University of Tartu
Faculty of Mathematics and Computer Science
Institute of Computer Science

Pille Pullonen

Actively Secure Two-Party Computation: Efficient Beaver Triple Generation

Master's thesis (30 ETCS)

Supervisors:	Sven Laur, Ph.D.
	Tuomas Aura, Ph.D.
Instructor:	Dan Bogdanov, Ph.D.

Author:	....	" ..... "	May 2013
Supervisor:	....	" ..... "	May 2013
Supervisor:	....	" ..... "	May 2013
Instructor:	....	" ..... "	May 2013

Allowed for defence

Professor:	....	" ..... "	May 2013

Tartu 2013

Contents

1 Introduction **4**

1.1 Motivation . . . 4

1.2 Contribution of the author . . . 4

1.3 Structure of the thesis . . . 5

2 Preliminaries **6**

2.1 Cryptographic primitives . . . 6

2.1.1 Additive secret sharing . . . 6

2.1.2 Chinese remainder theorem . . . 7

2.1.3 Universal composability . . . 7

2.1.4 Paillier cryptosystem . . . 7

2.1.5 Elliptic curves . . . 9

2.1.6 Lifted Elgamal cryptosystem . . . 10

2.1.7 Zero-knowledge proofs . . . 12

2.1.8 Dual-mode commitment schemes . . . 13

2.1.9 Message authentication codes . . . 17

2.2 Secure multi-party computation . . . 19

2.2.1 Overview of SMC techniques . . . 19

2.2.2 General SMC threat model . . . 20

2.2.3 Achieving actively secure two-party computation . . . 21

2.3 The Sharemind SMC framework . . . 21

2.3.1 Application model . . . 21

2.3.2 Computation primitives . . . 22

2.3.3 Programming applications . . . 22

3 Principles of the SPDZ framework **24**

3.1 Precomputation model . . . 24

3.2 Oblivious MAC . . . 26

3.3 Beaver triples . . . 27

3.4 Basic protocols . . . 27

3.5 Initialising actively secure two-party computation . . . 29

3.5.1 Asymmetric setup . . . 29

3.5.2 Symmetric setup . . . 30

3.5.3 Shared key setup . . . 30

4 Asymmetric two-party computation **31**

4.1 Protection domain setup . . . 31

4.2 Publishing shared values . . . 32

4.3 Random share generation 38
4.4 Beaver triples generation 41
4.5 Receiving inputs from the input party 45
4.6 Efficiency of the protocols 48
4.6.1 Computational cost 49
4.6.2 Communication cost 50

5 Protocols for Beaver triple generation 52
5.1 Setup for triple generation protocols 52
5.2 Packing several shares into one generation 54
5.2.1 Packing as base-B numbers 54
5.2.2 Triple generation with partial base-B packing 56
5.2.3 Packing using the Chinese remainder theorem 59
5.3 Share conversion 59
5.3.1 Converting binary shares to any modulus 60
5.3.2 Problems with converting the third triple element 60
5.3.3 Triple generation with share conversion 61
5.4 Comparison of proposed triple generation ideas 63

6 Symmetric two-party computation 65
6.1 Protection domain setup 65
6.2 Publishing shared values 66
6.3 Receiving inputs from the input party 67
6.4 Publishing a secret to the result party 69
6.5 Precomputation 69
6.5.1 Random share generation 70
6.5.2 Beaver triples generation 71
6.6 Efficiency of the protocols 72
6.6.1 Computational cost 72
6.6.2 Communication cost 73

7 Implementation 75
7.1 Implementation platform 75
7.2 Secure computation capabilities 75
7.3 Performance measurements 76
7.3.1 Online protocols 76
7.3.2 Precomputation protocols 78

8 Conclusions 80

Eestikeelne resümee 82

Bibliography 84

Chapter 1

Introduction

1.1 Motivation

Information is among the most important resources of the modern world, allowing to make wiser business choices, estimate future trends or foresee natural catastrophes. The world wide web makes collecting and also sharing information fast and easy, so in an honest world everyone could just combine their data and analyse it as they like. However, as a resource information might mean business secrets or confidential data that should not be freely published, thus, we need methods to process this information without losing privacy.

Secure multi-party computation is a cryptographic tool that enables sharing data for analysis without actually revealing it. For example, consider a chain of food-stores that is interested in learning how well they do compared to other retailers. It could use secure computation techniques to collaborate with other stores in the area to get such aggregated results without leaking their client databases or their behaviours to their competitors.

Each store can expect the others to help the computation as long as they believe that other companies are also interested in getting accurate results. However, there might be a shop that deliberately fiddles with the computations, for example, to trick its competitors into thinking that others have some very popular products that actually no-one buys. The competitors are then likely to increase their supply of these goods and, hence, suffer an economical loss. *Passively* secure computation can be used in the first case, whereas *active* security is needed to ensure that all parties follow the computations correctly. Actively secure secure computation means that the misbehaving shop could not affect the computation outcomes with anything other than its inputs.

Secure computation is currently an active research field and has reached the state where is is efficient enough for practical applications. SHAREMIND is one of the more mature secure multi-party computation frameworks that currently offers *passive* security [11, 12]. This thesis uses the principles also combined in the SPDZ framework [26] to add an *actively* secure protocol set to SHAREMIND framework.

1.2 Contribution of the author

The aim of this work is to adapt the SPDZ general actively secure computation framework [26] for the two-party case and focus on optimising the precomputation phase.

An important distinction between our work and SPDZ is the usage of additively homomorphic cryptosystem instead of the somewhat-homomorphic cryptosystem for the precomputation phase. The resulting protocol set is implemented in Sharemind version 3 [53].

The author was responsible for working out the details of the proposed protocol sets, including the share representation, local operations on this representation, and the precomputation phase. In addition, the share representations required protocols to communicate with data donors and data analysts. The author also implemented and benchmarked the proposed asymmetric and symmetric protocol set, as well as the triple generation protocols as part of the Sharemind framework.

The main outcomes of this thesis are the implementation of the asymmetric and symmetric protection domains, as well as ideas for generating Beaver triples using additively homomorphic cryptosystem.

1.3 Structure of the thesis

In the following, Chapter 2 gives an overview of tools used to build following secure computation protocols. At first, it summarises the necessary cryptographic concepts and gives specific initialisations of schemes used further in this thesis. Secondly, it provides a short overview of secure multi-party computation and security related issues. Finally, it briefly describes the key aspects of Sharemind.

The SPDZ framework is introduced separately in Chapter 3 together with some insights to how the following work uses ideas from it. This chapter focuses on the key principles used in SPDZ and gives additional background for the rest of the thesis.

Our protocol set for two-party computation with asymmetric setup is introduced in Chapter 4. This includes both the necessary protocols and theoretical efficiency analysis.

Chapter 5 introduces some ideas to generate Beaver triples using Paillier cryptosystem. The main focus is on achieving triple generation for arbitrarily chosen modulus.

The online phase of the symmetric protocol set is described in Chapter 6. This chapter also gives hints about achieving suitable precomputation phase using the Beaver triple generation ideas from Chapter 5.

Chapter 7 focuses on the implementation details and benchmark results, giving an overview of the efficiency of our protocols. This helps to summarise and compare the ideas from the rest of the thesis.

Finally, Chapter 8 concludes this thesis and gives additional directions for further work.

Chapter 2

Preliminaries

This chapter introduces the necessary building blocks and background notions for the proposed protocols. At first, Section 2.1 introduces the cryptographic tools used throughout this thesis. Secondly, Section 2.2 gives an introduction to secure multi-party computation and related problems. Thirdly, Section 2.3 introduces the SHAREMIND framework where we set up our proposed protocols.

2.1 Cryptographic primitives

This section introduces different cryptographic notions used for building secure two-party protocols. We introduce the basic concepts and initialise them with the exact instantiations used in the following sections.

2.1.1 Additive secret sharing

Secret sharing schemes are methods of distributing data between participants so that some subsets of the participants are able to restore the initial information using their shares of the data [52]. A secret sharing scheme is a (t, n)-*threshold* scheme if the data is shared among n participants and any subset of $t \leq n$ or more participants is able to restore it. A secret sharing scheme is *correct* if t shares uniquely determine the secret. In addition, the scheme is *private* if any set of $t - 1$ or less shares does not give any information about the secret.

Additive secret sharing is usually defined in a ring $\mathbb{Z}_N$ for some integer $N > 0$. To share a secret $x \in \mathbb{Z}_N$, one defines shares $x_1, \ldots, x_n$ where $x_1 + \ldots + x_n = x$ and all arithmetic is performed in $\mathbb{Z}_N$. This defines an (n, n)-threshold scheme, where a secret is divided to n parts and all of those are needed to restore the secret. Restoring the secret given all the shares is straightforward and only requires summing the shares. Each computing participant $\mathcal{CP}_i$ only receives the value x_i for secret x. Moreover, additive secret sharing is information-theoretically secure, meaning that having access to less than n shares does not reveal any information about the secret value. Such share representation also allows us to perform computations on the shares.

In the following, we use a two-party protocol where a secret $x \in \mathbb{Z}_N$ is distributed to two additive shares where $x_1 + x_2 \equiv x \bmod N$. Shared values will be denoted as $[\![x]\!]_N$. We omit the modulus if it is clearly inferred from the context.

2.1.2 Chinese remainder theorem

The Chinese remainder theorem (CRT) is a number theoretic result that has found many usages also in cryptography. The CRT states that a set of equations in the form

$$x \equiv a_i \bmod m_i \ , \quad i \in \{1, \ldots, k\}$$

where all m_i are pairwise coprime has a solution to x and it is uniquely fixed modulo $M = m_1 \cdot \ldots \cdot m_k$. In addition, the solution can be found as

$$x \equiv \sum_{i=1}^{k} a_i \cdot b_i \cdot \frac{M}{m_i} \bmod M$$

where

$$b_i \equiv \left(\frac{M}{m_i}\right)^{-1} \bmod m_i \ .$$

CRT is commonly used to reduce computations modulo M to many smaller computations modulo m_i and restore the result, or to unify computations for different m_i by computing them modulo M and then dividing back to separate moduli.

2.1.3 Universal composability

The framework of universal composability (UC) was proposed to provide a unified method for proving that protocols are secure even if executed sequentially or in parallel with other protocols [16, 47]. Security of a protocol is described in terms of securely implementing an ideal black-box behaviour of the protocol. An ideal functionality is described by a trusted third party (TTP), who securely collects all the inputs and then computes and returns the outputs of the protocol.

A protocol is said to be UC if for every adversary and computational context where the protocol is executed there exists an ideal world adversary that has the same computational advantage against the ideal world protocol. A universally composable protocol keeps its security guarantees in every context, provided that is is used in a black box manner. Thus, the protocol can receive inputs and give outputs, but all the intermediate messages are not used after the execution. UC security definitions give very strong security guarantees, but are, in general, also very restrictive.

Many two-party protocols can not be realized in the universally composable manner in the plain model, where the only setup assumption is the existence of authenticated communication [17]. The feasibility of universally composable two-party computation in the common reference string model (CRS) was shown in [18]. In addition, it is possible to avoid the assumption for trusted setup and base the protocols on the assumption of existence of public-key infrastructure like setup where each party has registered a public key, but no registration authority needs to be fully trusted [2].

2.1.4 Paillier cryptosystem

The Paillier public-key cryptosystem [45] uses an RSA modulus $N = pq$ where the secret components p and q are large primes of equal bit length. The requirement for equal bit length ensures that N is co-prime with the Euler totient function $\phi(N)$, namely

$$\gcd(pq, (p-1)(q-1)) = 1$$

where gcd is the *greatest common divisor*. The public key pk is (N, g), where $g \in \mathbb{Z}^*_{N^2}$ and the private key sk is the Carmichael function of N which can be computed as

$$\lambda = \operatorname{lcm}(p-1, q-1)$$

where lcm denotes the *least common multiple*.

For a shorthand, we define the Paillier cryptosystem as a set of algorithms for key generation, encryption and decryption as $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. The setup algorithm Gen is used to generate the keys pk and sk. The encryption function $\mathsf{Enc}_{pk}(m, r)$, where $m \in \mathbb{Z}_N$, requires a randomness $r \in \mathbb{Z}^*_N$ and defines the ciphertext as

$$c = \mathsf{Enc}_{pk}(m, r) = g^m r^N \bmod N^2 \ ,$$

where $c \in \mathbb{Z}^*_{N^2}$. In addition, the ciphertext space of the Paillier cryptosystem is equal to $\mathbb{Z}^*_{N^2}$ as Paillier showed that encryption is a bijection $\mathbb{Z}_N \times \mathbb{Z}^*_N \mapsto \mathbb{Z}^*_{N^2}$. This means that for each element c parties can publicly verify if it is a valid ciphertext. The decryption function

$$\mathsf{Dec}_{sk}(c) = \frac{L(c^\lambda \bmod N^2)}{L(g^\lambda \bmod N^2)} \bmod N$$

uses a helper function

$$L(x) = \left\lfloor \frac{x-1}{N} \right\rfloor \ .$$

The Paillier cryptosystem is additively homomorphic, allowing to compute the sum of the messages under encryption

$$\mathsf{Enc}_{pk}(m_1 + m_2, r_1 \cdot r_2) = \mathsf{Enc}_{pk}(m_1, r_1) \cdot \mathsf{Enc}_{pk}(m_2, r_2) \ .$$

This property also allows to evaluate the multiplication of an encrypted message and a plain value k under encryption $\mathsf{Enc}_{pk}(km) = \mathsf{Enc}_{pk}(m)^k$. We omit the randomness if its exact value is not important and in such case it is chosen uniformly during encryption.

Additive homomorphism also allows to re-randomize ciphertexts. Given a valid encryption $c = \mathsf{Enc}_{pk}(x)$ and $\mathsf{Enc}_{pk}(0)$ then $\hat{c} = c \cdot \mathsf{Enc}_{pk}(0)$ has the same distribution as $\mathsf{Enc}_{pk}(x, r), \ r \leftarrow \mathbb{Z}^*_N$ and nothing else than x can be learned from $\hat{c}$. In addition, for valid ciphertexts $c = \mathsf{Enc}_{pk}(x)$ and $d = \mathsf{Enc}_{pk}(y)$, the combination $c \cdot d \cdot \mathsf{Enc}_{pk}(0)$ reveals nothing else than $x + y$.

We say that a cryptosystem is (t, ε)-indistinguishable under a chosen plaintext attack (IND-CPA) if, for two known messages m_0 and m_1 and any t-time adversary $\mathcal{A}$, the probability of distinguishing between the encryptions of these messages is

$$Adv^{IND-CPA}(\mathcal{A}) = \left| \Pr[G^{\mathcal{A}}_{0,IND-CPA} = 1] - \Pr[G^{\mathcal{A}}_{1,IND-CPA} = 1] \right| \le \varepsilon \ .$$

where

$$G^{\mathcal{A}}_{0,IND-CPA} \left[\begin{array}{l} pk, sk \leftarrow \mathsf{Gen} \\ m_0, m_1 \leftarrow \mathcal{A}(pk) \\ \textbf{return } \mathcal{A}(\mathsf{Enc}_{pk}(m_0)) \end{array} \right. \qquad G^{\mathcal{A}}_{1,IND-CPA} \left[\begin{array}{l} pk, sk \leftarrow \mathsf{Gen} \\ m_0, m_1 \leftarrow \mathcal{A}(pk) \\ \textbf{return } \mathcal{A}(\mathsf{Enc}_{pk}(m_1)) \ . \end{array} \right.$$

The decisional composite residuosity assumption (DCRA) assumes that it is difficult to distinguish a random element of $\mathbb{Z}^*_{N^2}$ from a random N-th power in $\mathbb{Z}^*_{N^2}$. Let $\mathcal{N}$ be a randomised algorithm for generating Paillier moduli. Then $\mathcal{N}$ is considered to be (t, ε)-secure against DCRA if for any t-time adversary $\mathcal{A}$ the probability of distinguishing random elements from random N-th residues is at most

$$Adv^{DCRA}(\mathcal{A}) = \left|\Pr[G^{\mathcal{A}}_{0,DCRA} = 1] - \Pr[G^{\mathcal{A}}_{1,DCRA} = 1]\right| \le \varepsilon \ .$$

where

$$G^{\mathcal{A}}_{0,DCRA} \left[\begin{array}{l} N \leftarrow \mathcal{N} \\ x \leftarrow \mathbb{Z}^*_{N^2} \\ \textbf{return } \mathcal{A}(x, N) \end{array}\right. \qquad G^{\mathcal{A}}_{1,DCRA} \left[\begin{array}{l} N \leftarrow \mathcal{N} \\ x \leftarrow \mathbb{Z}^*_{N^2} \\ \textbf{return } \mathcal{A}(x^N \bmod N^2, N) \ . \end{array}\right.$$

Paillier cryptosystem is indistinguishable under chosen plaintext attacks (IND-CPA) under DCRA, which implies that the modulus N is hard to factor.

There are several known efficiency improvements to the basic definition of the Paillier cryptosystem, for example [45] and [22]. The decryption can be simplified by precomputing constant values in function $L(x)$ and using CRT. More precisely, we at first compute the decryption separately modulo p and q and use CRT to restore the decryption result modulo N [45].

In addition, we can define $g = N + 1$ which results in faster encryption function

$$\mathsf{Enc}_{pk}(m, r) = (Nm + 1)r^N \bmod N^2 \ .$$

This simplification results from the Binomial theorem which gives a general expression

$$(a + b)^c = \sum_{i=0}^{c} \binom{c}{i} a^i b^{c-i} \ ,$$

and from the fact that the generator does not need to be randomly chosen [22]. For our case, we need to compute

$$(N + 1)^m = \sum_{i=0}^{c} \binom{c}{i} N^i = 1 + cN + \binom{c}{2} N^2 + \cdots \ .$$

Encryption is a modular operation, thus, we can discard all the elements that have high powers of N since they are divisible by N^2 and we obtain

$$\mathsf{Enc}_{pk}(m, r) = (N + 1)^m r^N = (Nm + 1)r^N \bmod N^2 \ .$$

Furthermore, the encryption function can be computed more efficiently using the private key and CRT to compute separately modulo p^2 and q^2. Someone in possession of the private key might, for example, compute the encryption as a commitment to the encrypted value or to enable the other party to evaluate a function on the encrypted inputs.

2.1.5 Elliptic curves

Elliptic curves are plane curves with the general equation

$$y^2 = x^3 + ax + b$$

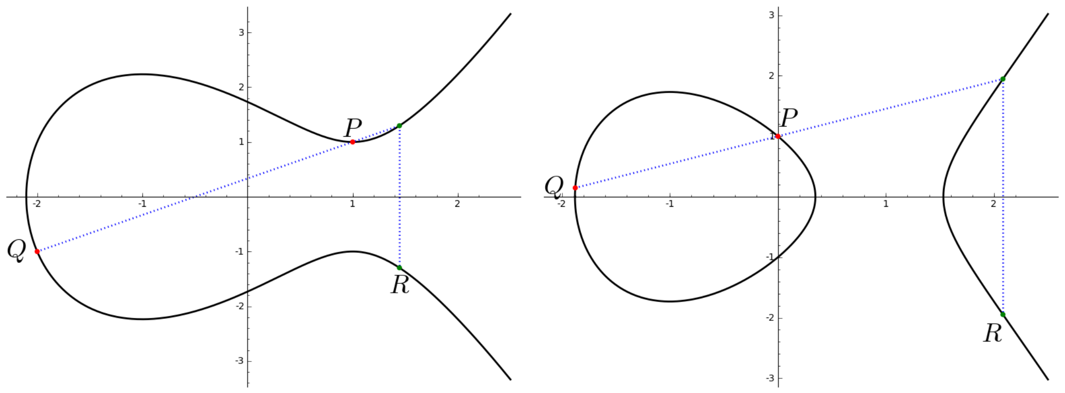


Figure 2.1: Elliptic curve addition $R = P + Q$ for curves $y^2 = x^3 - 3x + 1$ and $y^2 = x^3 - 3x + 1$ over real numbers.

where a and b are constants that define a specific curve. Cryptography usually only consider curves where $x, y, a, b \in \mathbb{F}$, for some finite field $\mathbb{F}$. In addition, there is a specific infinity point ∞ and a definition of a group operation so that elliptic curves form an Abelian group, where ∞ is the identity element. We usually use additive notation for operations between elliptic curve points which extends to multiplication of a point and an integer.

Figure 2.1 illustrates two different shapes of elliptic curves over real numbers. The geometric idea is that to find the sum of two points we must draw a line through them and find the third place where this line cuts the curve. The sum is this point mirrored by the x-axis or ∞ if there is no such point. Elliptic curves are much used in cryptography because their structure makes computing the discrete logarithm hard. Cryptography uses elliptic curves over finite fields where we can not draw illustrations as Figure 2.1, but the formulas derived from this planar interpretation still hold.

Elliptic curves are used as a basis for public key cryptosystems to be able to use shorter keys. For example, in our implementation, we can use 256-bit elliptic curve to achieve the same security level as 2048-bit Paillier key [3, 33]. To fix a curve, we must fix the field $\mathbb{F}$, constants a and b, base point (generator) of the elliptic curve and the order of the base point. There are several standard curves, where the parameters have been optimized for security and computational efficiency, for example see [43].

We use the elliptic curve *P-256* recommended by NIST [43] that is given in the Weierstrass form. Weierstrass form elliptic curves have a potential side-channel vulnerability because the addition of two different points and doubling one point have different algorithms. This is a serious threat as the point and scalar multiplication in elliptic curves is often implemented with an additive analogue of the *square-and-multiply* exponentiation algorithm and, therefore, analysing the power trace may leak the scalar used in the multiplication [15]. However, this risk can be mitigated with a common method of unifying the computation in these algorithms. We use Crypto++ [20] implementation of elliptic curves that avoids this vulnerability.

2.1.6 Lifted Elgamal cryptosystem

The Elgamal cryptosystem [29] is defined for a cyclic group $\mathbb{G}$ with generator g of prime order q. The setup phase defines the public key as $h = g^x$ and private key as x, where the latter can be chosen randomly. To encrypt a message $m \in \mathbb{G}$ we compute

$$\mathsf{E.Enc}_h(m, r) = (g^r, h^r \cdot m) \ ,$$

where $r \leftarrow \mathbb{Z}_q$. The decryption is defined as

$$\mathsf{E.Dec}_x(c_1, c_2) = c_2 \cdot c_1^{-x} \ ,$$

which is correct as

$$\mathsf{E.Dec}_x(\mathsf{E.Enc}_h(m, r)) = h^r \cdot m \cdot g^{-x \cdot r} = h^{r-r} \cdot m = m \ .$$

The Elgamal cryptosystem is multiplicatively homomorphic meaning that

$$\mathsf{E.Enc}_h(m) \cdot \mathsf{E.Enc}_h(n) = \mathsf{E.Enc}_h(m \cdot n) \ .$$

The Lifted Elgamal cryptosystem shares the setup phase with the aforementioned Elgamal cryptosystem, but differs from it as the encrypted message is used as an exponent rather than a multiplicand. In the following, we will use the Lifted Elgamal cryptosystem defined over elliptic curves. It is used as part of the commitment scheme and we do not use the expensive decryption operation. For a shorthand, we denote Lifted Elgamal cryptosystem as $(\mathsf{LE.Gen}, \mathsf{LE.Enc}, \mathsf{LE.Dec})$. The encryption function is defined as

$$\mathsf{LE.Enc}_h(m, r) = (g^r, h^r \cdot g^m)$$

and decryption requires computing a discrete logarithm on base g, denoted as $\log_g$, as follows

$$\mathsf{LE.Dec}_x(c_1, c_2) = \log_g(c_2 \cdot (c_1^{-x})) \ ,$$

which is correct as

$$\mathsf{LE.Dec}_x(\mathsf{LE.Enc}_h(m)) = \log_g(g^m) = m \ .$$

The Lifted Elgamal cryptosystem is additively homomorphic, as

$$\mathsf{LE.Enc}_h(m) \cdot \mathsf{LE.Enc}_h(n) = \mathsf{LE.Enc}_h(m + n) \ .$$

More precisely, homomorphism is correct because

$$\begin{aligned}\mathsf{LE.Enc}_h(m, r_m) \cdot \mathsf{LE.Enc}_h(n, r_n) &= (g^{r_m} \cdot g^{r_n}, h^{r_m} \cdot g^m \cdot h^{r_n} \cdot g^n) \\ &= (g^{r_m + r_n}, h^{r_m + r_n} \cdot g^{m+n}) = \mathsf{LE.Enc}_h(m + n) \ .\end{aligned}$$

We can also blind the ciphertext with $\mathsf{LE.Enc}_h(0)$ and having $\mathsf{LE.Enc}_h(m)$, $\mathsf{LE.Enc}_h(n)$ and $\mathsf{LE.Enc}_h(0)$ only allows us to learn $\mathsf{LE.Enc}_h(m + n)$.

Elgamal and Lifted Elgamal cryptosystems are IND-CPA secure under the Decisional Diffie-Hellman assumption (DDH). We say that a group $\mathbb{G}$ with generator g of order q is a (t, ε)-secure DDH-group if for any t-time adversary $\mathcal{A}$ the advantage

$$Adv^{DDH}(\mathcal{A}) = \left| \Pr[G^{\mathcal{A}}_{0,DDH} = 1] - \Pr[G^{\mathcal{A}}_{1,DDH} = 1] \right| \leq \varepsilon \ .$$

where

```
$G^{\mathcal{A}}_{0,DDH}$
    $x, y \leftarrow \mathbb{Z}_q$
    return $\mathcal{A}(g, g^x, g^y, g^{xy})$

$G^{\mathcal{A}}_{1,DDH}$
    $x, y, z \leftarrow \mathbb{Z}_q$
    return $\mathcal{A}(g, g^x, g^y, g^z)$ .
```

2.1.7 Zero-knowledge proofs

Zero-knowledge proofs are interactive proofs for the correctness of a statement, whereas the proofs should not reveal any additional information. Zero-knowledge proofs have three important properties: completeness, soundness and zero-knowledge. Completeness means that in case of an honest verifier and prover the verifier accepts the proof. Soundness, on the other hand, gives the guarantee that a malicious prover can not make the verifier accept a faulty statement. Zero-knowledge property ensures that in case of an honest prover and a correct statement, the verifier learns nothing aside from the proof outcome.

We need a zero-knowledge protocol to prove that we have three ciphertexts of elements $x_1,\ x_2,\ x_3$ in multiplicative relation $x_3 = x_1 \cdot x_2$ without revealing these elements, but assuming that we have a valid public key pk. This proof is defined in Algorithm 1 and is built from a conditional disclosure of secrets protocol based on [40]. The encryption scheme defines the randomness space $\mathcal{R}$ and the space of suitable messages $\mathcal{M}$. The space of secrets $\mathcal{S}$ of the conditional disclosure of secrets protocol serves as a randomness to check, if the prover received the correct message.

We use a randomised encoding function

$$\mathsf{Encode}(s, r) = s + 2^\ell \cdot r$$

for ℓ-bit secrets s and a randomness $r \leftarrow \mathbb{Z}_{\lfloor N/2^\ell \rfloor}$. Therefore, the randomness space $\mathcal{R}_1$ is defined by the encoding function as $\mathcal{R}_1 = \mathbb{Z}_{\lfloor N/2^\ell \rfloor}$. The corresponding decoding function is defined as

$$\mathsf{Decode}(s + 2^\ell \cdot r) = s + 2^\ell \cdot r \bmod 2^\ell = s$$

meaning that

$$\mathsf{Decode}(\mathsf{Encode}(s)) = s \ .$$

The latter ensures that the secret can be correctly restored if the queries q_1, q_2, q_3 were valid.

The encoding is said to be ε-secure if for any secret s and an additive non-zero subgroup $\mathcal{G} \subseteq \mathcal{M}$ the distribution of $\mathsf{Encode}(s) + \mathcal{G}$ is statistically ε-close to the uniform distribution over $\mathcal{M}$. This encoding is $\frac{2^{\ell-1}}{p}$-secure where p is the smallest factor of N [39]. The encoding function is used to add noise to hide the secret if the queries are not in the multiplicative relation.

The general idea of the protocol is that the prover should only be able to correctly get the secret $s' = s$ if the initial queries were in the multiplicative relation. This results from the conditional disclosure of secrets protocol that enables the prover to only learn the secret if the multiplicative condition is satisfied. Round 3 clearly gives correct result for multiplicative elements as

$$s' = \mathsf{Decode}(x_3 \cdot e_1 + x_2 \cdot e_2 + \mathsf{Encode}(s) - (x_1 \cdot e_1 + e_2) \cdot x_2) = \mathsf{Decode}(\mathsf{Encode}(s)) \ .$$

Nothing is leaked to the verifier in the honest case, as the prover only sends encryptions and the element $s' = s$ and an honest verifier already knows s. The verifier releases all used randomness to the prover in round 4 to show that it behaved honestly and really knows s. This ensures that the verifier can not learn new information from the provers secret s'. For security, we need a hiding and binding commitment scheme, secure randomised encoding function, and an IND-CPA secure cryptosystem.

Algorithm 1 CdsZkMul - zero-knowledge protocol for multiplicative relation of ciphertexts for an additively homomorphic cryptosystem

```
Setup: Commitment parameters $ck$,
       $Prover$ has a keypair $(pk, sk)$,
       $Verifier$ has $pk$
Data: $Prover$ has $x_1, x_2, x_3$
Result: $True$ for successful proof of $\mathsf{Enc}_{pk}(x_1) \cdot \mathsf{Enc}_{pk}(x_2) = \mathsf{Enc}_{pk}(x_3)$,
        $False$ in case of any failure or abort
 1: Round: 1
 2:     $Prover$ sets $q_1 = \mathsf{Enc}_{pk}(x_1)$, $q_2 = \mathsf{Enc}_{pk}(x_2)$, $q_3 = \mathsf{Enc}_{pk}(x_3)$
 3:     $Prover$ sends $q = (q_1, q_2, q_3)$ to $Verifier$
 4: Round: 2
 5:     $Verifier$ checks that $q_1, q_2, q_3$ are valid ciphertexts
 6:     $Verifier$ generates $e_1, e_2 \leftarrow \mathcal{M}$, $r_1, r_2 \leftarrow \mathcal{R}$, $r \leftarrow \mathcal{R}_1$, $s \leftarrow \mathcal{S}$
 7:     $Verifier$ computes $a_1 = q_1^{e_1} \cdot \mathsf{Enc}_{pk}(e_2, r_1)$
 8:     $Verifier$ computes $a_2 = q_3^{e_1} \cdot q_2^{e2} \cdot \mathsf{Enc}_{pk}(\mathsf{Encode}(s, r), r_2)$
 9:     $Verifier$ sends $a = (a_1, a_2)$ to $Prover$
10: Round: 3
11:     $Prover$ computes $s' = \mathsf{Decode}(\mathsf{Dec}_{sk}(a_2) - \mathsf{Dec}_{sk}(a_1) \cdot x_2)$
12:     $Prover$ computes $(c, d) = \mathsf{Com}_{ck}(s')$
13:     $Prover$ sends commitment $c$ to $Verifier$
14: Round: 4
15:     $Verifier$ sends $(s, e_1, e_2, r_1, r_2, r)$ to $Prover$
16: Round: 5
17:     $Prover$: if $a_1 \neq q_1^{e_1} \cdot \mathsf{Enc}_{pk}(e_2, r_1)$ return $False$
18:     $Prover$: if $a_2 \neq q_3^{e_1} \cdot q_2^{e2} \cdot \mathsf{Enc}_{pk}(\mathsf{Encode}(s, r), r_2)$ return $False$
19:     $Prover$ sends decommitment $d$ to $Verifier$
20: Round: 6
21:     $Verifier$: if $\mathsf{Open}_{ck}(c, d) \neq s$ return $False$
22: return $True$
```

In the following, we will use this protocol with the Paillier cryptosystem and a commitment scheme defined in Section 2.1.8. We will obtain two special cases of this protocol defined in Algorithm 12 and Algorithm 10.

2.1.8 Dual-mode commitment schemes

Commitment schemes allow one participant to commit to a value, but keep it private from other participants. Afterwards, the participant can open the commitment to prove that he initially committed to the value that he claims to have committed to.

We define a commitment scheme as the set of protocols, namely setup, commitment and opening, $(\mathsf{Gen}, \mathsf{Com}, \mathsf{Open})$. In general, they rely on an additively homomorphic IND-CPA cryptosystem. More precisely, we will use Lifted Elgamal on elliptic curves in our initialisation.

The commitment parameters of a binding commitment are

$$ck = (pk, e)$$

where $e = \mathsf{LE.Enc}_{pk}(1)$. We use the homomorphic properties of the cryptosystem to compute the commitment and decommitment to m as

$$\mathsf{Com}_{ck}(m) = (c, d)$$

where commitment is $c = e^m \cdot \mathsf{LE.Enc}_h(0, r) = \mathsf{LE.Enc}_{pk}(m)$ and decommitment $d = (m, r)$. A commitment is opened by releasing all values used to compute c and the opening function just recalculates the commitment and verifies the correctness as

$$\mathsf{Open}_{ck}(c, d) = \begin{cases} m, & \textbf{if } c = e^m \cdot \mathsf{LE.Enc}_h(0, r) \wedge c \in \mathcal{C} \\ \perp, & \text{otherwise} \end{cases}$$

where $\mathcal{C}$ is the set of valid ciphertexts.

Commitment schemes have two important properties: hiding and binding. A commitment scheme is (t, ε)-hiding if, for any t-time adversary $\mathcal{A}$, the probability of distinguishing commitments to two different messages is

$$Adv^{hiding}(\mathcal{A}) = \left| \Pr[G^{\mathcal{A}}_{0,hiding} = 1] - \Pr[G^{\mathcal{A}}_{1,hiding} = 1] \right| \leq \varepsilon \ ,$$

where

```
$G^{\mathcal{A}}_{0,hiding}$
    $ck \leftarrow \mathsf{Gen}$
    $(m_0, m_1) \leftarrow \mathcal{A}(ck)$
    $(c, d) \leftarrow \mathsf{Com}_{ck}(m_0)$
    return $\mathcal{A}(c)$

$G^{\mathcal{A}}_{1,hiding}$
    $ck \leftarrow \mathsf{Gen}$
    $(m_0, m_1) \leftarrow \mathcal{A}(ck)$
    $(c, d) \leftarrow \mathsf{Com}_{ck}(m_1)$
    return $\mathcal{A}(c)$ .
```

A commitment scheme is (t, ε)-binding if for any t-time adversary $\mathcal{A}$ the probability of creating a double opening is

$$Adv^{binding}(\mathcal{A}) = \Pr[G^{\mathcal{A}}_{0,binding} = 1] \leq \varepsilon \ ,$$

where

```
$G^{\mathcal{A}}_{0,binding}$
    $ck \leftarrow \mathsf{Gen}$
    $(c, d_0, d_1) \leftarrow \mathcal{A}(ck)$
    if $\mathsf{Open}_{ck}(c, d_0) \neq \perp \wedge \mathsf{Open}_{ck}(c, d_1) \neq \perp$
        return $\mathsf{Open}_{ck}(c, d_0) \neq \mathsf{Open}_{ck}(c, d_1)$
    else
        return $0$ .
```

We call a commitment perfectly equivocal, if a valid commitment can be efficiently opened to any message given some trapdoor information. We say that a commitment scheme is (t, ε)-equivocal, if for any t-time adversary $\mathcal{A}$ the advantage of distinguishing real and equivocal commitments is bounded as

$$Adv^{equivocal}(\mathcal{A}) = \left| \Pr[G^{\mathcal{A}}_{0,equivocal} = 1] - \Pr[G^{\mathcal{A}}_{1,equivocal} = 1] \right| \leq \varepsilon \ ,$$

where

```
$G^{\mathcal{A}}_{0,equivocal}$
    $ck \leftarrow \mathsf{Gen}$
    $\mathcal{A}(ck)$
    for as long as $\mathcal{A}$ wants
        $m \leftarrow \mathcal{A}$
        $(c, d) \leftarrow \mathsf{Com}_{ck}(m)$
        $\mathcal{A}(c, d)$
    end for
    return $\mathcal{A}(ck)$
```

```
$G^{\mathcal{A}}_{1,equivocal}$
    $ck, ek \leftarrow \mathsf{FakeGen}$
    $\mathcal{A}(ck)$
    for as long as $\mathcal{A}$ wants
        $m \leftarrow \mathcal{A}$
        $c, r \leftarrow \mathsf{FakeCom}_{ck,ek}()$
        $d \leftarrow \mathsf{Equivocation}_{ck,ek}(d, m, r)$
        $\mathcal{A}(c, d)$
    end for
    return $\mathcal{A}(ck)$ .
```

According to the requirements of the aforementioned zero-knowledge proof protocols, we need our commitment to be an equivocal and binding dual-mode commitment [40]. We need computationally indistinguishable setup phases that yield either statistically binding or perfectly equivocal commitment scheme to fulfil the requirements of the dual-mode commitment.

The previously described commitment scheme allows us to define a suitable equivocal setup. In addition, the ideal setup and defined algorithms $(\mathsf{Gen}, \mathsf{Com}, \mathsf{Open})$ yield a computationally hiding and unconditionally binding commitment as proved in Theorem 2.1.1. The security proofs in this section give guarantees up to a constant factor in terms of the running time.

Theorem 2.1.1. *The commitment scheme in algorithms* $(\mathsf{Gen}, \mathsf{Com}, \mathsf{Open})$ *yields a* (t, ε)*-hiding and unconditionally binding commitment given a* (t, ε)*-IND-CPA secure cryptosystem.*

Proof Sketch. According to the commitment algorithm, the commitment to message m is $c = \mathsf{Enc}_{pk}(m)$. Hence, the hiding property of the commitment scheme directly follows from the definition of IND-CPA security of the cryptosystem.

Similarly, the binding property follows from the fact that the public key pk defines a valid secret key and, thus, it is theoretically possible to decrypt the commitment, whereas the decryption always succeeds and yields m. □

We can use an altered setup so that the commitment key $ck = (h, e)$ and equivocation key ek are fixed according to $\mathsf{FakeGen}$ to obtain a perfectly equivocal commitment. $\mathsf{FakeGen}$ defines $ck = (h, \mathsf{LE.Enc}_h(0, r_*))$ and $ek = r_*$. Equivocal commitment is computed as

$$\mathsf{FakeCom}_{ck} = (c, r')$$

where r' is the trapdoor for equivocation and $c = \mathsf{LE.Enc}_h(0, r')$. Such a commitment and opened to any chosen message m by

$$\mathsf{Equivocation}_{ck,r_*}(m, c, r') = (m, r' - r_* \cdot m) \ .$$

The result can be verified using Open as defined previously. The correctness of this equivocal setup is shown in Theorem 2.1.2.

Theorem 2.1.2. *Algorithms* $(\mathsf{FakeGen}, \mathsf{FakeCom}, \mathsf{Equivocation}, \mathsf{Open})$ *yield a perfectly equivocal commitment of a random message given a* (t, ε)*-IND-CPA-secure Lifted Elgamal cryptosystem.*

Proof. According to the definition of Com, any commitment would be $c = \mathsf{LE.Enc}_h(0)$ and knowing r_* it can be opened to any message m. If we initially computed the commitment as $\mathsf{FakeCom}_{ck}$ and obtained

$$c = \mathsf{LE.Enc}_h(0, r') = (g^{r'}, h^{r'}) \ ,$$

then we can easily compute the decommitment to any m as $r = r' - r_* \cdot m$ as defined by $\mathsf{Equivocation}$. This is accepted by Open because

$$\begin{aligned} e^m \cdot \mathsf{LE.Enc}_h(0, r) &= \mathsf{LE.Enc}_h(0, r_*)^m \cdot \mathsf{LE.Enc}_h(0, r) = \mathsf{LE.Enc}_h(0, r_* \cdot m + r) \\ &= \mathsf{LE.Enc}_h(0, r_* \cdot m + r' - r_* \cdot m) = \mathsf{LE.Enc}_h(0, r') = c \ . \end{aligned}$$

Secondly, we need that the distributions of $(c, d) \leftarrow \mathsf{Com}_{ck}(m)$ and (c_*, d_*) where $c_* \leftarrow \mathsf{FakeCom}_{ck}$ and $d_* \leftarrow \mathsf{Equivocation}_{ck,r_*}(m)$ coincide. The distributions of c and c_* coincide because they are both random encryptions of zero. If we compute the commitment according to Com_{ck} with setup from Gen, we obtain

$$\begin{aligned} c &= e^m \cdot \mathsf{LE.Enc}_h(0, r) = ((g^{r_*})^m, (h^{r_*} g^0)^m) \cdot (g^r, h^r g^0) \\ &= (g^{r_* \cdot m + r}, h^{r_* \cdot m + r} g^0) = (g^{r_* \cdot m + r}, h^{r_* \cdot m + r}) = (g^{r'}, h^{r'}) \\ &= \mathsf{LE.Enc}_h(0, r') \ . \end{aligned}$$

In addition, fixing a message m and having a commitment c uniquely fixes the only possible decommitment $d = (m, r)$ as the value r is uniquely fixed. Thus, if the distributions of commitments c and c_* coincide then so do the joint distributions of (c, d) and (c_*, d_*).

Thus, we have a perfectly equivocal commitment. □

Theorem 2.1.3. *The setups for binding* (Gen) *and equivocal* ($\mathsf{FakeGen}$) *commitment schemes are* (t, ε)*-indistinguishable for parties who only see the commitment parameters* $ck = (pk, e)$*, given a* (t, ε)*-IND-CPA secure cryptosystem.*

Proof Sketch. Assume that there is an adversary $\mathcal{A}$ who can distinguish between these two setup phases. This means that $\mathcal{A}$ can differentiate between $(pk, \mathsf{LE.Enc}_h(0))$ and $(h, \mathsf{LE.Enc}_h(1))$. Hence, we can build an adversary $\mathcal{B}$ for the IND-CPA security of the cryptosystem. At first $\mathcal{B}$ sends the messages $m_0 = 0$ and $m_1 = 1$ and receives the ciphertext $\mathsf{LE.Enc}_h(m_b)$. $\mathcal{B}$ then forwards the message $(h, \mathsf{LE.Enc}_h(m_b))$ to $\mathcal{A}$ and outputs the result as $\mathcal{A}$. Hence, $\mathcal{B}$ breaks the IND-CPA security exactly when $\mathcal{A}$ successfully distinguishes the setups and the success of t-time adversary $\mathcal{A}$ is bounded by ε. The running time of $\mathcal{B}$ needs only to be constant time longer than $\mathcal{A}$ to forward the messages. □

Corollary 2.1.4. *According to Theorems 2.1.2 and 2.1.3 commitment setups are* (t, ε)*-indistinguishable even if the adversary* $\mathcal{A}$ *sees pairs of correct commitments* (c, d) *or fake commitments* (c_*, d_*)*.*

For practical purposes we need a protocol to implement the setup phase. It can be combined from the Diffie-Hellman key exchange [28], homomorphic properties of Lifted Elgamal cryptosystem and Schnorr Σ-protocols [51]. It is important that the setup should yield perfectly binding commitment in case computing parties execute it, but there also has to exist a simulator who can achieve equivocal setup. Full specification of this protocol is out of the scope of this thesis.

2.1.9 Message authentication codes

A message authentication code (MAC) is an extra piece of information about a message that enables to detect modifications to the initial message. MACs are often described as keyed hash functions that output a tag from a message and a secret key. A MAC is secure, if an adversary can not substitute messages or generate valid message and tag pairs. A substitution attack means changing message and tag pair (x, z) with a different pair $(\overline{x}, \overline{z})$ where $\overline{z}$ is a valid tag for $\overline{x}$. An impersonation attack means generating a valid pair $(\overline{x}, \overline{z})$ without seeing any authentication pairs before.

We define a MAC for secret-shared elements as follows. We have a key k and a secret $[\![x]\!]$, where $x = x_1 + x_2$ and $x_1,\ x_2$ are the additive shares. We define $z = k \cdot x$ as the authentication code for x and keep it as shares $z_{1,x}, z_{2,x}$, hence

$$z_{1,x} + z_{2,x} = k \cdot (x_1 + x_2) \ .$$

Verifying the code is trivial for anyone in possession of the secret key k. The key should be chosen from the same algebraic structure as used for the secret sharing.

Keeping MAC in shares allows us think of it as $z_{2,x} = k \cdot (x_1 + x_2) - z_{1,x}$ where $z_{2,x}$ is the tag for secret x and $z_{1,x}$ is part of the secret key which becomes $(k, z_{1,x})$. Hence, the view of $\mathcal{CP}_2$ is like having tags for unknown messages x where all these pairs share the sub-key k, but differ by the second part of the key. Such construction was introduced by Rabin and Ben-Or as *Information Checking* or *Check Vectors* for verifiable secret sharing [49].

Our attack scenario results from the opening of the shares where $\mathcal{CP}_2$ might receive x_1 and therefore learn x before sending x_2 and z_2 to $\mathcal{CP}_1$. Hence, we need that $\mathcal{CP}_2$ must not be able to come up with $\hat{x}_2$ and $\hat{z}_2$ such that $\mathcal{CP}_1$ would accept the MAC. This is a special substitution attack, because the attacker only gets to see one message and tag pair before the attack, however it is sufficient as the second part of the key is always different and the the attacker can not see tags for more messages of the same key. We can more precisely define it as a security game for $[\![x]\!]_N$ in ring R as $G^{\mathcal{A}}_{MAC}$. We say that a MAC is statistically ε-secure, if for any adversary $\mathcal{A}$ the probability of winning in $G^{\mathcal{A}}_{MAC}$ is bounded by ε:

$$\Pr[G^{\mathcal{A}}_{MAC} = 1] \leq \varepsilon$$

where

```
$G^{\mathcal{A}}_{MAC}$
  $k \leftarrow R$
  **for** as long as $\mathcal{A}$ wants
      $x \leftarrow \mathcal{A}$
      $z_1, x_2 \leftarrow R$
      $z_2 = k \cdot x - z_1$
      $\mathcal{A}(z_2, x - x_2, x_2)$
  **end for**
  $x \leftarrow \mathcal{A}$
  $z_1, x_2 \leftarrow R$
  $z_2 = k \cdot x - z_1$
  $x_1 = x - x_2$
  $\hat{z}_2, \hat{x}_2 \leftarrow \mathcal{A}(z_2, x_1, x_2)$
  **return** $\hat{z}_2 == (\hat{x}_2 + x_1) \cdot k - z_1 \wedge x_2 \neq \hat{x}_2$ .
```

Theorem 2.1.5. *The adversaries success in $G^{\mathcal{A}}_{MAC}$ for a finite field $\mathbb{F}_{p^n}$ is bounded by $\frac{1}{p^n}$.*

Proof Sketch. First, the views of the adversary $\mathcal{A}$ in $G^{\mathcal{A}}_{MAC}$ and $G'^{\mathcal{A}}_{MAC}$ are indistinguishable and the advantage is the same, where

```
$G'^{\mathcal{A}}_{MAC}$
    for as long as $\mathcal{A}$ wants
        $x \leftarrow \mathcal{A}$
        $z_2, x_2 \leftarrow \mathbb{F}_{p^n}$
        $\mathcal{A}(z_2, x - x_2, x_2)$
    end for
    $x \leftarrow \mathcal{A}$
    $z_2, x_2 \leftarrow \mathbb{F}_{p^n}$
    $x_1 = x - x_2$
    $\hat{z}_2, \hat{x}_2 \leftarrow \mathcal{A}(z_2, x_1, x_2)$
    $k \leftarrow \mathbb{F}_{p^n}$
    $z_1 = k \cdot x - z_2$
    return $\hat{z}_2 == (\hat{x}_2 + x_1) \cdot k - z_1 \wedge x_2 \neq \hat{x}_2$
```

The values of z_2 in $G^{\mathcal{A}}_{MAC}$ are randomly uniform, because z_1 is chosen uniformly and, therefore, $kx - z_1 \bmod p$ is also uniformly random element of $\mathbb{F}_{p^n}$. In addition, the values that adversary sees do not depend on the key k, so, it can be chosen later.

Clearly, the advantage in $G'^{\mathcal{A}}_{MAC}$ is the same as the possibility of coming up with a pair $\hat{z}_2, \hat{x}_2$ such that $\hat{z}_2 == (\hat{x}_2 + x_1) \cdot k - z_1$. After $\mathcal{A}$ picks $\hat{z}_2, \hat{x}_2$ there is exactly one

$$k_* = (z_1 + \hat{z}_2) \cdot (\hat{x}_2 + x_1)^{-1} \quad ,$$

such that $\hat{z}_2 == (\hat{x}_2 + x_1) \cdot k_* - z_1$. To conclude, the possibility of picking k such that the verification succeeds is $\frac{1}{p^n}$ because $\mathbb{F}_{p^n}$ has p^n different elements and only one uniquely fixed k_*. □

Our message space $\mathcal{M}$ may have a composite order and, therefore, this is not as secure MAC as it would be in case of finite fields. However, we will have $R = \mathbb{Z}_N$, where $N = pq$ is the Paillier modulus and both of its factors are large and the security is the same as it would be for either of the prime factors.

Theorem 2.1.6. *The success of adversary $\mathcal{A}$ in $G^{\mathcal{A}}_{MAC}$ with a Paillier modulus N that defines $R = Z_N$, where $N = pq$, p and q are primes, and $p < q$ is bounded by $\frac{1}{p}$.*

Proof Sketch. Assume that there is an adversary $\mathcal{A}$ against $G^{\mathcal{A}}_{MAC}$ for some modulus $N = pq$ who is very successful. Then, there exists an adversary $\mathcal{B}$ in G_{MAC} for modulus p, who can use this $\mathcal{A}$ to win in its game. Adversary $\mathcal{B}$ has a fixed modulus p, picks a prime $q = N/p$ and uses the adversary $\mathcal{A}$ for modulus N. For every x that $\mathcal{A}$ picks $\mathcal{B}$ forwards it to the challenger and gives the result pack to $\mathcal{A}$. It behaves similarly with the final challenge. According to the Chinese remainder theorem, if $\mathcal{A}$ gives a correct result modulo N then it also holds modulo p and $\mathcal{B}$ wins it its game exactly when $\mathcal{A}$ is successful. The runtime of $\mathcal{B}$ is only a constant factor longer than $\mathcal{A}$. However, the maximal success of $\mathcal{B}$ is limited by Theorem 2.1.5 as $\mathbb{Z}_p$ is a finite field and, thus, the maximal success of $\mathcal{A}$ is also $\frac{1}{p}$. From this we know that for any Paillier modulus $N = pq$, the maximal success is $\frac{1}{p}$ where $p < q$. □

This MAC also has homomorphic properties making it possible to compute the new MAC and new key for the sum of two messages given the tags of the initial messages.

2.2 Secure multi-party computation

Secure multi-party computation (SMC) is a mechanism that allows several participants to evaluate a function without revealing their inputs. A classical SMC problem known as the Millionaires' problem was proposed by Yao [56]. There are two millionaires who wish to know who has more money without revealing their wealth to the other millionaire.

2.2.1 Overview of SMC techniques

Garbled circuits

Together with the Millionaires' problem Yao proposed a solution for securely evaluating boolean circuits [56]. This approach is known as garbled circuit evaluation and has developed a lot since it was first proposed. Garbled circuits are commonly used for two-party computations in the passive model, but this approach can be extended to more parties [5, 55].

The general idea of garbled circuits is that the original circuit of a function is transformed so that the wires only contain random bitstrings. Each gate is encoded so that its output bitstring can be computed from the inputs and only the random bitstrings of output gates can be mapped back to actual results. This way the evaluation computes the function, but does not leak information about the values on separate wires. The main drawback of the garbled circuit technique are inefficient evaluation and inability to reuse the circuit. However, they have been used in SMC frameworks [42, 35].

Secret sharing

Secret sharing was introduced by Shamir [52] and Blakley [8]. Since then Shamir's scheme has provided a basis for different verifiable secret sharing [30, 46, 49], SMC and threshold encryption ideas [31]. SMC frameworks can be obtained from the secret sharing schemes based on the homomorphic properties of the schemes and by defining protocols for operations not directly supported by the homomorphism.

This thesis and the Sharemind framework are based on additive secret sharing as introduced in Section 2.1.1. Share computation is mainly aimed at securely evaluating arithmetic circuits. For many use-cases arithmetic circuits are more efficient than boolean circuits and, therefore, share computation is likely to be more efficient than garbled circuits.

Homomorphic encryption

Homomorphic encryption is especially useful for building secure client-server model applications, but it can be extended to more general settings. For example, the a sends encrypted inputs to a server who then computes the desired function on the ciphertexts.

Currently we know of several additively homomorphic cryptosystems, such as Paillier or Lifted Elgamal, or multiplicatively homomorphic cryptosystems, for example Elgamal. We can use them to obtain frameworks where one side can compute some operations locally, but others require collaboration. These difficulties can be overcome with *fully homomorphic* cryptosystems where both multiplication and addition can be performed locally [32]. However, at current state, fully homomorphic encryption is too inefficient for practical SMC frameworks.

The main limitation of SMC frameworks based on homomorphic cryptosystems is the inability to use common data types. The cryptosystem defines a modulus and all the arithmetic is with respect to these moduli. However, to achieve reasonable security, we commonly need moduli that are thousands of bits long. We use some of the ideas from this setup in the precomputation phase of our protocol sets and, therefore, we also suffer from this restriction on our modulus.

2.2.2 General SMC threat model

An important privacy goal of SMC is that all inputs and outputs should remain private, unless specifically declassified. However, we can not avoid that the output of a function may leak information about the inputs. Furthermore, we often require that the correctness of the outputs is guaranteed or at least verifiable.

The passive or *honest-but-curious* security model defines an adversary who always follows the protocol specification, but may try to extract additional information from its view of the protocol. An active or *malicious* security model proposes no restrictions to the behaviour of the adversary whereas the security aim is to catch the adversary with overwhelming probability. Consequently, the adversary can control all aspects of the corrupted parties and communication channels. *Covert* security model is somewhere in between the previous two, as the adversary can behave maliciously and must be caught with certain arbitrarily fixed probability. However, there is a bigger risk of leaking secrets than when achieving security against an active adversary.

In addition, adversarial behaviour can either be *static*, *adaptive*, or *mobile*. A static adversary picks the set of corrupted parties in the beginning of the protocol and is unable to change it later. An adaptive adversary can increase the set of corrupted parties over time. Finally, a mobile adversary can adaptively corrupt and release parties, thus varying the set of corrupted parties during the protocol execution.

Although we mostly concentrate on the computing parties, it is possible that the adversary is not any of the computing nodes, but, for example, someone in the network. Such adversary can eavesdrop or modify the network and disrupt the communication. We can use classical techniques to secure the channels against eavesdropping or modification, but we can not solve different denial of service attacks on the network.

A commonly used threshold limitation of SMC is that achieving unconditional security against a passive adversary is only possible if less than $\frac{n}{2}$ parties of n are corrupted or correspondingly $\frac{n}{3}$ for active adversary [19, 6]. These results are special cases of more general result with adversary structures that allow for stronger results [36, 37]. An adversary structure consists of sets of parties where the adversary is allowed to corrupt any of these sets. Let $Q^{(2)}$ ($Q^{(3)}$) be the conditions that no two (three) of these sets cover the whole set of parties. Every function can then be unconditionally securely computed by an active adaptive adversary if it is in $Q^{(3)}$. Analogous result holds for adaptive passive adversary for $Q^{(2)}$.

2.2.3 Achieving actively secure two-party computation

The active security model allows the adversary to behave maliciously and do anything it likes, for example, send incorrect messages. Hence, we need to ensure the correctness of computations and the privacy of the inputs. In addition, we also require universal composability to use the basic protocols as building blocks for more general functions. However, we restrict our adversary so that only one of the computing parties can be statically compromised and the two computing parties can not collude. The properties of the additive secret sharing scheme clearly define that the two computing parties can not collude and, hence, the adversary can not corrupt both of the computing parties.

We take the same approach as SPDZ [26] to ensure the correctness of computation results, where we only verify the correctness when we publish a result and not during the computation. Here we rely of the security properties of the used verification mechanisms. In addition, we assume that the setup of the protection domain has been securely fixed and other protocols must be secure in the shared setup model.

In general, we require universal composability, but for brevity we do not give full proofs for this. In the following, security means both privacy of the inputs and the correctness of the outputs. The security is achieved using protection mechanisms on the shares. All protocols with only local computation trivially protect privacy, but we need to ensure privacy in collaborative protocols.

In the following, we prove security of the protocols in the stand-alone model with shared setup so that it also implies security in sequential compositions. For some protocols, we only show the simulatability of the communication so that the adversary can not distinguish between simulated and real protocol run.

We actually assume that there are three conditions that a protocol needs to fulfil to achieve security. Firstly, the communication of the protocol should be simulatable. Secondly, the parties can notice if others cheat. Thirdly, after the protocol, the parties are convinced about the consistency of the share. However, proving that these are sufficient, is out of the scope of this thesis.

2.3 The Sharemind SMC framework

Sharemind is an SMC framework [11, 12, 9] with three main goals: (1) it must be usable for securely processing confidential data, (2) it must be efficient enough for practical applications, and (3) it must be usable by non-cryptographers. This thesis is based on version 3 of the Sharemind framework where we can easily define new secure computation schemes in addition to the traditional one with three miners and a passive security guarantee.

2.3.1 Application model

Sharemind is designed as a general tool for SMC and privacy preserving data mining. The model has three different kinds of parties: (1) computing parties, (2) input parties, and (3) result parties. One participant can belong to all of these classes.

Input parties use secret sharing to distribute their inputs between the computing parties and are denoted as $\mathcal{IP}_i$. Input parties correspond to the data donors or owners of the data and can often be the same as the result parties. Computing parties, a.k.a miners, perform computations on the shared data following the protocols specific to the sharing method. Computing party will be denoted as $\mathcal{CP}_i$ and can be thought of as a dedicated server, we denote the set of computing parties by $\mathcal{CP}$. Finally, result parties map to data analysts who initiate the queries and computations and learn the final public outcomes. They will be denoted as $\mathcal{RP}_i$. Result parties get to aggregate the data from the input parties without actual access to confidential inputs.

The number of input and result parties is not limited, but the number of computing parties is often defined by the computation protocols. For example, classical Sharemind protocols use three computing parties, but this thesis focuses on the case with two miners.

The important trust requirements are that the input parties must believe that the computing miners do not collude and the result parties must believe that the computing parties give correct results. The latter can be ensured at a protection domain level as long as the miners do not collude. In addition, the miners should either believe or check the consistency of inputs. However, in any case we can not avoid the attack where input party decides to classify false information.

2.3.2 Computation primitives

In general, a query from the result party means that the computing parties must execute some algorithm to compute the result. These algorithms are collections of operations such as addition or multiplication. Each of these operations in turn correspond to a secure computation protocol. The computing parties execute the corresponding protocols in order to securely evaluate the query.

A protection domain kind (PDK) is a set of algorithms that define the data representation and computation protocols. Different protection domain kinds can define different elementary operations. For example, some may support division, but others do not have to. A protection domain (PD) is a concrete initialisation of a protection domain kind. A protection domain is defined by the algorithms from the corresponding protection domain kind and its configuration, for example, the keys of the participants. In addition, a protection domain also consists of the protected data. Common Sharemind PDK is based on additive secret sharing among three miners and is secure in the passive adversary model, where a PD is fixed by the computing nodes.

All elementary protocols of the PDK must be reusable and composable with each other to achieve provably secure query evaluation. It has been shown that following simple rules when designing protocols for elementary operations yields a provably secure composition of protocols for the traditional three miner PDK [9]. In general we require universal composability of the computation protocols.

2.3.3 Programming applications

Sharemind 3 uses the SecreC 2 programming language to specify the query algorithms. SecreC is a Sharemind specific C-like language designed to be privacy-preserving and easy to use. SecreC is strongly typed whereas the type of the private variables includes the PD. In fact, all public values can be seen as belonging to some public PD and also including this in their type. The programmer does not need to be aware of the underlying PDK protocols for operations on the private data and can call them as any predefined functionality.

SecreC can be used for different protection domain kinds and for writing code that is not specific to any fixed PDK [10]. The domain-polymorphic code clearly only works if the PDK defines all the necessary protocols. Polymorphism means that integrating new PDKs to applications is simple and one can easily test their application against several PDs or develop libraries independently of the PDK. In addition, it is possible to implement a general function and then specify a special version of it for some PDK where, for example, the required functionality can be achieved more efficiently than in the generic code.

Algorithm 2 Example of SecreC

```
kind additive2pa;
kind additive2paSym;
domain pd_a2a additive2pa;
domain pd_a2a_sym additive2paSym;

template <domain D>
D uint32 sum (D uint32 [[1]] arr) {
    D uint32 out = 0;
    for (uint64 i = 0; i < size(arr); i++){
        out = out + arr[i];
    }
    return out;
}

void main () {
    uint64 n = 10;
    pd_a2a uint32 [[1]] arr1 (n) = 2;
    pd_a2a uint32 s1 = sum(arr1);
    assert (declassify(s1) == (20 :: uint32));

    pd_a2a_sym uint32 [[1]] arr2 (n) = 3;
    pd_a2a_sym uint32 s2 = sum(arr2);
    assert (declassify(s2) == (30 :: uint32));

    return;
}
```

Algorithm 2 gives an example of SecreC code that uses two different PDK where the domain fixes the setup of given PD. The main function defines one dimensional matrix (vector) of length n with secret shared elements equal to either 2 or 3 and computes the sum of the vector elements for both of these PD. The function sum is defined independently of the used PDK and can be used as long as the PDK defines type *uint32* and an addition operation. Finally the main function publishes the result and verifies that it has the value that was expected.

Chapter 3

Principles of the SPDZ framework

This chapter introduces the main aspects of SPDZ (pronounced *Speedz*) that is an actively secure SMC framework [26] that also has a covertly secure version [23]. An important characteristic of SPDZ is the usage of precomputations, which separate the protocols to two parts as also used in [24, 21, 25, 7, 44]. Firstly, the precomputation phase is independent of the secret information and produces some random shares. Secondly, the online phase uses the secrets and precomputation results to evaluate necessary functions.

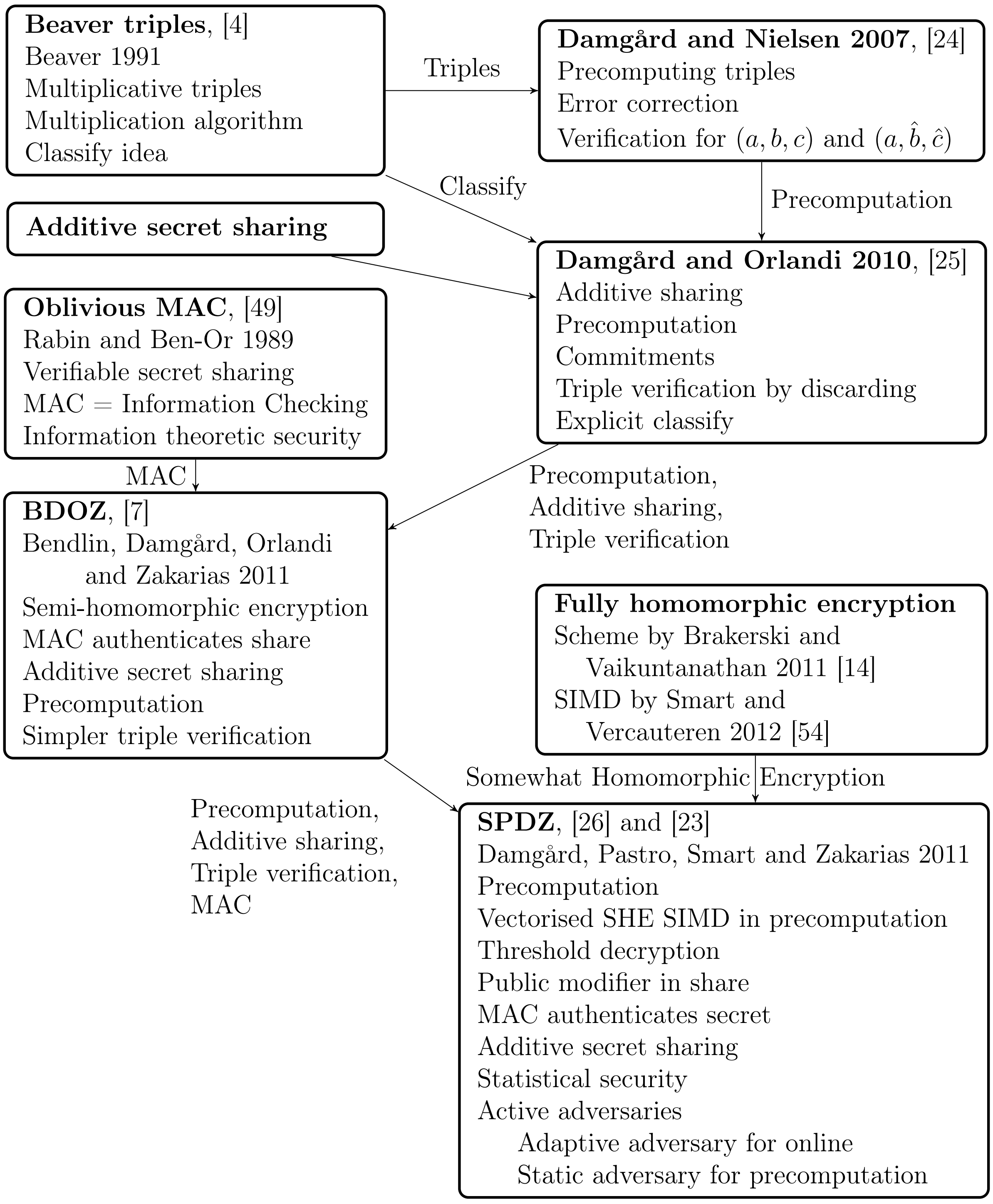
The SPDZ framework utilises three important tools: (1) Oblivious Message Authentication Codes, (2) Beaver triples, and (3) vectorized homomorphic encryption. The first two are used to ensure security against an active adversary and the second as precomputation for multiplication. These two have been previously used together for SMC in BDOZ [7]. However, SPDZ adds an important idea that MAC is used to authenticate the shared secret as a whole and not for authenticating independent shares. Thirdly, vectorised somewhat-homomorphic encryption is used to generate Beaver triples in a communication-efficient way and is a SPDZ-specific property.

It can be seen that SPDZ is a mixture of different previously existing ideas. We actually omit the usage of the somewhat homomorphic encryption, which is the most specific idea of SPDZ, but we use the idea of authenticating the secret, not the separate shares. In the following, we, in general, use the name SPDZ to refer to the collection of these ideas and reference the origins separately as we introduce the concepts. Our vision on the development of SPDZ is given on Figure 3.1.

3.1 Precomputation model

The precomputation and online phases are essentially independent and can be optimized separately, as long as they have consistent share representations. However, having a separate precomputation phase is meaningful only as long as preprocessing gives some benefit to the online phase.

Currently, SPDZ precomputes Beaver triples and single random shares. The covertly secure extension [23] also precomputes squaring pairs analogous to Beaver triples and shared bits for comparison, bit-decomposition, fixed point and floating point operations. It is an open question if other operations can be efficiently precomputed. The precomputation model originates from Beaver [4] and has found wider usage in SMC after [24].



Beaver triples, [4]
Beaver 1991
Multiplicative triples
Multiplication algorithm
Classify idea
Triples
Damgård and Nielsen 2007, [24]
Precomputing triples
Error correction
Verification for (a, b, c) and $(a, \hat{b}, \hat{c})$
Classify
Precomputation
Additive secret sharing
Damgård and Orlandi 2010, [25]
Additive sharing
Precomputation
Commitments
Triple verification by discarding
Explicit classify
Oblivious MAC, [49]
Rabin and Ben-Or 1989
Verifiable secret sharing
MAC = Information Checking
Information theoretic security
MAC
Precomputation, Additive sharing, Triple verification
BDOZ, [7]
Bendlin, Damgård, Orlandi and Zakarias 2011
Semi-homomorphic encryption
MAC authenticates share
Additive secret sharing
Precomputation
Simpler triple verification
Fully homomorphic encryption
Scheme by Brakerski and Vaikuntanathan 2011 [14]
SIMD by Smart and Vercauteren 2012 [54]
Somewhat Homomorphic Encryption
Precomputation, Additive sharing, Triple verification, MAC
SPDZ, [26] and [23]
Damgård, Pastro, Smart and Zakarias 2011
Precomputation
Vectorised SHE SIMD in precomputation
Threshold decryption
Public modifier in share
MAC authenticates secret
Additive secret sharing
Statistical security
Active adversaries
Adaptive adversary for online
Static adversary for precomputation

Figure 3.1: The development of SPDZ

Although precomputation is used to achieve efficient online computations, it may mean that the overall cost of the protocols increases. For example, it could be possible to use an expensive multiplication protocol M to precompute Beaver triples and then use the triples to do online multiplication in protocol O. In such case, we use computation time for both M and O, whereas, in theory, only the time of the precomputation M suffices for multiplication. However, dividing it to two parts allows for more efficient online phase. Thus, precomputation model allows us to gain online performance but may not reduce total workload. This model is usable if O is reasonably more efficient than M or if we can do precomputations without actually defining a multiplication protocol M.

Precomputation is meaningful in situations where the overall system has uneven workload so it can do precomputations in the background. Precomputations could be performed when the users are not active or in parallel with online computations. The latter is reasonable if it does not significantly reduce online performance. For example, we can consider a data analyst who likes to get fast results during the workday, but does not use the system outside common working hours. The latter indicating that the night-time can be easily used for precomputations.

The precomputation model assumes that the online phase always has sufficient precomputed values to proceed. However, it is not trivial to ensure this in practice. Therefore, it is important to consider the desired behaviour of online protocols when they can not retrieve all necessary precomputation products. One possibility is to signal the precomputing process and then compute the protocol incrementally as the precomputation results become available. The other case would be to define a separate slower protocol for the same functionality that does not require precomputations and use it instead. In addition, depletion of precomputation results can be avoided if the algorithms to be executed as well as the input sizes are well known beforehand.

The difficulty from dependence on the precomputation protocol speed indicates that this model is not well designed for all SMC use-cases. For example, the classical Millionaires' problem would have the best solution if the millionaires can set up the framework, insert their input and get the output at once. The alternative with the precomputation is unsatisfactory, as they may not wish to wait a while to allow the machine to perform all kinds of precomputations. In conclusion, the precomputation model is best suited for applications where the data is used for an extended time period.

3.2 Oblivious MAC

Oblivious MAC algorithms resemble threshold cryptosystems in a way that no party can check the MAC tag independently of others as no party can decrypt alone. Furthermore, the MAC tag value of a secret shared input will be stored as a secret shared value. For example, a secret value $[\![x]\!]$ might be protected using a MAC, where the tag is in turn kept in shares as $[\![z]\!]$. Together, these requirements indicate that the MAC key k must be a secret value. In addition, the MAC algorithm must have homomorphic properties to be able to compute the tag for the computation result from the tags of the inputs. The idea of producing MAC tags to unknown values originates from Rabin and Ben-Or as *Information Checking* for verifiable secret sharing [49].

The SPDZ framework uses unconditionally secure MAC to verify the correctness of shared value instead of the correctness of each share. The idea of checking the shared value and not each share results in less storage for MAC tags, but also means that we can not check the validity of the computations before declassifying the value. MAC key is shared using additive secret sharing together with meta-information so that all parties can verify the correctness of the key. Each party has a share of the value and a share of the tag on that value for each secretly shared element. An analogous algorithm to MAC from Section 2.1.9 is also used by SPDZ. All their arithmetic is in a finite field $\mathbb{F}_{p^k}$ for a prime p and integer k and thus, according to Theorem 2.1.5 the MAC is secure.

The security requirements from Section 2.1.9 apply also to SPDZ when extended to more than two parties. In addition, SPDZ proposes an efficiency improvement that either verifies that all the elements in a vector or none of them. The idea is to combine the MAC tags to reduce network communication when checking the tags. In this case, we do not exactly learn, which share was faulty, but in practice we only need to learn the fact that some party might be malicious. Besides, shares could be verified separately to sort out the false ones.

3.3 Beaver triples

Beaver triples are multiplicative triples $\langle a, b, c \rangle$ such that $c = a \cdot b$ proposed to simplify multiplication on secret shared inputs [4]. The initial idea was to randomize every input of an arithmetic circuit and evaluate the circuit on these random shared values to obtain $\hat{y} = f(r_1, \dots, r_k)$. Afterwards, the difference δ_x of the random input r_i and real input x_i is computed as $\delta_i = x_i - r_i$ and made public. The second time the circuit is evaluated using public differences and initial random inputs to find the difference δ_y for the output $y = f(x_1, \dots, x_k)$. The real output of the circuit is $y = \hat{y} + \delta_y$. The main question is correctly fixing the difference δ_y.

This idea is used in the following by the Multiplication protocol in Algorithm 3. For multiplication, the difference is computed as $\delta_y = \delta_1 \cdot r_2 + \delta_2 \cdot r_1 + \delta_1 \cdot \delta_2$, which follows trivially from the definition $y = (r_1 + \delta_1) \cdot (r_2 + \delta_2)$ as $x_i = r_i + \delta_i$ and $\hat{y} = r_1 \cdot r_2$.

The idea was proposed for secret sharing schemes that allow local addition and, actually, the randomization can be avoided during the addition step. However, computing multiplication results requires collaboration and computing the multiplication of the random inputs. Thus, Beaver triples are actually two random inputs a and b used to hide the protocol inputs and their multiplication $c = a \cdot b$ used together with public differences to restore the correct multiplication result.

Beaver triples are currently a common precomputation mechanism for SMC, as they can be computed before the inputs are known. The triples are used as helper values in multiplication according to the original proposal as shown in Multiplication protocol. Beaver triples were originally described for linearly shared secrets, but can easily be extended to shares with linear protection mechanisms such as the aforementioned MAC tags.

3.4 Basic protocols

The description of some SPDZ protocols is independent from the secret sharing method as long as the scheme defines protocols for publishing shares privately to each computing or result party, generating a random share, and generating random Beaver triples. There are three main protocols: (1) classifying the secret input, (2) opening the secret, and (3) multiplication of shared values.

Classifying and multiplication both require protocols to publish shares, which are dependant on the share representation. In general, we require three versions of the Publish protocol: (1) to declassify shares to all computing parties at the same time, (2) to publish to all computing parties separately, and (3) to declassify to non-computing parties. The declassification protocols are not specified here as they depend on the share description. However, the general idea is that a party receives information about the secret from others and verifies its correctness.

We assume that the share representation enables a local addition operation, meaning that to obtain $[\![x]\!] + [\![y]\!]$ all computing parties $\mathcal{CP}$ only need to compute on their own shares. In addition, this implies local operations for subtraction and multiplication with a public value. The multiplication Algorithm 3 assumes the existence of precomputed and verified Beaver triples and is directly based on Beaver's ideas [4]. It is derived from triples and local computations together with publishing a value, combining those to obtain $[\![xy]\!]$ from $[\![x]\!]$ and $[\![y]\!]$.

Algorithm 3 Multiplying two secret values (Multiplication)

```
Data: Shared secrets $[\![x]\!]$ and $[\![y]\!]$
Result: Shared result $[\![w]\!]$, where $w = x \cdot y$
1: $\mathcal{CP}$ collaboratively choose a triple $[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$, where $c = a \cdot b$
2: $\mathcal{CP}$ compute $[\![e]\!] = [\![x]\!] - [\![a]\!]$ and $[\![d]\!] = [\![y]\!] - [\![b]\!]$
3: $\mathcal{CP}$ collaboratively open $[\![e]\!]$ and $[\![d]\!]$ to all $\mathcal{CP}$
4: $\mathcal{CP}$ compute $[\![w]\!] = [\![c]\!] + e \cdot [\![b]\!] + d \cdot [\![a]\!] + e \cdot d$
5: return $[\![w]\!]$
```

The classifying protocol in Algorithm 4 enables input and computing parties to share their secret value among all computing parties. This is a straightforward extension of the circuit randomization idea from Beaver [4]. It assumes the existence of precomputation that produces random shared values to all parties.

Algorithm 4 Classifying a private input Classify-$\mathcal{IP}_i$

```
Data: Input party $\mathcal{IP}_i$ has a secret $x$
Result: Computing parties $\mathcal{CP}$ have $[\![x]\!]$
1: $\mathcal{CP}$ collaboratively choose a precomputed randomness $[\![r]\!]$
2: $\mathcal{CP}$ open $[\![r]\!]$ to $\mathcal{IP}_i$
3: $\mathcal{IP}_i$ computes $e = x - r$ and sends $e$ to $\mathcal{CP}$
4: $\mathcal{CP}$ compute $[\![x]\!] = [\![r]\!] + e$
5: return $[\![x]\!]$
```

These online protocols are claimed to be statically secure against an adaptive active adversary, if we have an ideal precomputation phase. However, only the case of static adversaries is proved as only this can be achieved by the precomputation protocols [26]. The adversary is allowed to corrupt at most $n - 1$ parties out of n.

Although the precomputation phase is not defined here due to dependencies on the share representation, we can still define one important step to verify the correctness of multiplicative triples. This ensures that the triple really has multiplicative relation. The triple verification process in Algorithm 5 takes two multiplicative triples and performs computations analogously to multiplication. For a finite field $\mathbb{Z}_p$ where p is prime, the probability of cheating in the verification is $\frac{1}{p}$ assuming ideal opening phase [25].

The behaviour of these protocols somewhat depends on either working with an honest minority or majority. Everything is the same in case all the protocols succeed—everyone communicates and all checks in the opening phase succeed. However, the difference comes when something fails. For example, if a two-party protocol fails then none of the parties can continue and they also can not restore the secrets. However,

Algorithm 5 Verifying the correctness of multiplicative triples

```
Data: Secret shared random triple $[\![x]\!]$, $[\![y]\!]$, $[\![w]\!]$
Result: $True$ if $w = x \cdot y$, $False$ in case any check fails
1: $\mathcal{CP}$ collaboratively choose a random value $[\![r]\!]$ and open it to all parties
2: $\mathcal{CP}$ collaboratively choose a triple $[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$, where presumably $c = a \cdot b$
3: $\mathcal{CP}$ compute $[\![e]\!] = r \cdot [\![x]\!] - [\![a]\!]$ and $[\![d]\!] = [\![y]\!] - [\![b]\!]$
4: $\mathcal{CP}$ open $[\![e]\!]$ and $[\![d]\!]$ to all $\mathcal{CP}$
5: $\mathcal{CP}$ compute $[\![h]\!] = r \cdot [\![w]\!] - [\![c]\!] - e \cdot [\![b]\!] - d \cdot [\![a]\!] - e \cdot d$
6: $\mathcal{CP}$ open $[\![h]\!]$ to all $\mathcal{CP}$
7: return $h == 0$
```

for (t, n)-threshold scheme a set of t honest parties could point out the malicious participants and continue the computations without them.

3.5 Initialising actively secure two-party computation

Frameworks analogous to SPDZ can be used with two computing parties and could have three considerably different initial setups. The main difference between them is how the MAC keys are defined and who knows them. In all cases, we require a homomorphic MAC. In addition, we would like the secret sharing method and homomorphic MAC to have the same operations that can be computed locally. Two parties are denoted by $\mathcal{CP}_1$ and $\mathcal{CP}_2$.

3.5.1 Asymmetric setup

Asymmetric setup differentiates the computing parties so that one gets the role of a master node ($\mathcal{CP}_1$) who defines the MAC key and the client ($\mathcal{CP}_2$) is using the keys from the master. Using the MAC to either authenticate the secret value or the share of the other party enables $\mathcal{CP}_1$ to easily verify the correctness of the declassification result. However, $\mathcal{CP}_2$ is unable to verify the MAC as it must not know the MAC secret key. It is up to the master to also define something that $\mathcal{CP}_2$ can check.

For example, the master can publish a homomorphic commitment to its input shares. That way the homomorphic properties of the commitment enable $\mathcal{CP}_2$ to compute valid commitments to all computation results and validate the declassification result. In addition, the master node must have a way to compute the openings for all commitments derived during the computation.

The MAC tag for the whole value or the share of $\mathcal{CP}_2$ can not be kept by the master node. MAC algorithms are not designed to protect the privacy of the message, thus, seeing the whole tag might leak the secret to the master node who also knows the MAC secret key. In addition, storing it on the side of $\mathcal{CP}_2$ might also leak some information about the secret or the key. Hence, for best security we need to store the tag z in a secret shared manner as $[\![z]\!]$ and both parties must be able to update their parts of the tags during computation. We can use the MAC from Section 2.1.9 where the key k is defined by $\mathcal{CP}_1$.

The used commitment has to be binding so that the client node can believe that it received a correct share in the opening phase. In addition, the hiding property of the commitment ensures the privacy of secret information in all phases but the declassifying. A complete initialization of a protocol set with asymmetric setup is described in Chapter 4.

3.5.2 Symmetric setup

A symmetric setup means that both computing parties define similar parameters. A direct continuation of the previous asymmetric setting would be that both parties $\mathcal{CP}_i$ in the symmetric setting define their own MAC keys k_i. This would mean that on top of the secret sharing method we have two MAC tags $z^{(1)}$, $z^{(2)}$ where both parties can verify one of them during the declassification phase. As in the asymmetric, case we need a to keep the tags in shares $[\![z^{(1)}]\!]$ and $[\![z^{(2)}]\!]$ to avoid revealing the secrets. For example, we can use the MAC from Section 2.1.9 where both parties define their own key.

The main benefit of this setup over the asymmetric one is that the protocol descriptions would also become symmetric. This simplifies the notation and also means that the parties can do exactly the same workload in parallel. In some sense, this enables us to gain more efficient time usage. More precisely, it is unlikely to have protocols where one party has to wait between sending and receiving network message without having any computations to perform. Furthermore, we can only use the cheap MAC algorithm and do not have a need for more expensive homomorphic commitments that we used in the asymmetric case.

Our specification of a protocol set with symmetric setup can be found in Chapter 6. Symmetric setup with MACs was also used by BDOZ [7]. A setup with using only commitments to the secrets was introduced in [25].

3.5.3 Shared key setup

The shared key model is a further extension changing the symmetric setup so that instead of both parties defining a key they share one key $[\![k]\!]_*$ between them. This defines a threshold MAC algorithm where all parties must participate in the verification of the tag. It can give additional efficiency gains as now the parties only have to update a single tag $[\![z]\!]$ during the computations. However, the sharing $[\![k]\!]_*$ is special as it has to define some additional information, allowing parties to verify the correctness of the restored key and checked tags. The shared key setup is the approach currently used by the SPDZ framework.

However, there are well-known difficulties with this approach as the knowledge of the secret key is usually needed to verify the MAC tags. One possible solution is to not verify any opened results before all computation is done. Afterwards, it is possible to restore the MAC key and verify all the results at once. However, there are drawbacks because parties can only notice cheating very late and they must agree on a new key before next computations. In addition, changing the key means that after verifying the correctness of opened values, the shares of the outputs or intermediate results from the checked computations can not be reused.

The first version of SPDZ used the previous approach but they substituted it to a way to collaboratively check the MAC without revealing the keys [23]. The idea is that if the secret value is made public and the tag is a linear combination of this public value and the MAC key then it can be checked by computing on the shares and publishing only the verification result.

Chapter 4

Asymmetric two-party computation

This section introduces our initialisation of an asymmetric two-party secure computation scheme. It includes the share representation as well as protocols specific to this representation, including precomputation.

4.1 Protection domain setup

We consider an additive secret sharing scheme in $\mathbb{Z}_N$ where N is a Paillier modulus and we have a Paillier keypair (pk, sk) corresponding to this modulus. The party $\mathcal{CP}_1$ knows this keypair, while $\mathcal{CP}_2$ only knows the public key pk. In addition, $\mathcal{CP}_2$ must be convinced that it is a valid key. $\mathcal{CP}_1$ uses $\mathsf{Enc}_{pk}(x)$ to commit to a value x and stores the encryption randomness as the decommitment. $\mathcal{CP}_1$ can also define a key $k \leftarrow \mathbb{Z}_N$ for message authentication together with a commitment $\mathsf{Enc}_{pk}(k)$, which is also known by $\mathcal{CP}_2$. To distinguish a commitment from encrypting, we denote $(\!\![k]\!\!)_{pk} = \mathsf{Enc}_{pk}(k, r_k)$ which is a fixed value depending on the randomness r_k that $\mathcal{CP}_1$ chose when initially encrypting it. We use the same notation to represent encryptions that $\mathcal{CP}_1$ has published as commitments during the computation. These encryption and MAC keys must be usable throughout the life of the shares computed with them.

Each secret value x is is represented by a tuple

$$[\![x]\!]_N = \langle \Delta, x_1, x_2, r, (\!\![x_1]\!\!)_{pk}, z_1, z_2 \rangle$$

such that $x = x_1 + x_2 + \Delta$ and $z_1 + z_2 = k \cdot (x_1 + x_2)$. The values Δ and $(\!\![x_1]\!\!)_{pk} = \mathsf{Enc}_{pk}(x_1, r)$ are public whereas $\mathcal{CP}_i$ has private values z_i and x_i. The public modifier Δ is always 0 for random values and is used to enable fast addition of a share and public constant. Value r is kept by $\mathcal{CP}_1$ to open the commitment to $(\!\![x_1]\!\!)_{pk}$ of share $[\![x]\!]_N$. This randomness also enables us to write protocols so that actually only $\mathcal{CP}_2$ computes $(\!\![x_1]\!\!)_{pk}$ and $\mathcal{CP}_1$ recomputes the encryption if needed. This is a reasonable step because, in reality, $\mathcal{CP}_1$ only needs the encryption result during the zero-knowledge proofs in the precomputation and avoiding computation on ciphertexts enables faster online computation. We sometimes use labels as $z_1^{(x)}$ and $\Delta^{(x)}$ to denote that these part of the share representation $[\![x]\!]$. For security, we need to rely on the security of MAC as showed in Theorem 2.1.6 and security of the commitment shown by Theorem 4.1.1.

Theorem 4.1.1. *The commitment scheme based on (t, ε)-IND-CPA secure cryptosystem where the commitment $c = \mathsf{Enc}_{pk}(m, r)$ is the encryption and opening $d = (m, r)$*

is the message together with the encryption randomness is (t, ε)-hiding and unconditionally binding if the public key of the cryptosystem is publicly verifiable or proved to be valid using a zero-knowledge proof.

Proof Sketch. The perfect binding property follows from the fact that public key uniquely fixes the secret key and, hence, it is possible to decrypt the commitment.

The hiding property follows from the definition of the IND-CPA security of a cryptosystem. □

Furthermore, addition is a local operation as we can just sum the additive share elements pairwise and use the homomorphic properties of Paillier cryptosystem. In addition, the existence of an addition protocol (Addition) also defines a subtraction protocol (Subtraction) and a protocol for multiplying the shared value with a public constant (Constant Multiplication). Moreover, adding a public value to the shared secret (Constant Addition) only requires modifying the value Δ. In a way, public value v can also be thought of as having a fixed share representation

$$[\![v]\!]_N = \langle \Delta = v, v_1 = 0, v_2 = 0, r = 1, (\![v_1]\!)_{pk} = 1, z_1 = 0, z_2 = 0 \rangle \ .$$

Every time when $\mathcal{CP}_2$ receives a ciphertext and uses it to compute a response to $\mathcal{CP}_1$, it has to verify that it is valid. For the Paillier cryptosystem, the validity means that the ciphertext c belongs to $\mathbb{Z}^*_{N^2}$. Checking that $c \in \mathbb{Z}^*_{N^2}$ is equivalent to checking that $\gcd(c, N) = 1$. However, it is actually unlikely for either party to send invalid ciphertexts. If $\mathcal{CP}_2$ sends an invalid ciphertext, it means that $\mathcal{CP}_2$ can actually factor N and break the security of this setup, thus, it is as likely as factoring. On the other hand, if $\mathcal{CP}_1$ sends an invalid ciphertext then it deliberately leaks its secret key to $\mathcal{CP}_2$.

Computing parties must also be able to communicate with result parties $\mathcal{RP}_i$ and input parties $\mathcal{IP}_i$. We need to enable the computing parties to receive inputs from input parties and to make sure that results parties can learn the correct outputs of the protocol. Input and result parties know the Paillier public key N and have received the commitment $(\![k]\!)_{pk}$ of the MAC key, thus they are in a similar role to $\mathcal{CP}_2$.

In the security proofs of this section, the computational security of the protocols results from the computational security of the Paillier cryptosystem. Therefore, by choosing suitable keys, we can make the protocols as secure as we require. However, we specially stress the probability $\frac{1}{p}$ that a party can cheat against the MAC (Theorem 2.1.6), as for some cases, picking securer keys may not mean that also this probability $\frac{1}{p}$ lessens. Therefore, it could be seen as a fixed value rather than a value that we can make arbitrarily negligible. However, when using the Paillier cryptosystem, increasing the modulus would also increase the value of p.

We assume the existence of secure authenticated communication channels and exclude eavesdropping and modification of network messages from the security analysis. In practice, secure network channels are achieved using standard secure channel implementations like TLS [27].

4.2 Publishing shared values

The secret value may either be made public to $\mathcal{CP}_1$, $\mathcal{CP}_2$ or $\mathcal{RP}_i$ and we need to have different protocols for these cases. We can use a combination of the first two to publish the value to both computing participants at the same time (Publish-both-$\mathcal{CP}_i$).

Clearly, sending the corresponding share to other party is a similar step in all of these protocols. However, the mechanisms for verifying the correctness of the given share value are different.

Algorithm 6 Publishing a shared value to $\mathcal{CP}_1$ ($\mathsf{Publish}\text{-}\mathcal{CP}_1$)

```
Data: Shared secret $[\![x]\!]_N$
Result: $\mathcal{CP}_1$ learns the value $x$
1: $\mathcal{CP}_2$ sends $x_2$ and $z_2$ to $\mathcal{CP}_1$
2: $\mathcal{CP}_1$ verifies $z_1 + z_2 = k \cdot (x_1 + x_2)$
3: return $\mathcal{CP}_1$ outputs $x_1 + x_2 + \Delta$
```

It is easy to see that $\mathcal{CP}_1$ can perform the verification in $\mathsf{Publish}\text{-}\mathcal{CP}_1$ (Algorithm 6) because $\mathcal{CP}_1$ knows all the plain values in the verification equation. The security of the MAC algorithm ensures that $\mathcal{CP}_2$ is unlikely to pass the verification when submitting a share or a tag not obtained from correct computations.

As the value $(\![x_1]\!)_{pk}$ is a commitment to value x_1 from $\mathcal{CP}_1$ then $\mathcal{CP}_2$ can verify the correctness of received x_1 in $\mathsf{Publish}\text{-}\mathcal{CP}_2$ (Algorithm 7) by successfully opening the commitment. The perfectly binding property of the commitment ensures that $\mathcal{CP}_1$ can only pass the verification with the unique correct share and randomness pair.

Algorithm 7 Publishing a shared value to $\mathcal{CP}_2$ ($\mathsf{Publish}\text{-}\mathcal{CP}_2$)

```
Data: Shared secret $[\![x]\!]_N$
Result: $\mathcal{CP}_2$ learns the value $x$
1: $\mathcal{CP}_1$ sends $x_1$ and $r$ to $\mathcal{CP}_2$
2: $\mathcal{CP}_2$ verifies $(\![x_1]\!)_{pk} = \mathsf{Enc}_{pk}(x_1, r)$
3: return $\mathcal{CP}_2$ outputs $x_1 + x_2 + \Delta$
```

In the following, we give a full proof for the security of $\mathsf{Publish}\text{-}\mathcal{CP}_i$ protocol, later in the thesis we give a more brief overview about the ideal world and simulator for the security proofs.

Theorem 4.2.1. *Algorithms* $\mathsf{Publish}\text{-}\mathcal{CP}_1$ *and* $\mathsf{Publish}\text{-}\mathcal{CP}_2$ *for publishing the value to one computing party are correct.* $\mathsf{Publish}\text{-}\mathcal{CP}_1$ *is computationally secure against cheating* $\mathcal{CP}_2$ *with an additional statistical* $\frac{1}{p}$*-error probability, where* p *is the smaller prime factor of* N *and computationally secure against cheating* $\mathcal{CP}_1$*. Protocol* $\mathsf{Publish}\text{-}\mathcal{CP}_2$ *is perfectly secure against a cheating* $\mathcal{CP}_1$ *and computationally secure against a cheating* $\mathcal{CP}_2$*.*

Proof sketch. For correctness, we need that $x = x_1 + x_2 + \Delta$ or the protocol aborts. The former is trivially true by the definition of the share representation and, in case of honest participants, the verification always succeeds. In the following, we show the security in the stand-alone setting.

We describe the ideal secure execution of this protocol using the model with a trusted third party (TTP) who always behaves honestly. The ideal functionality of publishing to $\mathcal{CP}_i$ is such that the TTP notifies $\mathcal{CP}_j$ that it is about to declassify x. On input *Continue* it sends x and Δ to $\mathcal{CP}_i$ and on input *Abort* it cancels the publishing. The real setup where the publish protocol is executed is such that the parties have executed some protocols and received the output x and correction value Δ and then $\mathcal{CP}_j$ sends the declassification values to $\mathcal{CP}_i$. However, the previous protocol runs are secure and can be replaced by a TTP, who gives the shares of x to the computing parties. These two execution models are illustrated on Figure 4.1 for Publish-$\mathcal{CP}_1$, the case for Publish-$\mathcal{CP}_2$ is analogous.

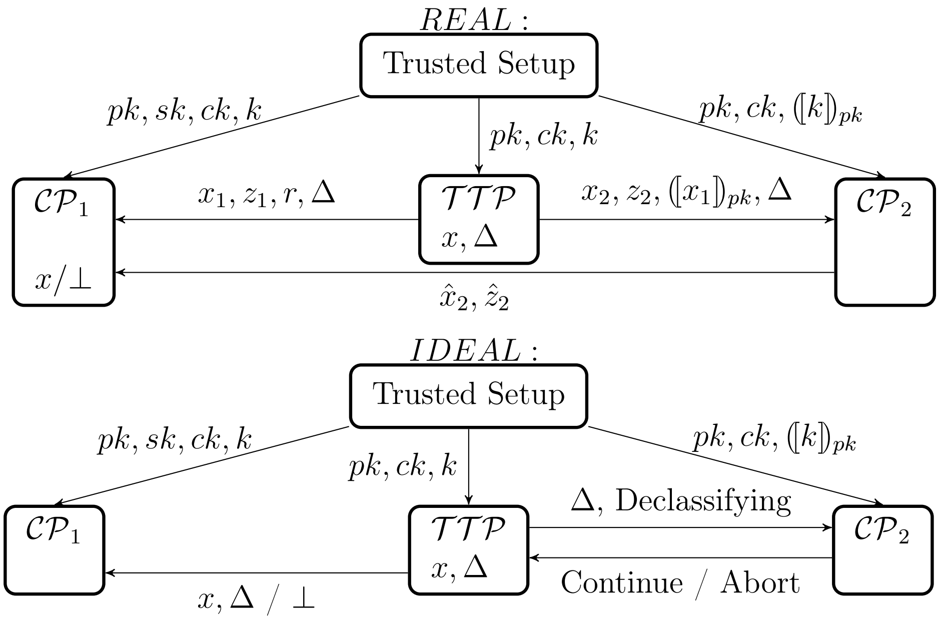


Figure 4.1: Ideal and real protocol execution of Publish-$\mathcal{CP}_1$

Therefore, for both protocols, we need to show a simulator, such that the output distributions of the simulated adversary and the $\mathcal{CP}_i$ of the ideal world coincide with the outputs of the adversary and $\mathcal{CP}_i$ in the real world.

Firstly, consider a corrupted $\mathcal{CP}_2$ in Publish-$\mathcal{CP}_2$. The corresponding simulator at first receives x and Δ from TTP if the ideal publishing succeeds and can easily create suitable values $x_1, x_2, z_2, ([x_1])_{pk} = \mathsf{Enc}_{pk}(x_1, r), r$, where $x_1 + x_2 = x$. The simulator can forward these to the corrupted $\mathcal{CP}_2$ as two messages in the real protocol execution. The outputs clearly coincide as the corrupted $\mathcal{CP}_2$ always gets the same x and honest $\mathcal{CP}_1$ has seen the same Δ.

Analogously, the simulator for corrupted $\mathcal{CP}_1$ in Publish-$\mathcal{CP}_1$ is straightforward. It receives x and Δ from TTP and can prepare $x_1, x_2, z_2, ([x_1])_{pk} = \mathsf{Enc}_{pk}(x_1, r), r$, where $x_1 + x_2 = x$. However, it can not fix a correct z_1, but it can compute $\mathsf{Enc}_{pk}(z_1) = \mathsf{Enc}_{pk}(k)^x \cdot \mathsf{Enc}_{pk}(-z_2)$. Therefore, it can simulate the protocol for the equivalent of $\mathcal{CP}_1$, who expects z_1 in an encrypted form.

In the following, for publishing to $\mathcal{CP}_i$ we only consider the case where the other party $\mathcal{CP}_j$ is corrupted. We assume that there has been a setup phase beforehand, where all parties and the simulator learned $([k])_{pk}$ and N. In addition, $\mathcal{CP}_1$ learned the private key for the Paillier cryptosystem and the TTP also has k. The setup is shared between protocol runs and is not a part of this protocol execution.

Publishing to $\mathcal{CP}_1$. The simulator $\mathcal{S}$ at first picks a MAC key $k_{\mathcal{S}}$ for itself. It then generates the share $x_2, z_2, ([x_1^*])_{pk}$ to the adversary $\mathcal{A}$ who responds with $\hat{x}_2, \hat{z}_2$. The simulator either finishes with *Abort* or *Continue* where the output Ψ_1 of $\mathcal{CP}_1$ will be $\perp$ in the former and x in the latter case as given by the ideal world TTP. The general game for this is specified as $G_1^{\mathcal{A}}(x)$ and $G_2^{\mathcal{A}}(x)$ rewrites this with the specific details of the simulator.

The corresponding real protocol execution can be seen from $G_3^{\mathcal{A}}(x)$ with the exact details of the honest $\mathcal{CP}_1$ and TTP in $G_4^{\mathcal{A}}(x)$. It can be seen that the games $G_2^{\mathcal{A}}(x)$

```
$G_1^{\mathcal{A}}(x, \Delta)$
    $sk, pk, k, ck, (\![k]\!)_{pk} \leftarrow$ Setup
    $\Delta, x_2, z_2, c^* \leftarrow \mathcal{S}(pk, (\![k]\!)_{pk})$
    $\hat{x}_2, \hat{z}_2 \leftarrow \mathcal{A}(\Delta, x_2, z_2, c^*)$
    $\Psi_2 \leftarrow \mathcal{A}$
    $b \leftarrow \mathcal{S}(\hat{x}_2, \hat{z}_2)$
    **if** $b = Continue$
        **then** $\Psi_1 = x$
        **else** $\Psi_1 = \perp$
    **return** $(\Psi_1, \Psi_2)$
```

```
$G_2^{\mathcal{A}}(x, \Delta)$
    $sk, pk, k, ck, (\![k]\!)_{pk} \leftarrow$ Setup
    $\Delta \leftarrow \mathcal{TTP}(pk, ck, k)$
    $x_2, z_2, x_1^*, k_{\mathcal{S}} \leftarrow \mathbb{Z}_N$
    $z_1 = k_{\mathcal{S}} \cdot (x_1^* + x_2) - z_2$
    $c^* \leftarrow \mathsf{Enc}_{pk}(x_1^*)$
    $\hat{x}_2, \hat{z}_2 \leftarrow \mathcal{A}(\Delta, x_2, z_2, c^*)$
    $\Psi_2 \leftarrow \mathcal{A}$
    **if** $k_{\mathcal{S}} \cdot (x_1^* + \hat{x}_2) = z_1 + \hat{z}_2$
        **then** $\Psi_1 = x$
        **else** $\Psi_1 = \perp$
    **return** $(\Psi_1, \Psi_2)$
```

Security game 4.2.1: Publishing to $\mathcal{CP}_1$ with simulator

and $G_4^{\mathcal{A}}(x)$ are almost equivalent, except for the usage of a different key, different commitments $(\![x_1]\!)_{pk}$, $(\![x_1^*]\!)_{pk}$ and some additional computations in $G_4^{\mathcal{A}}(x)$. By IND-CPA security we know that $(\![x_1]\!)_{pk}$ and $(\![x_1^*]\!)_{pk}$ are computationally indistinguishable. We know that starting $\mathcal{A}$ with the same randomness ϕ_2 will always result in the same output Ψ_2.

As the simulator does not know the real key k, it may falsely accept when the real protocol run rejects the inputs. However, according to the MAC security, the simulator falsely accepts with probability less than $\frac{1}{p}$ which is the same as in the real protocol run. From the IND-CPA security of the cryptosystem we know that $(\![k]\!)_{pk}$ hides k, therefore using $k_{\mathcal{S}}$ instead of k is computationally indistinguishable. However, in the ideal protocol run we know that $\mathcal{CP}_1$ always gets the correct output x, but due to the possible cheating in the MAC algorithm there is a $\frac{1}{p}$ possibility that the real protocol run finishes with $\hat{x}$. Therefore, the outputs of the ideal and real world coincide except with probability $\frac{1}{p}$.

```
$G_3^{\mathcal{A}}(x, \Delta)$
    $sk, pk, k, ck, (\![k]\!)_{pk} \leftarrow$ Setup
    $\Delta, x_1, z_1, r, x_2, z_2, c \leftarrow \mathcal{TTP}(x, \Delta, pk, ck, k)$
    $\mathcal{CP}_1(\Delta, x_1, z_1, r)$
    $\hat{x}_2, \hat{z}_2 \leftarrow \mathcal{A}(\Delta, x_2, z_2, c)$
    $\Psi_2 \leftarrow \mathcal{A}$
    $\Psi_1 \leftarrow \mathcal{CP}_1(\hat{x}_2, \hat{z}_2)$
    **return** $(\Psi_1, \Psi_2)$
```

```
$G_4^{\mathcal{A}}(x, \Delta)$
    $sk, pk, k, ck, (\![k]\!)_{pk} \leftarrow$ Setup
    $x_1, z_1 \leftarrow \mathbb{Z}_N, r \leftarrow \mathbb{Z}_N^*$
    $c \leftarrow \mathsf{Enc}_{pk}(x_1, r)$
    $x_2 = x - x_1 - \Delta$
    $z_2 = k \cdot x - z_1$
    $\hat{x}_2, \hat{z}_2 \leftarrow \mathcal{A}(\Delta, x_2, z_2, c)$
    $\Psi_2 \leftarrow \mathcal{A}$
    **if** $k \cdot (x_1 + \hat{x}_2) = z_1 + \hat{z}_2$
        **then** $\Psi_1 = x$
        **else** $\Psi_1 = \perp$
    **return** $(\Psi_1, \Psi_2)$
```

Security game 4.2.2: Publishing to $\mathcal{CP}_1$ in real protocol run

Publishing to $\mathcal{CP}_2$. The simulator $\mathcal{S}$ picks the shares that $\mathcal{CP}_1$ should receive and sends them to $\mathcal{A}$ to simulate the TTP in the real protocol execution. Then $\mathcal{A}$ sends the declassification message $\hat{x}_1, \hat{r}$. The simulator outputs *Continue* in case $\mathsf{Enc}_{pk}(x_1, r) = \mathsf{Enc}_{pk}(\hat{x}_1, \hat{r})$ which is the same check that an honest $\mathcal{CP}_2$ would do to check the commitment. The simulated protocol run can be seen in the game $G_5^{\mathcal{A}}(x)$ and the simulator specifics have been written out in $G_6^{\mathcal{A}}(x)$.

```
$G_5^{\mathcal{A}}(x, \Delta)$
    $sk, pk, k, ck, [\![k]\!]_{pk} \leftarrow$ Setup
    $\Delta, x_1, z_1, r \leftarrow \mathcal{S}(pk, [\![k]\!]_{pk})$
    $\hat{x}_1, \hat{r} \leftarrow \mathcal{A}(\Delta, x_1, z_1, r)$
    $\Psi_1 \leftarrow \mathcal{A}$
    $b \leftarrow \mathcal{S}(\hat{x}_1, \hat{r})$
    if $b = Continue$
        then $\Psi_2 = x$
        else $\Psi_2 = \perp$
    return $(\Psi_1, \Psi_2)$
```

```
$G_6^{\mathcal{A}}(x, \Delta)$
    $sk, pk, k, ck, [\![k]\!]_{pk} \leftarrow$ Setup
    $\Delta \leftarrow \mathcal{TTP}(x, \Delta)$
    $x_1, z_1 \leftarrow \mathbb{Z}_N$
    $r \leftarrow \mathbb{Z}_N^*$
    $\hat{x}_1, \hat{r} \leftarrow \mathcal{A}(\Delta, x_1, z_1, r)$
    $\Psi_1 \leftarrow \mathcal{A}$
    if $\mathsf{Enc}_{pk}(\hat{x}_1, \hat{r}) = \mathsf{Enc}_{pk}(x_1, r)$
        then $\Psi_2 = x$
        else $\Psi_2 = \perp$
    return $(\Psi_1, \Psi_2)$
```

Security game 4.2.3: Publishing to $\mathcal{CP}_2$ with simulator

An analogous game of the real protocol run with the TTP representing the previous computations and a real honest $\mathcal{CP}_2$ is shown in $G_7^{\mathcal{A}}(x)$. Finally, the game $G_8^{\mathcal{A}}(x)$ shows the real execution with the exact workings of TTP and $\mathcal{CP}_2$.

```
$G_7^{\mathcal{A}}(x, \Delta)$
    $sk, pk, k, ck, [\![k]\!]_{pk} \leftarrow$ Setup
    $x_1, z_1, r, x_2, z_2, c \leftarrow \mathcal{TTP}(x, \Delta)$
    $\mathcal{CP}_2(\Delta, x_2, z_2, c)$
    $\hat{x}_1, \hat{r} \leftarrow \mathcal{A}(\Delta, x_1, z_1, r)$
    $\Psi_1 \leftarrow \mathcal{A}$
    $\Psi_2 \leftarrow \mathcal{CP}_2(\hat{x}_1, \hat{r})$
    return $(\Psi_1, \Psi_2)$
```

```
$G_8^{\mathcal{A}}(x, \Delta)$
    $sk, pk, k, ck, [\![k]\!]_{pk} \leftarrow$ Setup
    $x_1, z_1 \leftarrow \mathbb{Z}_N$
    $r \leftarrow \mathbb{Z}_N^*$
    $c \leftarrow \mathsf{Enc}_{pk}(x_1, r)$
    $x_2 = x - x_1 - \Delta$
    $z_2 = k \cdot x - z_1$
    $\hat{x}_1, \hat{r} \leftarrow \mathcal{A}(\Delta, x_1, z_1, r)$
    $\Psi_1 \leftarrow \mathcal{A}$
    if $\mathsf{Enc}_{pk}(\hat{x}_1, \hat{r}) = c$
        then $\Psi_2 = x$
        else $\Psi_2 = \perp$
    return $(\Psi_1, \Psi_2)$
```

Security game 4.2.4: Publishing to $\mathcal{CP}_2$ in real protocol run

We can see that besides some additional computations, the games and outputs of $G_6^{\mathcal{A}}(x)$ and $G_8^{\mathcal{A}}(x)$ coincide and the simulation is perfect. The simulator always accepts the same cases as the real protocol run because the commitment is perfectly binding. Therefore, the outputs of the real and ideal world coincide. $\square$

There are two special cases when result parties $\mathcal{RP}_i$ may need to learn the computation outcomes. In one scenario, the computing parties $\mathcal{CP}_i$ are allowed to also learn the outcome, whereas in the other case, only the output party $\mathcal{RP}_i$ can learn the declassification result.

It is straightforward to satisfy the first case. The computing parties just run the declassification protocol Publish-both-$\mathcal{CP}_i$ and they both forward the declassified result to $\mathcal{RP}_i$. The result party only has to verify that both computing parties sent the same declassified result. Differently from either Publish-$\mathcal{CP}_i$, in Publish-$\mathcal{CP}$&$\mathcal{RP}_i$, the result party $\mathcal{RP}_i$ can not easily check which of the computing parties has tried to cheat, if the verification does not succeed.

Theorem 4.2.2. *Publishing values to both computing parties and to result parties* ($\mathsf{Publish}\text{-}\mathcal{CP}\&\mathcal{RP}_i$) *is correct and as secure against corrupted $\mathcal{CP}_j$ as* $\mathsf{Publish}\text{-}\mathcal{CP}_i$ *and perfectly secure against a corrupted $\mathcal{RP}_i$.*

Proof sketch. These properties result from the correctness and security of publishing to either of the computing parties in $\mathsf{Publish}\text{-}\mathcal{CP}_i$. The fact that the result party verifies that both computing parties sent the same same result also detects cheating after finishing the predefined publishing protocols.

It is trivial to simulate the protocol run for a corrupted $\mathcal{RP}_i$ as the simulator can just forward the value x from the TTP to $\mathcal{RP}_i$. □

The second case requires more work on the side of the result party as given in $\mathsf{Publish}\text{-}\mathcal{RP}_i$ (Algorithm 8). The idea is that the result party $\mathcal{RP}_i$ can verify the commitment similarly to $\mathcal{CP}_2$, but it can not verify the MAC tag as it does not know the secret key k. However, $\mathcal{RP}_i$ can verify that it has the right share using the proof of correct share representation similarly to the $\mathsf{Singles}$ protocol. The main drawback of this protocol is that the corresponding zero-knowledge proof protocol, that we instantiate with a version of CdsZkMul protocol ($\mathsf{CdsZKTags}$), can be quite expensive. However, in real life we would like to avoid heavy computational needs on the side of the input and result parties in order to make this usable in a variety of different settings, including, for example, those where the input and result parties are using mobile devices.

Algorithm 8 Publishing a shared value to $\mathcal{RP}_i$ ($\mathsf{Publish}\text{-}\mathcal{RP}_i$)

```
Data: Shared secret $[\![x]\!]_N$
Result: Result party $\mathcal{RP}_i$ learns the value $x$
1: $\mathcal{CP}_1$ sends $\Delta$, $x_1$ and $r$ to $\mathcal{RP}_i$
2: $\mathcal{CP}_2$ sends $\Delta$, $x_2$, $z_2$, and $(\![x_1]\!)_{pk}$ to $\mathcal{RP}_i$
3: $\mathcal{RP}_i$ verifies that $\mathcal{CP}_1$ and $\mathcal{CP}_2$ sent the same $\Delta$
4: $\mathcal{RP}_i$ verifies that $(\![x_1]\!)_{pk} = \mathsf{Enc}_{pk}(x_1, r)$
5: $\mathcal{CP}_1$ proves $z_1 + z_2 = k \cdot (x_1 + x_2)$ to $\mathcal{RP}_i$ using CdsZKTags
6: return $\mathcal{RP}_i$ computes $x = x_1 + x_2 + \Delta$
```

Intuitively, the protocol is secure if neither party can make $\mathcal{RP}_i$ accept a faulty x.

Theorem 4.2.3. *Algorithm* $\mathsf{Publish}\text{-}\mathcal{RP}_i$ *for declassifying shared secrets to result parties is correct. Protocol* $\mathsf{Publish}\text{-}\mathcal{RP}_i$ *is computationally secure against corrupted $\mathcal{CP}_2$ with possible $\frac{1}{p}$ error probability, where p is the smaller prime factor of N. Protocol* $\mathsf{Publish}\text{-}\mathcal{RP}_i$ *is perfectly secure against a corrupted $\mathcal{CP}_1$. It is also computationally secure against malicious $\mathcal{RP}_i$, assuming a simulatable* $\mathsf{CdsZKTags}$ *protocol.*

Proof sketch. The correctness means that in case of honest participants, $\mathcal{RP}_i$ receives the result x. It follows trivially from the share description and correctness of the used ZK proof.

For security in the stand-alone setting, we are interested in the cases where either of the parties is corrupted alone. We show the simulator construction for these cases. The ideal model and real world execution scenarios are analogous to those of the $\mathsf{Publish}\text{-}\mathcal{CP}_i$, except that $\mathcal{RP}_i$ is supposed to learn the final outcome.

The simulation in both $\mathcal{CP}_j$ cases begins by using the corresponding simulator from $\mathsf{Publish}\text{-}\mathcal{CP}_i$ where $\mathcal{CP}_j$ is corrupted. It at first acts on behalf of the previous protocols and gives the shares of a secret x^* to the corrupted $\mathcal{CP}_j$. The $\mathcal{CP}_j$ has to release the same values as in Publish-$\mathcal{CP}_i$ with the additional values Δ on both sides and $(\!\{x_1\}\!)_{pk}$ from $\mathcal{CP}_2$. The simulator checks the correctness of these by either opening the commitment $(\!\{x_1\}\!)_{pk}$ or verifying the tags with respect to its own key as in Publish-$\mathcal{CP}_i$.

In addition, the simulator of corrupted $\mathcal{CP}_2$ verifies the correctness of $(\!\{x_1\}\!)_{pk}$ as an honest $\mathcal{RP}_i$ would by checking the opening $(\!\{x_1\}\!)_{pk} = \mathsf{Enc}_{pk}(x_1, r)$. Finally, the simulator also has to check the validity of Δ, which it can do by storing the same Δ that the simulator learned from the TTP and forwarded to the corrupted $\mathcal{CP}_j$.

For the CdsZKTags part of the corrupted $\mathcal{CP}_1$, the simulator can define $x_2 = -x_1$ and $z_2 = -z_1$. This way the proof has the statement $0 = k \cdot 0$, which is correct independently of the modulus and the key, and the simulator can behave as an honest $\mathcal{RP}_i$ who is the verifier in this proof. Hence, we know that CdsZKTags has the correct inputs and that the proof does not leak z_2 and x_2, which mean that this special case of the proof is indistinguishable from the real case for the corrupted $\mathcal{CP}_1$. The simulations of corrupted computing parties give the same output distribution as the real protocol run, as the corrupted party has the same view and the result party learns either x or $\perp$, depending on the computing parties correctly participating in the protocol.

For corrupted $\mathcal{RP}_i$, the simulator receives x and Δ from the TTP and can fix x_1, x_2, z_2, r, $(\!\{x_1\}\!)_{pk}$ that it forwards to the result party. In addition, the simulator can compute $\mathsf{Enc}_{pk}(z_1) = \mathsf{Enc}_{pk}(k)^x \cdot \mathsf{Enc}_{pk}(-z_2)$ for the CdsZKTags. Hence, it can act as a simulator of the proof because it has all the correct queries $(\!\{x_1\}\!)_{pk}$, $(\!\{k\}\!)_{pk}$ and $(\!\{z_1\}\!)_{pk}$. The output distributions coincide as the corrupted $\mathcal{RP}_i$ has the same view and the output of the computing parties depends on the final decision of $\mathcal{RP}_i$. $\square$

An interesting aspect is that cheating in CdsZKTags can not help $\mathcal{CP}_1$ to make $\mathcal{RP}_i$ accept a wrong x. According to our assumptions, the $\mathcal{CP}_i$ are not allowed to collude and this proof can only make RPi to accept a faulty value from $\mathcal{CP}_2$. However, it can easily make the publishing protocol fail.

On the downside, as in Publish-$\mathcal{CP}\&\mathcal{RP}_i$ the Publish-$\mathcal{RP}_i$ also doest not allow $\mathcal{RP}_i$ to easily verify which of the parties $\mathcal{CP}_i$ tried to cheat if any of the checks fails. Theoretically, it would be possible to achieve by having both parties prove the correctness of all their previous computations.

4.3 Random share generation

The random share protocol (Singles) must generate a valid share representation of $[\![x]\!]_N = \langle \Delta, x_1, x_2, r, (\!\{x_1\}\!)_{pk}, z_1, z_2 \rangle$ for a random x where the participants do not know the value of x. This is a necessary protocol for sharing the inputs and producing random multiplicative triples. Both parties choose a random additive share and collaborate to fix the MAC tag as described in Algorithm 9. In addition, this gives a uniformly distributed random value $[\![x]\!]_N$ as a result as the sum of two uniformly distributed values is uniform. Furthermore, the value is uniformly distributed even if only one of the participants generated its share correctly.

Our initialisation of the proof of $z_1 + z_2 = k \cdot (x_1 + x_2)$ follows CdsZkMul (Algorithm 1), except for the initial messages, because a part of the query can be computed from the share by the verifier $\mathcal{CP}_2$. However, this actually means that the security of this protocol does not follow easily from the CdsZkMul. The best possibility would be to make $\mathcal{CP}_2$ to prove that q_3 is computed correctly. This way we could ensure the

Algorithm 9 Generating a random share (Singles)

```
Data: No inputs
Result: Shares $[\![x]\!]_N$ of random value $x$
 1: Round: 1
 2:     $\mathcal{CP}_i$ sets $\Delta = 0$
 3:     $\mathcal{CP}_1$ generates $x_1 \leftarrow \mathbb{Z}_N$
 4:     $\mathcal{CP}_1$ generates $r \leftarrow \mathbb{Z}_N^*$
 5:     $\mathcal{CP}_1$ sends $(\![x_1]\!)_{pk} = \mathsf{Enc}_{pk}(x_1, r)$ to $\mathcal{CP}_2$
 6:     $\mathcal{CP}_2$ generates $x_2, z_2 \leftarrow \mathbb{Z}_N$
 7:     $\mathcal{CP}_2$ computes $c = (\![k]\!)_{pk}^{x_2} \cdot \mathsf{Enc}_{pk}(-z_2)$
 8:     $\mathcal{CP}_2$ sends $c$ to $\mathcal{CP}_1$
 9: Round: 2
10:     $\mathcal{CP}_1$ computes $z_1 = k \cdot x_1 + \mathsf{Dec}_{sk}(c)$
11:     $\mathcal{CP}_2$ verifies that $(\![x_1]\!)_{pk}$ is a valid ciphertext
12: Verification:
13:     $\mathcal{CP}_1$ proves the correctness of MAC tags $z_1 + z_2 = k \cdot (x_1 + x_2)$ to $\mathcal{CP}_2$
14: return $[\![x]\!]_N$
```

simulatability and, hence, zero-knowledge property of this protocol, but would lose a lot of efficiency for the additional zero-knowledge proof. For now, we just assume that $\mathcal{CP}_1$ verifies that q_3 contains the right plaintext, which leaves a small hole that $\mathcal{CP}_2$ might use $\mathcal{CP}_1$ as kind of a decryption oracle, to check if it formed the q_3 correctly to contain the multiplication of the plaintexts from q_1 and q_2. We keep this protocol in hopes that we can define a simulatable protocol with the same form queries. In the future we should specify a simulatable version of CdsZKTags. A simple way to add some additional verification would be that occasionally the parties decide to discard the random value and open all values used for computing this or proving the correctness.

The idea of CdsZKTags is to ensure to $\mathcal{CP}_2$ that $\mathcal{CP}_1$ has all the values to accept this share during the opening phase in Publish-$\mathcal{CP}_1$. More precisely, after this proof, the $\mathcal{CP}_2$ knows that the share is correctly formed and that, if $\mathcal{CP}_2$ uses it correctly in the following computations, then $\mathcal{CP}_1$ should be able to open all the following results. Thus, $\mathcal{CP}_2$ can avoid malicious $\mathcal{CP}_1$ framing $\mathcal{CP}_2$ it as a malicious party. An honest $\mathcal{CP}_1$ already has this property because the commitment $(\![x_1]\!)_{pk}$ is public. This property is important as the Singles protocol is the basis for input sharing protocol Classify-$\mathcal{CP}_i$ which is the first step of all computations. Thus, verifying the correctness of $[\![x]\!]$ in Singles can be a basis for showing the correctness of all outputs.

Theorem 4.3.1. *Algorithm* Singles *for generating random shares is correct.*

Proof. The correctness of $(\![x_1]\!)_{pk} = \mathsf{Enc}_{pk}(x_1, r)$ is trivial in case of honest $\mathcal{CP}_1$. It is also trivial that $x_1 + x_2 + \Delta = x$ as the value of x is not predefined. For correctness we need to show that $z_1 + z_2 = k \cdot (x_1 + x_2)$ so that the verification succeeds:

$$\begin{aligned} z = z_1 + z_2 &= k \cdot x_1 + \mathsf{Dec}_{sk}(c) + z_2 = k \cdot x_1 + \mathsf{Dec}_{sk}((\![k]\!)_{pk}^{x_2} \cdot (\![-z_2]\!)_{pk}) + z_2 \\ &= k \cdot x_1 + k \cdot x_2 - z_2 + z_2 = k \cdot x_1 + k \cdot x_2 = k \cdot (x_1 + x_2) \ . \end{aligned}$$

□

The basic ideal functionality of the Singles protocol would be such that both parties notify the TTP that they are interested in sharing a random value. Then, the TTP

Algorithm 10 CdsZkMul for correctness of MAC tags (CdsZKTags)

```
Setup: Commitment parameters $ck$,
       Paillier keypair $(pk, sk)$ from the protection domain setup
Data: Shared secret $[\![w]\!]_N$
Result: $True$ for successful proof of $z_{1,w} + z_{2,w} = k(w_1 + w_2)$, $False$ in case of any failure
 1: Round: 1
 2:     $\mathcal{CP}_1$ computes and sends $(\![z_1^{(w)}]\!)_{pk} = \mathsf{Enc}_{pk}(z_1^{(w)})$ to $\mathcal{CP}_2$
 3: Round: 2
 4:     $\mathcal{CP}_2$ checks that $(\![z_1^{(w)}]\!)_{pk}$ is a valid ciphertext
 5:     $\mathcal{CP}_i$ sets $q_1 = (\![w_1]\!)_{pk}$, $q_2 = (\![k]\!)_{pk}$, $q_3 = (\![z_1^{(w)}]\!)_{pk} \cdot ((\![k]\!)_{pk}^{w_2} \cdot \mathsf{Enc}_{pk}(-z_2^{(w)}))^{-1}$
 6:     $\mathcal{CP}_2$ generates $e_1, e_2 \leftarrow \mathcal{M}$, $r_1, r_2 \leftarrow \mathcal{R}$, $r \leftarrow \mathcal{R}_1$, $s \leftarrow \mathcal{S}$
 7:     $\mathcal{CP}_2$ computes and sends $a_1 = q_1^{e_1} \cdot \mathsf{Enc}_{pk}(e_2, r_1)$ to $\mathcal{CP}_1$
 8:     $\mathcal{CP}_2$ computes and sends $a_2 = q_3^{e_1} \cdot q_2^{e2} \cdot \mathsf{Enc}_{pk}(\mathsf{Encode}(s, r), r_2)$ to $\mathcal{CP}_1$
 9: Round: 3
10:     $\mathcal{CP}_1$ computes $s' = \mathsf{Decode}(\mathsf{Dec}_{sk}(a_2) - \mathsf{Dec}_{sk}(a_1) \cdot k)$
11:     $\mathcal{CP}_1$ computes $(c, d) = \mathsf{Com}_{ck}(s')$ and sends $c$ to $\mathcal{CP}_2$
12: Round: 4
13:     $\mathcal{CP}_2$ sends $(s, e_1, e_2, r_1, r_2, r, q_3)$ to $\mathcal{CP}_1$
14: Round: 5
15:     $\mathcal{CP}_1$ verifies that $\mathsf{Dec}_{sk}(q_3) = k \cdot w_1$
16:     $\mathcal{CP}_1$: if $a_1 \neq q_1^{e_1} \cdot \mathsf{Enc}_{pk}(e_2, r_1)$ return $False$
17:     $\mathcal{CP}_1$: if $a_2 \neq q_3^{e_1} \cdot q_2^{e2} \cdot \mathsf{Enc}_{pk}(\mathsf{Encode}(s, r), r_2)$ return $False$
18:     $\mathcal{CP}_1$ sends $d$ to $\mathcal{CP}_2$
19: Round: 6
20:     $\mathcal{CP}_2$: if $\mathsf{Open}_{ck}(c, d) \neq s$ return $False$
21: return $True$
```

would generate the value x and give the share representation back to the computing parties. However, it is straightforward to see that we can not achieve this, as in our protocol both parties can choose their own x_i. We would like to consider a slightly different case where the TTP takes x_i as inputs from $\mathcal{CP}_i$. However, in this case we can also only fully simulate the case of corrupted $\mathcal{CP}_1$ that sends $\mathsf{Enc}_{pk}(x_1)$ in the real protocol where the simulator could learn x_1. However, for corrupted $\mathcal{CP}_2$ the only message c that it sends is independent of the input x_2 and, therefore, the corresponding simulator could not learn x_2. For now we show the simulatability of the communication assuming that x_i are private inputs of the protocol. For achieving fully simulatable protocol we should include the proof that $\mathcal{CP}_2$ knows x_2 and z_2 that it uses to compute the massage c.

Theorem 4.3.2. *The communication in algorithm* Singles *for generating random shares is computationally simulatable and the final shared value x is computationally uniformly distributed in $\mathbb{Z}_N$ given a computationally IND-CPA secure cryptosystem.*

Proof sketch. We show that there exists a non-rewinding simulator for the steps before the verification phase that manages to compute all the values needed for the verification. Cheating can be discovered during the verification, if the checks pass then the sharing is valid. We assume that x_i are actually the inputs of this protocol that the honest party would choose uniformly.

If $\mathcal{CP}_1$ is corrupted, then the simulator has to simulate the reply c from $\mathcal{CP}_2$. By definition $c = \mathsf{Enc}_{pk}(k)^{x_2} \cdot \mathsf{Enc}_{pk}(-z_2)$ is independent of x_2, therefore the simulator can simulate this efficiently by picking random values x_2^* and z_2^* and computing c according to the definition. If the adversary $\mathcal{A}$ sends $(\![x_1]\!)_{pk}$ that is not a valid ciphertext, then the simulator aborts, otherwise it finished successfully. In the end, the simulation has all the values x_2^*, z_2^*, $\mathsf{Enc}_{pk}(x_1)$ and $\mathsf{Enc}_{pk}(k)$ that it needs to continue with the zero-knowledge proof as an honest verifier. In can do continue with the proof as it has all the values for the queries and because the proof does not leak information about x_2^* and z_2^*, meaning that the proof with these value in indistinguishable from the one with real x_2 and z_2 for corrupted $\mathcal{CP}_1$.

If $\mathcal{CP}_2$ is corrupted, then the simulator must simulate the message $\mathsf{Enc}_{pk}(x_1)$. It can do this efficiently by publishing an encryption of a random value x_1^* because by the IND-CPA security of the cryptosystem, this in computationally indistinguishable from the encryption of the real input. The simulator sends *Abort* to the ideal functionality, if the message c from the adversary is not a valid ciphertext. Finally, the simulator can compute $\mathsf{Enc}_{pk}(z_1) = \mathsf{Enc}_{pk}(k)^{x_1^*} \cdot c$ for the zero-knowledge proof as its initial input to CdsZKTags. It could continue as a simulator of the proof if the simulator is defined.

Clearly, the result x is uniformly random, if at least one of the computing parties is honest. An honest party chooses its share uniformly as $x_i \leftarrow \mathbb{Z}_N$ and we know that in a ring the sum $x + r$ is uniformly distributed if x is uniform, independently of the distribution of r. However, we only achieve computationally uniform because $\mathcal{CP}_2$ also knows $(\![x_1]\!)_{pk}$ and might choose x_2 based on that. However, for a computationally IND-CPA secure cryptosystem, the probability of x_2 depending on x_1 is bounded by the computational indistinguishability. □

4.4 Beaver triples generation

Beaver triples [4] are multiplicative triples, so we need $[\![w]\!]_N = [\![xy]\!]_N$ from $[\![x]\!]_N$ and $[\![y]\!]_N$, where x and y are random values. We can easily find random shares $[\![x]\!]_N$ and $[\![y]\!]_N$ with Singles, so the main task of Triples protocol in Algorithm 11 is to correctly obtain $[\![w]\!]_N$. Random multiplicative triples are necessary to perform multiplication of the shares (Multiplication).

Verifying the correctness of $\mathcal{CP}_2$ requires using another unverified triple and thus, it can not be used after all runs of this protocol. However, in practice the protocols are commonly run on vectorised inputs and we could use half of the triples from generation to verify the other half.

Theorem 4.4.1. *Algorithm* Triples *for generating random triples is correct.*

Proof. The correctness of the commitment $(\![w_1]\!)_{pk}$ is trivial for an honest $\mathcal{CP}_1$. For correctness, we need to show that if both parties are following the protocol, then

Algorithm 11 Generating a random multiplicative triple (Triples)

```
Data: No inputs
Result: $[\![x]\!]_N$, $[\![y]\!]_N$, $[\![w]\!]_N$, where $w = x \cdot y$
 1: $\mathcal{CP}_i$ generate $[\![x]\!]_N$ and $[\![y]\!]_N$ with Singles
 2: Round: 1
 3:     $\mathcal{CP}_i$ sets $\Delta^{(w)} = 0$
 4:     $\mathcal{CP}_2$ generates $r,\ z_2^{(w)} \leftarrow \mathbb{Z}_N$
 5:     $\mathcal{CP}_2$ computes $w_2 = x_2 \cdot y_2 - r$
 6:     $\mathcal{CP}_2$ sends $v = (\![x_1]\!)_{pk}^{y_2} \cdot (\![y_1]\!)_{pk}^{x_2} \cdot \mathsf{Enc}_{pk}(r)$ to $\mathcal{CP}_1$
 7:     $\mathcal{CP}_2$ sends $t = (\![k]\!)_{pk}^{w_2} \cdot \mathsf{Enc}_{pk}(-z_2^{(w)})$ to $\mathcal{CP}_1$
 8: Round: 2
 9:     $\mathcal{CP}_1$ computes $w_1 = x_1 \cdot y_1 + \mathsf{Dec}_{sk}(v)$
10:     $\mathcal{CP}_1$ computes $z_1^{(w)} = k \cdot w_1 + \mathsf{Dec}_{sk}(t)$
11:     $\mathcal{CP}_1$ generates $r^{(w)} \leftarrow \mathbb{Z}_N^*$
12:     $\mathcal{CP}_1$ sends $(\![w_1]\!)_{pk} = \mathsf{Enc}_{pk}(w_1, r^{(w)})$ to $\mathcal{CP}_2$
13: Verification: Verify that $\mathcal{CP}_1$ is correct
14:     $\mathcal{CP}_2$ verifies that $(\![w_1]\!)_{pk}$ is a valid ciphertext
15:     $\mathcal{CP}_1$ proves $w = x \cdot y$ to $\mathcal{CP}_2$ using CdsZKMult
16:     $\mathcal{CP}_1$ proves $z_1^{(w)} + z_2^{(w)} = k \cdot (w_1 + w_2)$ to $\mathcal{CP}_2$ using CdsZKTags
17: Verification: Verify that $\mathcal{CP}_2$ is correct
18:     $\mathcal{CP}$ collaboratively run Triple Verification to learn $h$
19:     $\mathcal{CP}_1$: if $h \neq 0$ return $\perp$
20: return $[\![x]\!]_N, [\![y]\!]_N, [\![w]\!]_N$
```

$w = x \cdot y$ and $z^{(w)} = k \cdot (w_1 + w_2)$. Is is straightforward, as

$$\begin{aligned}
w &= w_1 + w_2 + \Delta^{(w)} = x_1 \cdot y_1 + \mathsf{Dec}_{sk}(v) + x_2 \cdot y_2 - r + 0 \\
&= x_1 \cdot y_1 + x_1 \cdot y_2 + y_1 \cdot x_2 + r + x_2 \cdot y_2 - r \\
&= x_1 \cdot (y_1 + y_2) + x_2 \cdot (y_1 + y_2) = (x_1 + x_2 + 0) \cdot (y_1 + y_2 + 0) = x \cdot y \\
z^{(w)} &= z_1^{(w)} + z_2^{(w)} = k \cdot w_1 + \mathsf{Dec}_{sk}(t) + z_2^{(w)} = k \cdot w_1 + k \cdot w_2 - z_2^{(w)} + z_2^{(w)} \\
&= k \cdot (w_1 + w_2)\ .
\end{aligned}$$

□

Analogously to the Singles protocol, the basic ideal functionality should be such that the parties notify TTP that they want a set of triples and the TTP then gives them the share. However, in the Singles protocol we are actually more likely to achieve the case where parties can pick their own inputs x_i and the TTP gives the other elements in the share representation. In the triples protocol the party $\mathcal{CP}_2$ can also actually pick w_2 for itself, therefore we also consider this as an input to the triple generation in the ideal world. Hence, the ideal functionality of the triple generation protocol is such that the TTP receives x_1, x_2, y_1, y_2, w_2 and computes w_1 such that the multiplicative relation holds, as well as fixes the tags and commitments. However, for now we can only show the simulatability of the communication of this protocol.

Theorem 4.4.2. *The communication of the* Triples *protocol for generating random multiplicative triples is computationally simulatable.*

Proof sketch. We show that the communication to either side is simulatable. We except the verification as it is straightforward to see that $\mathsf{Triple\ Verification}$ is simulatable and we have specially addressed the problems with $\mathsf{CdsZKTags}$ and $\mathsf{CdsZKMult}$. However, we know that if the verification succeeds then the triple has multiplicative relation and that the shares of the triple elements are correctly formed. We assume that the shares of the singles and w_2 are the inputs to this protocol.

The simulator can use the simulation for protocol $\mathsf{Singles}$ from Theorem 4.3.2 for generating random x and y. In case these protocols should abort, the simulator also aborts. Otherwise, it continues simulation.

The simulation for a malicious $\mathcal{CP}_1$ behaves exactly as an honest $\mathcal{CP}_2$ would, except that it has to pick w_2^* at random. The simulator can simulate v and t efficiently by picking a random w_2^* and using the values x_2^*, y_2^* that it picked for the simulation of the $\mathsf{Singles}$. It succeeds unless $\llbracket w_1 \rrbracket_{pk}$ is an invalid ciphertext. This simulation is perfect as the sent messages are independent of the input. By definition it has honestly computed the values needed in the zero-knowledge proofs and can behave as an honest verifier in the proof, because the proof does not leak its private input w_2^*.

We actually assume that the simulator for corrupted $\mathcal{CP}_2$ behaves slightly differently from the previous simulators. Namely, it also modifies the trusted setup, by defining a simulator of the setup, that picks $k_\mathcal{S}$ as a MAC key of the simulator and gives $\llbracket k_\mathcal{S} \rrbracket_{pk}$ instead of $\llbracket k \rrbracket_{pk}$ to $\mathcal{CP}_2$. Due to the IND-CPA security, this is computationally indistinguishable from the real setup. However, the limitation is that this simulated setup has to occur before any protocol runs, because the setup is shared between protocols. Therefore, all simulated runs for a corrupted $\mathcal{CP}_2$ must use the same $k_\mathcal{S}$ if they also contain the $\mathsf{Triples}$ protocol.

The simulator for a malicious $\mathcal{CP}_2$ can compute the only message $\llbracket w_1 \rrbracket_{pk}$ that it has to simulate as $c = \mathsf{Enc}_{pk}(x_1^* \cdot y_1^*) \cdot v$. It aborts, if v or t are invalid ciphertexts. This simulation is computationally indistinguishable from the real protocol run, given a computationally IND-CPA secure cryptosystem. This holds because the maximal advantage that adversary $\mathcal{A}$ might have for distinguishing $\mathcal{S}$ from honest $\mathcal{CP}_1$ occurs if it actually knows w_1 and can distinguish c from $\llbracket w_1 \rrbracket_{pk}$. Finally, the simulator has $\llbracket w_2 \rrbracket_{pk}$ and $\llbracket k_\mathcal{S} \rrbracket_{pk}$ as the queries to the zero-knowledge proof. It can define also the message $\llbracket z_1 \rrbracket_{pk} = \llbracket w_1 \rrbracket_{pk}^{k_\mathcal{S}} \cdot t$. Therefore it has all the correct values for inputs of $\mathsf{CdsZKTags}$. In addition, the simulator has all the values $\llbracket x_1^* \rrbracket_{pk}$, $\llbracket y_1^* \rrbracket_{pk}$ and $\llbracket w_1 \rrbracket_{pk} \cdot v^{-1}$ that it needs as queries in $\mathsf{CdsZKMult}$. Hence, it can run as a simulator for the proofs if the simulator is defined. □

The verification in $\mathsf{Triples}$ proves the correctness of $\mathcal{CP}_2$ to $\mathcal{CP}_1$ and vice versa. In general, we need that if $\mathcal{CP}_i$ has behaved correctly in the triple generation protocol, then this verification convinces $\mathcal{CP}_i$ that the other party $\mathcal{CP}_j$ also behaved correctly.

Our initialisation of the correctness proof of $\mathcal{CP}_1$ uses the special cases of CdsZkMul (Algorithm 1) and fails, if the multiplicative relation does not hold ($\mathsf{CdsZKMult}$) or the MAC tag is not correctly formed ($\mathsf{CdsZKTags}$). Their main difference from CdsZkMul is that the *prover* $\mathcal{CP}_1$ does not send the full initial query, but the query messages are fixed by the *verifier* $\mathcal{CP}_2$. Previously, we stressed that $\mathsf{CdsZKTags}$ can not be simulated because we need a zero-knowledge proof for the correctness of q_3 for that. The same holds for $\mathsf{CdsZKMult}$, because, also in there, $\mathcal{CP}_1$ has to be convinced that q_3 is computed correctly. However, for now we do not specify this proof and it is up to the follow-up work to define a simulatable initialisation for $\mathsf{CdsZKMult}$ and $\mathsf{CdsZKTags}$. As for the $\mathsf{CdsZKTags}$, we expect the simulatable version of $\mathsf{CdsZKMult}$ to also have the same input queries as the current version and use this assumption as a basis for our security proofs.

In addition, the verification step should ensure to $\mathcal{CP}_2$ that at this stage the sharing $[\![w]\!]$ is correct and $\mathcal{CP}_1$ can not frame it later. Analogously to the Singles protocol, anti-framing property holds for $\mathcal{CP}_1$ because of the commitment it made to w_1.

Proving the correctness of $\mathcal{CP}_2$ uses Triple Verification and needs to pick another unverified triple $[\![a]\!]_N$, $[\![b]\!]_N$, $[\![c]\!]_N$. If any of the two triples in the protocol are faulty then the correctness proof of $\mathcal{CP}_2$ fails, because $h \neq 0$. Actually, by the original definition, Triple Verification is used to prove the multiplicative relation to both $\mathcal{CP}_i$. However, here we only use this to prove the correctness of $\mathcal{CP}_2$ and similar proof about $\mathcal{CP}_1$ is done using CdsZKMult. This is due to the fact that using Triple Verification to also prove the multiplicative relation to $\mathcal{CP}_2$ could avoid CdsZKMult, but would introduce additional CdsZKTags. Currently $\mathcal{CP}_2$ does not have any knowledge about the correctness of the verification triple $[\![a]\!]_N, [\![b]\!]_N, [\![c]\!]_N$ and, therefore, we should use CdsZKTags to also prove that the tag of the third element of the verification triple is correct, meaning $z_1^{(c)} + z_2^{(c)} = k \cdot (c_1 + c_2)$. The latter is needed to prove to $\mathcal{CP}_2$ that the commitments used to open h in Triple Verification were computed correctly. However, CdsZKMult can be implemented slightly more efficiently than CdsZKTags and, therefore, our current specification is reasonable if using these proofs. A different approach should be considered if a more efficient analogue of CdsZKTags is used.

Algorithm 12 CdsZkMul for proving the multiplicative relation of the shares (CdsZKMult)

```
Setup: Commitment parameters $ck$,
       Paillier keypair $(pk, sk)$ from the protection domain setup
Data: Shares $[\![x]\!]_N, [\![y]\!]_N, [\![w]\!]_N$
Result: $True$ for successful proof of $x \cdot y = w$, $False$ in case of any failure
 1: Round: 1
 2:     $\mathcal{CP}_i$ sets $q_1 = ([x_1])_{pk}$, $q_2 = ([y_1])_{pk}$
 3:     $\mathcal{CP}_2$ sets $q_3 = ([w_1])_{pk} \cdot \mathsf{Enc}_{pk}(w_2) \cdot (([x_1])_{pk}^{y_2} \cdot ([y_1])_{pk}^{x_2} \cdot \mathsf{Enc}_{pk}(x_2 \cdot y_2))^{-1}$
 4:     $\mathcal{CP}_2$ generates $e_1, e_2 \leftarrow \mathcal{M}$, $r_1, r_2 \leftarrow \mathcal{R}$, $r \leftarrow \mathcal{R}_1$, $s \leftarrow \mathcal{S}$
 5:     $\mathcal{CP}_2$ computes and sends $a_1 = q_1^{e_1} \cdot \mathsf{Enc}_{pk}(e_2, r_1)$ to $\mathcal{CP}_1$
 6:     $\mathcal{CP}_2$ computes and sends $a_2 = q_3^{e_1} \cdot q_2^{e2} \cdot \mathsf{Enc}_{pk}(\mathsf{Encode}(s, r), r_2)$ to $\mathcal{CP}_1$
 7: Round: 2
 8:     $\mathcal{CP}_1$ computes $s' = \mathsf{Decode}(\mathsf{Dec}_{sk}(a_2) - \mathsf{Dec}_{sk}(a_1) \cdot y_1)$
 9:     $\mathcal{CP}_1$ computes $(c, d) = \mathsf{Com}_{ck}(s')$ and sends $c$ to $\mathcal{CP}_2$
10: Round: 3
11:     $\mathcal{CP}_2$ sends $(s, e_1, e_2, r_1, r_2, r, q_3)$ to $\mathcal{CP}_1$
12: Round: 4
13:     $\mathcal{CP}_1$ verifies that $\mathsf{Dec}_{sk}(q_3) = x_1 \cdot y_1$
14:     $\mathcal{CP}_1$: if $a_1 \neq q_1^{e_1} \cdot \mathsf{Enc}_{pk}(e_2, r_1)$ return $False$
15:     $\mathcal{CP}_1$: if $a_2 \neq q_3^{e_1} \cdot q_2^{e2} \cdot \mathsf{Enc}_{pk}(\mathsf{Encode}(s, r), r_2)$ return $False$
16:     $\mathcal{CP}_1$ sends $d$ to $\mathcal{CP}_2$
17: Round: 5
18:     $\mathcal{CP}_2$: if $\mathsf{Open}_{ck}(c, d) \neq s$ return $False$
19: return $True$
```

Theorem 4.4.3. *The verification steps in triple generation algorithm* Triples *are correct.*

Proof. The verification is correct, if it accepts correctly formed triples. We only need to show the correctness of the computations by $\mathcal{CP}_2$ as the correctness of $\mathcal{CP}_1$ is verified by two versions of the correct and universally composable CdsZkMul and the validity of the ciphertext.

Hence, we need to show that for a correct triple $[\![x]\!]_N$, $[\![y]\!]_N$, $[\![z]\!]_N$ and a verification triple $[\![a]\!]_N$, $[\![b]\!]_N$, $[\![c]\!]_N$ we get $h = 0$ from Triple Verification (Algorithm 5). We assume the correctness of necessary subprotocols of Triple Verification: Constant Multiplication, Subtraction, Constant Addition and Publish. By definition, we have

$$\begin{aligned} h &= s \cdot [\![w]\!]_N - [\![c]\!]_N - d \cdot [\![a]\!]_N - g \cdot [\![b]\!]_N - gd \\ &= s \cdot x \cdot y - a \cdot b - (y - b) \cdot a - (sx - a) \cdot b - (sx - a) \cdot (y - b) \\ &= sxy - ab - ya + ab - sxb + ab - sxy + sxb + ay - ab = 0 \ . \end{aligned}$$

Therefore, in the case of a correctly formed triple the verification succeeds. □

Theorem 4.4.4. *The verification steps in triple generation algorithm* Triples *are statistically $\frac{1}{p}$-secure against a cheating $\mathcal{CP}_2$ and as secure against a cheating $\mathcal{CP}_1$ as the proofs* CdsZKTags *and* CdsZKMult*, where $N = pq$, p, q are primes and $p < q$.*

Proof sketch. The security of $\mathcal{CP}_2$ depends on the security of CdsZKTags and CdsZKMult, that we should improve in the future to make them simulatable. However, the soundness property from CdsZkMul still holds, therefore, a successful proof indicates that $\mathcal{CP}_1$ has computed correctly.

Hence, we need to show the security of verification of the correctness of $\mathcal{CP}_2$. The verification algorithm for multiplicative relation is information-theoretically secure for finite fields [25]. According to CRT, breaking the security of the verification in case of modulus $N = pq$ also means breaking it separately modulo primes p and q, therefore the verification phase is $\frac{1}{p}$ secure for $\mathcal{CP}_1$ where $p < q$. □

4.5 Receiving inputs from the input party

The previously described Publish-$\mathcal{RP}_i$ (Algorithm 8) can be combined with the SPDZ classification protocol Classify-$\mathcal{IP}_i$ (Algorithm 4) in a straightforward manner to obtain a protocol for sharing the input of any third party. However, this version of the algorithm requires heavy computation and communication from the input party, who has to take part in a zero-knowledge proof. This is not efficient in many practical settings where we could have a variety of input devices, for example, smartphones and tablets.

There is a different protocol Classify-$\mathcal{IP}_i^\star$ in Algoritm 13 that uses ideas from the Singles protocol. In a way, the input party runs the single generation by itself and sends the corresponding share parts to the computing parties. Of course, instead of making a random share, it creates a share for its secret. The only addition is that the computing parties have to notify the input party if they accept this share. This means that the input party should wait and check that its input was accepted before it can know that it has correctly inserted the data.

Algorithm 13 Receiving an input from (non-computing) $\mathcal{IP}_i$ ($\mathsf{Classify}$-$\mathcal{IP}_i^\star$)

```
Data: Input party $\mathcal{IP}_i$ has a secret $x$
Result: Computing parties $\mathcal{CP}_i$ have a valid share representation $[\![x]\!]_N$
 1: Round: 1
 2:     $\mathcal{CP}_i$ fixes $\Delta = 0$
 3:     $\mathcal{IP}_i$ generates $x_1, z_2 \leftarrow \mathbb{Z}_N$, $r, t \leftarrow \mathbb{Z}_N^*$
 4:     $\mathcal{IP}_i$ computes $x_2 = x - x_1$
 5:     $\mathcal{IP}_i$ computes $c = (\![k]\!)_{pk}^{x_2} \cdot \mathsf{Enc}_{pk}(-z_2, t)$
 6:     $\mathcal{IP}_i$ sends $x_1$, $c$, $r$ to $\mathcal{CP}_1$
 7:     $\mathcal{IP}_i$ sends $x_2$, $z_2$, $(\![x_1]\!)_{pk} = \mathsf{Enc}_{pk}(x_1, r)$, $t$ to $\mathcal{CP}_2$
 8: Round: 2
 9:     $\mathcal{CP}_1$ computes $z_1 = k \cdot x_1 + \mathsf{Dec}_{sk}(c)$
10:     $\mathcal{CP}_2$ computes $c^* = (\![k]\!)_{pk}^{x_2} \cdot \mathsf{Enc}_{pk}(-z_2, t)$
11:     $\mathcal{CP}_2$ sends $(\![x_1]\!)_{pk}, c^*$ to $\mathcal{CP}_1$
12: Verification:
13:     $\mathcal{CP}_i$ verifies that $(\![x_1]\!)_{pk}$ is a valid ciphertext
14:     $\mathcal{CP}_1$ verifies that $c^* = c$
15:     $\mathcal{CP}_1$ verifies that it received $(\![x_1]\!)_{pk} = \mathsf{Enc}_{pk}(x_1, r)$
16:     $\mathcal{CP}_1$ proves $z_1 + z_2 = k \cdot (x_1 + x_2)$ to $\mathcal{CP}_2$ using CdsZKTags
17:     $\mathcal{CP}_1$ and $\mathcal{CP}_2$ notify $\mathcal{IP}_i$ about the verification outcome
```

Theorem 4.5.1. *Protocol* $\mathsf{Classify}$-$\mathcal{IP}_i^\star$ *in Algorithm 13 for collecting inputs from input parties is correct.*

Proof sketch. The correctness of additive shares is clear by the definition as $x = x_1 + x_2 + \Delta = x_1 + x - x_1 + 0 = x$. If $\mathcal{CP}_1$ accepts the commitment then it is valid. Finally, we need that the MAC tag is correct:

$$z = z_1 + z_2 = k \cdot x_1 + k \cdot x_2 - z_2 + z_2 = k \cdot (x_1 + x_2) \ ,$$

thus, in case of honest participants the zero-knowledge proof succeeds and the share is correctly formed. □

There are three potential security risks in this protocol: (1) $\mathcal{CP}_i$ might modify the share, (2) $\mathcal{CP}_2$ might modify the share, and (3) $\mathcal{IP}_i$ might try to give inconsistent share representation. However, $\mathcal{IP}_i$ can always affect the outcome by inputting a maliciously chosen value x instead of the valid input value. Still, by security definition regardless of the choice of x it can only input a valid representation $[\![x]\!]$. This version of the protocol is only usable if $\mathcal{IP}_i$ is not one of the computing parties and does not collude with them. The former is not a restriction as we have a simpler protocol for $\mathcal{CP}_i$ to classify inputs, but the latter may be too restrictive for practical applications.

Theorem 4.5.2. *Protocol* $\mathsf{Classify}$-$\mathcal{IP}_i^\star$ *for collecting inputs from input parties is perfectly secure against corrupted* $\mathcal{CP}_1$ *and* $\mathcal{IP}_i$ *and computationally secure against corrupted* $\mathcal{CP}_2$ *with additional* $\frac{1}{p}$ *error probability, assuming a computationally IND-CPA secure cryptosystem and modulus* $N = pq$*, where* p *is the smaller of its prime factors.*

Proof sketch. The principal ideal functionality of this protocol would be such that the $\mathcal{IP}_i$ gives x to the TTP and TTP gives shares of x to the computing parties. However, we can not achieve this as the distribution of the shares of x is controlled by $\mathcal{IP}_i$. Hence, we define an ideal model where $\mathcal{IP}_i$ gives x, x_1, z_2 and r to TTP, who creates the remaining z_1 and forwards these to the computing parties. After that, the computing parties notify the TTP about accepting or rejecting these shares and the TTP forwards the outcome as *Success* or *Failure* to all parties.

Corrupted $\mathcal{CP}_1$. The simulator at first receives x_1, z_1, r from the TTP. It then computes $c = \mathsf{Enc}_{pk}(z_1) \cdot [\![k]\!]_{pk}^{-x_1} = \mathsf{Enc}_{pk}(z_1 - k \cdot x_1)$ and sends x_1, c and r to the corrupted $\mathcal{CP}_1$. By definition $\mathcal{CP}_1$ gets the output z_1 as the one given by the TTP. In the following, the simulator simulates the messages from $\mathcal{CP}_2$ as $\mathsf{Enc}_{pk}(x_1, r)$ and $c^* = c$. For the zero-knowledge proof, the simulator can define $x_2^* = -x_1$ and $z_2^* = -z_1$ to behave as an honest verifier for the case $0 = k \cdot 0$. This can be done as the proof does not leak x_2^* and z_2^* and, therefore, the corrupted $\mathcal{CP}_1$ sees the same view that it would in the real world. Finally, the simulator receives *Continue* or *Abort* from the corrupted $\mathcal{CP}_1$ and forwards this to the TTP. The outputs of the real and simulated ideal world coincide as in case the sharing succeeds both computing parties have the same output shares in both worlds and in case the sharing does not succeed they have both seen the same shares in these two versions of the protocol.

Corrupted $\mathcal{CP}_2$. In case of the simulation for corrupted $\mathcal{CP}_2$, we assume that the shared setup step was also simulated, so that the simulator has $k_{\mathcal{S}}$, whereas the $\mathcal{CP}_2$ has $[\![k_{\mathcal{S}}]\!]_{pk}$. The simulator receives x_2, z_2 and $[\![x_1]\!]_{pk}$ from the TTP. It has to specify the value t, that it can do by generating it as honest $\mathcal{IP}_i$ would. It then forwards x_2, z_2, $[\![x_1]\!]_{pk}$ and t to $\mathcal{CP}_2$. On messages c^* and $[\![x_1]\!]_{pk}$ from the corrupted $\mathcal{CP}_2$, the simulator verifies that the $[\![x_1]\!]_{pk}$ is the same as sent to $\mathcal{CP}_2$ and that $c^* = [\![k_{\mathcal{S}}]\!]_{pk}^{x_2} \cdot \mathsf{Enc}_{pk} - z_2, t$. If these are incorrect, then the simulator outputs *Failure*, otherwise it continues the simulation. For the zero-knowledge proof, the simulator has $[\![x_1]\!]_{pk}$ and $[\![k_{\mathcal{S}}]\!]_{pk}$ meaning that it can also compute $[\![z_1]\!]_{pk} = ([\![x_1]\!]_{pk} \cdot \mathsf{Enc}_{pk}(x_1))^{k_{\mathcal{S}}} \cdot \mathsf{Enc}_{pk}(-z_2) = \mathsf{Enc}_{pk}(x \cdot k_{\mathcal{S}} - z_2)$. Therefore the simulator has all the correct inputs for $\mathsf{CdsZKTags}$ and it could behave as a simulator for the proof. The outputs coincide as for a corrupted $\mathcal{CP}_1$.

Corrupted $\mathcal{IP}_i$. The simulator for a corrupted $\mathcal{IP}_i$ at first receives x_1, c, r, x_2, z_2, $[\![x_1]\!]_{pk}$ and t from the corrupted $\mathcal{IP}_i$. In then verifies that $[\![x_1]\!]_{pk} = \mathsf{Enc}_{pk}(x_1, r)$ and $c = [\![k]\!]_{pk}^{x_2} \cdot \mathsf{Enc}_{pk}(-z_2, t)$. If these hold, then it sends x, x_1, z_2 and r to the TTP and forwards the *Success* or *Failure* to the corrupted $\mathcal{IP}_i$ as messages from $\mathcal{CP}_i$. If the initial check does not verify, then it sends *Failure* to the $\mathcal{IP}_i$ and notifies the TTP that it does not participate in the protocol, which means that the protocol is a failure for all parties. The output distributions coincide because the simulator performs the same checks as the real functionality would to ensure the correctness of the $\mathcal{IP}_i$. □

Actually, $\mathsf{Classify}\text{-}\mathcal{IP}_i^\star$ is also secure if $\mathcal{CP}_1$ and $\mathcal{IP}_i$ are corrupted by the same adversary, but insecure if $\mathcal{CP}_2$ and $\mathcal{IP}_i$ are corrupted together. In the latter case, the adversary could use one run of the protocol to check if some ciphertext that it sends as c is an encryption of some fixed message m. In practice, this protocol can be fairly securely used if $\mathcal{CP}_1$ only publishes encryptions of uniformly distributed elements in $\mathbb{Z}_N$ and the number of expected inputs is small, or if the input party is authenticated and one party should input a limited number of elements. In this case, the probability of an input party guessing the correct m within the expected number of input attempts can be made arbitrarily small. For full security, we could define a zero-knowledge proof where $\mathcal{IP}_i$ or $\mathcal{CP}_2$ proves to $\mathcal{CP}_1$ that c or c^* is computed correctly. It would be more reasonable to define this for c^* because we are more likely to have miners with bigger computing capability. Besides, we can always use $\mathsf{Classify}\text{-}\mathcal{IP}_i$ with $\mathsf{Publish}\text{-}\mathcal{RP}_i$, if we expect $\mathcal{IP}_i$ to have enough computational power to efficiently participate in an analogous proof.

We have the anti-framing property, as after this protocol, all parties know that the share was correctly formed. Intuitively, as in the Singles protocol, framing $\mathcal{CP}_2$ is infeasible because of the zero-knowledge proof, which shows that in this step $\mathcal{CP}_1$ could correctly open this share. Analogously, framing $\mathcal{CP}_1$ is impossible because it can convince itself that the commitment $(\![x_1]\!)_{pk}$ is correct. However, if any of the checks fail during the protocol, then the computing parties can not easily verify whether the input party or the other computing party is acting maliciously.

4.6 Efficiency of the protocols

This section analyses the theoretical cost of the proposed protocols. We have two important criteria: (1) computational cost and (2) communication cost. These allow to compare these protocols as well as to estimate the cost of future protocols that are built from these existing blocks. As an overview, Figure 4.2 illustrates the current state of existing primitive protocols and protocols combined from them. The protocol Classify-$\mathcal{IP}_i$ has actually two versions, one that is derivated from publishing algorithm Publish-$\mathcal{RP}_i$ (which is equal to Publish $\mathcal{IP}_i$), and the second, that is a standalone protocol Classify-$\mathcal{IP}_i^\star$.

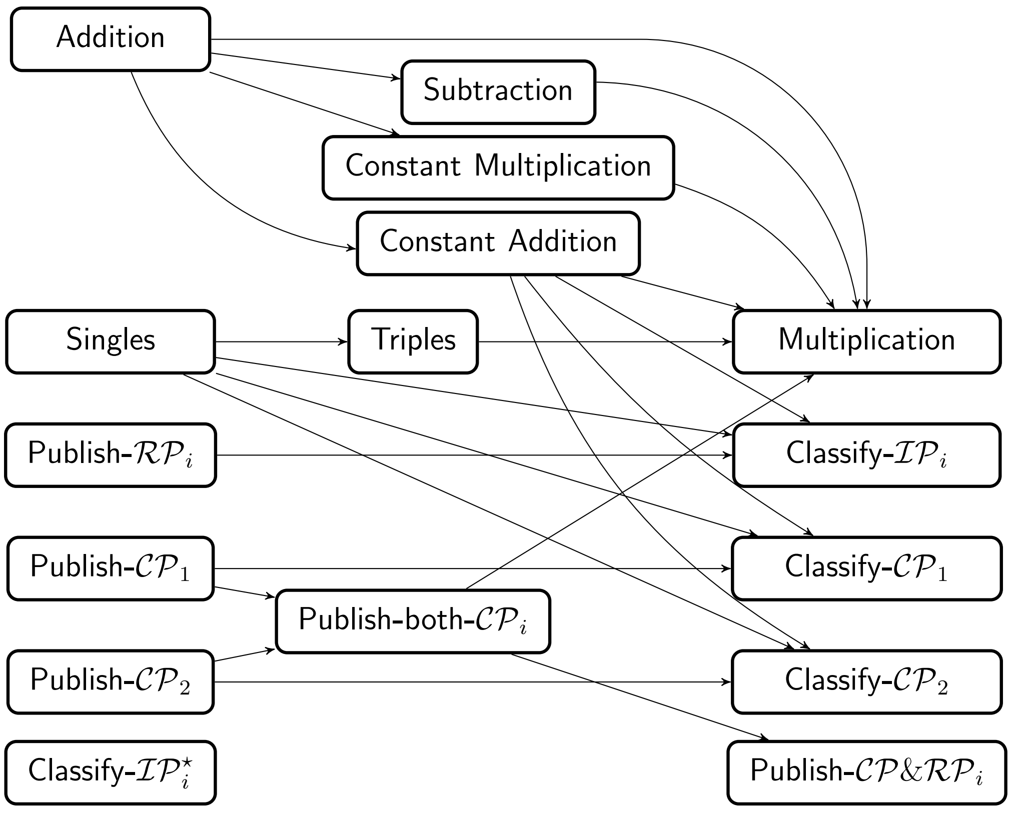

Addition
Subtraction
Constant Multiplication
Constant Addition
Singles
Triples
Multiplication
Publish-$\mathcal{RP}_i$
Classify-$\mathcal{IP}_i$
Publish-$\mathcal{CP}_1$
Classify-$\mathcal{CP}_1$
Publish-both-$\mathcal{CP}_i$
Publish-$\mathcal{CP}_2$
Classify-$\mathcal{CP}_2$
Classify-$\mathcal{IP}_i^\star$
Publish-$\mathcal{CP}\&\mathcal{RP}_i$

Figure 4.2: The hierarchy of protocols for the asymmetric setup

For a shorthand we define Add for Addition, Subtract for Subtraction, ConstMult for Constant Multiplication and Multiply for Multiplication.

4.6.1 Computational cost

This section analyses the computational requirements if the protocols are applied to one element, the extension to vectors in most cases just requires that amount of computation for each element. We focus on the multiplication and exponentiation operations as addition and subtraction are always considerably more efficient.

For a $|N|$-bit exponent we assume that we need to perform approximately $|N|$ multiplications, as using square-and-multiply it can definitely be done with $2|N|$. Furthermore, Paillier encryption and decryption have approximately the cost of one exponentiation, where exponent has length $|N|$, but multiplied elements have length $2 \cdot |N|$. Finally, modular inversion has the cost of a few multiplications, denoted by i in the following. For simplicity, we also assume that computing gcd for Paillier ciphertext validity check also has cost i as both of these can be done by Euclidean algorithm. Hence, we can estimate the computational complexity as the number of multiplication on either values of $\mathbb{Z}_N$ or Paillier ciphertexts in $\mathbb{Z}_{N^2}$ of length $2|N|$. Because of this we divide the analysis to two parts and give separate results results for both of these lengths. In total, the following should be taken as a rough estimate for comparing these protocols and the total cost is the sum of both length with more relative cost for length $2|N|$.

Party	Length	Publish	Add	Subtract	ConstMult	Multiply
$\mathcal{CP}_1$	$\|N\|$	1	1	$i+1$	$\|N\|+3$	$2\|N\|+2i+13$
	$2\|N\|$	0	0	0	0	0
$\mathcal{CP}_2$	$\|N\|$	0	0	0	3	7
	$2\|N\|$	$\|N\|$	1	$i+1$	$\|N\|$	$4\|N\|+2i+4$

Table 4.1: The computational cost of computation protocols as a number of multiplications

Table 4.1 summarises the multiplicative cost of our basic computation protocols. The length denotes the bitlength of the multiplication operands and other fields stand for separate protocols. For Publish-$\mathcal{CP}_i$ protocol, we consider the work that either party $\mathcal{CP}_i$ has to do in order to verify the result if the value is opened to it. We omit Classify-$\mathcal{CP}_i$ as it has exactly the same multiplicative cost as Publish-$\mathcal{CP}_i$. The results are obtained by counting the corresponding operations in the protocols.

Party	Length	Singles	Triples	TripleVerif
$\mathcal{CP}_1$	$\|N\|$	1	2	$4\|N\|+5i+22$
	$2\|N\|$	$2\|N\|$	$3\|N\|$	0
$\mathcal{CP}_2$	$\|N\|$	0	1	13
	$2\|N\|$	$2\|N\|+i+1$	$5\|N\|+i+3$	$8\|N\|+5i+5$

Table 4.2: The computational cost of precomputation as a number of multiplications

The cost of precomputation can be seen from Table 4.2. The costs of zero-knowledge proofs have been omitted from the precomputation protocols and can be found in Table 4.3. We omit the cost of our commitment scheme from the analysis of the zero-knowledge protocols as we use a lot smaller field for elliptic curves than in our general computations. Likewise, the cost of Singles values has been omitted from the Triples protocol, which here only illustrates the cost of obtaining one unverified triple as given in Algorithm 11. Triple Verification is an exceptional protocol as its amortized cost per vector element is slightly less than given in the table because we can pick one random element to verify a set of triples.

Party	Length	CdsZKTags	CdsZKMult
Prover	$\|N\|$	2	2
	$2\|N\|$	$9\|N\|+3$	$8\|N\|+3$
Verifier	$\|N\|$	0	1
	$2\|N\|$	$7\|N\|+i+5$	$9\|N\|+i+7$

Table 4.3: The computational cost of zero-knowledge proofs as a number of multiplications

Table tbl:zk-comp-requirements illustrates the computational requirements of the zero-knowledge proofs. It can be seen that CdsZKMult is more efficient on the side of $\mathcal{CP}_1$ and actually it can easily be implemented more efficiently also for $\mathcal{CP}_2$. The main complexity on the side of $\mathcal{CP}_2$ results from the computation of the third query q_3, which actually does several computation already performed in Triples and we could reduce $9|N|+i+7$ to approximately $5|N|+i+4$. Therefore, in our context, CdsZKMult is more efficient than CdsZKTags when used to verify the triple generation procedure.

Table 4.4 summarises the cost of protocols used to communicate with input and result parties. Similarly to precomputation, the zero-knowledge proofs have been omitted from this analysis. It can be seen that with additional proofs, the workload of $\mathcal{RP}_i$ in Publish-$\mathcal{RP}_i$ is approximately the same as that of $\mathcal{CP}_1$ whereas $\mathcal{CP}_2$ does not need to do any computations. However, Classify-$\mathcal{IP}_i^\star$ adds the complexity of the proof to the computing parties. Thus, in total $\mathcal{IP}_i$ has lower workload than $\mathcal{RP}_i$.

Party	Length	Classify-$\mathcal{IP}_i^\star$	Publish-$\mathcal{RP}_i$
$\mathcal{CP}_1$	$\|N\|$	1	0
	$2\|N\|$	$2\|N\|+i$	0
$\mathcal{CP}_2$	$\|N\|$	0	0
	$2\|N\|$	$2\|N\|+i+1$	0
$\mathcal{IP}_i \setminus \mathcal{RP}_i$	$\|N\|$	0	0
	$2\|N\|$	$3\|N\|+1$	$\|N\|$

Table 4.4: The computational cost of protocols for communicating with a third party as a number of multiplications

In total, the precomputation and zero-knowledge proofs form the most expensive part of this protection domain. Therefore, they remain the most important point for further optimisations.

4.6.2 Communication cost

This section analyses the cost of communication per protocol output in terms of the number and length of the sent messages. As in the previous section, we have two clear classes of messages with different length: plain elements of $\mathbb{Z}_N$ and Paillier ciphertext. In addition, we now include commitments for zero-knowledge protocols. These commitments are pairs of elliptic curve elements and for a prime P each element can be encoded as $|P|+1$ bits. Decommitments consist of two values that are both at most $|P|$ bits. In addition, we can assume that the secret value in zero-knowledge proofs is at most length $|P|$, and the randomness of the encoding function is bounded by N.

All the local computation protocols Addition, Subtraction, Constant Addition and Constant Multiplication do not require any communication. In addition, protocols Triple Verification and Multiplication only use communication during Publish protocols.

Party	Length	CdsZKTags	CdsZKMult	Singles	Triples	Publish
$\mathcal{CP}_1$	$\|N\|$	0	0	0	0	2
	$2\|N\|$	1	0	1	1	0
	$\|P\|$	4	4	0	0	0
$\mathcal{CP}_2$	$\|N\|$	5	5	0	0	2
	$2\|N\|$	3	3	1	2	0
	$\|P\|$	1	1	0	0	0

Table 4.5: The communication cost of computation protocols as number of messages

Therefore, it is sufficient to only analyse the communication cost of the zero-knowledge protocols, precomputation and publishing.

Table 4.5 summarises the communication cost for each class of messages. As previously, the precomputation protocols only include the cost of their specific functions and not proofs or other precomputation protocols. The amount or messages for a player means how much messages of which length he has to send for each protocol. Publish protocols only have a communication cost for one of the participants and the table should be read so that $\mathcal{CP}_j$ has to send that many messages in Publish-$\mathcal{CP}_i$. It can be seen that the precomputations and Publish-$\mathcal{CP}_i$ are quite efficient compared to the zero-knowledge protocols.

Party	Length	Classify-$\mathcal{IP}_i^\star$	Publish-$\mathcal{RP}_i$
$\mathcal{CP}_1$	$\|N\|$	0	3
	$2\|N\|$	0	0
$\mathcal{CP}_2$	$\|N\|$	0	3
	$2\|N\|$	2	1
$\mathcal{IP}_i \setminus \mathcal{RP}_i$	$\|N\|$	5	0
	$2\|N\|$	2	0

Table 4.6: The communication cost of protocols for communicating with a third party as number of messages

Table 4.6 gives an analogous overview for protocols including $\mathcal{IP}_i$ or $\mathcal{RP}_i$. Separately, they also prove more communication efficient than zero knowledge proofs. Thus, also communication-wise, the zero-knowledge proofs are currently the main bottleneck of this protection domain.

Chapter 5

Protocols for Beaver triple generation

This section describes our efforts to efficiently generate Beaver triples for various moduli using the additively homomorphic Paillier cryptosystem. This section only considers the *semi-honest* security setting as the main goal of this section is to propose new ideas for precomputation and it is easier to reason about them in the passive model. Besides, it is reasonable as we often can do precomputation by firstly fixing unprotected shares, then protection mechanisms and finally, we can verify that the sharing is correct. More specifically, we only consider the case of semi-honest static adversary in our security proofs.

5.1 Setup for triple generation protocols

This chapter considers the case where we have additively shared secrets $[\![x]\!]_M$ and $[\![y]\!]_M$ for some modulus M and the goal is to obtain $[\![w]\!]_M$ where $w = x \cdot y \bmod N$. Hence, $[\![x]\!]_M = \langle x_1, x_2 \rangle$.

Algorithm 14 Multiplication of additively shared secrets based on the Paillier cryptosystem (Paillier Multiplication)

```
Setup: Paillier keypair with modulus $N$, where $\mathcal{CP}_1$ knows the secret key
Data: Shared secrets $[\![x]\!]_M,\ [\![y]\!]_M$
Result: Shared result $[\![w]\!]_N$, where $w = (x_1 + x_2) \cdot (y_1 + y_2) \bmod N$
1: $\mathcal{CP}_1$ encrypts and sends $\mathsf{Enc}_{pk}(x_1)$, $\mathsf{Enc}_{pk}(y_1)$ to $\mathcal{CP}_2$
2: $\mathcal{CP}_2$ generates $r \leftarrow \mathbb{Z}_N$
3: $\mathcal{CP}_2$ computes and sends $t = \mathsf{Enc}_{pk}(x_1 \cdot y_2 + y_1 \cdot x_2 + r)$ to $\mathcal{CP}_1$
4: $\mathcal{CP}_2$ computes $w_2 = x_2 \cdot y_2 - r \bmod N$
5: $\mathcal{CP}_1$ computes $w_1 = x_1 \cdot y_1 + \mathsf{Dec}_{sk}(t) \bmod N$
```

Protocol Paillier Multiplication in Algorithm 14 shows how to do simple share multiplication using the Paillier cryptosystem and additive secret sharing. This will be an important tool throughout this chapter. This protocol is actually all that is needed to generate triples $[\![x]\!]_N, [\![y]\!]_N, [\![w]\!]_N$ for a Paillier modulus N.

Theorem 5.1.1. *The* Paillier Multiplication *protocol for share multiplication using the Paillier cryptosystem is correct.*

Proof. It remains to show that indeed $w = x \cdot y \bmod N$, which can be seen trivially, as

$$\begin{aligned} w &= w_1 + w_2 = x_1 \cdot y_1 + \mathsf{Dec}_{sk}(t) + x_2 \cdot y_2 - r \\ &= x_1 \cdot y_1 + x_1 \cdot y_2 + x_2 \cdot y_1 + r + x_2 \cdot y_2 - r = (x_1 + x_2) \cdot (y_1 + y_2) = x \cdot y \ . \end{aligned}$$

□

Theorem 5.1.2. *The* Paillier Multiplication *protocol for share multiplication using the Paillier cryptosystem is computationally secure against corrupted $\mathcal{CP}_2$ and statistically secure against a corrupted $\mathcal{CP}_1$ in the passive model.*

Proof. The ideal functionality of this protocol is such that it receives x_1, y_1 from $\mathcal{CP}_1$ and x_2, y_2 from $\mathcal{CP}_2$. It gives back w_1 to $\mathcal{CP}_1$ and w_2 to $\mathcal{CP}_2$.

In semi-honest case the simulator knows the inputs x_i and y_i of the corrupted party and can easily forward them to the TTP. In both cases it gets w_i back from the TTP and must now make sure that the output of corrupted $\mathcal{CP}_i$ is finally w_i.

In case of a corrupted $\mathcal{CP}_1$, the output is fixed as $w_1 = x_1 \cdot y_1 + \mathsf{Dec}_{sk}(t)$ where the simulator must fix t. Knowing w_1, x_1 and y_1, the simulator can easily compute $c = w_1 - x_1 \cdot y_1$ and fix $t = \mathsf{Enc}_{pk}(c)$. In is straightforward to see that the corrupted $\mathcal{CP}_1$ will output w_1. The outputs of the real and simulated world coincide.

In case of a corrupted $\mathcal{CP}_2$, we know that the output $w_2 = x_2 \cdot y_2 - r$, depends on the randomness r chosen by the corrupted $\mathcal{CP}_2$. However, as it is semi-honest, we know that it chooses $r \leftarrow \mathbb{Z}_N$, therefore, r is affected by the initial randomness of $\mathcal{CP}_2$. Knowing the desired output w_2 and inputs x_2, y_2 the simulator can compute $r = x_2 y_2 - w_2$ and pick a suitable randomness to run $\mathcal{CP}_2$ with. Therefore, the simulated run in the ideal world and the corresponding run in the real world always have coinciding output distributions. □

As stated in the algorithm description, we can actually obtain something more general, namely $[\![w]\!]_N$ for $w = (x_1 + x_2) \cdot (y_1 + y_2) \bmod N$ from $[\![x]\!]_M$ and $[\![y]\!]_M$, which is actually closer to what we will use in the following. If we use a modulus M such that $M^2 < N$, then we actually have $x \cdot y < N$. Therefore taking the final $w \bmod M$ should give us a valid triple. Actually, it is slightly more difficult as we have

$$w_1 + w_2 \geq N \text{ or } w_1 + w_2 < N \ .$$

In the latter case, this conversion works by reducing both shares separately. However, in the former case a simple modulo reduction gives invalid results. This remaining issue is discussed in Section 5.3.

For now, we can assume that we can get correct results for at least half of the executions of Paillier Multiplication with modulus M. For example, if we always toss a fair coin after the generation and then either try to correct the error by computing $w = w_1 + w_2 - N \bmod M$ or do not try to correct it, then we get the right triple half of the times. A similar algorithm has been used for precomputations also by predecessors of SPDZ [7, 25], but with restrictions to the size of the randomness r to avoid the overflow.

We say that the yield of the protocol is the ratio of the length of the produced triple elements to the length of the Paillier modulus and denote it by γ_{len}. Analogously, by γ_{net} we denote the ratio of triple length to the cumulative length of exchanged messages on the network. Finally, by γ_{comp} we denote the ratio of outputs bits to multiplications of ciphertexts that are elements of length $2|N|$. As in the asymmetric case, we assume that encryption and decryption require $|N|$ multiplications and that the ciphertexts have length $2|N|$. In general, we would like to maximise all these parameters for the best efficiency. For packing the best achievable bound is 1, for others there is no fixed limit. For now, we focus on maximising γ_{len} and give other for additional comparison in hope that they are easier to improve on using clever implementation tricks.

Clearly, for modulus N we have

$$\gamma_{len} = 1,\ \ \gamma_{net} = \frac{|N|}{3 \cdot 2|N|} = \frac{1}{6} \text{ and } \gamma_{comp} = \frac{|N|}{6|N| + 2} \approx \frac{1}{6} \ .$$

For an arbitrarily chosen modulus M, where $M^2 < N$, we achieve on average

$$\gamma_{len} = \frac{1}{2} \cdot \frac{|M|}{|N|} \leq \frac{1}{4},\ \ \gamma_{net} = \frac{1}{2} \cdot \frac{|M|}{3 \cdot 2|N|} \leq \frac{1}{24},\ \ \gamma_{comp} = \frac{1}{2} \cdot \frac{|M|}{4|N| + 2|M| + 2} \leq \frac{1}{20} \ ,$$

if we assume that we get half of the triples correctly.

5.2 Packing several shares into one generation

The introduced Paillier Multiplication was used for a general modulus M with a relation to the used Paillier modulus N, approximately $M^2 < N$, or for $M = N$. It could then be used for any such modulus M, but is clearly most efficient in terms of γ_{len} if the size of M is close to the bound $M^2 < N$. However, for practical sizes of N, this results in a very long $|M| \geq 1024$. This section explores how shorter types could be packed inside elements of $\mathbb{Z}_M$ so that we can most efficiently use the triple generation protocols and learn meaningful triples for shorter moduli.

The efficiency of packing is mainly shown by γ_{len}. The values for γ_{net} and γ_{comp} reflect more the communication and computation cost that we need to compute with given packing.

5.2.1 Packing as base-B numbers

Packing as B-ary numbers means that each element modulo $M < B$ represents a digit and we pack them as a numbers of base B. A three-digit base-B number could be written out as

$$x = B^2 \cdot x_3 + B \cdot x_2 + x_1 \ ,$$

where $x_i < B$ are digits. If we assume, that y is written out in a similar manner as

$$y = B^2 \cdot y_3 + B \cdot y_2 + y_1 \ ,$$

then the corresponding multiplication becomes

$$xy = B^4 x_3 y_3 + B^3(x_3 y_2 + x_2 y_3) + B^2(x_2 y_2 + x_1 y_3 + x_3 y_1) + B(x_2 y_1 + x_1 y_2) + x_1 y_1 \ .$$

This shows that we could get a triple x_1, y_1, $x_1 y_1$ assuming that $x_1 y_1$ does not overflow the B-ary digit. With restrictions to the initial x_i and y_i values we can ensure that this nor other combinations do not overflow and xy is a five-digit number. Thus, we could also get a triple x_3, y_3 and $x_3 y_3$.

Packing as straightforward B-ary numbers is, therefore, not very beneficial as we did not receive a triple x_2, y_2, x_2y_2. However, we could consider another example, with x as before, but y is modified, giving

$$\begin{aligned} x &= B^2 \cdot x_3 + B \cdot x_2 + x_1 \\ y &= B^6 y_3 + B^3 \cdot y_2 + y_1 \ . \end{aligned}$$

The resulting multiplication is

$$\begin{aligned} xy =& B^8 x_3y_3 + B^7 x_2y_3 + B^6 y_3x_1 + B^5 x_3y_2 + B^4 x_2y_2 + B^3 x_1y_2 + \\ &+ B^2 x_3y_1 + Bx_2y_1 + x_1y_1 \ . \end{aligned}$$

Here we can see that xy contains all the triples x_i, y_i and x_iy_i, but also some elements x_iy_j, $i \neq j$ that we do not need. Picking the powers in y as multiples of the number of digits in x always gives analogous results. More specifically, if both pack n elements, then the product would have n^2 digits where we are only interested in n of them in form x_iy_i. Thus, in this packing we only get the same number of triples as the square-root of the number digits in the base-B representation of xy.

What we need to achieve is actually that every result x_iy_i is multiplied by a unique power of B. On the other hand, we do not care about the other x_iy_j, which could share the same powers of B between them. This observation allows us to always make small adjustments to special cases. For example, using

$$y = B^4 y_3 + B^2 y_2 + y_1$$

would also enable us to get all the x_iy_i pairs with the gain of of two B-ary digits. The result xy is only a seven digit number as

$$\begin{aligned} xy =& B^6 x_3y_3 + B^5 x_2y_3 + B^4(x_1y_3 + x_3y_2) + B^3 x_2y_2 + B^2(x_1y_2 + x_3y_1) + \\ &+ Bx_2y_1 + x_1y_1 \ . \end{aligned}$$

However, we can not get rid of all the unnecessary products x_iy_j in this packed multiplication result.

The problem with using this approach in a straightforward manner is that we have to assume that $y_ix_i < B$, which essentially means that the result xy contains integer form results of all triples as digits. We can not use this directly with Paillier Multiplication, as the randomisation there would ruin this structure. However, we can not define it without any randomisation either because otherwise seeing y_ix_i and knowing x_i also leaks the secret y_i. A possible solution is that we actually redefine the randomising element as

$$r = \sum B^i r_i$$

and make sure that at least for all $r_i + x_iy_i$, there is no overflow from B from either side. On the downside, this is not completely secure because the values $r_i + x_iy_i$ will not be uniformly distributed and therefore may leak information about x_iy_i. Commonly, we would define a security parameter σ so that r is σ bits longer than x_iy_i to hide it with probability $1 - 2^{-\sigma}$. The hiding properties of this randomisation are addressed by Theorem 5.2.2.

Therefore, in Paillier Multiplication with base-B numbers we achieve

$$\gamma_{len} = \frac{|M| \cdot m}{|N|} \leq \frac{1}{2}, \ \gamma_{net} = \frac{|M| \cdot m}{3 \cdot 2|N|} \leq \frac{1}{12}, \ \gamma_{comp} = \frac{|M| \cdot m}{4|N| + 2|M| + 2} \leq \frac{1}{10} \ ,$$

because we always get the correct outcome $w_1 + w_2 < N$. However, the bounds are only achievable in very insecure settings where we do not use the randomisers.

The main benefit of this approach is that there are no restrictions to the length of the packed type because we can always pick a suitable B. The following Algorithm 15 uses a version of this linear packing.

5.2.2 Triple generation with partial base-B packing

Protocol B-Triples in Algorithm 15 is the protocol by Thomas Schneider as used in [48]. In uses the ideas of B-ary packing, however, these are used slightly differently from the previous initialisation. One party sends x_i separately and the other responds with $\sum B^i x_i y_i + r_i$. This is not very communication-efficient, but avoids the occurrence of elements $x_i y_j$ in the packed response.

The main drawback of this protocol is that although we use randomness to blind the encrypted response, we actually have secret sharing over integers in the response v and it may leak some information about the shares. The main idea of this protocol is to reduce the network communication compared to the basic Paillier Multiplication. The gain comes from the fact that the responder does not need to send back a ciphertext for each of the triple, but packs elements into one ciphertext.

Let σ denote a statistical security parameter. The efficiency of this protocol depends on σ and the length $k = |M|$ of the initial single values. Variable ℓ stands for the length of the packed values. We have $\ell = 2k + 2 + \sigma$ and thus, it is possible to pack $\lfloor |N|/\ell \rfloor$ responses into one ciphertext. This length also defines the randomness that is used to hide the actual value of the inputs.

Algorithm 15 Generating $m < |N|/\ell$ triples with B-ary packing (B-Triples)

```
Setup: Security parameter $\sigma$, modulus length $k = |M|$, $\ell = 2k + 2 + \sigma$
Data: Arrays of shared secrets $[\![x_1]\!]_M, \ldots, [\![x_m]\!]_M$ and $[\![y_1]\!]_M, \ldots, [\![y_m]\!]_M$
Result: Array $[\![w_1]\!]_M, \ldots, [\![w_m]\!]_M$, where $[\![w_i]\!]_M = [\![x_i \cdot y_i]\!]_M$
 1: $\mathcal{CP}_1$ sends $\mathsf{Enc}_{pk}(x_{1,1}), \ldots, \mathsf{Enc}_{pk}(x_{1,m})$ to $\mathcal{CP}_2$
 2: $\mathcal{CP}_1$ sends $\mathsf{Enc}_{pk}(y_{1,1}), \ldots, \mathsf{Enc}_{pk}(y_{1,m})$ to $\mathcal{CP}_2$
 3: $\mathcal{CP}_2$ fixes $r = 0$, $e = 0$, $\mathsf{Enc}_{pk}(e)$
 4: for $i \in \{1, \ldots, m\}$ do
 5:     $\mathcal{CP}_2$ generates a random $r_i \leftarrow \{0,1\}^{2\cdot k+1+\sigma}$
 6:     $\mathcal{CP}_2$ computes $\mathsf{Enc}_{pk}(t_i) = \mathsf{Enc}_{pk}(x_{1,i} \cdot y_{2,i} + y_{1,i} \cdot x_{2,i})$
 7:     $\mathcal{CP}_2$ computes $r = r \cdot 2^\ell + r_i$
 8:     $\mathcal{CP}_2$ computes $\mathsf{Enc}_{pk}(e) = \mathsf{Enc}_{pk}(e \cdot 2^\ell + t_i)$
 9:     $\mathcal{CP}_2$ computes $w_{2,i} = x_{2,i} \cdot y_{2,i} - r_i \bmod M$
10: end for
11: $\mathcal{CP}_2$ encrypts $\mathsf{Enc}_{pk}(r)$
12: $\mathcal{CP}_2$ sends $\mathsf{Enc}_{pk}(v) = \mathsf{Enc}_{pk}(e + r)$ to $\mathcal{CP}_1$
13: $\mathcal{CP}_1$ decrypts and unpacks single values $v_1||v_2||\ldots||v_m = \mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(v))$
14: for $i \in \{1, \ldots, m\}$ do
15:     $\mathcal{CP}_1$ computes $w_{1,i} = x_{1,i} \cdot y_{1,i} + v_i \bmod M$
16: end for
17: return $[\![c]\!]$
```

For example, choosing $|N| = 2048$, $\sigma = 112$ and $k = 32$ as in [48] allows us to pack 11 elements of 32-bits to 2048-bit modulus. The main strength of this approach is that there is no need for share conversion as the fixed length of elements also ensures that the response does not overflow N and the algorithm always yields correct triples modulo M.

Theorem 5.2.1. *Protocol* B-Triples *in Algorithm 15 for generating Beaver triples with partial B-ary packing is correct.*

Proof. We need to show that $w_i = x_i \cdot y_i$ for all $i \in 1, \dots, m$. For this, it is crucial to analyse what happens in the packing. We define $B = 2^\ell$ and the cycle computes the response

$$\begin{aligned} e =& B^{m-1} \cdot (x_{1,1} \cdot y_{2,1} + x_{2,1} \cdot y_{1,1}) + \ldots + B \cdot (x_{1,m-1} \cdot y_{2,m-1} + x_{2,m-1} \cdot y_{1,m-1}) + \\ &+ (x_{1,m} \cdot y_{2,m} + x_{2,m} \cdot y_{1,m}) \end{aligned}$$

and a randomness

$$r = B^{m-1} \cdot r_1 + \ldots + B \cdot r_{m-1} + r_m \enspace .$$

Finally, the sum of these is computed and sent back to $\mathcal{CP}_1$ who can decrypt and disassemble it to blocks

$$v_i = x_{1,i} \cdot y_{2,i} + x_{2,i} \cdot y_{1,i} + r_i \enspace .$$

We assume that the elements x_i and y_i have a length of k bits, thus, each $x_{1,i} \cdot y_{2,i} + x_{2,i} \cdot y_{1,i}$ is at most $2k+1$ bits long. The randomness r is defined as $2k+1+\sigma$ bits, which means that the value v_i is at most $2k+2+\sigma$ bits and if we define $\ell > 2k+2+\sigma$ then each of these values fits into an ℓ-bit slot. Thus, the packed value can always be restored correctly if this value is smaller than the encryption modulus, which is ensured as $m \cdot \ell < |N|$.

It remains to show that $w_i = x_i \cdot y_i$, which can be easily seen, as

$$\begin{aligned} w_i = w_{1,i} + w_{2,i} &= x_{1,i} \cdot y_{1,i} + v_i + x_{2,i} \cdot y_{2,i} - r_i \\ &= x_{1,i} \cdot y_{1,i} + x_{1,i} \cdot y_{2,i} + x_{2,i} \cdot y_{1,i} + r_i + x_{2,i} \cdot y_{2,i} - r_i \\ &= (x_{1,i} + x_{2,i}) \cdot (y_{1,i} + y_{2,i}) = x_i \cdot y_i \enspace . \end{aligned}$$

□

Theorem 5.2.2. *Protocol* B-Triples *in Algorithm 15 for generating Beaver triples with B-ary packing is statistically secure against a corrupted $\mathcal{CP}_1$ and computationally secure against a corrupted $\mathcal{CP}_2$, given a (t, ε)-IND-CPA secure cryptosystem and statistical security constant σ for packing.*

Proof sketch. The simulator can adjust the outputs of either party to correspond to the results from the TTP as in the Paillier Multiplication. In the following we consider, how the simulator can simulate the communication of this protocol.

Corrupted $\mathcal{CP}_1$. The simulator can replace each v_i with an encryption of a random element $r^* \leftarrow R$ where $R = \{t_{max}/2, \ldots, 2^{2k+\sigma+1} + t_{max}/2\}$ where $t_{max} = 2^{2k+1} - 2^{k+2} + 2 < 2^{2k+1}$ is the maximal value that t_i might have. It is easy to see that the minimal value that t_i might have is 0, therefore, $t_{max}/2$ is the median value of t_i. The advantage of the adversary in this case is bounded by the statistical distance of $v = t_i + r_i$ and

r^*. The value $v = t_i + r_i$ is uniformly distributed in $T = \{t_i, \dots, t_i + 2^{2k+\sigma+1}\}$ where $0 \le t \le t_{max}$. Hence, the statistical distance is

$$\begin{aligned}
\mathsf{sd}(r^*, v) &= \frac{1}{2} \cdot \sum_{x \in R \cup T} \Big| \Pr[r^* = x] - \Pr[v = x] \Big| \\
&= \frac{1}{2} \cdot \Big(\sum_{x \in R \cap T} \Big| \Pr[r^* = x] - \Pr[v = x] \Big| + \sum_{x \in T \setminus R} \Big| \Pr[r^* = x] - \Pr[v = x] \Big| + \\
&+ \sum_{x \in R \setminus T} \Big| \Pr[r^* = x] - \Pr[v = x] \Big| \Big) \\
&= \frac{1}{2} \cdot \Big(\sum_{x \in T \cap R} \Big| \frac{1}{|R|} - \frac{1}{|T|} \Big| + \sum_{x \in T \setminus R} \Big| 0 - \frac{1}{|T|} \Big| + \sum_{x \in R \setminus T} \Big| \frac{1}{|R|} - 0 \Big| \Big) \ .
\end{aligned}$$

It is possible to continue this evaluation as we know that $|R| = |T| = 2^{2k+\sigma+1} + 1$ and that $0 \le |R \setminus T| = |T \setminus R| \le t_{max}/2$. Therefore we can give an upper bound to $\mathsf{sd}(r^*, v)$ as

$$\begin{aligned}
\mathsf{sd}(r^*, v) &= \frac{1}{2} \cdot \Big(\sum_{x \in T \cap R} 0 + \sum_{x \in T \setminus R} \frac{1}{2^{2k+\sigma+1} + 1} + \sum_{x \in R \setminus T} \frac{1}{2^{2k+\sigma+1} + 1} \Big) \\
&\le \frac{1}{2} \cdot t_{max} \cdot \frac{1}{2^{2k+\sigma+1} + 1} = \frac{t_{max}}{2^{2k+\sigma+2} + 2} \le \frac{2^{2k+1}}{2^{2k+\sigma+2} + 2} \le 2^{-\sigma} \ .
\end{aligned}$$

Therefore, sending an encryption of a random element from R is statistically $2^{-\sigma}$ indistinguishable from a correctly computed reply. It follows that the simulator can pick r^* such that the output of the corrupted $\mathcal{CP}_1$ is as desired.

Corrupted $\mathcal{CP}_2$. The simulator should send the encryptions of $x_{1,i}$ and $y_{1,i}$ to $\mathcal{CP}_2$, which it can do efficiently by sending the encryptions of random values. Due to the IND-CPA security, the simulation is at distance $2m \cdot \varepsilon$ from the real protocol run for any t-time adversary. □

The efficiency of this protocol depends on the chosen parameters, but for now, we write it out in terms of $m < |N|/\ell$ and $|M|$, where $\ell = 2|M| + \sigma + 2$. In total, we learn $|M| \cdot m$ bits of valid triples which gives

$$\gamma_{len} = \frac{|M| \cdot m}{|N|} \le \frac{1}{2} - \frac{\sigma \cdot m}{2|N|} - \frac{m}{|N|} < \frac{1}{2} \ ,$$

where we get the estimate as we know that $\ell \cdot m < |N|$ which gives

$$|M| \cdot m < \frac{|N|}{2} - \frac{\sigma \cdot m}{2} - m \ .$$

Communication-wise this protocol gives

$$\gamma_{net} = \frac{|M| \cdot m}{(2m+1) \cdot 2|N|} \le \frac{1}{4 \cdot (2m+1)} \ .$$

Finally, in terms of computation this protocol is also quite expensive due to the separate encryption operations resulting in

$$\gamma_{comp} = \frac{|M| \cdot m}{(2m+2) \cdot |N| + 2m \cdot |M| + m \cdot \ell + 2m + 1} \le \frac{1}{4m+4} \ .$$

5.2.3 Packing using the Chinese remainder theorem

We can also use the Chinese remainder theorem for packing several elements into one ciphertext. However, the used mechanism is quite different from the previously described B-ary packing and can only be used, if we are using elements with pairwise coprime moduli p_i. By definition, CRT can be used to combine all those single random values modulo p_i for a modulus

$$M = p_1 \cdot \ldots \cdot p_k$$

and execute the triple generation protocol to obtain the corresponding third triple elements modulo M.

For example, we start with values x_i and y_i and we interpret them as

$$\begin{cases} x = x_i \bmod p_i \\ y = y_i \bmod p_i \end{cases} .$$

Then, we combine them using CRT to learn $x \bmod M$ and $y \bmod M$ which are inputs to the triple generation protocol for learning $xy \bmod M$. We know that, by definition, we have

$$\begin{cases} xy \quad = x_i y_i \bmod p_i \end{cases} ,$$

where we are interested in learning the shares for $x_i y_i$. The CRT allows us to reduce the final result respectively for all moduli p_i to learn the third triple element for all initial random value pairs. Therefore, we can learn $x_i y_i$ from $xy \bmod M$ as

$$x_i y_i = xy \bmod p_i .$$

Packing with CRT enables us to get exactly $|M|$-bit triples from one execution of the triple generation protocol. However, we could also use this with the idea to later convert all these shares of different moduli to one shared modulus as discussed in Section 5.3.2. This would result in approximately $\frac{|M|}{2}$-bit triple elements.

This packing can be trivially well used with the Paillier Multiplication protocol because all we need is to learn a valid triple modulo M to be able to get all separate triples modulo p_i. However, we have to ensure that, in this case, $M^2 < N$. Therefore, we can get approximately $\frac{|N|}{2}$-bit triples when using a modulus M. Using Paillier Multiplication with CRT packing gives exactly the same yields as it does for any arbitrarily chosen modulus M as given in Section 5.1. We need to do additional computations for packing and unpacking, but these do not significantly affect our computational cost as they are not operations on ciphertexts.

5.3 Share conversion

Share conversion is the process of transforming a shared value $[\![x]\!]_M$ for one fixed modulus M to a valid share $[\![v]\!]_{M^*}$ of the same secret value $x = v$ under a different modulus M^*. It will be necessary as we use the Paillier cryptosystem that has homomorphic properties modulo N for generating triples of a generic modulus M. This section focuses on transforming $[\![x]\!]_2$ to $[\![v]\!]_M$ and $[\![x]\!]_N$ to $[\![v]\!]_M$, where $N > M$ that are needed for triple generation. In addition, we stress additional restrictions that must be met in order to successfully convert the triple from Paillier Multiplication so that the multiplicative relation still holds after the conversion.

5.3.1 Converting binary shares to any modulus

A protocol for obtaining $[\![v]\!]_M$ from $[\![x]\!]_2$ is a simpler subcase of all conversion protocols as x in $[\![x]\!]_2$ has only two potential values. Thus, if $x = 0$ we should have $[\![v]\!]_M$ as $v_1 \leftarrow \mathbb{Z}_M$ and $v_2 = M - v_1$ or, correspondingly, $v_2 = M - v_1 + 1$ for $x = 1$.

		$\mathcal{CP}_1$ input	
		$x_1 = 0$	$x_1 = 1$
$\mathcal{CP}_2$	$x_2 = 0$	$v_2 = M - v_1$	$v_2 = M - v_1 + 1$
input	$x_2 = 1$	$v_2 = M - v_1 + 1$	$v_2 = M - v_1$

Table 5.1: Oblivious transfer for share conversion $[\![x]\!]_2$ to $[\![v]\!]_M$

Such a replacement of the shares can be achieved using oblivious transfer (OT). The *1-out-of-2* OT is a communication protocol for transporting information so that the sender does not know which of its two inputs it forwarded to the receiver. In addition, the receiver is only able to learn one of the sender's inputs per protocol. The idea for share conversion is that the sender defines $v_1 \leftarrow \mathbb{Z}_M$ and sends $v_2 = M - v_1$ or $v_2 = M - v_1 + 1$ to the receiver based on the value of x. More precisely, we do not open the value x, but do OT based on the shares of $[\![x]\!]_2$ being equal ($x = 0$) or not ($x = 1$). $\mathcal{CP}_2$ learns the outcomes as specified in Table 5.1 for each potential input combination and $\mathcal{CP}_1$ always outputs v_1.

For the efficiency analysis, we use the well known AIR OT protocol [1] for the *1-out-of-2* case. We use it with the Paillier cryptosystem as defined in Algorithm 16. Any OT protocol could be used in future implementations, but we will use this due to its simplicity to analyse the efficiency of using this share conversion in the triple generation protocols.

Algorithm 16 Aiello-Ishai-Reingold oblivious transfer

```
Setup: Receiver has defined a Paillier keypair $(pk, sk)$, which defines a modulus $N$
Data: Receiver has input $x \in \{0, 1\}$, Sender has two secrets $s_0$ and $s_1$
Result: Receiver learns $s_x$
1: Receiver computes and sends $c = \mathsf{Enc}_{pk}(x)$ to the Sender
2: Sender generates $r_0, r_1 \leftarrow \mathbb{Z}_M$
3: Sender computes and sends $c_0 = c^{r_0} \cdot \mathsf{Enc}_{pk}(s_0)$ to the Receiver
4: Sender computes and sends $c_1 = (c \cdot \mathsf{Enc}_{pk}(-1))^{r_1} \cdot \mathsf{Enc}_{pk}(s_1)$ to the Receiver
5: Receiver decrypts $s_x = \mathsf{Dec}_{sk}(c_x)$
```

The idea of AIR OT is straightforward, as $c_0 = \mathsf{Enc}_{pk}(0 \cdot r_0 + s_0)$, if $x = 0$ and analogously, if $x = 1$ then $c_1 = \mathsf{Enc}_{pk}(0 \cdot r_1 + s_1)$. The role of the randomiser r_i is to ensure that the other secret is not leaked. For example, in case the query was $x = 0$, then c_1 is an encryption of a random value $\mathsf{Enc}_{pk}(-1 \cdot r_1 + s_1)$ and does not reveal s_1.

5.3.2 Problems with converting the third triple element

We can expect the first two elements x and y of the triple to be generated according to some fixed modulus M. However, the triple generation protocol may change the modulus for the outcome $w = x \cdot y$. The possibility to convert this outcome back to the original modulus means that we can actually generate triples for any modulus. However, there are losses in how many bits we use in the multiplication and afterwards receive as triples.

One place where such share conversion is needed is from the Paillier modulus to our chosen modulus M. At some point in triple generation, we use Paillier Multiplication, where the result will be given modulo N. Namely, if we have some uniformly generated values $[\![x]\!]_M$ and $[\![y]\!]_M$ modulo M, then we know that $x \cdot y < N$ if $M^2 < N$. Hence, we can avoid modular reductions in the product. However, working with additive share representation requires more care, as we actually have

$$(x_1 + x_2) \cdot (y_1 + y_2) < N \ ,$$

where $x_1 + x_2 < 2M$ and

$$(x_1 + x_2) \cdot (y_1 + y_2) < 4M^2 \ .$$

Hence, we require that $4M^2 < N$. With this restriction to initial values, we can apply general share conversion to the third triple element and achieve a multiplicative triple with respect to modulus M.

The main challenge in the share conversion is to differentiate between having either

$$w_1 + w_2 \geq N \text{ or } w_1 + w_2 < N \ .$$

The latter case means that if $x \cdot y = w_1 + w_2 \bmod N$ then also $x \cdot y = w_1 + w_2 \bmod M$ and share conversion can be obtained by converting both $w_i \bmod M$ separately. However, the former gives us $x \cdot y = w_1 + w_2 - N \bmod M$ and means that we have to check for this error when converting triples. Achieving this error correction is one of the important goals of the triple generation algorithms based on Paillier Multiplication. The main idea is that the parties can collaboratively decide if they have $w_1 + w_2 \geq N$ or $w_1 + w_2 < N$ and in the former case they can fix the shares as $w = w - N$ before modular reduction.

There are many different ways for performing this check. For example, one possibility is to do computations as in Triple Verification and check for the value of h. It is straightforward to fix values of h that mean that a triple was correct or had $w_1 + w_2 \geq N$. For security, h should not be published and the correction could be done based on the value of h using oblivious transfer. A more efficient method is introduced as part of Algorithm 17 and a more general idea analogous to that is also specified later in Chapter 6 as Algorithm 20. The ideas in Algorithm 17 and Algorithm 20 can be used to perform general conversion from $[\![x]\!]_N$ to $[\![v]\!]_M$, independently of the fact that we are working with a multiplication result.

In addition, share conversion of the third triple element can be used with CRT packing to get results that all use the same modulus. For example, the initial shares of x_i and y_i are fixed with relation to some modulus P and then we choose primes $p_i > P^2$ and interpret these shares each for a different modulus p_i. After the triple generation we learn $xy \bmod p_i$, but as previously, we know that $xy < p_i$ and now converting them from modulus p_i to P is the same as converting from N to M after Paillier Multiplication. This approach decreases the efficiency of the CRT packing by half, but helps to get rid of the need to use different moduli.

5.3.3 Triple generation with share conversion

The problem with using Paillier Multiplication for triple generation in a straightforward manner was mentioned in Section 5.3.2. It means that occasionally this protocol gives a valid triple for moduli other than the Paillier modulus, but sometimes the result is invalid. This section introduces a way that uses the Paillier Multiplication algorithm, but adds additional checks to always get the correct result for a chosen modulus.

The idea of ShareConv-Triples in Algorithm 17 is to use a modulus 2 to test if the shares of the third element $w_1 + w_2 < N$ or overflow N and base the share conversion on the result of this check. This clearly leaks some information about the result as we need to declassify the least significant bit of the inputs, but this may not be an issue for all use-cases. The idea is that by declassifying the least significant bits of x and y we learn what the parity of w should be without any modular reductions. It could be used with CRT packing trivially as long as we do not use modulus 2 in the packing. Using this idea exactly like this is actually not secure as it leaks the least significant bits of x and y.

However, we can easily avoid the leakage, by actually using an odd modulus P that is one bit shorter that maximum length of M in Paillier Multiplication. In such case we just define

$$x_i = 0 \bmod 2 \text{ and } y_i = 0 \bmod 2$$

and use the CRT to compute the representation of x and y for modulus $2P$. We can learn the correct triple modulo P by reducing the final shares of $[\![w]\!]_{2P}$ modulo P.

According to the CRT, the multiplicative relation modulo $2P$ holds exactly, if it holds for all its prime divisors, including 2. The latter means that by checking the relation modulo 2, we actually verify that it holds for modulus $2P$.

Algorithm 17 Triple generation with share conversion (ShareConv-Triples)

```
Setup: Paillier keypair $(pk, sk)$ with modulus $N$
       $2|P| + 5 < |N|$ and $P$ is odd
Data: Shared secrets $[\![x]\!]_P, [\![y]\!]_P$
Result: Third triple element $[\![w]\!]_P$ where $w = xy \bmod P$
1: $\mathcal{CP}_i$ uses CRT with inputs $x_i = 0 \bmod 2$ and $x_i = x_i \bmod P$ to learn $x_i \bmod 2P$
2: $\mathcal{CP}_i$ uses CRT with inputs $y_i = 0 \bmod 2$ and $y_i = y_i \bmod P$ to learn $y_i \bmod 2P$
3: $\mathcal{CP}$ compute $[\![w]\!]_{2P}$ from $[\![x]\!]_{2P}, [\![y]\!]_{2P}$ with Paillier Multiplication
4: $\mathcal{CP}_i$ computes $c_i = w_i \bmod 2$
5: $\mathcal{CP}$ convert $c = c_1 + c_2$ from $[\![c]\!]_2$ to $[\![c]\!]_{2P}$
6: $\mathcal{CP}_i$ computes $w_i = w_i - N \cdot c_i \bmod 2P$ to correct the potential mistakes
7: $\mathcal{CP}_i$ fixes $w_i = w_i \bmod P$ to get the correct final modulus
```

Theorem 5.3.1. *The* ShareConv-Triples *protocol in Algorithm 17 for generating Beaver triples with simple share conversion is correct assuming the correctness of share conversion from $[\![c]\!]_2$ to $[\![c]\!]_{2P}$ and* Paillier Multiplication.

Proof. For correctness, we need to analyse the meaning of c in this algorithm. We assume that the share conversion from $[\![c]\!]_2$ to $[\![c]\!]_{2P}$ is correct and Paillier Multiplication always gives either $w = w_1 + w_2$ or $w = w_1 + w_2 - N$. By definition, $c \in \{0, 1\}$ and $c = c_1 + c_2$, where

$$c_1 = w_1 \bmod 2 \text{ and } c_2 = w_2 \bmod 2 \ .$$

We know that as both x and y are defined as being even then w also has to be even and we have

$$c = w_1 + w_2 \bmod 2 \ .$$

We also know that N is odd and therefore w can be even if either $w_1 + w_2$ is even which means $w_1 + w_2 < N$ or if $w_1 + w_2$ is odd which gives $w_1 + w_2 \geq N$. Hence, if $c = 0$, then both w_i have the same parity and $w_1 + w_2$ is even. Knowing that w has to be even gives us $w = w_1 + w_2 < N$. On the other hand, if $c = 1$, then w_i have different parity and $w_1 + w_2 \geq N$ as integer is odd. Thus, we have $w = w_1 + w_2 - N$.

This clearly corresponds to how we compute w as $w_i = w_i - N \cdot c_i \bmod 2P$ gives us

$$w = w_1 + w_2 = w_1 - N \cdot c_1 + w_2 - N \cdot c_2 = w_1 + w_2 - c \cdot N \bmod 2P \ .$$

The final modular conversion $w_i = w_i \bmod P$ is correct according to the CRT. □

Theorem 5.3.2. *The* ShareConv-Triples *protocol in Algorithm 17 for generating Beaver triples with simple share conversion is secure, assuming the security of binary share conversion and* Paillier Multiplication.

Proof sketch. Te security follows trivially, as this protocol is just a combination of share conversion, Paillier Multiplication and local operations. □

Protocol ShareConv-Triples works trivially well with CRT packing, if the packing does not include modulus 2, because we require P to be odd. There is no need to use this with B-ary packing as in that case, the randomisation in Paillier Multiplication is defined so that we always get $w_1 + w_2 < N$ and there is no need for additional correction.

This protocol is more efficient than the previous, enabling us to get one $\frac{|N|}{2} - 2$-bit triple at the cost of one Paillier multiplication and error correction. However, it proposes additional restrictions to the choice of P, which has to be odd. In terms of achieved bits we clearly have

$$\gamma_{len} = \frac{|P|}{|N|} \leq \frac{1}{2} \ .$$

In terms of communication we have to take into account that we do Paillier Multiplication and share conversion, we currently use share conversion with AIR OT for comparison. This gives us

$$\gamma_{net} = \frac{|P|}{6 \cdot 2|N|} \leq \frac{1}{24}$$

because AIR OT has the same communication cost as Paillier Multiplication. Finally, in terms of computation, we also require ciphertext operations in both Paillier Multiplication and OT, which gives

$$\gamma_{comp} = \frac{|P|}{10|N| + 2(|P| + 1) + 5} \leq \frac{1}{22} \ .$$

5.4 Comparison of proposed triple generation ideas

In a real life setting, we would like to generate a set of triples for some fixed size. This section gives a comparison of the ideas from this chapter for the case where we are interested in learning triples for $M = 2^{32}$, we analyse the case for $|N| = 2048$. For packing, we assume that $M = 2^{32}$ for B-Triples and that we use 33-bit primes for CRT packing. In addition, for B-Triples we define $\sigma = 112$, which gives $\ell = 178$ and $m = 11$.

In B-ary packing we assume the same setup and basic packing with

$$x = B^{m-1}x_m + \ldots + x_1$$
$$y = B^{(m-1)\cdot m}y_m + B^m y_2 + \ldots + y_1 \ .$$

We need $|B| = 2 \cdot |M| + \sigma = 176$ and $B^{m^2} < N$, which enables us to pack $m = 3$ elements, because we need $m^2 < 11$. With CRT packing, we can pack at most 31 elements if we choose small primes, or 30 for general 33-bit primes, because we need that $4M^2 < N$. In addition, for ShareConv-Triples we can not use exactly $P = 2^{32}$, but we expect that $|P| = 32$ and therefore $|2P| = 33$.

Protocol	γ_{len}	γ_{net}	γ_{comp}
Paillier Multiplication	$\frac{1}{128} \approx 0.008$	$\frac{1}{768} \approx 0.001$	$\frac{8}{4129} \approx 0.002$
B-ary packing	$\frac{3}{64} \approx 0.047$	$\frac{1}{128} \approx 0.008$	$\frac{48}{4129} \approx 0.012$
CRT packing	$\frac{31}{128} \approx 0.242$	$\frac{31}{768} \approx 0.04$	$\frac{248}{4129} \approx 0.06$
B-Triples	$\frac{11}{64} \approx 0.171$	$\frac{11}{2944} \approx 0.004$	$\frac{352}{51837} \approx 0.007$
ShareConv-Triples	$\frac{1}{64} \approx 0.016$	$\frac{1}{768} \approx 0.001$	$\frac{32}{20551} \approx 0.002$
CRT packing	$\frac{1023}{2048} \approx 0.5$	$\frac{341}{8192} \approx 0.042$	$\frac{992}{22471} \approx 0.044$

Table 5.2: Comparison of Beaver triple generation protocols

The approximate values in Table 5.2 are given simply for making the comparison of these results more straightforward. Trivially, Paillier Multiplication with B-ary packing is three times more efficient than without as we can pack exactly 3 elements. However, there is an additional 2 times gain as the output triples are always correct. Packing with CRT is exactly 31 times more efficient that plain Paillier Multiplication. Though, the main trouble with these two cases is that although these are the average ratios assuming that the triple is rightfully corrected, we should also verify that they are correct. For example, if we use Triple Verification then learning one correct triple actually has the cost of two unverified triples.

Partial packing in B-Triples has significantly better packing count than using basic B-ary packing which results in better ratio of γ_{len}. The loss in other parameters is small enough to give this algorithm precedence over Paillier Multiplication with packing.

Basic ShareConv-Triples is close to Paillier Multiplication as expected, as we always get the correct triple, but the correction has approximately the same cost as the multiplication. The ratio for γ_{len} of ShareConv-Triples with CRT packing is actually ideal because $\frac{1}{2}$ is the best limit we can achieve with our current ideas about arbitrary modulus in Paillier Multiplication. This packing ratio also affects the efficiency of network usage as well as computations and clearly makes this the most efficient of our ideas resulting in bounds close to the theoretical ones.

In Chapter 7, we also give results for the implementation of ShareConv-Triples with CRT packing and B-Triples, as they are the more efficient and easier to use protocols according to given comparison. However, Table 5.2 also indicates that we possibly should consider only using Paillier Multiplication and CRT packing in cases where we can increase the probability of receiving a correct triple so that we are more likely to pass the Triple Verification check. It is especially meaningful if we need to perform Triple Verification to check for some other possible errors as well. In the follow-up work, we should compare the efficiency of this approach to ShareConv-Triples with CRT packing and different OT protocols.

Chapter 6

Symmetric two-party computation

This section introduces our ideas for setting up symmetric two-party computation. Currently, this section consists of the online phase, which is derived quite directly from the share representation and ideas from the asymmetric protocol set in Chapter 4. The question of achieving reasonably efficient precomputation of Beaver triples is currently unsolved, but this section give hints on how we might use the protocols from Chapter 5.

6.1 Protection domain setup

We consider additive secret sharing in a ring $\mathbb{Z}_p$ for some modulus p. Party $\mathcal{CP}_i$ defines a MAC key k_i so that $z^{(i)} = k_i \cdot x \bmod p$. It is clear from Section 2.1.9 that we can obtain a secure protection scheme for a prime p and moduli with only suitably large prime divisors. In case we use a modulus with a short bitlength, we can allow each party to define several keys to enhance the security. This way, we could achieve the necessary security level for any desired threshold, independently of the modulus. However, each additional key will make the computations less efficient. It is currently an open question, if a suitable efficient MAC algorithm could be obtained for other moduli. All arithmetic in this scheme is with respect to the modulus p. In the following, the security proofs give security guarantees with respect to using a prime modulus p.

We propose a share representation as

$$[\![x]\!]_p = \langle \Delta, x_1, x_2, z_1^{(1)}, z_2^{(1)}, z_1^{(2)}, z_2^{(2)} \rangle \ ,$$

where $x = x_1 + x_2 + \Delta$ and Δ is the public modifier. The remaining values belong to the MAC tags as $z_1^{(1)} + z_2^{(1)} = k_1 \cdot (x_1 + x_2)$ and $z_1^{(2)} + z_2^{(2)} = k_2 \cdot (x_1 + x_2)$. Both parties know Δ and, in addition, $\mathcal{CP}_i$ has values x_i, $z_i^{(1)}$ and $z_i^{(2)}$.

It is straightforward to obtain an addition protocol (Addition) as both parties can just locally add their shares to get their share of the sum. Analogously, we get protocols for subtraction (Subtraction) and multiplication with a public value (Constant Multiplication). Addition with a public value (Constant Addition) still only requires modifying the common value Δ. Thus, a public value v can be seen as

$$[\![v]\!]_p = \langle \Delta = v, v_1 = 0, v_2 = 0, z_1^{(1)} = 0, z_2^{(1)} = 0, z_1^{(2)} = 0, z_2^{(2)} = 0 \rangle \ .$$

For the sake of achieving protocols for communication with non-computing parties $\mathcal{IP}_i$ and $\mathcal{RP}_i$ and precomputation, we also assume that both computing parties $\mathcal{CP}_i$ have defined their own Paillier keypair (pk_i, sk_i) where pk_i is also known by the other parties and defines a modulus N_i. In addition, they have published a commitment $\mathsf{Enc}_{pk_i}(k_i) = (\!|k_i|\!)_{pk_i}$. The inconvenience in this is that our otherwise statistically secure setup becomes computationally secure, depending on the IND-CPA security of the cryptosystem that hides k_i.

We occasionally use the notation $\mathcal{CP}_i$ and $\mathcal{CP}_j$ where the idea is that $i \neq j$. For example, to specify that $\mathcal{CP}_i$ sends something to the other party $\mathcal{CP}_j$ where the meaning is that both computing parties send something to the other. We occasionally use an abbreviation $\mathcal{CP}$, that should be read as *computing parties*, to denote that both $\mathcal{CP}_i$ execute some sub-protocol together.

6.2 Publishing shared values

Due to the symmetric setup of the protection domain, we can give a general publishing protocol Publish-$\mathcal{CP}_i$ (Algorithm 18) to open share to party $\mathcal{CP}_i$. Party $\mathcal{CP}_i$ learns the correct result if the verification succeeds and should otherwise abort the protocol. $\mathcal{CP}_i$ is the party who should receive the output and by $\mathcal{CP}_j$ we mean the other party who sends its shares. We can combine two instances of this protocol to simultaneously declassify to both computing parties (Publish-both-$\mathcal{CP}_i$).

Algorithm 18 Publishing a shared value to $\mathcal{CP}_i$ (Publish-$\mathcal{CP}_i$)

```
Data: Shared secret $[\![x]\!]_p$
Result: $\mathcal{CP}_i$ learns the value $x$
1: $\mathcal{CP}_j$ sends $x_j$ and $z_j^{(i)}$ to $\mathcal{CP}_i$
2: $\mathcal{CP}_i$ verifies $z_1^{(i)} + z_2^{(i)} = k_i \cdot (x_1 + x_2)$
3: return $\mathcal{CP}_i$ outputs $x_1 + x_2 + \Delta$
```

Theorem 6.2.1. *Protocol* Publish-$\mathcal{CP}_i$ *for publishing a shared value to one party is correct.*

Proof. For correctness, we need that $x = x_1 + x_2 + \Delta$, which is trivially true in case the verification process is correct. In the case of honest participants, we know that the verified equations must hold by the definition of the shares. □

Theorem 6.2.1. *Protocol* Publish-$\mathcal{CP}_i$ *for publishing a shared value to one party is computationally secure with additional statistical* $\frac{1}{p}$ *error probability.*

Proof sketch. This proof is analogous to the part Publish-$\mathcal{CP}_1$ in Theorem 6.2.1. In the asymmetric setting, $\mathcal{CP}_1$ also verified the correctness using MAC tags. The computational requirement follows from the fact that $\mathcal{CP}_j$ knows $(\!|k_i|\!)_{pk_i}$. □

As in the asymmetric case, we have a simple possibility that we declassify an element to the computing parties, who check the MAC and then forward the declassified results to the result parties $\mathcal{RP}_i$. The result parties only have to verify that both computing parties forwarded them the same declassification result and accept the output. Thus, we can easily obtain the protocol Publish-$\mathcal{CP}$&$\mathcal{RP}_i$. The security and correctness of this protocol result from those of Publish-$\mathcal{CP}_i$ protocol. This is an easy way to make the results publicly known, a way to open shares only to $\mathcal{RP}_i$ is discussed later in Section 6.4.

6.3 Receiving inputs from the input party

This section defines a standalone protocol to receive inputs from $\mathcal{IP}_i$. We do not yet have a Publish-$\mathcal{RP}_i$ protocol, therefore we can not use the common Classify-$\mathcal{IP}_i$ protocol from Chapter 3 and need to define something independent from other protocols. Our Classify-$\mathcal{IP}_i^\star$ protocol is given in Algorithm 19.

Algorithm 19 Receiving inputs from $\mathcal{IP}_i$ (Classify-$\mathcal{IP}_i^\star$)

```
Data: $\mathcal{IP}_i$ has a secret $x$
Result: $\mathcal{CP}_i$ have a shared secret $[\![x]\!]_p$
1: Round: 1
2:     $\mathcal{CP}_i$ fixes $\Delta = 0$
3:     $\mathcal{IP}_i$ generates $x_1 \leftarrow \mathbb{Z}_p$, $z_1^{(2)} \leftarrow \mathbb{Z}_{N_2}$, $z_2^{(1)} \leftarrow \mathbb{Z}_{N_1}$, $r_1, r_2 \leftarrow \mathbb{Z}_N^*$
4:     $\mathcal{IP}_i$ computes $x_2 = x - x_1 \bmod p$
5:     $\mathcal{IP}_i$ computes $c_1 = \mathsf{Enc}_{pk_1}(k_1)^{x_2} \cdot \mathsf{Enc}_{pk_1}(-z_2^{(1)}, r_1)$
6:     $\mathcal{IP}_i$ computes $c_2 = \mathsf{Enc}_{pk_2}(k_2)^{x_1} \cdot \mathsf{Enc}_{pk_2}(-z_1^{(2)}, r_2)$
7:     $\mathcal{IP}_i$ sends $x_i, z_i^{(j)}, c_i, r_j$ to $\mathcal{CP}_i$
8: Round: 2
9:     $\mathcal{CP}_i$ computes $z_i^{(i)} = k_i \cdot x_i + \mathsf{Dec}_{pk_i}(c_i) \bmod N_i$ to learn $[\![z^{(i)}]\!]_{N_i}$
10:    $\mathcal{CP}_i$ computes $c_j^* = \mathsf{Enc}_{pk_j}(k_j)^{x_i} \cdot \mathsf{Enc}_{pk_j}(-z_i^{(j)}, r_j)$ and send to $\mathcal{CP}_j$
11: Round: 3 (share conversion)
12:    $\mathcal{CP}$ collaboratively convert $[\![z^{(1)}]\!]_{N_1}$ to $[\![z^{(1)}]\!]_p$ and $[\![z^{(2)}]\!]_{N_2}$ to $[\![z^{(2)}]\!]_p$ using ShareConv
13: Verification:
14:    $\mathcal{CP}_i$ checks that $c_i = c_i^*$
15:    $\mathcal{CP}_i$ notifies $\mathcal{IP}_i$ about the verification outcome
```

The idea of this algorithm is similar to some tricks from the triple generation algorithms. Namely, the input party uses the encryptions of the MAC keys to share the tags modulo N and the computing parties convert them to correct modulus M.

Theorem 6.3.1. *The protocol* Classify-$\mathcal{IP}_i^\star$ *for receiving inputs from* $\mathcal{IP}_i$ *is correct, assuming the correctness of* ShareConv.

Proof sketch. The correctness of $x = x_1 + x_2 \bmod p$ is trivial from the definition. Similarly, the correctness of $z^{(i)} = k_i \cdot x \bmod N_i$ is straightforward from the algorithm description. Furthermore, the correctness of the verification is trivial, as by definition $c = c^*$ if all parties are honest. □

For security, we need that neither the computing parties nor the input party can cheat during the protocol. Cheating on the side of input party means that it tries to give inconsistent share representations. On the side of computing parties cheating means that they try to modify the shares they received. However, as in the asymmetric case, we still have the limitation that $\mathcal{IP}_i$ can not collude with either $\mathcal{CP}_i$.

Theorem 6.3.2. *The protocol* Classify-$\mathcal{IP}_i^\star$ *for receiving inputs from* $\mathcal{IP}_i$ *is computationally secure against corrupted* $\mathcal{CP}_i$ *and perfectly secure against corrupted* $\mathcal{IP}_i$*, if the adversary is allowed to corrupt at most one party, assuming a computationally secure share conversion protocol.*

Proof sketch. Similarly to the asymmetric case, the ideal functionality of this protocol receives x, x_1, $z_1^{(2)}$ and $z_2^{(1)}$ from the $\mathcal{IP}_i$. It then fixes the remaining $z_1^{(1)}$ and $z_2^{(2)}$ and forwards the shares to the computing parties. The computing parties can either accept or reject the shares. If either party rejects then the output of all parties is $\perp$, otherwise the computing parties learn their shares and $\mathcal{IP}_i$ learns that the input was received correctly.

The simulator for a corrupted $\mathcal{CP}_i$ at first receives x_i, $z_i^{(1)}$, and $z_i^{(2)}$ from the TTP. It then computes $c_i = \mathsf{Enc}_{pk}(z_i^{(i)}) \cdot (\![k_i]\!)_{pk_i}^{-x_i} = \mathsf{Enc}_{pk}(z_i^{(i)} - k_i \cdot x_i)$ and picks a r_j as an honest $\mathcal{IP}_i$ would. The simulator forwards x_i, $z_i^{(j)}$, c_i and r_j to the corrupted $\mathcal{CP}_i$ as a message from the $\mathcal{IP}_i$. By definition, $\mathcal{CP}_i$ can compute $z_i^{(i)}$ as originally defined by the TTP. In addition, it forwards c_i also as a message from the other computing party $\mathcal{CP}_j$. For the share conversion, it can act as an honest party by picking a random input for the case where it has to send the initial query. In addition, it can simulate the conversion for the case where it is the sender. Finally, the simulator gets the output *Continue* or *Failure* from the corrupted $\mathcal{CP}_i$ and forwards this to the TTP. Clearly, the simulator can make the output shares of $\mathcal{CP}_i$ the same as they would be in the ideal world and the output of $\mathcal{CP}_j$ is also the same, therefore, the outputs of the real and simulated ideal world coincide.

The simulator for the corrupted $\mathcal{IP}_i$ receives all the values x_i, $z_i^{(j)}$, c_i, r_i from the $\mathcal{IP}_i$. It checks that $c_i = (\![k_i]\!)_{pk_i}^{x_j} \mathsf{Enc}_{pk}(-z_j^{(i)}, r_j)$ for both c_i and forwards x, x_1, $z_1^{(2)}$ and $z_2^{(1)}$ to the TTP, if the check succeeds. In this case, it gives the output *Continue* or *Failure*, that it receives from the TTP, back to the corrupted $\mathcal{IP}_i$. Otherwise, if the check did not pass, it gives *Failure* to both the corrupted $\mathcal{IP}_i$ and the TTP. Clearly, the check that the simulator does for c_i is sufficient to check that the $\mathcal{IP}_i$ gives correct inputs. In addition, the final states of the ideal and real world coincide as in case the sharing succeeds the shares are chosen by the $\mathcal{IP}_i$ and parties have also seen these shares in case the sharing does not succeed. □

This protocol does not give the anti-framing property, because in the end the computing parties have not verified that the other party has the value $z_i^{(i)}$ that it needs to very the tag. To achieve this property we must include corresponding zero-knowledge proofs as a part of this protocol. We should also include the proofs that c_i is correctly computed in order to achieve security against collaborating pairs $\mathcal{CP}_j$ and $\mathcal{IP}_i$.

We can use the $\mathsf{ShareConv}$ protocol in Algorithm 20 for share conversion. This is intended for the case introduced in Section 5.3.2 where we either have $z = z_1 + z_2 \bmod p$ or $z = z_1 + z_2 - N \bmod p$, which is exactly what we have in the $\mathsf{Classify}\text{-}\mathcal{IP}_i^\star$ protocol.

Algorithm 20 Share conversion from $[\![z]\!]_N$ to $[\![v]\!]_p$ ($\mathsf{ShareConv}$)

```
Data: Shared secret $[\![z]\!]_N$
Result: Shared secret $[\![v]\!]_p$, where $z = v$
1: $\mathcal{CP}_i$ computes $t_i = 2 \cdot z_i \bmod N$
2:        This ensures that $t = t_1 + t_2$ as an integer is even
3: $\mathcal{CP}_i$ computes $t_i^* = t_i \bmod 2$
4: $\mathcal{CP}$ collaboratively perform simple share conversion from $[\![t^*]\!]_2$ to $[\![c]\!]_p$,
5:        This can be done with OT, as discussed in Section 5.3
6: $\mathcal{CP}_i$ computes $t_i = t_i - c_i \cdot N \bmod p$
7: $\mathcal{CP}_i$ computes $v_i = 2^{-1} \cdot t_i \bmod p$
```

Theorem 6.3.3. *The protocol* ShareConv *in Algorithm 20 for converting* $[\![z]\!]_N$ *to* $[\![v]\!]_p$ *is correct and secure, assuming secure share conversion from binary to any modulus.*

Proof sketch. The idea of computing $t_i = 2 \cdot z_i \bmod N$ is to ensure that $t_1 + t_2$ share an even number $[\![2z]\!]_N$. This enables us to do the share conversion using OT from Section 5.3 where, in the end, computing parties have a shared secret $c = 0$, if the parity of t_1 and t_2 was the same and $c = 1$ in other case. Here, different parity indicates that $t_1 + t_2 \geq N$ and same parity ensures $t_1 + t_2 < N$. By computing $t_i = t_i - c_i \cdot N \bmod p$ the parties learn $[\![2z]\!]_p$, where $2z = t_1 + t_2 \bmod p$. Trivially, computing $v_i = 2^{-1} \cdot t_i \bmod p$ gives $[\![v]\!]_p$, where $v = z \bmod p$. The correctness of this protocol follows from the correctness of the general share conversion idea.

The security clearly results from the security of the conversion from $[\![t^*]\!]_2$ to $[\![c]\!]_p$. According to Section 5.3.1, it depends on the security of the oblivious transfer. □

6.4 Publishing a secret to the result party

Previously, we defined a protocol for declassifying a secret to both computing and result parties (Publish-$\mathcal{CP}\&\mathcal{RP}_i$). However, we also have to satisfy the case where we need to open the secret privately only to $\mathcal{RP}_i$. Algorithm 21 defines protocol Publish-$\mathcal{RP}_i$ that achieves this by combining Classify-$\mathcal{IP}_i^\star$ and Publish-$\mathcal{CP}\&\mathcal{RP}_i$ in a very general manner.

Algorithm 21 Publishing a shared value to $\mathcal{RP}_i$ (Publish-$\mathcal{RP}_i$)

```
Data: secret $[\![x]\!]_p$
Result: $\mathcal{RP}_i$ learns $x$
1: $\mathcal{RP}_i$ shares a uniformly random value $y$ using Classify-$\mathcal{IP}_i^\star$
2: $\mathcal{CP}$ compute $[\![w]\!]_p = [\![x]\!]_p + [\![y]\!]_p$
3: $\mathcal{CP}$ and $\mathcal{RP}_i$ execute Publish-$\mathcal{CP}\&\mathcal{RP}_i$ to learn $w = x + y$
4: return $\mathcal{RP}_i$ corrects $x = w - y$
```

We could probably use a simpler version of Classify-$\mathcal{IP}_i^\star$ for this purpose, because the verification of correct sharing is actually a part of Publish-$\mathcal{CP}\&\mathcal{RP}_i$. Hence, we could omit all the verification steps from Classify-$\mathcal{IP}_i^\star$. The idea of this protocol is very simple, as the value y randomises the published result w and, thus, w reveals nothing about x to $\mathcal{CP}_i$.

Theorem 6.4.1. *The* Publish-$\mathcal{RP}_i$ *protocol for declassifying shared secrets to result parties is correct and as secure against a cheating* $\mathcal{CP}_i$ *as* Publish-$\mathcal{CP}\&\mathcal{RP}_i$.

Proof sketch. Both, correctness and security, result from the properties of the used subprotocols Addition, Classify-$\mathcal{IP}_i^\star$ and Publish-$\mathcal{CP}\&\mathcal{RP}_i$. The correctness of the output $x = w - y$ follows trivially from the definition $w = x + y$. □

6.5 Precomputation

We do not have a full precomputation phase for this protection domain at the moment. We introduce a protocol for generating random shares and discuss how the Beaver triple protocols from Chapter 5 could be used to achieve the necessary share representation and security guarantees.

6.5.1 Random share generation

Generating MAC tags is actually the same task as generating Beaver triples, only the inputs are slightly different. An additively shared secret that needs the tag can be used as one of the random inputs. The second is clearly the key so that the third element of the triple is actually the tag value.

Although the key is not kept in the form of additive shares, we can assume that $\mathcal{CP}_i$ has the share as the value of the key k_i and $\mathcal{CP}_j$ has the share value as 0. This, as well as the fact that we always use the same k_i for all shares, allows us to somewhat simplify the triple generation protocols, but the core ideas remain the same. All in all, Beaver triple generation protocols can also be the key for generating protection mechanisms to shares and thus, to generating single random values as well as triples.

More specifically, $\mathsf{Classify}\text{-}\mathcal{IP}_i^\star$ already introduced a way, how a third party can generate a valid share representation. The value x is not known when the computing parties generate a random value, but they could collaborate to share the tags modulo N as shown in Algorithm 22. Afterwards, they could do the same conversion as in $\mathsf{Classify}\text{-}\mathcal{IP}_i^\star$.

Algorithm 22 Generating a random share (Singles)

```
Data: No input
Result: $\mathcal{CP}_i$ have a shared secret $[\![x]\!]_p$ for uniformly random $x \leftarrow \mathbb{Z}_p$
1: Round: 1
2:     $\mathcal{CP}_i$ fixes $\Delta = 0$
3:     $\mathcal{CP}_i$ generates $x_i \leftarrow \mathbb{Z}_p$, $z_i^{(j)} \leftarrow \mathbb{Z}_{N_j}$
4:     $\mathcal{CP}_i$ computes and sends $c_j = \mathsf{Enc}_{pk_j}(k_j \cdot x_i - z_i^{(j)})$ to $\mathcal{CP}_j$
5: Round: 2
6:     $\mathcal{CP}_i$ computes $z_i^{(i)} = k_i \cdot x_i + \mathsf{Dec}_{pk_i}(c_i) \bmod N_i$ to learn $[\![z^{(i)}]\!]_{N_i}$
7:     $\mathcal{CP}$ collaboratively perform ShareConv to get $[\![z^{(1)}]\!]_p$ from $[\![z^{(1)}]\!]_{N_1}$ and $[\![z^{(2)}]\!]_p$
       from $[\![z^{(2)}]\!]_{N_2}$
```

Theorem 6.5.1. *The* Singles *protocol for generating random shares with correct tags is correct.*

Proof sketch. We need that $z_1^{(i)} + z_2^{(i)} = k_i \cdot (x_1 + x_2) \bmod p$. We assume the correctness of the share conversion and only show

$$z_1^{(i)} + z_2^{(i)} = k_i \cdot (x_1 + x_2) \bmod N \ .$$

In addition, we assume that $k_i \cdot (x_1 + x_2) < N$. We know that, by definition

$$z_i^{(i)} = k_i \cdot x_i + k_i \cdot x_j - z_j^{(i)} \bmod N \ ,$$

where it trivially follows that $z^{(i)} = k_i \cdot x \bmod N$. □

Similarly to the asymmetric version of the Singles protocol, we would actually need a zero-knowledge proof that the messages c_i are correctly formed. Currently, we could not define a simulator for the ideal version of the Singles protocol where the parties input x_i because the messages that the parties send are independent of their inputs. The ideal functionality that we would like to achieve is that the computing parties input x_i to the TTP who computes all the tag values and gives them back to the computing parties.

Theorem 6.5.2. *The communication of the* Singles *protocol for generating random shares with correct tags is simulatable and the final value of x is uniformly distributed in $\mathbb{Z}_p$, assuming computationally secure share conversion.*

Proof sketch. The communication in the generation part of this protocol is clearly simulatable as by definition c_i is an encryption of a random value that does not depend on the protocol inputs and can be simulated by sending an encryption of a random value. The share conversion part can be simulated using the corresponding simulator.

Finally, if at least one participant is honest, then x is a uniformly random element in $\mathbb{Z}_p$. This holds, because if one participant $\mathcal{CP}_i$ is honest, then $x_i \leftarrow \mathbb{Z}_p$ is uniformly distributed in $\mathbb{Z}_p$ and so is $x_1 + x_2$, because party $\mathcal{CP}_j$ does not receive any information about the value of x_i. □

Differently from the asymmetric case this Singles protocol does not specify verification and therefore we can not be sure that the share $[\![x]\!]_p$ is correctly formed. This means that we can not get the anti-framing property, but does not make the protocol less secure as we would discover the wrongly formed share during the publishing. We could add some verification as, for example, we could generate the values in pairs. For each pair, the parties would randomly choose one value that they open and where both $\mathcal{CP}_i$ show that they know x_i and $z_i^{(j)}$ together with the randomness that they used to compute c_j. That means that with probability $\frac{1}{2}$ we could notice cheating in this protocol. A more general solution would be to add a zero-knowledge protocol about the correctness of c_i.

6.5.2 Beaver triples generation

As previously, we mainly check the correctness of the computations during the opening phase, where the information-theoretic security of the MAC ensures the correctness of the verification. We can use the protocol Triple Verification to ensure that both the multiplicative relation of the triple holds and the MAC tags have been computed correctly. The security of this check results from the security of the triple verification and the security of the MAC algorithm.

One way to generate valid Beaver triples with all protection mechanisms would be to at first generate two random values x and y together with MAC tags as in Singles protocol. Then we could choose a Beaver triple protocol from Chapter 5 and run this with input shares as additive shares of x and y to learn the additive shares of $w = x \cdot y$. It would require some extra care to ensure that the Beaver triple protocol defined in the semi-honest model protects the privacy of the inputs even in the presence of active adversaries. Finally, the tags could be generated for w similarly to the tag generation the the Singles protocol. This process should be finished with a full Triple Verification protocol to ensure that the triples really have the multiplicative relation.

It currently seems most beneficial to use the basic Paillier Multiplication protocol with CRT packing for modulus M that is k-bits shorter than the maximal length allowed by $M^2 < N$. This way, we can actually always optimistically correct the result w from the Paillier Multiplication as $w = w_1 + w_2 - N \bmod M$ and with probability more than $1 - 2^{-2k}$ we have a correct w modulo M. This probability results from the fact that with probability $1 - 2^{-2k}$ we pick $w_2 \leftarrow \mathbb{Z}_N$ such that $w_2 > xy$ and compute $w_1 > xy$, giving $w_1 + w_2 > N$. After generating the protection mechanisms, we anyway have to perform Triple Verification the check the tags which also checks if the parties received a correct $w \bmod p_i$. The other possibility would be to use ShareConv-Triples with an efficient OT protocol. It would require testing the corresponding implementation in order to verify which can be more efficient.

6.6 Efficiency of the protocols

This section analyses the theoretical cost of the proposed protocols. We have two important criteria: (1) computational cost and (2) communication cost. Figure 6.1 illustrates all our protocols for the symmetric protocol set and also marks the place for the Triples protocol that was not specified.

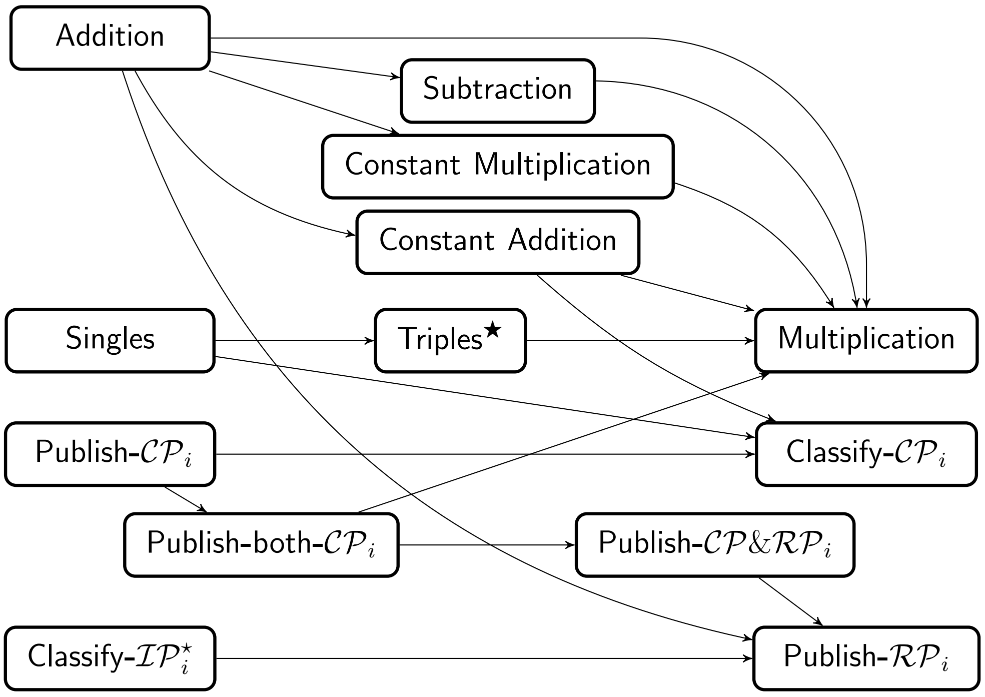


Figure 6.1: The hierarchy of the protocols for the symmetric setup

For simplicity of the analysis, we assume that both parties have chosen Paillier keys of the same length $|N_1| \approx |N_2|$. We need to consider three different classes of operations: (1) operations on additive shares of length $|p|$ bits, (2) operations on Paillier ciphertexts of length $2|N|$ bits, and (3) operations of Paillier plaintext space of length $|N|$ bits.

6.6.1 Computational cost

Differently from the asymmetric setting, most of the online protocols do not need any multiplications. Actually, the only computation protocols requiring multiplication operations are Multiplication, Constant Multiplication and Publish-$\mathcal{CP}_i$, whereas Multiplication only uses multiplication operations from the Constant Multiplication and Publish-$\mathcal{CP}_i$ protocols. In all these cases, we only use $|P|$ bit operands and get the result of the same length. However, the protocols to communicate with third parties add some complexity because there we also have to operate with ciphertexts of length $2|N|$ bits and plaintext space of $|N|$ bits. Out of these, we only analyse Classify-$\mathcal{IP}_i^\star$ because Publish-$\mathcal{RP}_i$ combines this with Publish-both-$\mathcal{CP}_i$ and uses no additional multiplications. Finally, we also include the Singles precomputation protocol.

Party	Length	ConstMult	Publish-$\mathcal{CP}_i$	Classify-$\mathcal{IP}_i^\star$	Singles
$\mathcal{CP}_i$	$\|p\|$	4	1	0	0
	$\|N\|$	0	0	1	1
	$2\|N\|$	0	0	$3\|N\| + \|p\| + 1$	$2\|N\| + \|p\| + 1$
$\mathcal{IP}_i$ $\mathcal{RP}_i$	$\|p\|$	-	-	0	-
	$\|N\|$	-	-	0	-
	$2\|N\|$	-	-	$2\|N\| + 2\|p\| + 2$	-

Table 6.1: The computational cost of protocols as a number of multiplications

As in the asymmetric setting, we assume that the Paillier encryption and decryption have the computational complexity of $|N|$ multiplications on elements of length $2|N|$. Addition under encryption has the cost of one multiplication and $\mathsf{Enc}_{pk}(m)^k$ has the cost of $|k|$ multiplications.

Table 6.1 summarises the computational complexity of our independent protocols. The complexity for Constant Multiplication results from the fact that all share elements have to be multiplied with the public value. These protocols are symmetric for the computing parties and the third party only participates in Classify-$\mathcal{IP}_i^\star$. However, for Publish-$\mathcal{CP}_i$, we mean that only $\mathcal{CP}_i$, to who the value is opened to, has to do this amount of work. From this we can see that actually the protocol Multiplication would only require 11 multiplications of length p.

The computational complexity of Classify-$\mathcal{IP}_i^\star$ for $\mathcal{IP}_i$ results from the computation of tag values under encryption where $\mathsf{Enc}_{pk}(k_1)^x$ requires approximately $|x|$ multiplications, which we estimate by $|p|$. It is similar for the Singles protocol, only that the work is done by $\mathcal{CP}_i$. We exclude the cost of share conversion from these protocols as it mainly depends on the complexity of the chosen oblivious transfer protocol.

It is easy to see that the need to use encryption makes Classify-$\mathcal{IP}_i^\star$ and Singles very expensive compared to the online computation protocols. In addition, compared to the asymmetric protocol set we have a very cheap Publish-$\mathcal{CP}_i$ protocol.

6.6.2 Communication cost

The protocols that require communication are Publish-$\mathcal{CP}_i$, Classify-$\mathcal{CP}_i$, Multiplication, Singles, Classify-$\mathcal{IP}_i^\star$ and Publish-$\mathcal{RP}_i$. However, Multiplication and Classify-$\mathcal{CP}_i$ only require communication as part of the Publish-$\mathcal{CP}_i$ protocol and Publish-$\mathcal{RP}_i$ is just a combination of Publish-$\mathcal{CP}_i$ and Classify-$\mathcal{IP}_i^\star$. Thus, it is reasonable to only analyse the communicational complexity of Publish-$\mathcal{CP}_i$, Classify-$\mathcal{IP}_i^\star$ and Singles.

Party	Length	Publish-$\mathcal{CP}_i$	Classify-$\mathcal{IP}_i^\star$	Singles
$\mathcal{CP}_j$	$\|p\|$	2	2	2
	$\|N\|$	0	0	0
	$2\|N\|$	0	1	1
$\mathcal{IP}_i$ $\mathcal{RP}_i$	$\|p\|$	-	2	-
	$\|N\|$	-	4	-
	$2\|N\|$	-	2	-

Table 6.2: The communication cost of the protocols as the number of messages of different length

Table 6.2 gives an overview of the communication complexity of the Publish-$\mathcal{CP}_i$, Classify-$\mathcal{IP}_i^\star$ and Singles protocols. The lines for Publish-$\mathcal{CP}_i$ should be read so, that in this protocol, $\mathcal{CP}_j$ has to send that many messages. The workload of the computing parties is equal for the other protocols and they both have to send the given amount of messages. Third parties have to send a given amount of messages in total and exactly half of those are sent to each $\mathcal{CP}_i$.

Protocols $\mathsf{Classify}\text{-}\mathcal{IP}_i^\star$ and $\mathsf{Singles}$ exclude the cost of the OT as the protocol could be used with different initialisations of OT or possibly with other binary share conversion ideas than our idea with OT.

Analogously to the computational cost the usage of a cryptosystem also makes the communication requirements of $\mathsf{Classify}\text{-}\mathcal{IP}_i^\star$ and $\mathsf{Singles}$ to stand out. Especially, since the additional cost of the share conversion further increases the amount of messages. However, from the straightforward relatively low cost of $\mathsf{Publish}\text{-}\mathcal{CP}_i$ we can also derive that multiplication and $\mathsf{Classify}\text{-}\mathcal{CP}_i$ are communication efficient protocols in this protection domain.

Chapter 7

Implementation

This chapter introduces the details of our implementation as well as the benchmark results of the proposed protocol sets and some precomputation ideas.

7.1 Implementation platform

Our protocols are part of SHAREMIND 3 which is implemented in C++ as are our protection domains. SHAREMIND currently uses RakNet [50] as a network layer and Boost [13] for multi-threading and configuration. We used a popular free C++ cryptography library Crypto++ [20] for the functionality of the elliptic curves. In addition, the GNU Multiple Precision library (GMP) [34] was used to get unbounded integers needed to represent shares and ciphertexts in our implementation. The implementation of the Paillier cryptosystem is similar to [48], but ported to GMP.

7.2 Secure computation capabilities

The asymmetric protocol set is implemented as a SHAREMIND protection domain kind defining share operations for addition, subtraction and multiplication. In addition, there are special protocols to multiply a share with a public value and to add a public value to the share. All these protocols have been implemented as operations on vectors applying the suitable function component-wise. Besides these online protocols, the asymmetric protection domain also contains protocols for precomputing random values and multiplicative triples. These protocols are executed as needed to keep some threshold of precomputations available. In the future we might also consider the restricted configuration where all triples needed to execute a previously define algorithm are generated beforehand and not replaced during the computations.

Currently, the controller side of this PD, consisting of communication with non-computing parties, has not been implemented because the infrastructure of SHAREMIND 3 does not yet fully support this functionality. It is future work to implement these as well as a full setup phase. Current implementation is sufficient to give insights to the usability of this PD.

The symmetric protocol set only includes the online protocols for working with the data. This includes classifying, publishing, addition, subtraction, and multiplication on the shares, as well as adding or multiplying a shared value and a public constant. This online phase is accompanied by an insecure precomputation phase to produce singles and triples necessary for testing online computations. It is future work to specify and implement a real precomputation phase. It is reasonable to test the online phase independently of the precomputations to see if it proves efficient enough to continue with these ideas. Similarly to the asymmetric case, the protocols to communicate with non-computing parties have not been implemented.

Two of the most promising protocols for Beaver triple generation were also implemented for testing purposes. At the moment they do not form a full precomputation phase for any PD. The B-Triples protocol is implemented with packing as built to the algorithm description. Protocol ShareConv-Triples is implemented in a general manner that is independent of the packing and tested using packing with CRT. We use a computationally private information retrieval (CPIR) [38] protocol to perform share conversion in ShareConv-Triples. CPIR is suitable for replacing OT in the semi-honest setting where our triple generation currently works. We used a *1-out-of-2* initialisation from [41] with the Paillier cryptosystem that is more communication efficient than AIR OT from Algorithm 16.

7.3 Performance measurements

The tests were executed on the SHAREMIND cluster where each miner ran in a different machine and they were communicating over LAN. Each of the cluster machines had 48 GB of RAM, two Intel Xeon X5670 CPUs and were connected with 1 GB/s LAN connection.

All the given results are average running times of the operations over at least ten repeated tests, more tests were used for faster operations. Column *length* denotes the length of the input and output vectors, other columns in the following tables denote various implemented protocols.

All the experiments were executed using a SECREC script. We recorded the running times of each independent execution of separate operations. These results are fixed at a miner level, thus allowing us to get separate measurements from both miners. The latter is mostly important for the online protocols of the asymmetric protocol set. It is important to note that the precomputations are running in parallel with online operations during the measurements of the online phase. This mostly affects the multiplication operation because it uses up a lot of precomputed triples that need to be replaced.

7.3.1 Online protocols

This section analyses the time requirements of the online phase of the asymmetric and symmetric protocol sets. We use the asymmetric setting with 2048-bit key and give the symmetric setting for 2048-bit prime as a comparison to that, as they represent similar data types. In addition, we compare the efficiency of the two computing parties in the asymmetric setting and give a 65-bit version of the symmetric PD.

Tables 7.1 and 7.2 illustrate the time requirements of the two computing parties in the asymmetric setting. Theoretical analysis in Section 4.6 indicated that this setup results in unbalanced workload for the two computing parties, and our measurements also reflect this. Local protocols of $\mathcal{CP}_1$ are two to three times faster than the same protocols for $\mathcal{CP}_2$ who also has to compute with ciphertexts. There is less difference for publishing or multiplication protocols as those are collaborative and it is likely that

Length	Publish	Add	Subtract	ConstAdd	ConstMult	Multiply
1	21.28	0.03	0.06	0.002	0.15	218.57
10	197.67	0.10	0.41	0.008	1.37	572.57
100	1974.02	0.62	3.93	0.037	13.49	4135.15
1000	19 732.16	6.27	38.87	0.170	134.19	39 866.97
10000	197 276.02	72.75	400.92	3.652	1343.81	392 461.09

Table 7.1: Time requirements of asymmetric computation protocols for party $\mathcal{CP}_1$ in SHAREMIND (milliseconds)

Length	Publish	Add	Subtract	ConstAdd	ConstMult	Multiply
1	24.76	0.02	0.11	0.003	0.47	222.54
10	210.76	0.15	0.98	0.004	4.65	599.92
100	2103.54	1.38	9.64	0.025	46.19	4399.50
1000	20 919.80	13.92	96.69	0.227	461.83	42 510.33
10000	209 190.81	172.94	989.36	5.749	4613.70	418 776.28

Table 7.2: Time requirements of asymmetric computation protocols for party $\mathcal{CP}_2$ in SHAREMIND (milliseconds)

$\mathcal{CP}_1$ has to wait until $\mathcal{CP}_2$ finishes some computations and answers on the network, before the parties can continue. Time requirements of both miners demonstrate a linear growth as the test inputs increase, illustrating that we actually do not gain much from vectorisation and that the computations are more likely to be CPU than network bounded.

The asymmetric setting can be compared to the symmetric setting with a 2048-bit modulus. Comparing the asymmetric results in Tables 7.1 and 7.2 to those of the symmetric protocols in Table 7.3 reveals that the gain from the symmetric protocol is significant. The declassifying and, thus, also multiplication protocols have gained most as there are no more encryption operations involved in the symmetric setting.

Length	Publish	Add	Subtract	ConstAdd	ConstMult	Multiply
1	10.28	0.01	0.01	0.003	0.01	110.48
10	10.56	0.03	0.05	0.004	0.03	112.36
100	9.94	0.26	0.39	0.024	0.24	127.89
1000	11.27	2.71	3.89	0.175	1.41	223.09
10000	22.65	34.83	48.63	2.534	12.27	1147.97

Table 7.3: Time requirements of symmetric computation protocols for 2048-bit modulus in SHAREMIND (milliseconds)

A new trend in the symmetric setting is that the times to declassify a value or multiply shares do not increase linearly as the input size grows, at least for small input sizes. This probably indicates that these protocols depend more on the network speed than computation power. The sudden growth in multiplication cost for *length* 10000 can be explained by the fact it has to perform several Publish operations and the network capacity may become a bottleneck. In addition, it requires as many triples as the input length and, thus, there is continuous precomputation in the background to replace those triples. These trends can be especially well seen from Table 7.4 which also includes longer input lengths.

The comparison of Table 7.3 to Table 7.4 shows, that the considerable differences in the data type size affect the running time less than we might expect. According

Length	Publish	Add	Subtract	ConstAdd	ConstMult	Multiply
1	10.51	0.02	0.01	0.005	0.01	55.79
10	10.27	0.04	0.02	0.007	0.01	56.76
100	10.16	0.23	0.19	0.023	0.05	54.33
1000	11.01	1.37	1.75	0.188	0.62	65.84
10000	24.77	13.56	17.84	0.886	4.49	203.48
100000	102.27	146.48	185.64	10.462	46.20	1880.76
1000000	846.05	1467.25	1682.50	97.189	460.60	14 084.73

Table 7.4: Time requirements of symmetric computation protocols for 65-bit modulus in SHAREMIND (milliseconds)

to Tables 7.3 and 7.4, computation with 65-bit modulus in only two to three times faster than computing with 2048-bit modulus. The difference between using 65-bit and 33-bit modulus illustrated the same trend where 33-bit modulus is only slightly faster than 65-bit. The surprising result that ConstMult is faster than Add results from the specifics of our setup where the public value is a uniformly random 32-bit element, which is small compared to general tested values. Measuring the symmetric setup with 33-bit prime gives a better estimate where ConstMult is actually approximately three times slower than Add.

These results clearly show that the symmetric setting can be more efficient than the asymmetric one, as expected. However, the symmetric PD can only be made usable if there also exists a reasonably efficient precomputation phase. In conclusion, the protocol set for the symmetric setup is a reasonable focus for future developments.

For simple comparison, in traditional SHAREMIND three miners PD multiplication of vectors of length 10000 took less than 100 milliseconds and was close to that also for shorter input lengths of 32-bit secrets [12]. Our asymmetric protocol set is significantly slower than that, but actually our symmetric protocol set can show similar speeds for 65 or 33-bit moduli. The main difference here is of course that [12] does not do precomputations. Covertly secure SPDZ [23] for two-parties reports doing 64-bit multiplications of input length 10000 in about 76 milliseconds for one thread and vectorised inputs. Our symmetric protocol set is currently slightly slower than that, but seems to be a good step from the asymmetric version.

7.3.2 Precomputation protocols

This section analyses the behaviour of our precomputation protocols. Table 7.5 gives the results of the time requirements of the precomputation of the asymmetric protection domain. The precomputation phase of the asymmetric protocol set is clearly less efficient than the online phase. In addition, measured results also indicate that the zero-knowledge proofs are the most expensive part of these protocols as also noted in Section 4.6. The proofs take approximately $\frac{4}{5}$ of time in the singles protocol and $\frac{3}{4}$ of total time in the triples protocol. We need approximately 1.6 seconds for one 2048-bit triple, whereas SPDZ [23] can prepare one 128-bit triple in 0.4 seconds.

Protocol B-Triples is used exactly as given in its protocol description in Algorithm 15 as packing smaller data types was native to this algorithm. The ShareConv-Triples from Algorithm 17 is benchmarked using the packing idea based on the Chinese remainder theorem. We only consider packings where all packed moduli are of equal bit length for simpler exposition and comparison. We chose 65-bit and 33-bit moduli as they are

Length	Singles with ZK	Triples with ZK	Singles	Triples
1	315	1852	42	529
10	2335	15 786	402	4699
100	22 492	154 853	4014	46 487
1000	226 923	1 544 571	40 257	464 853
10000	2 233 351	15 464 414	402 658	4 678 799

Table 7.5: Time requirements of asymmetric precomputation protocols in SHAREMIND (milliseconds)

sufficient to keep traditional 32-bit or 64-bit integers in them.

The CRT packing enables us to pack 15 elements of length 65-bits and 31 elements of length 33-bits into one ciphertext for 2048-bit modulus. This also explains the phenomena in Table 7.6 that lengths 1 and 10 take the same time for Algorithm 17—in both cases they are packed into one ciphertext and the main algorithm has the same workload. Difference between packing efficiency results in the approximately double difference between efficiency of 33-bit and 65-bit versions of these algorithms. Theoretical analysis in Section 5.4 showed that ShareConv-Triples is the most efficient of our proposals and the measurements clearly illustrate this. ShareConv-Triples can prepare about 186 packed 65-bit triples in a second, which is approximately 12 triple generation operations. In comparison, this means that ShareConv-Triples can prepare a semi-honestly secure 65-bit triple in 0.005 seconds, and SPDZ can prepare an actively secure 64-bit triple in 0.027 seconds [23].

	B-Triples		ShareConv-Triples	
Length	33-bit	65-bit	33-bit	65-bit
1	63	64	152	155
10	287	311	153	153
100	2617	2767	398	661
1000	25 686	27 199	2789	5458
10000	256 775	270 903	26 948	53 674

Table 7.6: Time requirements of Beaver triple protocols with packing in SHAREMIND (milliseconds)

For linear packing in B-Triples, we use a security constant $\sigma = 112$, which enabled us to pack 11 elements of 33-bits and 8 elements of length 65-bit into 2048-bit of plaintext space. Both this packing inefficiency and considerably higher requirements on the network made this less efficient than ShareConv-Triples. These packing counts also explain the relatively small difference in runningtimes for 33 and 65-bit cases. For both of these moduli, $\mathcal{CP}_1$ has to encrypt all *length* elements and the gain of packing only comes from a shorter result it gets back from $\mathcal{CP}_2$ which also lessens the amount of decryptions. Hence, the effect the packing has on the overall performance is substantially smaller than for packing with CRT, but the latter gain most from reducing the amount of necessary encryption and decryption functions.

In conclusion, it seems realistic to combine one of our Beaver triple protocols with CRT packing and share conversion to use it as full precomputation in the symmetric setting. The main open issue is defining efficient general share conversion that applies to additive shares and protection mechanisms.

Chapter 8

Conclusions

Secure multi-party computation is a general solution for privacy preserving data processing tasks. This thesis explores the subcase of SMC for two computing parties with the additional benefit that the parties can detect faults in the computation results. The main tools used to achieve this are an additively homomorphic cryptosystem, additive secret sharing and message authentication codes. We introduced a popular computation model that divides work to preprocessing and online phase. The latter is used to prepare some randomness that helps to speed up computations in the online phase, that performs all desired computations.

The goal of this thesis is to propose and implement new protocols for secure two-party computation for both online and precomputation phase. We concentrate mostly on common operations as sharing and publishing secret data as well as addition and multiplication. The latter is commonly implemented using Beaver triples, that are prepared in the offline phase. One of the important goals of our protocol sets is to define efficient generation of Beaver triples using an additively homomorphic cryptosystem.

The main result of this thesis is the introduction of three different flavours of setup for secure two-party computation, including asymmetric, symmetric and shared key setup. Their theoretical differences are stressed by the exact initialisation and implementation of the first two. For our initialisation, the symmetric setup is both more efficient and more flexible than the asymmetric setting. The shared key setup is presumably more efficient than the symmetric one, but adds additional complexity to verify the correctness of both computing parties.

The main goal of the Beaver triple generation protocols is to maximise the total bit length of the triples we can obtain from one multiplication using the Paillier cryptosystem. The main difficulties are coming up with a good way to pack smaller elements into the plaintext space of the Paillier cryptosystem and modifying the multiplication with the Paillier cryptosystem to give correct results for other moduli than the Paillier modulus. Two possibilities to pack smaller values into the plaintext space include linear packing and packing using the Chinese remainder theorem. The former is useful because it proposes no limits to the packed types, but the latter can be more efficient. We can also correct the results of the Paillier multiplication by analysing the potential outcomes of the protocol and collaboratively deciding which of those happened.

Current results show that actively secure multi-party computation is significantly slower than passively secure versions. However, our results indicate that fully implemented symmetric protocol set could be close to the performance of the SPDZ framework that is the current leader in actively secure multi-party computation frameworks. In addition, achieving security against malicious adversaries can be very important for data mining tasks that have important economical or societal outcomes. Therefore, in many cases the extra time consumption is a reasonable trade-off for the additional layer of security.

Future work should extend the symmetric setup to include a full precomputation phase and add new operations to both introduced protocol sets. In addition, an implementation of the shared key setup using precomputation with Paillier cryposystem would provide an interesting comparison to the existing asymmetric and symmetric setups. Furthermore, the protocols for collecting inputs or returning outputs should be implemented to allow us to use these protection domains in real world applications. Likewise, it would be important to fully specify the universally composability of each protocol as well as define protocols for setting up the necessary keys of the protection domains.

Kahe osapoolega turvaline ühisarvutus: efektiivne Beaveri kolmikute genereerimine

Magistritöö

Pille Pullonen

Resümee

Turvaline ühisarvutus võimaldab salajaste sisenditega funktsioone väärtustada ning seeläbi lahendada turvaliselt mitmeid andmetöötlusülesandeid. Passiivselt turvaline ühisarvutus kindlustab, et kui kõik osapooled järgivad protokolli, siis jäävad sisendid salajaseks ning väljundid on õiged. Aktiivne turvamudel tagab privaatsuse ka siis, kui osapooled ei käitu ausalt ning võimaldab kontrollida saadud tulemuste korrektsust.

Käesolev töö uurib turvaliste ühisarvutuste erijuhtu, kus on kaks arvutavat osapoolt. Neile lisaks võib olla ka kolmandaid osapooli, kes annavad arvutusele sisendeid või soovivad saada tulemusi. Töö peamiseks eesmärgiks on kirjeldada aktiivses mudelis turvalisi kahe osapoolega protokollistike ning implementeerida need turvalise ühisarvutuse raamistikus SHAREMIND. Meie protokollid on jagatud kahte osasse: ettearvutamine ning tööfaas. Efektiivse ettearvutamise saavutamiseks vaatleme eraldi, kuidas genereerida Beaveri kolmikuid, mis võimaldavad tööfaasis teha kiiret korrutamist.

Kahe osapoolega ühisarvutuse ülesseadmiseks on vähemalt kolm erinevat võimalust: asümmeetriline, sümmeetriline ja jagatud konfiguratsioon. Käesolev töö keskendub kahele esimesele ning defineerib kummagi jaoks konkreetse protokollistiku näite. Kolmas on olemas meie tööd oluliselt mõjutanud SPDZ protokollistikus. Meie põhiline tööriist aktiivses mudelis turvalisuse saavutamiseks on sõnumiautentimiskood, mille abil kontrollitakse salastatud väärtuste korrektsust. Ebasümmeetrilises protokollistikus kasutame lisaks ka kinnistusskeeme ja nullteadmustõestusi. Mõlemad protokollistikud põhinevad aditiivsel ühissalastusel. Nii meie teoreetiliste arutluste kui implementatsiooni järgi on sümmeetriline protokollistik efektiivsem ning paindlikum kui ebasümmeetriline. Eelkõige on sümmeetriline praktilisem, sest võimaldab vähese vaevaga defineerida erineva suurusega andmetüüpe.

Ettearvutamise osas keskendusime eelkõige Beaveri kolmikute ehk juhusliku väärtusega multiplikatiivsete kolmikute (a, b, c) genereerimisele, kusjuures a, b on juhuslikud, ning $c = a \cdot b$. Kasutame selleks aditiivselt homomorfset Paillier' krüptosüsteemi ning klassikalist algoritmi aditiivselt jagatud andmete korrutamiseks Paillier' krüptosüsteemi kasutades. Peamiseks väljakutseks on selle algoritmi kohandamine erinevatele andmetüüpidele sõltumata krüptosüsteemi jaoks defineeritud moodulist. Eelkõige vaatame, kuidas garanteerida, et korrutamisprotokoll annaks sõltumata moodulist korrektseid tulemusi. Selgub, et võimalikud tekkivad vead on hästi defineeritud ning arvutavad osapooled saavad turvaliselt kontrollida, kas viga esines või mitte.

Efektiivsuse tõstmiseks analüüsime ka erinevaid viise, kuidas väiksemaid andmetüüpe Paillier' avateksti sisse pakkida nii, et lõpptulemusena saame iga pakitud elemendi jaoks korrektse kolmiku. Elemente saab pakkida nii lineaarselt kui ka Hiina jäägiteoreemi kasutades. Meie tulemuste kohaselt on viimane neist pakkimise mõttes efektiivsem, kuid seab lisapiiranguid pakitud elementide moodulitele. Praktikas tähendab see, et Hiina jäägiteoreemi järgi pakkimisele lisaks võime me vajada ka algoritme jagatud andmete mooduli vahetamiseks.

Realiseerisime nii asümmeetrilise kui sümmeetrilise protokollistiku tööfaasi ja asümmeetrilise protokollistiku ettearvutamise faasi. Lisaks realiseerisime ühe lineaarse pakkimisega ning ühe ühe Hiina jäägiteoreemil põhineva pakkimisega Beaveri kolmikute genereerimise protokolli. Katsed näitavad, et aktiivselt turvalise sümmeetrilise protokollistiku tööfaas on rohkem kui kaks korda ajamahukam kui traditsiooniline kolme osapoolega passiivselt turvaline SHAREMINDi protokollistik. Samas on jõudluse vahe piisavalt väike selleks, et sümmeetriline protokollistik oleks praktikas kasutatav. Lisaks võivad tugevamated turvagarantiid paljude kriitilise tähtsusega andmetöötlusülesannete lahendamisel kaaluda üles jõudluse puudujäägid.

Ettearvutamise osas on selgelt näha, et asümmeetriline protokollistik jääb oluliselt alla SPDZ protokollistiku täishomomorfsel krüposüsteemil põhinevale ettearvutamisele. Samas on meie Hiina jäägiteoreemil põhinev pakkimismeetod koos sobiva kolmikute genereerimise meetodiga piisavalt efektiivne, et oleks võimalik selle alusel defineerida ettearvutusfaas sümmeetrilisele protokollistikule.

Bibliography

[1] Aiello, W., Ishai, Y., and Reingold, O. Priced oblivious transfer: How to sell digital goods. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques: Advances in Cryptology* (London, UK, UK, 2001), EUROCRYPT '01, Springer-Verlag, pp. 119–135.

[2] Barak, B., Canetti, R., Nielsen, J. B., and Pass, R. Universally composable protocols with relaxed set-up assumptions. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science* (Washington, DC, USA, 2004), FOCS '04, IEEE Computer Society, pp. 186–195.

[3] Barker, E., Barker, W., Burr, W., Polk, W., and Smid, M. Recommendation for key management – part 1: General (revision 3). Tech. rep., National Institute of Standards and Technology, 2012. NIST Special Publication 800-57.

[4] Beaver, D. Efficient multiparty protocols using circuit randomization. In *Proceedings of the 11th Annual International Cryptology Conference. CRYPTO '91* (1991), J. Feigenbaum, Ed., vol. 576 of *Lecture Notes in Computer Science*, Springer, pp. 420–432.

[5] Beaver, D., Micali, S., and Rogaway, P. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing* (New York, NY, USA, 1990), STOC '90, ACM, pp. 503–513.

[6] Ben-Or, M., Goldwasser, S., and Wigderson, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing* (New York, NY, USA, 1988), STOC '88, ACM, pp. 1–10.

[7] Bendlin, R., Damgård, I., Orlandi, C., and Zakarias, S. Semi-homomorphic encryption and multiparty computation. In *Proceedings of the 30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology* (Berlin, Heidelberg, 2011), EUROCRYPT'11, Springer-Verlag, pp. 169–188.

[8] Blakley, G. R. Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference* (1979), vol. 48, pp. 313–317.

[9] Bogdanov, D. *Sharemind: programmable secure computations with practical applications.* PhD thesis, University of Tartu, 2013. http://hdl.handle.net/10062/29041.

[10] Bogdanov, D., Laud, P., and Randmets, J. Domain-polymorphic programming of privacy-preserving applications.

[11] Bogdanov, D., Laur, S., and Willemson, J. Sharemind: A framework for fast privacy-preserving computations. In *Proceedings of the 13th European Symposium on Research in Computer Security - ESORICS'08* (2008), S. Jajodia and J. Lopez, Eds., vol. 5283 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 192–206.

[12] Bogdanov, D., Niitsoo, M., Toft, T., and Willemson, J. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec. 11*, 6 (2012), 403–418.

[13] Boost - C++ libraries. `http://www.boost.org/`. Last accessed 2013-04-02.

[14] Brakerski, Z., and Vaikuntanathan, V. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Proceedings of the 31st annual conference on Advances in cryptology* (Berlin, Heidelberg, 2011), CRYPTO'11, Springer-Verlag, pp. 505–524.

[15] Brier, E., and Joye, M. Weierstraß elliptic curves and side-channel attacks. In *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography* (2002), vol. 2274 of *Lecture Notes in Computer Science*, Springer, pp. 335–345.

[16] Canetti, R. Universally composable security: a new paradigm for cryptographic protocols. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on* (oct. 2001), pp. 136 – 145.

[17] Canetti, R., Kushilevitz, E., and Lindell, Y. On the limitations of universally composable two-party computation without set-up assumptions. In *Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques* (Berlin, Heidelberg, 2003), EUROCRYPT'03, Springer-Verlag, pp. 68–86.

[18] Canetti, R., Lindell, Y., Ostrovsky, R., and Sahai, A. Universally composable two-party and multi-party secure computation. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing* (New York, NY, USA, 2002), STOC '02, ACM, pp. 494–503.

[19] Chaum, D., Crépeau, C., and Damgård, I. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing* (New York, NY, USA, 1988), STOC '88, ACM, pp. 11–19.

[20] Dai, W. Crypto++ library. `http://www.cryptopp.com/`. Last accessed 2013-04-02.

[21] Damgård, I., Geisler, M., Krøigaard, M., and Nielsen, J. B. Asynchronous multiparty computation: Theory and implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09* (Berlin, Heidelberg, 2009), Irvine, Springer-Verlag, pp. 160–179.

[22] Damgård, I., and Jurik, M. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography* (London, UK, UK, 2001), PKC '01, Springer-Verlag, pp. 119–136.

[23] Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., and Smart, N. P. Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits. Cryptology ePrint Archive, Report 2012/642, 2012. `http://eprint.iacr.org/`.

[24] Damgård, I., and Nielsen, J. B. Scalable and unconditionally secure multiparty computation. In *Proceedings of the 27th annual international cryptology conference on Advances in cryptology* (Berlin, Heidelberg, 2007), CRYPTO'07, Springer-Verlag, pp. 572–590.

[25] Damgård, I., and Orlandi, C. Multiparty computation for dishonest majority: from passive to active security at low cost. In *Proceedings of the 30th annual conference on Advances in cryptology* (Berlin, Heidelberg, 2010), CRYPTO'10, Springer-Verlag, pp. 558–576.

[26] Damgård, I., Pastro, V., Smart, N., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. `http://eprint.iacr.org/`.

[27] Dierks, T., and Rescorla, E. RFC 5246 - The transport layer security (TLS) protocol version 1.2. `http://tools.ietf.org/html/rfc5246`, August 2008. Last accessed 2013-05-14.

[28] Diffie, W., and Hellman, M. E. New directions in cryptography. *IEEE Transactions on Information Theory 22*, 6 (1976), 644–654.

[29] El Gamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO'84 on Advances in cryptology* (New York, NY, USA, 1985), Springer-Verlag New York, Inc., pp. 10–18.

[30] Feldman, P. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1987), SFCS '87, IEEE Computer Society, pp. 427–438.

[31] Fouque, P.-A., Poupard, G., and Stern, J. Sharing decryption in the context of voting or lotteries. In *Proceedings of the 4th International Conference on Financial Cryptography* (London, UK, UK, 2001), FC '00, Springer-Verlag, pp. 90–104.

[32] Gentry, C. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing* (New York, NY, USA, 2009), STOC '09, ACM, pp. 169–178.

[33] Giry, D. BlueKrypt - cryptographic key length recommendation. `http://www.keylength.com`. Last accessed 2013-04-02.

[34] Granlund, T. GMP: The GNU multiple precision arithmetic library. `http://gmplib.org/`. Last accessed 2013-04-02.

[35] Henecka, W., Kögl, S., Sadeghi, A.-R., Schneider, T., and Wehrenberg, I. TASTY: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 451–462.

[36] Hirt, M., and Maurer, U. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1997), PODC '97, ACM, pp. 25–34.

[37] Hirt, M., and Maurer, U. Player simulation and general adversary structures in perfect multiparty computation. *JOURNAL OF CRYPTOLOGY 13* (2000), 31–60.

[38] Kushilevitz, E., and Ostrovsky, R. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1997), FOCS '97, IEEE Computer Society, pp. 364–.

[39] Laur, S., and Lipmaa, H. A new protocol for conditional disclosure of secrets and its applications. *Applied Cryptography and Network Security* (2007), 1–19.

[40] Laur, S., and Zhang, B. Lightweight zero-knowledge proofs for crypto-computing protocols. Cryptology ePrint Archive, Report 2013/064, 2013. `http://eprint.iacr.org/`.

[41] Lipmaa, H. First cpir protocol with data-dependent computation. In *Proceedings of the 12th international conference on Information security and cryptology* (Berlin, Heidelberg, 2010), ICISC'09, Springer-Verlag, pp. 193–210.

[42] Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. Fairplay - a secure two-party computation system. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 20–20.

[43] Recommended elliptic curves for federal government use. Tech. rep., National Institute of Standards and Technology, 1999. `http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf`.

[44] Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 681–700.

[45] Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th international conference on Theory and application of cryptographic techniques* (Berlin, Heidelberg, 1999), EUROCRYPT'99, Springer-Verlag, pp. 223–238.

[46] Pedersen, T. P. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, UK, 1992), CRYPTO '91, Springer-Verlag, pp. 129–140.

[47] Pfitzmann, B., and Waidner, M. A model for asynchronous reactive systems and its application to secure message transmission. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2001), SP '01, IEEE Computer Society, pp. 184–.

[48] Pullonen, P., Bogdanov, D., and Schneider, T. The design and implementation of a two-party protocol suite for Sharemind 3. Tech. rep., Cybernetica AS Institute of Information Security, 2012. `http://research.cyber.ee`.

[49] Rabin, T., and Ben-Or, M. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing* (New York, NY, USA, 1989), STOC '89, ACM, pp. 73–85.

[50] RakNet - multiplayer game network engine. `http://www.jenkinssoftware.com/`. Last accessed 2013-04-02.

[51] Schnorr, C.-P. Efficient identification and signatures for smart cards. In *Advances in Cryptology - EUROCRYPT '89*, J.-J. Quisquater and J. Vandewalle, Eds., vol. 434 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1990, pp. 688–689.

[52] Shamir, A. How to share a secret. *Communications of the ACM 22*, 11 (Nov. 1979), 612–613.

[53] Sharemind. `http://sharemind.cyber.ee`. Last accessed 2013-04-19.

[54] Smart, N., and Vercauteren, F. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography* (2012), 1–25.

[55] Tate, S. R., and Xu, K. On garbled circuits and constant round secure function evaluation. Tech. rep., University of North Texas, 2003.

[56] Yao, A. C. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1982), SFCS '82, IEEE Computer Society, pp. 160–164.