; ++i) {//receive shares sent by other parties
      if (channels[i]->hasMsg()) {//T+1 shares will be enough, others will remain as zero
          (effectively excluding them from the upcoming dot product).
        fmpz_set(shares+i, channels[i]->recv()->getShare());
        receivedShareCount++;
      }
    }
    if (receivedShareCount > D) {//need at least T = D+1 shares for interpolation
      _fmpz_vec_dot(value, recombinationVector, shares, N);
      fmpz_mod(value, value, FIELD_PRIME);
      cout << "Evaluation␣result␣:␣" << MathUtil::fmpzToStr(value) << endl;
    } else {
      cout << "Data␣user␣did␣not␣receive␣enough␣shares␣to␣recover␣evaluation␣result.␣"
        << "(Protocol␣cannot␣tolerate␣active␣cheaters.)" << endl;
    }
  }

  end();
}
```

A.6.2 Method `runPceas()`

```

/**
 * Protocol 'CEAS' (Circuit Evaluation with Active Security)
 */
void Party::runPceas(bool circuitRandomization, bool finalRun) {

  sanityChecks();
  setRecombinationVector();//note : we will recalculate recombination vector each time we
      mark a party as corrupt

  if (circuitRandomization) {
    // Preprocessing phase for 'CEAS with Circuit Randomization' - generates
        multiplication triples
    runPreprocessing();
    commitments->cleanUp();//to keep commitment table size managable, we remove records
        which are no longer needed
  }

  {// Step 1 of 3 :input sharing
    const unsigned long CIRCUIT_INPUT_NUM = circuit->getInputCount();
    auto const& secretsMap = secrets->getSecrets();
    auto it = secretsMap.begin();
    ulong inputSharingLoopCounter = 0;//any single party will loop at most
        CIRCUIT_INPUT_NUM times. we use this fact to break loop and end protocol if any
        dishonest refuse to distribute a share.
    while (commitments->getInputShareCountReceivedBy(pid) < CIRCUIT_INPUT_NUM &&
        inputSharingLoopCounter < CIRCUIT_INPUT_NUM) {
      const string inputSharingUniqueSuffix = to_string(inputSharingLoopCounter);
      if (it != secretsMap.end()) {//distribute shares of each secret
        fmpz_set_ui(value, it->second);
        distributeVerifiableShares(value, inputSharingUniqueSuffix, it->first, false, true
            );
        it++;
      } else {
        /*
         * Note : Ideally, if a party does not have any input to distribute, it should just
         * passively participate in distribution process of others (VSS is highly interactive).
         * For simplifying the implementation, we make these parties distribute some arbitrary value,
         * which will be ignored. This does not increase the number of rounds required.
         */
        fmpz_zero(value);
        distributeVerifiableShares(value, inputSharingUniqueSuffix, NONE, false, true);
      }
      inputSharingLoopCounter++;
    }
    commitments->cleanUp();//to keep commitment table size managable, we remove records which are no longer needed
    vector<CommitmentRecord*> inputShares = commitments->getInputSharesReceivedBy(pid);
    if (inputShares.size() < CIRCUIT_INPUT_NUM) {
      throw PceasException("Missing␣inputs.");
    }
    if (inputShares.size() > CIRCUIT_INPUT_NUM) {
      throw PceasException("Received␣more␣inputs␣than␣expected.");
    }
    for (auto const& is : inputShares) {
      circuit->assignInputCid(is->getCommitid(), is->getInputLabel());
#ifdef VERBOSE
      cout << "Party␣" << to_string(pid) << "␣assigns␣wire␣" + is->getInputLabel() + "␣:␣\nCID␣=␣" << is->getCommitid()
        << "\nOpenedVal␣=␣" << MathUtil::fmpzToStr(is->getOpenedValue()) << endl;
#endif
    }
  }
  interact();
#ifdef VERBOSE
  cout << "Gate␣computation␣phase␣starts." << endl;
#endif
  // Step 2 of 3 : computation
  Gate* g;
```

```
 while ((g = circuit->getNext()) != nullptr) {
  switch (g->getType()) {
  case ADD:
  {
   AdditionGate* ag = static_cast<AdditionGate*>(g);
   for (PartyId k = 1; k <= N; ++k) {
    /*
     * To keep the commitment records synchronized, we do all other parties local
        computations, in addition to our own.
     */
    const CommitmentId share_k_1 = getShareNameFor(k, ag->getInputCid1());
    const CommitmentId share_k_2 = getShareNameFor(k, ag->getInputCid2());
    CommitmentId add_k = addCommitments(share_k_1, share_k_2);
    CommitmentId result_k = makeShareName(NOPARTY, k, to_string(g->getGateNumber()),
       false, false, true);
    commitments->rename(add_k, result_k);
    CommitmentRecord* cr_k = commitments->getRecord(result_k);
    if (cr_k == nullptr || cr_k->getOwner() != k) {//should not happen
     throw PceasException("Wire␣is␣assigned␣invalid␣commitment.");
    }
    cr_k->setPermanent();
    if (k == pid) {
     g->assignResult(cr_k->getCommitid());
#ifdef VERBOSE
     cout << "Party␣" << to_string(pid) << "␣assigns␣output␣to␣gate#␣" << g->
        getGateNumber() << "␣(addition␣gate)␣:␣\nCID␣=␣"
       << cr_k->getCommitid() << "\nOpenedValue␣=␣" << MathUtil::fmpzToStr(cr_k->
          getOpenedValue()) << endl;
#endif
    }
   }
  }
  break;
  case CONST_MULT:
  {
   ConstantMultGate* cmg = static_cast<ConstantMultGate*>(g);
   for (PartyId k = 1; k <= N; ++k) {
    /*
```

```
       * To keep the commitment records synchronized, we do all other parties local computations, in addition to our own.
       */
      const CommitmentId share_k = getShareNameFor(k, cmg->getInputCid());
      CommitmentId mult_k = constMultCommitment(cmg->getConstant(), share_k);
      CommitmentId result_k = makeShareName(NOPARTY, k, to_string(g->getGateNumber()), false, false, true);
      commitments->rename(mult_k, result_k);
      CommitmentRecord* cr_k = commitments->getRecord(result_k);
      if (cr_k == nullptr || cr_k->getOwner() != k) {//should not happen
       throw PceasException("Wire␣is␣assigned␣invalid␣commitment.");
      }
      cr_k->setPermanent();
      if (k == pid) {
       g->assignResult(cr_k->getCommitid());
#ifdef VERBOSE
       cout << "Party␣" << to_string(pid) << "␣assigns␣output␣to␣gate#␣" << g->getGateNumber() << "␣(const.␣mult.␣gate)␣:␣\nCID␣=␣"
         << cr_k->getCommitid() << "\nOpenedValue␣=␣" << MathUtil::fmpzToStr(cr_k->getOpenedValue()) << endl;
#endif
      }
     }
   }
   break;
   case MULT:
   {
    MultiplicationGate* mg = static_cast<MultiplicationGate*>(g);
    if (circuitRandomization) {
      //construct a common representation (common to all honest parties) for a * b, using existing multiplication triples (generated in preprocessing phase)
      auto it = triples.find(g->getGateNumber()); // x, y, x*y
      if (it == triples.end()) {
       throw PceasException("Missing␣triple.");
      }
      CommitmentId e_pid, d_pid;
      for (PartyId k = 1; k <= N; ++k) {//To keep the commitment records synchronized, we do all other parties local computations, in addition to our own.
       const CommitmentId input1_k = getShareNameFor(k, mg->getInputCid1());
       const CommitmentId input2_k = getShareNameFor(k, mg->getInputCid2());
       CommitmentId e = substractCommitments(input1_k, makeTripleName(k,
           MultiplicationTriple::M1, g->getGateNumber())); // a - x
       CommitmentId d = substractCommitments(input2_k, makeTripleName(k,
           MultiplicationTriple::M2, g->getGateNumber())); // b - y
       CommitmentId eNew = makeTripleName(k, MultiplicationTriple::E, g->getGateNumber()
           );
       CommitmentId dNew = makeTripleName(k, MultiplicationTriple::D, g->getGateNumber()
           );
       commitments->rename(e, eNew);
       commitments->rename(d, dNew);
       if (k == pid) {
        e_pid = eNew;
        d_pid = dNew;
       }
      }
      /*
       * We open e and d, and other parties will open theirs(e' = a - x', d' = b - y')
       * Note that, these 'open's are the only interactions we need in order to process
           the multiplication gate.
       * Via circuit randomization, much of the cost due to interactions for
           multiplications are pushed
       * to the preprocessing phase, in which triples are (ideally - see '
           runPreprocessing') generated
       * in parallel, rather than one at a a time.
       */
      open(e_pid);//INTERACTIVE
      open(d_pid);//INTERACTIVE
      MultiplicationTriple& triple = it->second;
      auto& receivedShares = triple.receivedShares;
      CommitmentId result_pid;
      //eliminate shares for which the sender of share is known to be dishonest (we
          marked parties as corrupt in previous steps)
      receivedShares.erase(remove_if(receivedShares.begin(), receivedShares.end(), [this
          ](CommitmentRecord* cr){return isCorrupt(cr->getDistributer());}),
          receivedShares.end());
      if (receivedShares.size() > 2*D) {
          sort(receivedShares.begin(), receivedShares.end(), [](CommitmentRecord* is1, CommitmentRecord* is2){return is1->getDistributer() < is2->getDistributer();});
          result_pid = runDegreeReduction(receivedShares, g->getGateNumber());
#ifdef VERBOSE
          this_thread::sleep_for(chrono::milliseconds(pid*700));
          cout << "Party␣" << to_string(pid) << "␣recombined␣x.y␣:␣" << MathUtil::fmpzToStr(commitments->getRecord(result_pid)->getOpenedValue()) << endl;
#endif
        } else {
          /*
           * Since deg(h) = 2D, we needed more than 2D shares for recombination.
           * This protocol tolerates <= N / 3 dishonest.
           * Not having enough shares means, our assumption failed. We stop execution..
           */
          throw PceasException("More␣dishonest␣than␣the␣protocol␣can␣handle.");
        }
        for (PartyId k = 1; k <= N; ++k) {//To keep the commitment records synchronized, we do all other parties local computations, in addition to our own.
          CommitmentRecord* ek = commitments->getRecord(makeTripleName(k, MultiplicationTriple::E, g->getGateNumber()));
          CommitmentRecord* dk = commitments->getRecord(makeTripleName(k, MultiplicationTriple::D, g->getGateNumber()));
          if (ek == nullptr || !ek->isOpened() || dk == nullptr || !dk->isOpened()) {
            addCorrupt(k);//all honest will agree
            if (k == pid) {//keep corrupt parties alive for running test cases
              g->assignResult(result_pid);
            }
            continue;
          }
          const CommitmentId input1_k = getShareNameFor(k, mg->getInputCid1());
          const CommitmentId input2_k = getShareNameFor(k, mg->getInputCid2());
          const CommitmentId result_k = makeShareName(NOPARTY, k, to_string(g->getGateNumber()), false, false, true);
          CommitmentId temp_k = makeTripleName(k, MultiplicationTriple::PROD, g->getGateNumber());
          //[[a * b]] = [[x * y]] + e[[b]] + d[[a]] - e.d
          commitments->rename(result_k, temp_k);//initialize temp_k with [[x * y]]
          temp_k = addCommitments(temp_k, constMultCommitment(ek->getOpenedValue(),
              input2_k)); // result_k += e[[b]]
          temp_k = addCommitments(temp_k, constMultCommitment(dk->getOpenedValue(),
              input1_k)); // result_k += d[[a]]
          fmpz_mul(value, ek->getOpenedValue(), dk->getOpenedValue());
#ifdef VERBOSE
          if (k == pid) {
            cout << "Party␣" << to_string(pid) << "a,b␣:␣" << MathUtil::fmpzToStr(
                commitments->getRecord(input1_k)->getOpenedValue()) << "\t" << MathUtil::
                fmpzToStr(commitments->getRecord(input2_k)->getOpenedValue()) << endl;
            cout << "␣a.b␣+␣e.d␣:␣" << MathUtil::fmpzToStr(commitments->getRecord(temp_k)->
                getOpenedValue()) << "␣␣e.d␣:␣" << MathUtil::fmpzToStr(value) << endl;
          }
#endif
          fmpz_neg(value, value);
          temp_k = constAddCommitment(value, temp_k); // result_k -= e.d
          commitments->rename(temp_k, result_k);
          CommitmentRecord* cr_k = commitments->getRecord(result_k);
          if (cr_k == nullptr || cr_k->getOwner() != k) {//should not happen
            throw PceasException("Wire␣is␣assigned␣invalid␣commitment.");
          }
          cr_k->setPermanent();
          if (k == pid) {
            g->assignResult(cr_k->getCommitid());
#ifdef VERBOSE
            cout << "Party␣" << to_string(pid) << "␣assigns␣output␣to␣gate#␣" << g->
                getGateNumber() << "␣(mult.␣gate)␣:␣\nCID␣=␣"
              << cr_k->getCommitid() << "\nOpenedValue␣=␣" << MathUtil::fmpzToStr(cr_k->
                  getOpenedValue()) << endl;

            cout << "Triple␣used␣were␣(M1,␣M2,␣E,␣D)␣:␣" << MathUtil::fmpzToStr(triple.
                firstMult->getOpenedValue()) << "\t" << MathUtil::fmpzToStr(triple.
                secontMult->getOpenedValue()) << "\t" << MathUtil::fmpzToStr(commitments->
                getRecord(e_pid)->getOpenedValue()) << "\t" << MathUtil::fmpzToStr(
                commitments->getRecord(d_pid)->getOpenedValue()) << endl;
            cout << "Received␣shares␣were␣:␣" << endl;
            for (auto const& s : receivedShares) {
              cout << MathUtil::fmpzToStr(s->getOpenedValue()) << "\t";
```

```
         }
         cout << endl;
#endif
       }
     }
   } else {
     /*
      * [[ab;f.g]]_2t = [[a;f]]_t * [[b;g]]_t
      */
     CommitmentId localMult = multiplyCommitments(mg->getInputCid1(), mg->getInputCid2());
     distributeVerifiableShares(localMult, to_string(g->getGateNumber()));
     vector<CommitmentRecord*> receivedShares = commitments->getVSSharesReceivedBy(pid);
     //eliminate shares for which the sender of share is known to be dishonest (we marked parties as corrupt in previous steps)
     receivedShares.erase(remove_if(receivedShares.begin(), receivedShares.end(), [this](CommitmentRecord* cr){return isCorrupt(cr->getDistributer());}), receivedShares.end());
     if (receivedShares.size() > 2*D) {
       sort(receivedShares.begin(), receivedShares.end(), [](CommitmentRecord* is1, CommitmentRecord* is2){return is1->getDistributer() < is2->getDistributer();});
       CommitmentId result = runDegreeReduction(receivedShares, g->getGateNumber());
       g->assignResult(result);
#ifdef VERBOSE
       cout << "Party␣" << to_string(pid) << "␣assigns␣output␣to␣gate#␣" << g->getGateNumber() << "␣(mult.␣gate)␣:␣\nCID␣=␣"
         << result << "\nOpenedValue␣=␣" << MathUtil::fmpzToStr(commitments->getRecord(result)->getOpenedValue()) << endl;
#endif
     } else {
       /*
        * Since deg(h) = 2D, we needed more than 2D shares for recombination.
        * This protocol tolerates <= N / 3 dishonest.
        * Not having enough shares means, our assumption failed. We stop execution..
        */
       throw PceasException("More␣dishonest␣than␣the␣protocol␣can␣handle.");
```

```
    }
   }
  }
  break;
  }
  commitments->cleanUp();//to keep commitment table size managable, we remove records which are no longer needed
 }
 interact();
#ifdef VERBOSE
 cout << "Gate␣computation␣phase␣ends." << endl;
#endif
 {
  // Step 3 of 3 : output reconstruction
  // find output gate's output and send it privately to the data user
  CommitmentId result = circuit->retrieveOutputCid();
  for (ulong i = 0; i < N; ++i) {//since parties can not designatedOpen to the same party in parallel, they will take turns
   PartyId k = i + 1;
   if (k != dataUser) {
    if (k == pid) {//our turn to 'designatedOpen' share to dataUser
     designatedOpen(result, dataUser, true);//INTERACTIVE
    } else {//We will not 'designatedOpen' anything, but will participate in other's 'designatedOpen's.
     //note that single share per party is automatically enforced due to target selection scheme used in 'designatedOpen'
     const PartyId target = getTargetFromSource(pid, k, dataUser);//(when party k is opening to dataUser, we can only open to...)
     designatedOpen(NONE, target, true);//INTERACTIVE
    }
   }
  }
  if (pid == dataUser) {
   //We mark the share we have as output. (we did not 'designatedOpen' to self.
   //Shares from other parties have been marked during the 'designatedOpen's above.)
   commitments->getRecord(result)->markAsOutput();
   vector<CommitmentRecord*> outputShares = commitments->getOutputShares();
   if (outputShares.size() > N) {
    throw PceasException("Too␣many␣output␣shares");
   }
   //eliminate shares for which we have no access to the value, and for which it is known that sender of share (owner of 'designatedOpen'ed commitment) is known to be dishonest
   outputShares.erase(remove_if(outputShares.begin(), outputShares.end(), [this](CommitmentRecord* cr){return !cr->isValueOpenToUs() || isCorrupt(cr->getOwner());}), outputShares.end());
   if (outputShares.size() > D) {
    //use Lagrange interpolation to find output value and print it
    _fmpz_vec_zero(shares, N);
    for (auto const& os : outputShares) {//T = D+1 shares will be enough, others will remain as zero (effectively excluding them from the upcoming dot product).
     ulong arrIndex = os->getOwner() - 1;
     fmpz_set(shares+arrIndex, os->getOpenedValue());
    }
    _fmpz_vec_dot(value, recombinationVector, shares, N);
    fmpz_mod(value, value, FIELD_PRIME);
    cout << "Evaluation␣result␣:␣" << MathUtil::fmpzToStr(value) << endl;
   } else {
    /*
     * Since deg(f) = D, we needed more than D shares for recombination.
     * This protocol tolerates <= N / 3 dishonest.
     * Not having enough shares means, our assumption failed. We stop execution..
     */
    cout << "Data␣user␣did␣not␣receive␣enough␣shares␣to␣recover␣evaluation␣result.␣"
       << "(More␣dishonest␣than␣the␣protocol␣can␣handle)" << endl;
   }
  }
 }
 if (finalRun) {
  end();
 } else {
  interact();
 }
}
```