Implementing two Multi-party Threshold Private Set Intersection Protocols based on Homomorphic Encryption

Anton Jeppsson
anton@ontanj.se

Department of Electrical and Information Technology
Lund University

Supervisor: Elena Pagnin, Alexander Nilsson

Examiner: Thomas Johansson

February 12, 2021

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Private set intersection is a technique for finding the intersection of (two) parties' sets without disclosing anything else, and it finds it use in e.g. contact discovery, finding friends who are already on a social platform. Multi-party threshold private set intersection is an extension of this which allows multiple parties and only reveals the intersection if the intersection is larger than a given threshold. Badrinarayanan, Miao and Rindal [BMR20] proposed three protocols for performing multi-party threshold private set intersection. Their proposals rely on homomorphic encryption and are the first solution to realize multi-party threshold private set intersection with communication complexity linear in threshold size, and thus sublinear in set size.

In the this thesis, we investigate how to implement two of the three protocols in [BMR20] and their actual efficiency in practice. Our implementation is in the Go programming language and it is independent of the encryption schemes. Although our implementation is meant to run on a single computer, it is possible to extend it to multiple computers in an easy way. In order to implement the protocols in [BMR20], we present an algorithm for homomorphically finding the minimal polynomial of linearly recurrent sequences in this setting. This is to the best of our knowledge the first development of such an algorithm. In addition we also implemented, and made publicly available, a Go library for performing basic matrix operations with elements being of any type, particularly useful for working with matrices homomorphically.

Popular Science Summary

As bigger and bigger parts of our society get digital, the need for safe storage constantly increases. One way to protect sensitive information is by using encryption. Encryption transforms your data into a seemingly random number, so that only the keeper of a secret key can retrieve the original data, and an eventual hacker cannot. One special kind of encryption is called *homomorphic encryption*. Homomorphic encryption protects the data as above, but in addition to that it also preserves certain arithmetic properties. For example an *additive homomorphic* encryption scheme allows to you to add two ciphertexts, and upon decryption of the sum you will receive the sum of the original plaintexts. This property is not always desirable, but in some cases it is highly useful. One scenario is if you upload your health data to the cloud, you might not want to disclose this data as it is sensitive, thus you wish to encrypt it. By using homomorphic encryption though, it's possible for a third-party service to make calculations on your data, e.g. likeliness of certain medical conditions and give you notes on this that only you (and not the third-party service provider) can see.

In this thesis we investigate another scenario, named multi-party threshold private set intersection. This is a technique that allows multiple parties, all carrying a set of some sort, to learn whether their intersection (elements present in every set) is large enough and in that case learn which elements are in this set, but not revealing elements that are not. Badrinanrayanan, Miao and Rindal [BMR20] proposed recently three ways to achieve this. Their solution make use of an aforementioned additive homomorphic encryption scheme and a fully homomorphic encryption scheme (that is a scheme that allows both addition and multiplication of ciphertexts). In our thesis we implement two of these protocols in the Go programming language, and we investigate how such an implementation can be constructed in an efficient way. We construct it in a way that makes it independent of the encryption scheme (i.e. you can plug in any implementation of an additive homomorphic encryption scheme for the additive part and the same for the fully homomorphic part).

As a motivating example, we look at a use-case where several parties are interested in starting a support group, discussing issues they have, e.g. alcoholism. These issues might however be sensitive, so the parties only want to reveal their intersection after answering a questionnaire, and discuss those issues at meetings. The threshold here could be set so that they have at least three issues in common.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

While our society gets more and more based around digital solutions, the need for safe storage of data increases. Standard cryptosystems keep the data safe, but they do not allow for processing of data, unless decrypted. Homomorphic encryption addresses this need as it allows computation on encrypted data and thus eliminates the exposure of data during the computational step. This, however, makes homomorphic encryption malleable by design, and thus is not useful for every application. Nonetheless, there are several applications that can be realized efficiently only thanks to homomorphic encryption. Some applications are advertising, medical applications and data mining [Arm+15]. One recent use case is given by [BMR20], where the authors use homomorphic encryption to design Multi-party Threshold Private Set Intersection protocols. This thesis extends [BMR20] by providing a working implementation of two of the three proposed protocols in the Go programming language.

1.1 What is homomorphic encryption?

Homomorphic cryptosystems are cryptosystems with homomorphic properties, which means that we are able to perform operations on the data while it is encrypted. This differs from usual cryptosystems where data has to be decrypted in order to be manipulated, thus potentially exposing it. The term homomorphism comes from ancient Greek, meaning *of the same shape* [Aca+18]. In abstract algebra it is used to describe a map that preserves the algebraic structure between the domain and the range of an algebraic set. The same is true for homomorphism in cryptology were we have this homomorphism between the plaintext space and the ciphertext space.

Several types of homomorphic cryptosystems exist, with *partially homomorphic* being the first discovered. *Partially* in this case means homomorphic with respect to one property, most commonly addition or multiplication. These cryptosystems are often referred to as *additive homomorphic* (AHE) or *multiplicative homomorphic* [Pai99] [Elg85].

Levelled homomorphic cryptosystems are more expressive than partially homomorphic cryptosystems in that they allow for both addition and multiplication, however the number of multiplications one can perform on a ciphertext is limited by a level L, set by the scheme.

Somewhat homomorphic cryptosystems are not well-defined and used inconsistently in the literature. Typically a somewhat homomorphic encryption scheme allows any number of addition and a few numbers of multiplications. In other words, the value of L may vary according to what kind of computation is performed.

The last type in this hierarchy are *fully homomorphic encryption* (FHE). Fully homomorphic encryption allows for an arbitrary number of additions and multiplications, and according to theory, this is enough to construct any other operation.

Another property of homomorphic cryptosystems, or rather cryptosystems in general is *multi-party* schemes. In a *multi-party homomorphic cryptosystem* there is a predetermined number of parties n. Any one party can encrypt and perform operations, but it takes the joint effort of all n parties to decrypt. In *threshold homorphic cryptosystems* there is an additional number t, and the difference from multi-party is that it only takes t parties out of the total n to decrypt.

A homomorphic cryptosystem can be formalized as one containing the following four functions [Arm+15].

- KeyGen$(1^\lambda, \alpha)$: Given a security parameter λ and an auxiliary input α, outputs three keys (pk, sk, evk) where pk is the encryption key, sk is the decryption key and evk is the evaluation key. pk and evk can be made public.

- Encrypt(pk, m): Encrypts the plaintext message m using the encryption key pk, and outputs a ciphertext c.

- Evaluate$(evk, C, c_1, \ldots, c_n)$: Evaluates (possibly a mix of) ciphertexts and evaluations c_i on circuit C using the evaluation key evk. It outputs an evaluation output c.

- Decrypt(sk, c): Decrypts a ciphertext or an evaluation c using the decryption key sk and outputs a plaintext m.

1.1.1 History of homomorphic encryption

The first homomorphic cryptosystem was the well-known RSA cryptosystem presented 1978 [RSA78]. However the concept of homomorphic encryption was described short thereafter [RD78]. This is a homomorphism over the group $(\mathbb{Z}, \cdot)$, i.e. multiplication of integers.

Goldwasser and Micali described the first additive homomorphic cryptosystem, being homomorphic over the group $(\mathbb{Z}_2, +)$ [GM84]. Benaloh continued their work and presented an additive homomorphic cryptosystem over $\mathbb{Z}$ [Ben94]. Katz and Yung also extended the work of Goldwasser and Micali and described a threshold additive homomorphic cryptosystem [KY02] which later was further improved by Desmedt and Kurosawa [DK07]. Meanwhile several other additive homomorphic cryptosystems were described with Paillier's [Pai99] being the most well-known and versatile for implementations. This was expanded into a threshold additive cryptosystem both by Fouque, Poupard and Stern [FPS01] and by Damgård and Jurik [DJN03] using a trick presented by Shoup [Sho00].

The first fully homomorphic encryption scheme was described by Gentry [Gen09] in 2009 and used ideal lattices, a mathematical structure new to cryptography. Gentry showed how to transform a somewhat homomorphic scheme into a fully homomorphic one by squashing and boostrapping. The squashing step simplifies the encryption function as much as possible and the bootstrapping reencrypts the ciphertext homomorphically. This brings the noise in the ciphertext (which grows upon evaluation), back to a fixed size. After this breakthrough van Dijk et al. [Dij+09] described a second fully homomorphic encryption scheme based on some of Gentry's ideas, but without using ideal lattices and thus reducing the complexity of the scheme. Over time several different schemes have emerged and the problems over ideal lattices are left in favor of *Learning with errors* and *Ring learning with errors*, although still following the road map of Gentry. One such scheme is the *scale-invariant* scheme proposed by Brakerski [Bra12]. This scheme was improved by Junfeng Fan and Frederik Vercauteren, resulting in the BFV scheme [FV12].

In 2017 Cheon et al. proposed a new interesting encryption technique with approximate calculations [Che+17]. As some applications, e.g. machine learning, have inherent approximations and small errors, this scheme integrates that in itself.

1.2 Applications of homomorphic encryption

The most wide-spread motivation for homomorphic encryption stems from modern day cloud services. It it common today to store your data in the cloud at a cloud service provider, but this raises privacy concerns if the could service provider can not be trusted. By using homomorphic encryption, the data can be uploaded to an untrusted service provider and safely be operated on.

A similar example is provided by Armknecht et al. [Arm+15]: If one company A has sensitive data, e.g. a stock portfolio, and another company B has secret algorithms that make predictions about the stock prices, it is possible for A to encrypt its data and let B perform operations on it homomorphically, that way no extra information is disclosed.

In [NLV11] Naehrig et al. sketch a scenario where individuals continuously send their health data in encrypted form to a service provider. This way the individual is the owner of the data and has full control over it. By using homomorphic encryption the service provider can continuously send health feedback to the individual or give predictions on likelihood of certain medical states.

Bösch et al. [Bös+14] propose a tool for forensic image recognition. Law enforcement might have a database containing hash values to detect illegal images on disks and data streams. To detect and mitigate the spreading of such images it is beneficial to spread the database to internet service providers or companies to monitor the traffic. However this database is strictly confidential to prevent people from, e.g., scanning their illegal images with this database and changing the images so they aren't detected. What Bösch et al. describe is a tool called SOFIR which allows law enforcement to receive statistics on matches in the database, while both the database and the traffic is kept private.

1.3 A special case: Private set intersection

One area where homomorphic encryption techniques are useful is for private set intersection protocols [BMR20]. Private set intersection explores the case where mutually distrustful parties wish to find the intersection of their respective sets, but don't want to disclose anything else. This finds its use in a range of applications from botnet detection [Nag+10] to advertising [Ion+17] and private contact discovery [Mar14].

A special case of private set intersection is threshold private set intersection, where the intersection is disclosed only if the size of the intersection exceeds a certain threshold T. This could be the case for a privacy-preserving ride sharing where the parties only want to reveal their path if the common path is long enough.

Badrinarayanan, Miao and Rindal suggested [BMR20] three protocols to perform multi-party (i.e., more than two participants) threshold private set intersection, using threshold homomorphic encryption. This is a very recent work, and according to [BMR20] they are the first to propose a "regular" multi-party private set intersection protocol where the communication complexity depends on the threshold value rather than the set size. We found this work cutting edge, yet realized it was missing an implementation. To this end we asked ourselves the questions: 1) How can this protocols be implemented? 2) Which libraries do we need to implement it? 3) How efficient will it be? These questions find their answers in chapters 3, 2.3 and 4

This thesis implements two of the protocols proposed by Badrinarayanan, Miao and Rindal by building on existing homomorphic encryption libraries.

1.3.1 A motivating example

The use case we build around is a mechanism for support group creation. A group of 4-5 people answer a questionnaire about sensitive issues that they wish to talk about. The questionnaire might contain approximately 10 common issues, such as *Do you suffer from alcoholism?*. If there are at least e.g. three questions that everybody in the group answers yes to, a group can be created.

1.4 Contributions

We provide implementations of the protocols $\mathcal{F}_{\text{TPSI-diff}}$ using AHE, and a slightly modified version of $\mathcal{F}_{\text{TPSI-int}}$ using FHE, as described by [BMR20], in the Go programming language [Jepb]. The discussions in the paper can be used as guidance for implementations in other languages. The implementation is modular when it comes to cryptosystem; as long as the homomorphic properties are satisfied, any cryptosystem implementation can be run with our implementations. To realize $\mathcal{F}_{\text{TPSI-diff}}$ using AHE we describe a way of finding the minimal polynomial of a linearly recurrent sequence with multi-party additive homomorphic encryption, this step was omitted in [BMR20], but we follow the blueprint outlined there. For the $\mathcal{F}_{\text{TPSI-int}}$ we provide an alternative algorithm to realize the same functionality as in [BMR20]. The main reason for this change is that the FHE library we build on lacks the crucial bootstrapping step but instead provides "collective bootstrapping". Finally, we also provide a mathematical library in Go for matrix operations with elements of any type [Jepa].

Chapter 2
Preliminaries

2.1 Notation

In this work we use n parties, each party has an encryption key and a set S_i. Since the setting is multi-party homomorphic, all are needed in order to decrypt a plaintext. Badrinarayanan, Miao and Rindal assumes that every party has the same number of elements in their set $|S_i|$, denoted m. Furthermore, they work in the star topology, visualized in Figure 2.1, which means that one party is the central party which is communicating with all other parties. No communication takes place between the other parties, everything goes through the central party. In this thesis the central party will be denoted P_1 and some functions will be prefixed with `Central` or `Outer` to clarify which party are to use it.

The threshold size which needs to be passed to reveal the intersection is denoted T.

Brackets are used to distinguish the ciphertext $[\![a]\!]$ from the plaintext a.

The elements are denoted a, $S_i = \{a_{1,i}, \dots, a_{m,i}\}$.

2.2 The work of Badrinarayanan, Miao and Rindal

In [BMR20] Badrinarayanan, Miao and Rindal describe three protocols for performing multi-party threshold private set intersection using homomorphic encryption. These are $\mathcal{F}_{\text{TPSI-diff}}$ using AHE, $\mathcal{F}_{\text{TPSI-diff}}$ using FHE and $\mathcal{F}_{\text{TPSI-int}}$ (using FHE). The difference in functionality between $\mathcal{F}_{\text{TPSI-diff}}$ and $\mathcal{F}_{\text{TPSI-int}}$ lies in the interpretation of T. For $\mathcal{F}_{\text{TPSI-int}}$ we check whether the intersection $I = \bigcap_{i=1}^{n} S_i$ is sufficiently large $|I| \geq m - T$. For $\mathcal{F}_{\text{TPSI-diff}}$ on the other hand we check if the set difference is sufficiently small $|(\bigcup_{i=1}^{n} S_i) \setminus I| \leq T$. Refer to 2.2 for a visualization.

All protocols are split into two parts, the first being cardinality testing, which determines whether or not the threshold property is satisfied. If it's not, the protocol is terminated with empty output, but if it is, the second part of the protocol is run which returns the intersection. This functionality for finding the intersection is shared between all protocols. It only requires an additive homomorphic property and can thus be performed by both AHE and FHE.

Badrinarayanan, Mian and Rindal work in the semi-honest setting where parties are expected to learn as much as possible from the data they receive, but they will never deviate from the protocol to gather extra information.

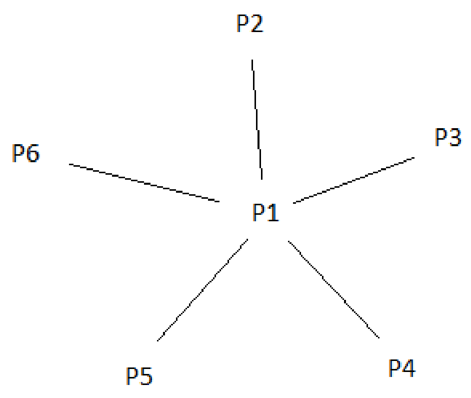


Figure 2.1: Visualization of the star topology. All communication goes through P_1.

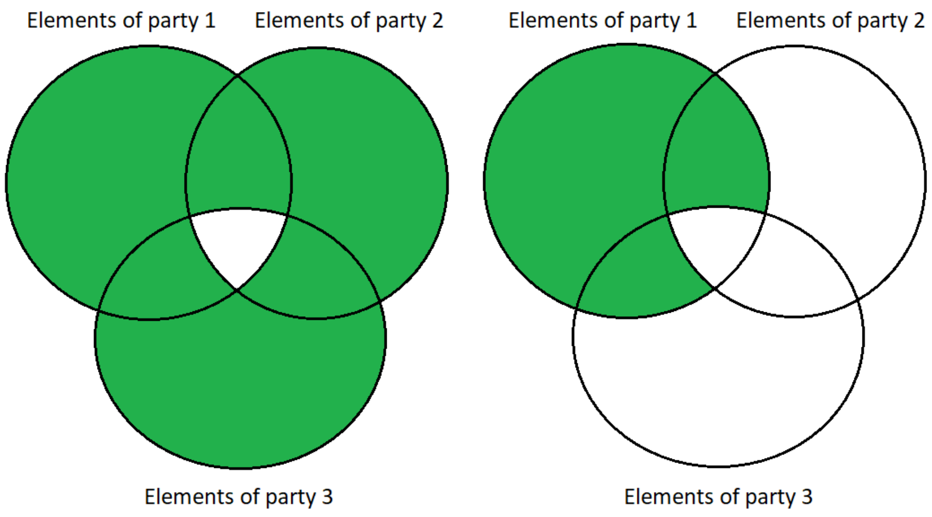


Figure 2.2: Venn diagrams of the differences of the protocols in [BMR20]. Every circle contains the elements of a party. In the middle of each diagram we have the intersection. Colored is the element we "count" in each protocol. This count needs to be $\leq T$ for the cardinality test to pass. The left-hand sketch shows $\mathcal{F}_{\text{TPSI-diff}}$ and the right-hand sketch shows $\mathcal{F}_{\text{TPSI-int}}$. Note that the colored area of $\mathcal{F}_{\text{TPSI-int}}$ are of the same size for all parties since all sets are of equal size.

Let us start by looking at $\mathcal{F}_{\text{TPSI-diff}}$ using AHE (described in Figure 9 in [BMR20]).

Each party P_i encodes their set elements a_j as exponents in a polynomial, $p_i(x) = \sum_{j=1}^{m} x^{a_{j,i}}$. Now if P_1 multiplies p_1 by the number of outer parties and subtract the sum of all outer parties polynomials, i.e. $p(x) = (n-1)p_1(x) - \sum_{i=2}^{n} p_i(x)$, we see that all monomials with exponents in the intersection cancel out. To determine whether the number of polynomials in p is $\leq T$ we apply the polynomial sparsity test of Grigorescu et al. [GJR10]. To to this we take a uniformly random u and construct a Hankel matrix H, shown below. A Hankel matrix is a matrix where the elements equals within each diagonal. The matrix H is singular if the the number of polynomials in p is $\leq T$. So we are left to perform a singularity test for H.

$$H = \begin{pmatrix} p(u^0) & p(u^1) & \cdots & p(u^T) \\ p(u^1) & p(u^2) & \cdots & p(u^{T+1}) \\ \vdots & \vdots & \ddots & \vdots \\ p(u^T) & p(u^{T+1}) & \cdots & p(u^{2T}) \end{pmatrix} \tag{2.1}$$

The path Badrinarayanan et al. take to perform a singularity test relies on the theory of [Kil+07]. By finding the linearly recurrent sequence $\mathbf{a} = \{\mathbf{u}^T H^i \mathbf{v}\}_{i \in \mathbb{N}}$ for random vectors $\mathbf{u}$ and $\mathbf{v}$. Using this linearly recurrent sequence $\mathbf{a}$ the problem of singularity testing is reduced to finding the minimal polynomial of $\mathbf{a}$. Badrinarayanan et al. provide a protocol for collective matrix multiplication of encrypted matrices, described in figure 7 of [BMR20]. For finding the minimal polynomial the authors state that there exists such a functionality with the desired communication bounds. A solution to this problem is provided by this thesis in section 3.7.

Finding whether H is singular or not concludes the first part of $\mathcal{F}_{\text{TPSI-diff}}$ using AHE and we reach the second part, where we find the intersection. This part is conceptually very simple, but its technical description is complicated by the fact that the parties polynomials need to be properly masked for security. For simplicity, this description leaves out masking polynomials and masking roots, refer to figure 10 of [BMR20] for further information.

To find the intersection, the elements a_j are reencoded as roots of a polynomial, $p_i(x) = \prod_{j=1}^{m}(x - a_{j,i})$. Each party receives the sum of all p_i, $V(x) = \sum_{i=1}^{n} p_i(x)$, or rather $3T+3$ sample values of V. As V is properly masked this is safely passed unencrypted. Now, for the rational polynomial $q_i(x) = \frac{V(x)}{p_i(x)}$, elements in the intersection will be cancelled out from the numerator and the denominator, so by performing rational polynomial interpolation of q, the elements belonging to P_i but not in the intersection, are the roots remaining in the denominator of q. And this solves our problem.

For the $\mathcal{F}_{\text{TPSI-int}}$ the idea is similar to the second part of $\mathcal{F}_{\text{TPSI-diff}}$ above. The elements are encoded as roots $p_i(x) = \prod_{j=1}^{m}(x - a_{j,i})$ and $2T+3$ sample values of this polynomial is passed encrypted to P_1 together with an encryption of $p_i(z)$ with z being randomly determined by P_1. P_1 now homomorphically performs a rational polynomial interpolation of $[\![p^*(x)]\!] = [\![\frac{\sum_{i=2}^{n} p_i}{p_1}]\!]$ using the $2T+3$ points.

If enough roots are cancelled out (i.e. the intersection is big enough), the degree of this rational polynomial will drop so that it can be correctly recreated with only $2T+3$ evaluation. After interpolation we check whether $p^*(z) = \frac{\sum_{i=2}^{n} p_i(z)}{p_1(z)}$, if we have equality, p^* was correctly restored, and thus the cardinality test passed and we can move on to determine the intersection.

The second part of $\mathcal{F}_{\text{TPSI-int}}$ is identical to the second part of $\mathcal{F}_{\text{TPSI-diff}}$.

2.3 Library review: Additive homomorphic libraries

One of the most fundamental parts of programming is the re-use of code. There exists multiple repositories publicly available that implement the homomorphic cryptosystems needed in this project. First, let us look at the additive homomorphic cryptosystems.

The most common issue with existing implementations is missing licenses. Although most of them signal that they are meant to be used, the lack of license makes them effectively not open source, and thus unfit for using. For the licenses present, they were of various kinds but all were permissive for our use. Another problem was libraries not stating which protocol they were based on. This makes it hard to verify whether the protocol actually is correct, and it would make it particularly hard to fix an eventual error.

Table 2.1 shows the threshold additive homomorphic schemes that were found. They are sorted roughly by how fit they are for the task, with *Paillier Threshold Encryption Scheme Implementation* being the one that was finally chosen.

The reviewed libraries can broadly be split into three categories, those based on Paillier [Pai99], Elgamal [Elg85] and BGN [BGN05]. Paillier schemes are the most numerous, as these include the schemes of Damgård-Jurik [DJN03], Fouque [FPS01] and Nishide [NS11]. BGN is labelled as *Somewhat homomorphic* as it allows one multiplication. If no other implementations suited our purpose this might had been useful, but primarily we wanted to favour purely additive cryptosystems. Elgamal is mainly a multiplicative homomorphic cryptosystem, but by raising the plaintext m to a fixed value before encryption transforms it to an additive cryptosystem. However, to retrieve m after decryption for this transformed cryptosystem we would have to solve an equation of the form $c = k^m$ in $\mathbb{Z}_n$, which might be difficult depending on e.g. n.

Implementations for threshold fully homomorphic encryption exists in Go and C++, making them favourable for the first part also. This deems Paillier Threshold Encryption Scheme Implementation the most suitable choice. It will be called tcpaillier for the rest of the thesis, as this is the package path.

2.4 Library review: Fully homomorphic libraries

Listed in Figure 2.2 are the most prominent existing libraries implementing fully homomorphic encryption. As can be seen, most of them are written in C++. Only two of them announce support for threshold fully homomorphic encryption, that is Lattigo and PALISADE.

Project name	Language	Implements	Notes
Paillier Threshold Encryption Scheme Implementation	Go	[DJN03]	[Lab]
Damgard-Jurik	Python	[DJN03]	[Cryb]
Somewhat homomorphic encryption over elliptic curve using BGN algorithm	Python	[BGN05]	
Cryptography	Java	[BGN05], [Pai99], [Ben94]	[Mal]
Threshold cryptography library	Python	[Elg85]	[Pet]
Paillier Threshold Encryption Toolbox	Java	[DJ01]	[KG]
Distributed Paillier Cryptosystem	Python	Not stated	[App], based on [KG]
Lua Multi Party	Lua / C	Not stated	[Dyn]
Damgård-Jurik Cryptosystem	C++	[DJN03]	Not licensed [jia]
paillier	Go	[Pai99]	Not licensed [Serb]
thresh-paillier-wo-td	Java	[NS11]	Not licensed [Chr]
Threshold-Paillier-with-ZKP	C++	[FPS01]	Not licensed [ziya]
Threshold-Paillier-without-Trust-Dealer	C++	[NS11]	Not licensed [ziyb]
Threshold-Paillier-without-Trust-Dealer-with-TCP-IP	Python	Not stated	Not licensed [ziyc]
BGN	C++	[BGN05]	Not licensed [Poo]
BGN 🔒	Go	[BGN05]	Not licensed [Sera]

Table 2.1: Considered additive homomorphic encryption implementations

Project name	Language	Implements	Notes
Lattigo	Go	[Che+17], [FV12]	Threshold for BFV, CKKS [Dat]
PALISADE	C++	[BGV11], [FV12], [Che+17], [DM14], [Chi+16]	Threshold for BGV, BFV, CKKS [PAL]
HElib	C++	[BGV11], [Che+17]	[hom]
Microsoft SEAL	C++ / C#	[Che+17], [FV12]	[20]
HEAAN	C++	[Che+17]	[Seo]
FHEW	C++	[DM14]	[Duc]
TFHE	C++	[Chi+16]	[tfh]
FV-NFLlib	C++	[FV12]	[Crya]
NuFHE	Python	[Chi+16]	[NuC]
Concrete	Rust	[Chi+16]	[Zam]

Table 2.2: Considered fully homomorphic encryption implementations

The choice was made to use Lattigo as the Go library tcpaillier was preferred for AHE and Lattigo is more light-weight. However, upon working with the library, the discovery was made that the crucial bootstrapping step was not implemented yet, this was later confirmed by the authors. Instead, they provide a "collective bootstrapping". Thus in practice a single party can only perform few multiplications on its own, thereafter, to continue calculations, all parties need to participate in this collective bootstrapping. Due to this, we developed an alternative scheme for $\mathcal{F}_{\text{TPSI-int}}$, which makes use of the collective bootstrapping instead. From the documentation of PALISADE it is unclear whether they implement the actual bootstrapping for the BGV or BFV schemes or if they take a similar approach as Lattigo.

The CKKS scheme [Che+17] provided by both Lattigo and PALISADE is not applicable to this work as it uses approximate numbers.

2.5 The Go programming language

Go was designed in 2007 at Google by Robert Griesemer, Rob Pike and Ken Thompson. In 2009, version 1.0 was released, and it was made open-source. The idea to create Go arose from problems seen at Google at the time. Back then, computers had seen extensive improvement over the years, but the languages in use, e.g. C++ and Java, had not seen so much improvement. As stated on golang.org, at the time *one had to choose either efficient compilation, efficient execution, or ease of programming.* Griesemer, Pike and Thompson wanted to create a modern, fast and easy-to-use language by looking at where current programming was and where it was going [Go].

Go is syntactically similar to C, but with memory safety, garbage collection, concurrency and structural typing. As opposed to nominal typing, which is used in e.g. Java and C++, in structural typing an element is compatible with another if all features of the first element also exists for the second. This is evident in how Go handles polymorphism, with `interface`. An interface can be described as a set of methods acting on a `struct` (a `struct` in Go is a collection of data, like in C). Go is not an object-oriented language, so methods do not belong to an object, instead some methods are declared with a so called *receiver argument.* In order for a `struct A` to *implement* an `interface B`, all functions stated in `B` must be implemented for `A` (with `A` being the receiver argument). Below is a simple example where `A` implements `B` and thus `A` can be used everywhere `B` is stated.

```
type A struct {
    num int
}

type B interface {
    Foo(int) int
}

func (a A) Foo(b int) int {
    return a.num + b
}

func Bar(b B) int {
    return b.Foo(2)
}

var a A
Bar(a) // this is valid
```

In Go lists are known as slices, they are much like `ArrayLists` in Java. A slice (with zero-elements) can be initialized using `make`, e.g. `make([]int, 3)`, or by initializing values directly, `[]int{1,2,3}`. The declared type of a slice is empty square brackets followed by the data type stored in the slice, e.g. `[]int`. Accessing elements in the slice is done similar to Python, i.e. stating an index or a range inside brackets, `a_slice[1:]`

Go's approach on errors is treating them as values. Go allows multiple return arguments from functions, and an `error` element are frequently passed as one of them. Although originally omitted, a `panic`/`recover`-mechanism (similar to `try-catch`) was eventually added, it's used for *truly exceptional unrecoverable* errors.

Go allows named returns, that is, the return arguments can be stated in the function header and their current values will be returned when `return` is called. The return arguments are stated after the function arguments. In `func Foo(a int) (b int, err error)`, `b` and `err` are return arguments. This feature is particularly useful in Go since eventual errors can create a lot of potential return points. The following snippet is thus a common sight in Go.

```
val, err := Foo()
if err != nil {
    return
}
```

2.6 Cryptosystems

This section gives a brief description of the additive homomorphic cryptosystem used, and its ancestors, as a taste. The description of the fully homomorphic cryptosystem, BFV, is out of scope for this thesis. Refer to [FV12] and [Mou+20] for a description.

2.6.1 Paillier cryptosystem

Paillier cryptosystem was presented in [Pai99]. It consists of three algorithms: KeyGen, Encrypt and Decrypt. Evaluation is trivial.

KeyGen: Choose two large prime numbers p and q, and set $n = pq$. Choose g such that

$$gcd(\frac{g^{\lambda(n)} - 1 \mod n^2}{n}, n) = 1.$$

The public key is (n, g) and the private key is (p, q). $\lambda(n)$ refers to Carmichael's function, $\lambda(n) = \mathrm{lcm}(p-1)(q-1)$.

Encrypt: Use plaintext $m < n$. Select random $r < n$. Ciphertext is $c = g^m \cdot r^n \mod n^2$.

Decrypt: $m = \frac{L(c^\lambda(n) \mod n^2)}{L(g^\lambda(n) \mod n^2)} \mod n$, where $L(x) = \frac{x-1}{n}$.

It it easy to see that the system exhibits the following homomorphisms on $\mathbb{Z}_n$.

$$\begin{aligned} \mathbf{Decrypt}(\mathbf{Encrypt}(m_1) \cdot \mathbf{Encrypt}(m_2) \mod n^2) =& m_1 + m_2 \mod n \\ \mathbf{Decrypt}(\mathbf{Encrypt}(m)^k \mod n^2) =& km \mod n \end{aligned}$$

2.6.2 Damgård-Jurik cryptosystem

Damgård-Jurik cryptosystem [DJN03] is a generalization of Paillier. It introduces the parameter s, extending the plaintext space to $\mathbb{Z}_{n^s}$, where s is a natural number smaller than p and q. For Paillier we have $s = 1$ [DJN03]. In our case we also use $s = 1$, since we want the plaintext space to be as similar as possible to a field, and we don't have any other requirements for the size of the plaintext space.

KeyGen: Set $n = pq$ for p, q prime. Choose g such that $g = (1+n)^j x \mod n^{s+1}$ for j relatively prime to n and x. It is possible to always choose $g = n+1$ without reducing security of the system [DJN03]. Choose d such that $d = 0 \mod \lambda(n)$. For Paillier $d = \lambda(n)$ was used. The public key is (n, g) and private key is d.

Encrypt: Use plaintext $m < n^s$. Select random $r < n$. Ciphertext is $c = g^m \cdot r^{n^s} \mod n^{s+1}$.

Decrypt: We note that

$$\begin{aligned} c^d &= (g^m r^{n^s})^d = ((1+n)^{jm} x^m r^{n^s})^d \\ &= (1+n)^{jmd \mod n^s} (x^m r^{n^s})^{d \mod \lambda} \\ &= (1+n)^{jmd \mod n^s} \end{aligned}$$

[DJN03] provides an algorithm to determine $jmd \mod n^s$ from $(1+n)^{jmd \mod n^s}$. After applying that algorithm it is possible to determine $m = jmd \cdot (jd)^{-1} \mod n^s$.

2.6.3 Damgård-Jurik threshold cryptosystem

To create the threshold variant of the scheme, we adopt a technique proposed by Shoup [Sho00]. Shoup describes how to collectively and effectively raise an input number to a secret exponent modulo an RSA modulo n. Damgård et al. transform this to their case which allows to raise the input c to the secret exponent d modulo n^{s+1}. By using $g = n+1$ and $d = 1 \mod n^s$ and $d = 0 \mod \lambda$ we will receive $c^d = (1+n)^{m \mod n^s}$ so that the rest of the decryption can be performed without knowledge of d.

KeyGen: Choose four primes p, p', q and q' such that $p = 2p' + 1$ and $q = 2q' + 1$, and set $n = pq$ and $n' = p'q'$. Choose s as before, $s = 1$ in our case. Choose $d = 0 \mod n'$ and $d = 1 \mod n^s$. We set the threshold w as $2w - 1 = l$, where l is the number of key's generated. Now by picking random

a_i (for $0 < i < w$) from $\{0, ..., n^s \cdot n' - 1\}$ and setting $a_0 = d$, we can create the polynomial $f(X) = \sum_{i=0}^{w-1} a_i X^i \mod n^s n'$. The secret key of party i ($1 \leq i \leq l$) is $s_i = f(i)$ and the public key is (n, s).

Encrypt: Pick a random $r \in \mathbb{Z}_n^*$, the ciphertext is $c = (n+1)^m r^{n^s} \mod n^{s+1}$.

Decrypt: A party contributes $c_i = c^{2l! s_i}$. For a subset S of w such partial decryptions we can calculate $c' = \prod_{i \in S} c_i^{2\lambda_{0,i}^S}$ where $\lambda_{0,i}^S = l! \prod_{i \in S i} \frac{-i}{i-i'} \in \mathbb{Z}$. This brings c' to the form $c' = (1+n)^{4l!^2 m} \mod n^{s+1}$. Here we use the same algorithm from [DJN03] as used in section 2.6.2 and then multiply by $(4l!^2)^{-1} \mod n^s$ to restore m.

Chapter 3

Implementation

In this chapter we will present all the different parts of the implementations we provide. In Figure 3.1 we see a visualization of the building block's part in the protocols. The implementation is available at [Jepb].

3.1 Cryptosystem interfaces

We aim to provide an implementation of the protocols that are independent of cryptosystem. To facilitate this two interfaces `AHE_Cryptosystem` and `FHE_Cryptosystem` were designed. They are defined as follows.

```
type AHE_Cryptosystem interface {
    // addition of two elements
    Add(Ciphertext, Ciphertext)
            (sum Ciphertext, err error)

    // scaling of an element by scalar factor
    Scale(cipher Ciphertext, factor *big.Int)
            (product Ciphertext, err error)

    // encrypt a plaintext message
    Encrypt(*big.Int) (Ciphertext, error)

    // combine partial decryptions to plaintext
    CombinePartials([]Partial_decryption)
            (*big.Int, error)

    // encrypted matrix evaluation
    EvaluationSpace() gm.Space

    // size of plaintext space
    N() *big.Int
}

type FHE_Cryptosystem interface {
    AHE_Cryptosystem
```

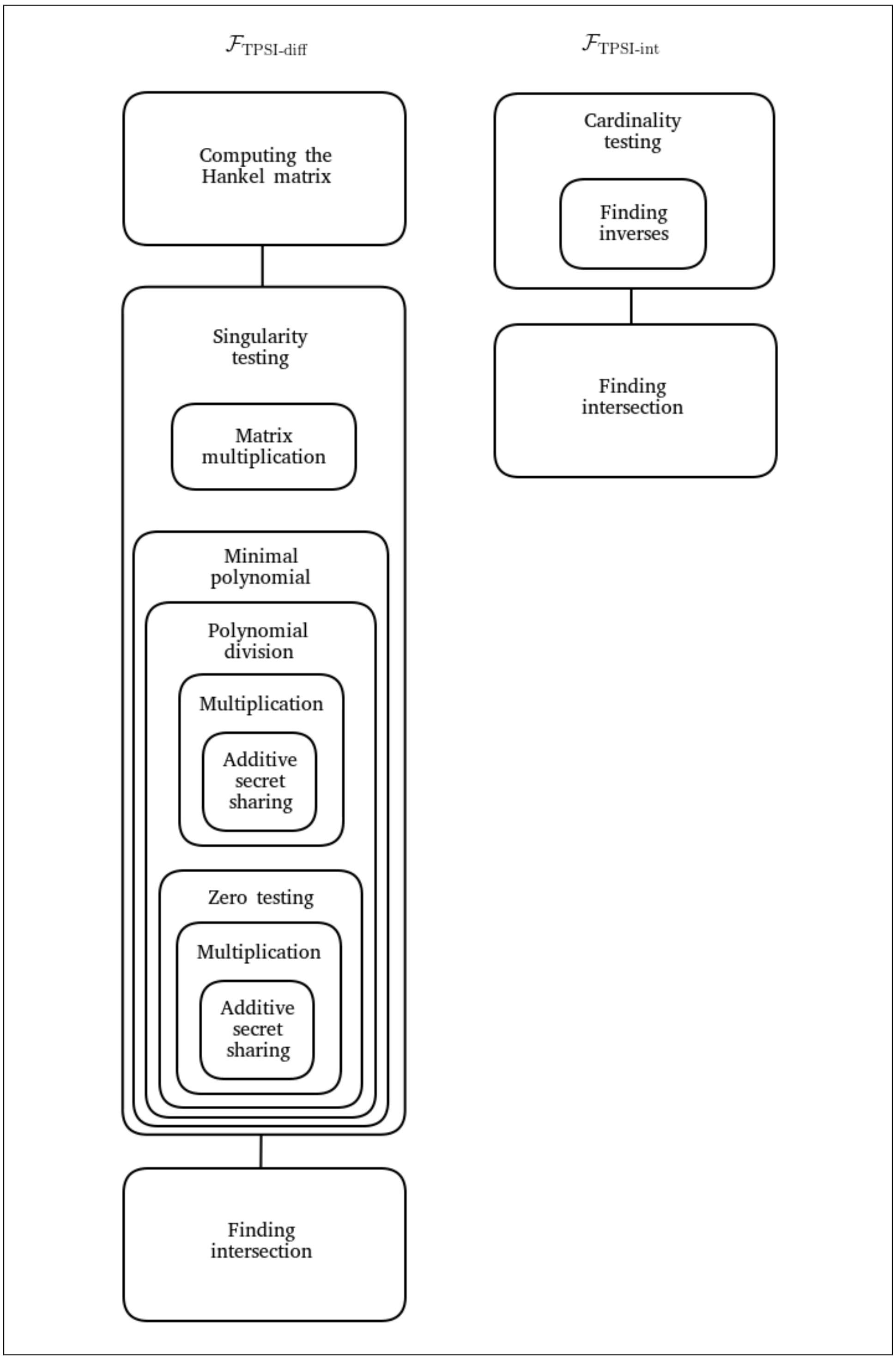


Figure 3.1: Flow chart of the building blocks that make up the protocols.

```
    // multiplication of two elements
    Multiply(Ciphertext, Ciphertext) (Ciphertext, error)
}

type Secret_key interface {
    PartialDecrypt(Ciphertext)
            (Partial_decryption, error)
}

type Partial_decryption interface {}

type Ciphertext interface {}
```

A plaintext message is typed as `*big.Int`. This is in fact true for tcpaillier, but for Lattigo plaintexts are typed as `uint64` (or rather `[]uint64`) but transforming `uint64` to `*big.Int` is injective and this is expected to be true for all crypto libraries' plaintext types.

For ciphertexts the interface `Ciphertext` was created. This interface is an empty interface and thus any type implements it and so a ciphertext can be of any type. The same is true for `Partial_decryption`, which is the decryption state between one party doing its partial decryption and parties combining all their partial decryptions into the plaintext.

We can see that `FHE_Cryptosystem` simply extends `AHE_Cryptosystem` by adding a `Multiply` function. A distinction is made between scaling and multiplication. For an AHE cryptosystem multiplication between encrypted values are not allowed, but multiplication between one ciphertext and one plaintext is allowed, this is what is denoted as scaling.

The function `EvaluationSpace` refers to evaluation space in Generic Matrix-library, see section 3.3.

`AHE_Cryptosystem` is implemented by `DJ_encryption` to use tcpaillier. This is straightforward and need no further description. However, implementing `BFV_encryption` to use Lattigo in the FHE setting raised som problems and is discussed further in section 3.13.

These structs will be described later.

3.2 Setting

The interfaces `AHE_setting` and `FHE_setting` describe the setting. We needed to distinguish them as they contain the cryptosystem which is of various types (FHE or AHE). Apart from this they also contain the number of parties, the threshold value and communication facilities.

```
type AHE_setting interface {
    // threshold value
    Threshold() int

    // number of parties
    Parties() int

    // ahe cryptosystem
    AHE_cryptosystem() AHE_Cryptosystem

    // used by central party to send a value to all
    Distribute(interface{})

    // used by outer parties to send to central party
    Send(interface{})

    // used by central party to send
    // a message to given party
    SendTo(int, interface{})

    // for central to await messages from all
    // and get them (ordered) in a slice
    ReceiveAll() []interface{}

    // receive a message from central party
    Receive() interface{}

    // true if this party is central
    IsCentral() bool
}

type FHE_setting interface {
    AHE_setting

    // fhe cryptosystem
    FHE_cryptosystem() FHE_Cryptosystem
}
```

For this work we run each party as separate threads on a single machine. For real scenarios however, one probably want to use separate machines. This construction takes and generic approach to communication and can easily be changed to another way of communicating.

The interfaces are implemented by the two structs `AHESetting` and `FHESetting`.

```
type AHESetting struct {
    cs AHE_Cryptosystem
    n int // number of participants
    T int // threshold
    channels []chan interface{}
    channel chan interface{}
}
```

```
type FHESetting struct {
    AHESetting
    cs FHE_Cryptosystem
}
```

3.3 Generic Matrix

In order to perform cardinality testing for AHE, we need support for matrix operations. There exists a widely used third-party mathematical library for the Go programming language, namely Gonum. Specifically, it includes a sub-package including matrix structures and linear algebra operations [Gon]. This sub-package, however, support 64 bit float elements only, which is unfit for the encrypted data we will use. To overcome this issue a *Generic Matrix* library was implemented as a part of this project. This library is available at [Jepa].

The aim of this package is to enable basic matrix operations, both in the plaintext space and in the ciphertext space, and to do it in a modular way for use in different scenarios. To allow this, an interface `Space` was constructed. It looks the following way

```
type Space interface {

    // addition of two elements in the space
    Add(interface{}, interface{})
        (sum interface{}, err error)

    // subtraction of two elements in the space
    Subtract(interface{}, interface{})
        (diff interface{}, err error)

    // multiplication of two elements in the space
    Multiply(interface{}, interface{})
        (product interface{}, err error)

    // scaling of an element by scalar factor
    Scale(spaced interface{}, factor interface{})
        (product interface{}, err error)

    // return true if this space (matrix) consists
    // of scalar factors i.e. if Scale are
    // to be used in matrix multiplication
    Scalarspace() bool
}
```

Every matrix stores a struct implementing this interface, to allow evaluation of the operations in the current space. In this way it's easy to extend the library with other evaluation spaces. To support the desired functionality of TPSI two types implementing this interface was constructed, `Bigint` and `DJ_public_key` to perform evaluations in the plaintext space and ciphertext space respectively.

The matrix struct is implemented as following, containing a `[]interface{}` with all the elements, two integers `Rows` and `Cols`, storing the number of rows and the number of columns and the cryptosystem at hand.

```
type Matrix struct {
    values []interface{}
    Rows, Cols int
    Space Space
}
```

The row and column numbering is zero-based (evident in `Set` and `At`), which is common in programming and gives a slightly easier formula for getting and setting.

The following operations was implemented.

- `NewMatrix(rows, cols int, data []interface{}, space Space) (m Matrix, err error)` - constructor
- `(m Matrix) At(row, col int) (interface{}, error)` - get value at given position
- `(m Matrix) Set(row, col int, value interface{}) error` - set value at given position
- `(a Matrix) Multiply(b Matrix) (c Matrix, err error)` - matrix multiplication
- `(a Matrix) MultiplyScalar(scalar interface{}) (Matrix, error)` - scalar multiplication when `scalar` and `a` share the same space
- `(a Matrix) Scale(factor interface{}) (Matrix, error)` - scalar multiplication to be used if `factor` is in a scalar space while `a` is not
- `(a Matrix) Add(b Matrix) (c Matrix, err error)` - matrix addition
- `(a Matrix) Subtract(b Matrix) (c Matrix, err error)` - matrix subtraction
- `(a Matrix) Concatenate(b Matrix) (Matrix, error)` - horizontal concatenation
- `(a Matrix) CropHorizontally(k int)` - horizontal crop
- `(a Matrix) Apply(f func(interface{}) (interface{}, error)) (b Matrix, err error)` - apply a function to all matrix elements

Below is the addition function shown. We can see how the functions calls the user-defined `a.Space.Add` to perform addition on each element.

```
// matrix addition
func (a Matrix) Add(b Matrix) (c Matrix, err error) {
    if a.Rows != b.Rows || a.Cols != b.Cols {
        err = fmt.Errorf("dimension␣mismatch")
        return
    }
    c_vals := make([]interface{}, len(a.values))
    for i := range c_vals {
        c_vals[i], err = a.Space.Add(a.values[i],
                b.values[i])
        if err != nil {return a, err}
    }
    return NewMatrix(a.Rows, a.Cols, c_vals, a.Space)
}
```

The elements are stored in row-major order, meaning that the formula for fetching element `(r,c)` is `Cols*r + c`. The reason for choosing this approach is that it is the most intuitive, as it is the way we read. The only place where this matters for implementation is in the creation of matrices (`NewMatrix`), and in `Concatenate` and `CropHorizontally`, where we merge and split matrices horizontally. In the latter case column-major order would have been given a faster implementation as then we could have just concatenated / split the value slices but constructing the matrix as a sequence of rows is more intuitive and user friendly.

Extra care was needed for differentiating various kinds of scalar multiplication. As the elements are `interface{}` it is not possible to distinguish between multiplication by plaintext scalar or ciphertext scalar. For that reason two functions exist, `Scale` and `MultiplyScalar`. In fact the latter won't be used in this project since it is not allowed with AHE, but the functionality is needed for the library to be useful in other cases.

The same applies to matrix multiplication, where we have even more scenarios; we might use matrices from the same or different spaces, and we might do the multiplication from left or right. To handle this we have the function `Scalarspace` in the interface `Space`. It returns true if matrix multiplication with this matrix should be performed with `Scale` and false if it should be performed with `Multiply`.

3.4 Computing the Hankel matrix

The first step of $\mathcal{F}_{\text{CTest-diff}}$ is to create a Hankel Matrix. This is done by each party setting a polynomial $p_i(x) = \sum_{j=1}^{m} x^{a_j}$, where a_j is the jth item in the party's set, and evaluating it at u^0, u^1 ... u^{2T} for u sampled by P_1. This is done unencrypted and is thus easily achieved. We use the slice `u1_list` which stores the values of u^{a_j} for each j throughout the execution and the slice `u_list`, which stores $(u^k)^{a_j} = u^{ka_j}$ for each j and the current iteration k. We use a `for`-loop that iterates over all diagonals, except the first, and thus runs $2T$ times. We know that H will be zero at the first entry since all parties have the same number of elements, so we start our iteration at the second diagonal, with `i = 1`. For each iteration we sum the elements of `u_list` and insert them on the corresponding diagonal. If we are to do another iteration, we do an element-wise multiplication of the elements in the slices `u_list` and `u1_list` to get the new state of `u_list`.

The Hankel matrix looks as follows

$$H = \begin{pmatrix} p(u^0) & p(u^1) & \cdots & p(u^T) \\ p(u^1) & p(u^2) & \cdots & p(u^{T+1}) \\ \vdots & \vdots & \ddots & \vdots \\ p(u^T) & p(u^{T+1}) & \cdots & p(u^{2T}) \end{pmatrix} \tag{3.1}$$

The position of the diagonal is decided by noting that if `i <= T` the starting column (`startCol`) is `0` and the final column is `i+1`, or else, if `i > T` the starting column is `i-T` and the final column is `T+1`. The starting row will always be `i-startCol` and now we can iterate through decreasing row and increasing column until we reach the final column.

The matrix is then encrypted item-wise and passed to P_1. The polynomial of P_1 looks slightly different: $p_1(x) = \sum_{j=1}^{m}(n-1)x^{a_j}$. This can be simplified as $p_1(x) = (n-1)\sum_{j=1}^{m} x^{a_j}$, or, (computationally) equivalent, calculating the matrix as for P_i and multiplying by $(n-1)$ lastly. P_1 sums the outer parties' matrices and subtracts with its own, this will cause all shared monomials to disappear, since P_1 uses the factor $n-1$.

P_1 initiates the calculations by sampling `u` uniformly random from the plaintext space and passing it to all parties. The parties send back their encrypted matrices, which each is subtracted from P_1's matrix. This will cause $p = p_1 - \sum_{i>1} p_i$ to cancel out all roots that are elements of all parties. Now we can perform a singularity test to determine whether p has low enough degree.

Finding the Hankel matrix is implemented in `CentralHankelMatrix` and `OuterHankelMatrix` and described in Figures 3.2 and 3.3.

3.5 Matrix multiplication

In order to perform the singularity testing we will need a protocol for performing matrix multiplication of encrypted matrices in the TAHE setting. Such a protocol is provided in [BMR20].

The protocol Π_{MMult} inputs two encrypted matrices $[\![A]\!]$ and $[\![B]\!]$ and outputs their product $[\![A \cdot B]\!]$. The implementation consists of four functions, each one representing one of the four steps of the protocol as described in [BMR20]. The protocol is described in Figure 3.4. Mathematically it works as follows.

Each party uniformly samples two matrices R_i^A and R_i^B with elements from the plaintext space, those are sent encrypted to P_1. P_1 sums $R^A = \sum_{i=1}^{n} R_i^A$, $M^A = A + R^A$ and $M^B = B + \sum_{i=1}^{n} R_i^B$. Each party calculates $ct_i = [\![R^A R_i^B - M^A R_i^B - R_i^A M^B]\!] = [\![R^A]\!] R_i^B - [\![M^A]\!] R_i^B - R_i^A [\![M^B]\!]$ which is possible since unencrypted R_i^A and R_i^B are stored locally. Parties collectively decrypt M^A and M^B too calculate $[\![M^A \cdot M^B]\!]$ which allows for the final calculation of

$$[\![A \cdot B]\!] = [\![M^A \cdot M^B]\!] + \sum_{i=1}^{n} ct_i$$

The functionality is implemented in the functions `CentralMatrixMultiplicationWorker` and `OuterMatrixMultiplicationWorker`.

Input slice `items` containing all elements; random `u`.
Output the party's Hankel matrix `H`.

```
1. Initiate a new matrix H with size $T \times T$ and set to 0 at H[0,0].
2. Initiate u_list and u1_list same length as items, containing the values: $u^{a_i} \mod q$ for each element $a_i$ in items and plaintext field size $q$.
3. For i := 1, 2, ...
    (a) If i <= $T$; then set startCol = 0 and stopCol = i + 1.
    (b) Else set startCol = i - $T$ and stopCol = $T$ + 1.
    (c) Sum the values of u_list into el.
    (d) Starting at row i-startCol and column startCol, traverse H diagonally up-right and set the elements on this diagonal to el, until stopCol is reached.
    (e) If i >= $2T$; return H
    (f) Else item-wise multiply u_list and u1_list and store to u_list.
```

Figure 3.2: Functionality of `ComputePlainHankelMatrix`

Input slice `items` containing all elements.
Output Hankel matrix `H` for singularity testing.

1. P_1 samples a value `u` uniformly random from the plaintext field and sends to all parties.
2. Outer parties call `ComputeHankelMatrix`, which
   (a) calls `ComputePlainHankelMatrix` to receive `H`
   (b) returns encryption of `H`
3. P_1 calls `CPComputeHankelMatrix`, which
   (a) calls `ComputePlainHankelMatrix` to receive `H`
   (b) set `H` to product of `H` times the number of outer parties
   (c) returns encryption of `H`
4. All parties send `H` to P_1.
5. P_1 subtracts all outer parties' `H` from its own.
6. This difference is distributed to all parties and returned from the function.

Figure 3.3: Functionality of `CentralHankelMatrix` and `OuterHankelMatrix`

Input matrix `a` of size $k \times s$; matrix `b` of size $s \times l$.
Output matrix product `AB`.

1. All parties call `SampleRMatrices`, which
   (a) uniformly samples two matrices `RA` and `RB` with elements from the plaintext space, and with the same size as `a` and `b`, respectively.
   (b) returns them both encrypted (`RAi_crypt`, `RBi_crypt`) and unencrypted (`RAi_clear`, `RBi_clear`).
2. All parties send `RAi_crypt` and `RBi_crypt` to P_1.
3. P_1 collects incoming matrices in two slices `RAs_crypt` and `RBs_crypt` and call `GetMulMatrices`, which
   (a) sums `RAs_crypt` into a matrix `RA` and sum `RBs_crypt` into `RB`,
   (b) adds `RA` and `A` into `MA`, and adds `RB` and `B` into `MB`,
   (c) returns `RA`, `MA` and `MB`.
4. P_1 sends `RA`, `MA` and `MB` matrices to all parties.
5. All parties call `GetCti` with `RA`, `MA`, `MB`, its `RAi_clear` and its `RBi_clear`. `GetCti`
   (a) calculates `cti` according to $ct_i = [\![R^A R_i^B - M^A R_i^B - R_i^A M^B]\!] = [\![R^A]\!] R_i^B - [\![M^A]\!] R_i^B - R_i^A [\![M^B]\!]$
   (b) returns `cti` together with partial decryptions of `MA` (`MA_part`) and `MB` (`MB_part`).
6. All parties send `cti`, `MA_part` and `MB_part` to P_1.
7. Central party collects incoming matrices in slices and calls `CombineMatrixMultiplication`, which
   (a) decrypts `MA` and `MB`
   (b) multiplies `MA` and `MB` and encrypts them back into `AB`
   (c) returns the sum of `AB` and all parties `cti`.
8. P_1 distributes the output of `CombineMatrixMultiplication` to all parties.

Figure 3.4: Matrix multiplication protocol, implemented in `CentralMatrixMultiplicationWorker` and `OuterMatrixMultiplicationWorker`

3.6 Singularity testing

The problem of singularity testing can be reduced to computing the minimal polynomial of the linear recurrent sequence $(u^T H^i v) = (h_i)$, with u, v being random vectors [BMR20]. The length of this sequence needs to be twice the dimension of H, i.e. $2(T+1)$.

The communication efficiency of finding the sequence elements is optimized by only computing H^{2^i} for $i \geq 1$ as $H^{2^i} = H^{2^{i-1}} \dot{H}^{2^{i-1}}$. E.g. if we need a sequence up to $i = 5$ it will suffice to compute $2 < \log_2 5$ matrix multiplications in this step. The function `NbrMMultInstances` is provided to find how many times this algorithm needs to run.

```
// calculates how many instances of MMult is needed to
// get all H, according to: n = ceil( log(matrix size) )
// H^2^n being the highest order needed
func NbrMMultInstances(m BigMatrix) int {
     return int(math.Ceil(math.Log2(float64(m.cols))))
}
```

Once we have all H^i we need, a second sequence of Π_{MMult} is initiated as such.

$$\begin{aligned} [\![Hv]\!] &= [\![H]\!] \cdot [\![v]\!] \\ [\![H^3v|H^2v]\!] &= [\![H^2]\!] \cdot [\![Hv|v]\!] \\ [\![H^7v|H^6v|H^5v|H^4v]\!] &= [\![H^4]\!] \cdot [\![H^3v|H^2v|Hv|v]\!] \end{aligned}$$

$X|Y$ denotes the horizontal concatenation of matrices X and Y. By incrementally concatenating the resulting matrices from Π_{MMult} and finally multiplying this matrix from left by u^T, we get the sequence as a row matrix. The function `HSeq` lets the central party sample u, multiply it by the matrix and additionally crop it to the desired length. This gives us the linearly recurrent sequence. Cropping and concatenating is provided in Generic Matrix library.

The following section describes how to find the minimal polynomial from the linearly recurrent sequence. After finding the minimal polynomial we perform a zero test to see whether the constant term of the polynomial is zero or not. If it is zero, that means the matrix `H` is singular thus the cardinality test passed.

3.7 Minimal polynomial of linearly recurrent sequence

[BMR20] doesn't provide a protocol for finding the minimal polynomial of a linearly recurrent sequence. The following protocol is a contribution of this thesis. We will use the Extended Euclidean Algorithm to create the protocol. First we define the minimal polynomial for a linearly recurrent sequence.

A infinite sequence $\mathbf{a} = (a_i)_{i \in N}$ is linear recurrent if there exists $f_0, \ldots, f_n$ with $f_n \neq 0$ such that $\sum_{j=0}^{n} f_j a_{i+j} = 0$ for all $i \in N$. The polynomial $f = \sum_{j=0}^{n} f_j x^j$ of degree n is called a characteristic polynomial of $\mathbf{a}$. There exists a unique monic polynomial m of least degree that divides all characteristic polynomials of $\mathbf{a}$. This polynomial m is the minimal polynomial of a linearly recurrent sequence $\mathbf{a}$. For a more formal definition, refer to [Kil+07].

The Extended Euclidean Algorithm extends the Euclidean Algorithm such that in addition to finding the greatest common divisor of polynomials a and b it also provides the polynomials u and v of *Bézout's identity* such that $au+bv = gcd(a, b)$. By applying this algorithm to $a = x^{2T+2}$ and $b = \sum_{i=0}^{2T+1} h_i x^{2T+1-i}$ we get a sequence of polynomials such that $r_i(x) = au_i + bv_i$. When $\deg(r_i) < T + 1$, the minimal polynomial m of b is equal to v_i, such that $au_i + bm = r_i$ [Kil+07].

The functionality of finding the minimal polynomial is implemented in `MinPolyWorker`. It iteratively calls `PolynomialDivisionWorker` and `nextT`. `PolynomialDivisionWorker` performs the division between polynomials `a` and `b` and returns the quotient and the remainder, while `nextT` finds the next polynomial v_i described above, using v_{i-1}, v_{i-2} and the quotient. `MinPolyWorker` is initiated by setting the polynomial $a = x^{2T+2}$ as mentioned above, and $v_0 = 0$, $v_1 = 1$ according to Extended Euclidean Algorithm. When the degree of the remainder is less or equal to v_i, the iteration is terminated.

The functionality of `PolynomialDivisionWorker` is described in section 3.11 and Figure 3.5. This functionality requires the availability of multiplication and testing for zero, which in turn require additive secret sharing. Multiplication using additive homomorphic cryptosystem in the multi-party setting and additive secret sharing are described in [CDN01]. Zero testing is a trivial functionality arising from the availability of multiplication. Those protocols are described below.

3.8 Additive secret sharing

Cramer et al. describe in [CDN01] how to construct arbitrary MPC protocols from Additive Threshold Homomorphic cryptosystems, which is what we use to solve the problem of minimal polynomial. For such an MPC we will need a way to perform multiplication, which in turn will need a subprotocol for Additive Secret Sharing (denoted ASS by [CDN01]). Secret sharing is a technique for sharing a secret unencrypted among parties. In additive secret sharing specifically, the secret is shared in such a way that the sum of all shares is the secret. ASS is implemented in `CentralASSWorker` and `OuterASSWorker`, the aim is to additively share encrypted `a` in plaintext among all parties. Every party samples a random value `d_plain` and encrypts it into `d_enc`. `d_enc` is sent to P_1 who distributes a slice `all_d` containing all `d_enc` back to each party. Each party then homomorphically sums `all_d` and adds it to `a` which therefore is safe to jointly decrypt, we call this decrypted value `e`. Every outer party keeps the additive inverse of its `d_plain` as their share while P_1 keeps `e` subtracted by its `d_plain` as its share. It's easy to see that the sum of all parties shares is `a`.

3.9 Multiplication

Multiplication of two encrypted values `a` and `b` is implemented in `CentralMultWorker` and `OuterMultWorker`. First ASS is run to secret share `a` among all parties. Once `a` is shared each party can multiply it's own share `a_share` with `b`, since `a_share` is unencrypted. Now the product of `a` and `b` is additively secret shared among all parties, and by distributing these shares the product is received by homomorphical summation.

3.10 Zero testing

The functionality of testing whether a value `a` is zero or not is implemented in `CentralZeroTestWorker` and `OuterZeroTestWorker`. Each party sample a mask from the field, encrypts it and sends it to P_1. P_1 collects all masks in a slice and distribute to all parties. All parties sum the masks and multiply by `a`. This product is decrypted and if it is different from zero, we know `a != 0`, or else `a != 0` with negligible probability, provided the field is big enough.

3.11 Polynomial division

As we are unable to perform division or find inverses, we separate the polynomial coefficients into numerator and denominator. The input to `PolynomialDivisionWorker` is two row-matrices `a` and `b`, containing the numerators of the polynomials, and two `Ciphertexts` containing the denominator for each polynomial as this will always be the same for our use of polynomial division. Using zero tests, we find the degree of `a` as `la` and the degree of `b` as `lb`. We find the maximum size of the quotient polynomial as `ql := 1+la-lb` and initiate the quotient polynomial (`q_num`, `q_den`) by setting `q_num` to a zero polynomial of length `ql` and `q_den` to an all-ones polynomial of the same length.

Now, for the main part of the algorithm we iterate over `i` through all coefficients of `a`. Initially `i := la` and then it decreases for each loop.

We start the loop by checking that the current coefficient of `a` is not equal to 0. For the first iteration we know this to be true, but it might not be the case in later iterations. In case it is equal to 0, we call `continue` to go to the next iteration. We assign `pos := i-lb` to be the position of the quotient polynomial this iteration evaluates. At this position we insert the division of current coefficient of `a` and the highest order coefficient of `b`, this is done by multiplying the numerator of `a` by the denominator of `b` and inserting it to `q_num` and vice versa for `q_den`, according to section 3.9. Next, we create a polynomial p as a row-matrix `p_den` and a `Ciphertext` `p_den`. This is the polynomial that will bring us the remainder polynomial for the current iteration. We initiate it as the product of the polynomial b and the current value of q. However we skip the highest degree coefficient as we know that it will cancel out with the highest degree coefficient of a. After multiplication we want to normalize it to have the same denominator as a in order to perform a subtraction. This is achieved by multiplying a with the denominator of p, we call this polynomial r, and multiplying p with the denominator of a. Both polynomials now share the same denominator `r_den`. We let the central party P_1 subtract p from r by item-wise subtracting the corresponding numerators and then P_1 distributes this to all the parties. The received r is assigned to `a_num` and `a_den` and we jump to a new iteration of the loop.

The loop terminates when `i < lb` as then the divisor b has a higher degree than the dividend a.

After termination of the loop we strip a of eventual leading zeroes and then return `q_num`, `q_den`, `a_num` and `a_den`.

This functionality is described in Figure 3.5.

3.12 Finding intersection

If the cardinality test passed successfully, that means we can move on to finding the intersection, this is done by `IntersectionWorker`.

Each party encodes its elements as roots of a polynomial $p_i(x)$ using `PolyFromRoots`. `PolyFromRoots` simply takes the first element a_0 and creates a linear polynomial from it $p(x) = x - a_0$, for the following elements we again create linear polynomials that are multiplied in to p. The multiplication is performed by `MultPoly` and multiplies by multiplying the coefficients. `CentralIntersectionPolyWorker` and `OuterIntersectionPolyWorker` are called with $p_i(x)$. Their task is to find the desired polynomial $V(x)$, defined as

$$V(x) = \sum_{i=1}^{n} \Big(p'_i(x) \cdot \Big(R_1(x) + \ldots + R_{i-1}(x) + \tilde{R}_i(x) + R_{i+1}(x) + \ldots + R_n(x) \Big) \Big)$$

Here $p'_i(x)$ is $p_i(x)$ with an extra randomizing root. R_i and $\tilde{R}_i$ are masking polynomials of degree $T+1$. Each party samples one polynomial R to distribute and mask all other parties p', and one polynomial $\tilde{R}$ to mask its own p'.

The first thing happening in `CentralIntersectionPolyWorker` and `OuterIntersectionPolyWorker` is that the extra root is added to p using `RootMask`. Next `EvalIntPolys` is called, which samples the polynomials R and $\tilde{R}$ and evaluates them and p' at $3T+4$ points.

Since we encounter problems if these evaluation points are roots to a polynomial we must ensure that they do not coincide with the elements, since the elements are in fact roots. To achieve this, the decision was made to evaluate at points $2i+1$ for $i = 0, 1, \ldots$ and encode the elements as $2e$ for an element e.

`EvalIntPolys` returns three row matrices: `R_values_enc` with encrypted evaluations of R, `R_tilde_values` with unencrypted evaluations of $\tilde{R}$ and `p_values` with unencrypted evaluations of p'. `R_values_enc` are sent to P_1 which for every party i sums `R_values_enc` of all parties except i. This row matrix, called `party_values`, is sent to i. Next, all parties call `MaskRootPoly` which encrypts `R_tilde_values`, sums it with `party_values` and item-wise multiplies by `p_values`. This row matrix `v` is sent to P_1 and by summation P_1 gets V which is distributed to all parties and jointly decrypted.

3.12.1 Rational polynomial interpolation

In `Interpolation` we input two row matrices `vs` and `ps` containing $3T+4$ evaluations of $V(x)$ and $p'(x)$, they are immediately used to calculate $3T+4$ evaluations of $q(x) = \frac{V(x)}{p'(x)}$ and not used apart from that. $3T+4$ is the maximum number

Input row matrices `a` and `b` expressing numerator coefficients; ciphertext values `a_den` and `b_den` expressing a denominator.
Output three row matrices: numerator of the quotient, denominator of the quotient, numerator of the remainder; one ciphertext: denominator of the remainder.

1. Set `la` to index of highest degree non-zero coefficient of `a` and `lb` to index of highest degree non-zero coefficient of `b`.
2. Set `ql = 1 + la - lb` and `a_num = a`.
3. P_1 encrypts an all zero row matrix `q_num` of length `ql` and an all ones row matrix `q_den` of length `ql`. `q_num` and `q_den` are distributed to all parties.
4. **For** `i := la, la-1, ..., lb`
   - (a) **If** the coefficient of `a_num` at `i` equals 0, **then continue**.
   - (b) Set `pos = i-lb`. This is the position in the quotient polynomial `q` we are currently at.
   - (c) Find the numerator `num` at `pos` of `q_num` by multiplying `a_num[i] * b_den` and the denominator `den` at `pos` of `q_den` by multiplying `b[lb] * a_den`.
   - (d) Multiply `b * num` into a new row matrix `p_num` and `b_den * den` into `p_den`.
   - (e) Give `a` and `p` the same denominator by
     - i. multiplying `a_num` and `p_den` into an new row matrix `r_num`
     - ii. updating `p_num` by multiplying by `a_den`
     - iii. multiplying `a_den` and `p_den` into `r_den`.
   - (f) `p` and `r` now share the same denominator and are eligable for subtraction which is performed by P_1 using a function `divSub` and distributed to all parties.
   - (g) `a_num` is set to this difference and `a_den` is set to `r_den`.
5. Remove intial zero coefficients from `a_num` and return `q_num`, `q_den`, `a_num` and `a_den`.

Figure 3.5: Functionality of `PolynomialDivisionWorker`.

Input a row matrix `RootPoly` with all elements encoded into roots.
Output row matrix of samples of V and p' to perform interpolation

1. Each party calls `RootMask` which adds a random root to `root_poly`.
2. Each party calls `EvalIntPolys` which
   (a) samples two random polynomials R and $\tilde{R}$,
   (b) computes $R(x)$, $\tilde{R}(x)$ and $p(x)$ for $x = 2i+1,\ i = 0, 1, \ldots, 3T+4$
   (c) returns evaluations of $[\![R(x)]\!]$, $\tilde{R}(x)$ and $p(x)$ in row matrices `R_values_enc`, `R_tilde_values` and `p_values`
3. Each party sends `R_values_enc` to P_1.
4. P_1 does the following: **For** each party P_i
   (a) sums all `R_values_enc` except P_i's into the row matrix `party_values`
   (b) sends `party_values` to P_i
5. Each party calls `MaskRootPoly` which calculates that party's contribution to $V(x)$ by
   (a) element-wise adding `R_tilde_values` and `party_values`
   (b) element-wise multiplying the sum with `p_values` into `v`

   `v` is passed to P_1.
6. P_1 updates `v` by summing all parties `v` and distributes it.
7. Each party partially decrypts `v` using `PartialDecryptMatrix` and sends to P_1.
8. P_1 decrypts `v` and distributes to all parties.
9. `v` and `p_values` are returned.

Figure 3.6: Functionality of `CentralIntersectionPolyWorker` and `OuterIntersectionPolyWorker`.

of evaluations needed to interpolate $\frac{V(x)}{p'(x)}$ and output $p'(x)$. There are $3T+5$ coefficients to interpolate

$$\frac{v_{2T+2}x^{2T+2}+\ldots+v_0}{d_{T+1}x^{T+1}+\ldots+d_0}=q(x) \tag{3.2}$$

which require $3T+5$ evaluations to be uniquely determined. In our case, we loose one constraint as we are only interested in the roots, which allows for any scaling of the polynomials, that means we can set the highest non-zero coefficient of p' to 1 for p' to be monic. We rename all unknown coefficients c such that $v_i = c_i$ and $d_i = c_{i+2T+3}$. `eq` is a row matrix that expresses the full equation for each evaluation of $q(x)$. It is received by multiplying both sides of equation 3.2 by the denominator and subtracting the right hand side. The ith element of `eq` corresponds to e_i in the equation. The mathematical formulation of `eq` is shown below (renaming v and d).

$$\begin{aligned}
0 =& v_0+\ldots+v_{2T+2}x^{2T+2}-d_0q(x)-\ldots-d_{T+1}q(x)x^{T+1} && (3.3)\\
=& c_0+\ldots+c_{2T+2}x^{2T+2}-c_{2T+3}q(x)-\ldots-c_{3T+4}q(x)x^{T+1} && (3.4)\\
=& c_0e_0+\ldots+c_{2T+2}e_{2T+2}+c_{2T+3}e_{2T+3}+\ldots+c_{2T+3}e_{3T+4} && (3.5)
\end{aligned}$$

This equation is used once for each evaluation of $q(x)$, or until we are able to determine $p'(x)$.
We also use a matrix of relations, `relations`. The ith row of `relations` expresses how c_i is related to the succeeding coefficients, such that $c_i = r_{i,j}c_j + r_{i,j+1}c_{j+1} + \ldots + r_{i,3T+4}c_{3T+4}$. When all relations are determined, we can set the last non-zero coefficient $c_k = 1$ and going from bottom up in `relations` we will be able to uniquely determine all coefficients, one per row. The matrix `relations` is as follows.

$$\text{relations} = \begin{pmatrix} 0 & r_{0,1} & r_{0,2} & \cdots & r_{0,3T+3} & r_{0,3T+4} \\ 0 & 0 & r_{1,2} & \cdots & r_{1,3T+3} & r_{1,3T+4} \\ \vdots & & & \ddots & & \vdots \\ 0 & 0 & 0 & \ldots & 0 & r_{3T+3,3T+4} \end{pmatrix}$$

To build this matrix we iterate through all c, starting from c_0. For each c_i, we start by expressing the equation `eq`. Then by fetching the jth row of `relations` for $j < i$ starting with $j = 0$ we can multiply it by e_j and add to the equation, causing the cancellation of c_j. Eventually we will get an equation where c_i is the first unknown coefficient. Now we can solve for this coefficient and add the relation to `relations`.

While iterating through all c in this way, we will eventually reach one of two cases. If we are precisely at the threshold T, the loop will terminate after doing the last evaluation of q. In this case we set $c_{3T+4} = p_{T+1} = 1$ and traverse `relations` from bottom up. If, on the other hand, the number of elements not shared by all is below T, the substituted equation will eventually read $0 = 0c_i + 0c_{i+1} + \ldots + 0c_{3T+4}$. In this case we know that c_i is unrelated to all succeeding coefficients. Thus we set $c_i = 1$, $c_j = 0$ for all $j > i$ and determine all preceding elements, starting from the $(i-1)$th row of `relations`.

```
Input vs, samples of $V$; ps, samples of $p'$.
Output denumerator of $\frac{V(x)}{p'(x)}$

1. By item-wise multiplication of vs and inverse of ps we get evaluations
   of $q(x)$ in q.
2. For each element at index coeff_pos of q
   (a) The row matrix eq is instantiated so that the sum of all elements
       is 0 according to equation 3.3.
   (b) For each row (index prev_coeff) of relations with
       prev_coeff < coeff_pos
       i. multiply the row relations[prev_coeff] by eq[coeff_pos]
          into crel
       ii. update eq by adding crel to it
       iii. set eq[prev_coeff] to 0
   (c) If eq[coeff_pos] == 0, terminate the loop.
   (d) Create a new row matrix rel_row and set to 0 for all indices less
       than coeff_pos + 1
   (e) For all remaining indices rem_coeff in rel_row
       i. set rem = 0 - eq[rem_coeff]
       ii. set rel_row[rem_coeff] to product of rem and inverse of
           eq[coeff_pos]
   (f) Insert rel_row at relations[coeff_pos].
3. Create a slice interpolated_coeffs and set
   interpolated_coeffs[coeff_pos] = 0.
4. For each line in relations with index solving_coeff < coeff_pos.
   (a) item-wise multiply this line with interpolated_coeffs
   (b) sum the all values into interpolated_coeffs[solving_coeff]
5. Create a matrix containing the elements from interpolated_coeffs
   starting from index $2T + 3$ up to coeff_pos. This is the interpolated
   denumerator excluding higher order zeroes.
```

Figure 3.7: Function for rational polynomial interpolation

3.13 Lattigo

For the protocols that uses FHE, we will use the BFV scheme implemented in Lattigo. The original BFV scheme contains a bootstrapping function that needs to be called after some multiplications for the inherent error not to affect the encrypted message. Lattigo, however, does not provide this function, instead they provide a *collective bootstrapping* function which refreshes the ciphertext alike the proper bootstrapping, but for this part all parties need to participate. This forces us to diverge from the protocol of [BMR20]. In [BMR20], the authors assume a cryptosystem where one party (P_1) can do arbitrary calculations which is not the case for our cryptosystem. We bypass this problem in the following way: Whenever the protocol tells P_1 to do something, all parties follow along and do the same calculations. In this way all parties are at the same state when the ciphertext needs to be refreshed.

This however doesn't fully solve the problem as the probabilistic structure of the scheme gives varying ciphertext output for the same input to the multiplication function. To avoid this, whenever a multiplication is performed it is performed by P_1 only and then distributed to all parties. This means that no party can perform any multiplication on its own, but that is not needed for this approach. This way of constraining the FHE scheme brings the functionality down to what was previously achieved by the AHE, where we too could do multiplication in collaboration. So in theory, it is possible to use AHE also for this part, if using our alternate protocol.

To know when it is time to refresh the ciphertext, a simple counter was embedded in the ciphertext.

```
type BFV_ciphertext struct {
    msg *bfv.Ciphertext
    mult_counter int
}
```

This counter is incremented for each multiplication and when a certain value is reached the ciphertext is refreshed. The threshold for this was set to `mult_limit := 6` which works for the current use-case. This limit and its incrementation is a point of improvement, but that is out of scope for this thesis.

The BFV implementation implements the interface `FHE_Cryptosystem` described in section 3.1. To enable the communication as described above communication facilities was needed to be embedded in the cryptosystem. Below is the struct `BFV_encryption` which is the one implementing `FHE_Cryptosystem`. The multiplication function is also shown.

```
type BFV_encryption struct {
    params *bfv.Parameters
    crs *ring.Poly
    crp []*ring.Poly
    pk *bfv.PublicKey
    rlk *bfv.EvaluationKey
    tpk *bfv.PublicKey
    tsk *bfv.SecretKey
    sk *bfv.SecretKey
    channels []chan interface{}
    channel chan interface{}
}

func (pk BFV_encryption) Multiply(a, b Ciphertext)
        (product Ciphertext, err error) {
    mult_limit := 6
    ac := a.(BFV_ciphertext)
    bc := b.(BFV_ciphertext)

    if ac.mult_counter >= mult_limit {
        if pk.channels != nil {
            ac = CentralRefresh(ac, pk)
        } else {
            ac = OuterRefresh(ac, pk)
        }
    }
    if bc.mult_counter >= mult_limit {
        if pk.channels != nil {
            bc = CentralRefresh(bc, pk)
        } else {
            bc = OuterRefresh(bc, pk)
        }
    }

    var prod *bfv.Ciphertext
    if pk.channels != nil {
        evaluator := bfv.NewEvaluator(pk.params)
        prod = evaluator.MulNew(ac.msg, bc.msg)
        evaluator.Relinearize(prod, pk.rlk, prod)
        for _, ch := range pk.channels {
            ch <- prod
        }
    } else {
        prod = (<-pk.channel).(*bfv.Ciphertext)
    }
    return BFV_ciphertext{msg: prod,
        mult_counter: mulMax(ac, bc) + 1}, nil
}
```

As seen, `BFV_encryption` contains `channel` and `channels` to enable communication. Unfortunately is also contains the secret key, as it is needed inside `CentralRefresh` and `OuterRefresh`. This blurrs the distinction between public and private, but that's however only a conceptual difference and not a practical one.

Above is also shown the implementation of `Multiply` with the multiplication counter and the refresh functions acting.

Another subtlety with this scheme is that after doing the partial decryption of a ciphertext, the original ciphertext is needed to combine into the plaintext. To solve this, the ciphertext was embedded in the partial decryption.

```
type BFV_partial struct {
     part dbfv.PCKSShare
     ciphertext BFV_ciphertext
}
```

3.14 Finding inverses

For implementing $\mathcal{F}_{\text{TPSI-int}}$ we need to be able to calculate the multiplicative inverse of a ciphertext. This is done using an interactive mini protocol among parties, implemented in `CentralInverseWorker` and `OuterInverseWorker`. Actually, those functions are convenience functions for `CentralInverseWorkerWithFactor` and `OuterInverseWorkerWithFactor`. The latter functions takes an extra `factor *big.Int` as an argument and returns the product of `factor` and the inverse. Since the FHE scheme we are using is limited to performing a few homomorphic multiplications before needing the collective bootstrapping we can instead use this technique when we have a plaintext factor to do plaintext multiplication and thus skip one homomorphic multiplication. Even if we had the proper bootstrapping function available, it is in most schemes very heavy and is preferable to avoid anyhow.

In `CentralInverseWorkerWithFactor` and `OuterInverseWorkerWithFactor` each party samples a mask value from the plaintext space and sends it to P_1 who distributes all masks to all parties. Now all masks are multiplied to `a` (the ciphertext to invert) and it is safe to decrypt it. In the decrypted state we find the inverse and multiplies by our `factor`. By encrypting and again multiplying by all masks, we receive our desired inverse.

3.15 FHE cardinality testing

For this part each party, again, encodes their elements as roots in a polynomial $p'(x)$ using `PolyFromRoots`, and add an extra random root as a mask. The goal is to see if the rational polynomial

$$q(x) = \frac{p'_2(x) + \ldots + p'_n(x)}{p'_1(x)}$$

can be correctly interpolated using $2T + 3$ evaluation points. If so, that implies that each $p'(x)$ only has at most T roots left after division (i.e. not shared by all) and thus the cardinality test passes. In `plain_evals` $2T+3$ evaluations of $q(x)$ for $x = 2i + 1, i = 0,\ 1,\ \ldots$ is stored, `evals` stores the same values encrypted. This is except for P_1 who first finds the multiplicative inverse before encrypting. P_1 also uniformly samples a value z from the plaintext field. This z works as the predicate to verify whether the interpolation is correctly determined. Each party evaluates and encrypts $p'_i(z)$ in `z_eval`, except P_1 who encrypts the inverse of $p'_1(z)$ into `z_eval`. The slices `evals` and `z_eval` is sent to P_1, who collects all `evals` into the slices of slices `all_evals` and all `z_evals` into the slice `z_evals`. They are then distributed back to all parties.

All parties homomorphically calculate the expected evaluation of $q(z)$ and store it in `z_exp` by summing `z_evals` of the outer parties and multiplying `z_eval` of P_1 as it is already inverted. $p'_1(z)$ is stored at the last index of the slice, `z_evals[n-1]`.

The same is done for each evaluation point in `all_evals`. All evaluations of $q(x)$ are stored in a slice `evals_sum`. This slice is passed to the function `FHEInterpolation` to perform the interpolation. The methodology for `FHEInterpolation` is the same as for `Interpolation` in section 3.12.1, except of course now it is done homomorphically. Refer to section 3.12.1 for a description. `Interpolation` returns only the polynomial of the denominator while in this case we need both the numerator and the denominator, so all coefficients are returned in one slice, the first $T + 2$ are the coefficients of the numerator, and the rest are the coefficients of the denominator (this might be shorter as highest order zero coefficients are removed in `FHEInterpolation`).

The function `FHEEvaluate` evaluates a polynomial homomorphically, given the polynomial as a slice and the evaluation point. All parties evaluate both the numerator and the denominator at z, and the results are stored in `num_eval` and `den_eval`. By finding the inverse of `den_eval`, multiplying by `num_eval` and subtracting from the `z_exp` we get the predicate `pred`. If `pred` is 0, we know that the polynomial q was correctly interpolated and the cardinality test has thus passed, otherwise the cardinality test failed. To see whether or not it is 0, `CentralZeroTestWorker` and `OuterZeroTestWorker` are called.

Chapter 4

Experimental results

4.1 Setup

For our experimental setup, we took as an use-case the creation of support groups. Each person answers a questionnaire with potential issues, such as *Do you suffer from alcoholism?*. If all parties share enough such issues, the support group can be created. To run the protocols implemented in this thesis, all parties must have sets of the same size m. So we have m *yes*- and *no*-questions. Each answer needs to be encoded into some element of the parties set. The yes-answers need of course be encoded into the same value for all, however for no-answers we need to make sure that they are encoded into at least two different values so that they are not in the intersection and thus disclosed. If we would encode all no-answers equally, no-answers would possibly appear in the intersection and the cardinality test would pass although the amount of shared issues are to small. To optimize $\mathcal{F}_{\text{TPSI-diff}}$ it is efficient to have as few elements as possible in the set difference, and thus we encode no-answers into only two different values. One easy way to do this is the following: For question i, encode *yes* as $i \cdot 3$, encode *no* as $i \cdot 3 + 1$ if you are P_1, encode *no* as $i \cdot 3 + 2$ if you are an outer party.

To run the protocols a simple main program was written, which in addition to running the protocols, encodes the values into even numbers, to avoid conflicting with the sampled values, as mentioned in section 3.12. The main program accepts as input a protocol `int` or `diff`, a cryptosystem `bfv` (FHE) or `dj` (AHE), a threshold value T, and a file containing all the elements, comma-separated, with one line per party. The basic test case was the elements in Figure 4.1, simulating the questionnaire result described above. The right section shows the contents of the input file. In this case we have $n = 5$ (5 parties) and $m = 6$ (6 questions). The tests were performed at a Intel® Core$^{\text{TM}}$ i7-6500U Processor with 2 cores and a base frequency of 2.50 GHz. We used the libraries Lattigo for $\mathcal{F}_{\text{TPSI-int}}$ and tcpaillier for $\mathcal{F}_{\text{TPSI-diff}}$ as provided by our implementation. The implementation, together with the main application and the input file, is available at [Jepb].

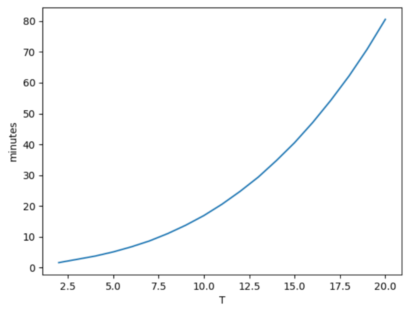
All tests run with $T = 2$ for $\mathcal{F}_{\text{TPSI-int}}$ and $T = 6$ for $\mathcal{F}_{\text{TPSI-diff}}$ as the smallest value of T. This is because that is the smallest value of T where the cardinality test passes, and thus the complete protocol is run.

```
Party 1: yes, yes, yes, yes,  no,  no     |     0,3,6,9,13,16
Party 2: yes, yes, yes, yes,  no,  no     |     0,3,6,9,14,17
Party 3: yes, yes, yes, yes,  no, yes     |     0,3,6,9,14,15
Party 4: yes, yes, yes, yes, yes,  no     |     0,3,6,9,12,17
Party 5: yes, yes, yes, yes, yes, yes     |     0,3,6,9,12,15
```

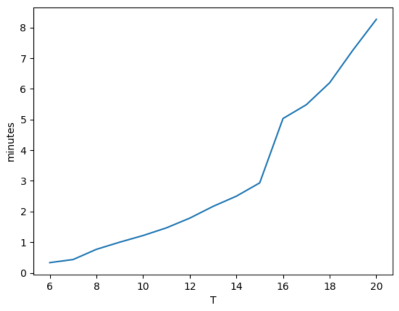
Figure 4.1: Left section contains the imagined questionnaire result. The right section contains those answered encoded according to description, this section is the input to the main application.

4.2 Running time

Figures 4.2a and 4.2b show the running times for this setup with various values of T. As seen, the running time is somewhat acceptable, especially for small T and especially for the AHE scheme (used in $\mathcal{F}_{\text{TPSI-diff}}$). We see that the running time is not linear, and the times quickly increase for higher T. We can make the conclusion that these algorithms are mostly useful for small T only.



(a) Running time for $\mathcal{F}_{\text{TPSI-int}}$ for various values of T.

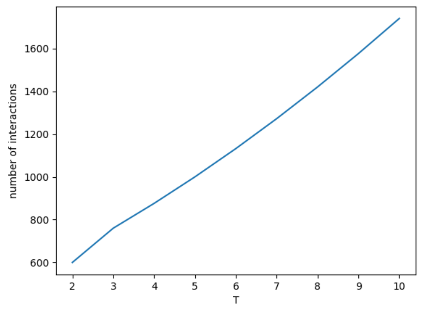


(b) Running time for $\mathcal{F}_{\text{TPSI-diff}}$ for various values of T.

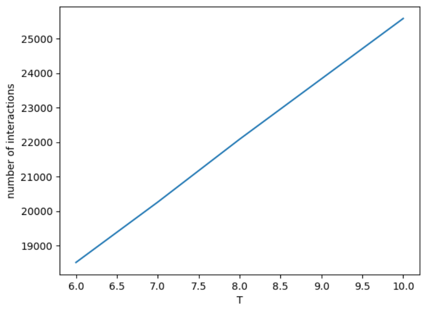
Figure 4.2: Running times for various values of T.

4.3 Communication complexity

In a real scenario, this application would probably run on remote machines, and thus the communication might add considerable time to the execution. Figures 4.3a and 4.3b show the communication cost for the above example. We can see that for the $\mathcal{F}_{\text{TPSI-diff}}$ the cost is considerable and might make the protocol useless for moderate size scenario like the one we consider. On the other hand, $\mathcal{F}_{\text{TPSI-int}}$ requires much less interaction. This is in line with the claims of [BMR20], where the authors provided the AHE solution as a computionally more efficent alternative, but at the expense of more communication. Also by looking at Figure 4.3a it seems like we have communication complexity linear in T, which was what [BMR20] achieved, and it seems to be true even in our modified protocol.



(a) Communication cost for $\mathcal{F}_{\textsf{TPSI-int}}$.

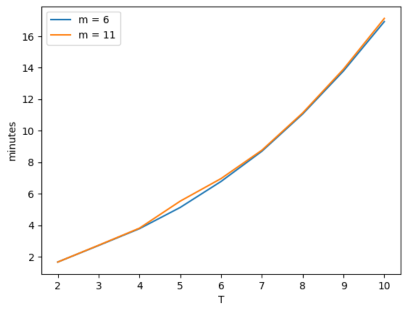


(b) Communication cost for $\mathcal{F}_{\textsf{TPSI-diff}}$.

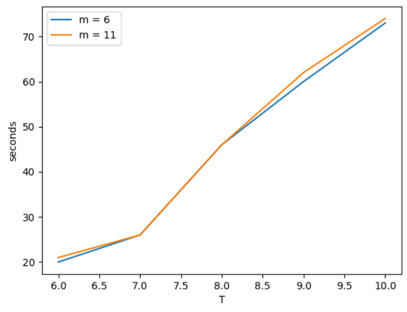
Figure 4.3: Communication cost for various values of T.

4.4 Varying set size

In Figures 4.4a and 4.4b we show our basic test case together with a run with extended set. We simply added 5 elements to the intersection. As seen, an increased set does not affect the running time. This is not surprising as the number of sampled values we work with throughout the protocols are only depending on T. Greater set size only gives some overhead in the start of the protocols, when the elements are encoded into polynomials, and the sampled values received.



(a) Running time for $\mathcal{F}_{\textsf{TPSI-int}}$ for various values of T and m.

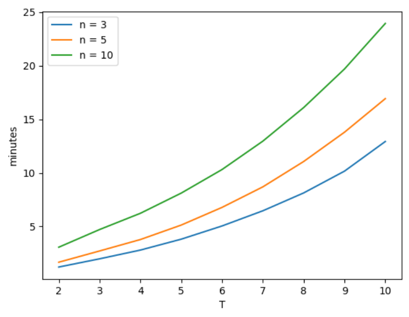


(b) Running time for $\mathcal{F}_{\textsf{TPSI-diff}}$ for various values of T and m.

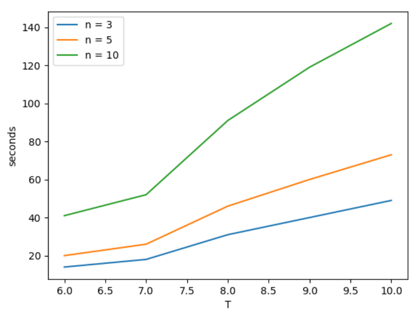
Figure 4.4: Running times for various values of T and m.

4.5 Varying number of parties

The functionalities implemented in this thesis are highly parallel, and if they are to run on multiple machines, we can expect concurrency speed up. These tests were all run on a dual-core computer. Figures 4.5a and 4.5b show execution times for three values of n, namely $n = 3, 5, 10$. In these tests we kept the sizes of the intersection fixed, but added and removed parties. As we can see, we can indeed expect faster execution if we were to use multiple computers, so that each party had their own dedicated processor to run on.



(a) Running time for $\mathcal{F}_{\text{TPSI-int}}$ for various values of T and n.



(b) Running time for $\mathcal{F}_{\text{TPSI-diff}}$ for various values of T and n.

Figure 4.5: Running times for various values of T and n.

Chapter 5

Conclusion

We have showed how to implement two of the three protocols described by Badrinarayanan, Miao and Rindal in [BMR20], $\mathcal{F}_{\text{TPSI-diff}}$ using AHE and $\mathcal{F}_{\text{TPSI-int}}$. We have also showed how to implement an alternate version of $\mathcal{F}_{\text{TPSI-int}}$ for the scenario when collective bootstrapping is available instead of ordinary bootstrapping. We have provided an implementation of this alternate version of $\mathcal{F}_{\text{TPSI-int}}$ and $\mathcal{F}_{\text{TPSI-diff}}$ using AHE in the Go programming language. Our implementation was structured in a way that allows the use of any implemented cryptosystem that satisfies the homomorphic properties. We have run the protocols and our experimental results show that the execution time might be satisfactory for some scenarios.

We have described how to homomorphically find the minimal polynomial of a linearly recurrent sequence, which was needed to implement $\mathcal{F}_{\text{TPSI-diff}}$ but not provided by [BMR20].

Lastly, we have provided a library *Generic Matrix* in the Go programming language that enables basic matrix operations on matrices with elements of any type.

Chapter 6
Acknowledgements

I would like to thank my supervisors for their help during this work. Thank you Elena Pagnin for being a great support throughout the full process with your theoretical knowledge, which gained me a lot of understanding for the topic. Thank you Alexander Nilsson for discussing the code with me. Also thank you to Advenica which offered me equipment and a place to work, although the circumstances wanted otherwise.

Bibliography

[20] *Microsoft SEAL (release 3.6)*. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. Nov. 2020.

[Aca+18] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. "A Survey on Homomorphic Encryption Schemes: Theory and Implementation". In: *ACM Comput. Surv.* 51.4 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3214303. URL: https://doi.org/10.1145/3214303.

[App] The Netherlands Organisation for Applied Scientific Research. *Distributed Paillier Cryptosystem*. Version d28041fd576a29febdebf62c1d7f138224a22db4. URL: https://github.com/TNO/Distributed-Paillier-Cryptosystem.

[Arm+15] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, and Martin Strand. *A Guide to Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2015/1192. https://eprint.iacr.org/2015/1192. 2015.

[Ben94] Josh Benaloh. "Dense Probabilistic Encryption". In: *Selected Areas of Cryptography*. May 1994. URL: https://www.microsoft.com/en-us/research/publication/dense-probabilistic-encryption/.

[BGN05] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. "Evaluating 2-DNF Formulas on Ciphertexts". In: *Theory of Cryptography*. Ed. by Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 325–341. ISBN: 978-3-540-30576-7.

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. *Fully Homomorphic Encryption without Bootstrapping*. Cryptology ePrint Archive, Report 2011/277. https://eprint.iacr.org/2011/277. 2011.

[BMR20] Saikrishna Badrinarayanan, Peihan Miao, and Peter Rindal. *Multi-Party Threshold Private Set Intersection with Sublinear Communication.* Cryptology ePrint Archive, Report 2020/600. `https://eprint.iacr.org/2020/600`. 2020.

[Bös+14] C. Bösch, A. Peter, P. Hartel, and W. Jonker. "SOFIR: Securely outsourced Forensic image recognition". In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 2014, pp. 2694–2698. DOI: `10.1109/ICASSP.2014.6854089`.

[Bra12] Zvika Brakerski. *Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP.* Cryptology ePrint Archive, Report 2012/078. `https://eprint.iacr.org/2012/078`. 2012.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper B. Nielsen. "Multiparty Computation from Threshold Homomorphic Encryption". In: *Advances in Cryptology — EUROCRYPT 2001.* Ed. by Birgit Pfitzmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 280–300. ISBN: 978-3-540-44987-4.

[Che+17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: *Advances in Cryptology – ASIACRYPT 2017.* Ed. by Tsuyoshi Takagi and Thomas Peyrin. Cham: Springer International Publishing, 2017, pp. 409–437. ISBN: 978-3-319-70694-8.

[Chi+16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds". In: *Advances in Cryptology – ASIACRYPT 2016.* Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 3–33. ISBN: 978-3-662-53887-6.

[Chr] ChristianMct. *thresh-paillier-wo-td.* Version `86be64fcbe584b16a3a09f38fa620ca0f2bfca68`. URL: `https://github.com/ChristianMct/thresh-paillier-wo-td`.

[Crya] CryptoExperts. *FV-NFLlib.* Version `4474b00196e9eec15b312e834bad9bc560b32059`. URL: `https://github.com/CryptoExperts/FV-NFLlib`.

[Cryb] Cryptovote. *Damgard-Jurik.* Version `5471ec2eb098381dd4dc37fac6b041a010290960`. URL: `https://github.com/cryptovoting/damgard-jurik`.

[Dat] Laboratory for Data Security. *Lattigo*. Version `bb095f1d82cd5c532171b138cd0ad83ecd785103`. URL: `https://github.com/ldsec/lattigo`.

[Dij+09] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. *Fully Homomorphic Encryption over the Integers*. Cryptology ePrint Archive, Report 2009/616. `https://eprint.iacr.org/2009/616`. 2009.

[DJ01] Ivan Damgård and Mads Jurik. "A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System". In: *Public Key Cryptography*. Ed. by Kwangjo Kim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 119–136. ISBN: 978-3-540-44586-9.

[DJN03] Ivan Damgård, Mads Jurik, and Jesper Nielsen. "A generalization of Paillier's public-key system with applications to electronic voting". In: *International Journal of Information Security* 9 (Apr. 2003), pp. 371–385. DOI: `10.1007/s10207-010-0119-9`.

[DK07] Yvo Desmedt and Kaoru Kurosawa. "A Generalization and a Variant of Two Threshold Cryptosystems Based on Factoring". In: *Information Security*. Ed. by Juan A. Garay, Arjen K. Lenstra, Masahiro Mambo, and René Peralta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 351–361. ISBN: 978-3-540-75496-1.

[DM14] Léo Ducas and Daniele Micciancio. *FHEW: Bootstrapping Homomorphic Encryption in less than a second*. Cryptology ePrint Archive, Report 2014/816. `https://eprint.iacr.org/2014/816`. 2014.

[Duc] Léo Ducas. *FHEW*. Version `f53cd4b9887218aded14f96abb5f54daf284a79f`. URL: `https://github.com/lducas/FHEW`.

[Dyn] Dyne.org. *Lua Multi Party*. Version `43b089a9f8b2b688855532e0970266fc564efde2`. URL: `https://github.com/dyne/lua-multiparty`.

[Elg85] Taher Elgamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *Advances in Cryptology*. Ed. by George Robert Blakley and David Chaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 10–18. ISBN: 978-3-540-39568-3.

[FPS01] Pierre-Alain Fouque, Guillaume Poupard, and Jacques Stern. "Sharing Decryption in the Context of Voting or Lotteries". In: *Financial Cryptography.* Ed. by Yair Frankel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 90–104. ISBN: 978-3-540-45472-4.

[FV12] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption.* Cryptology ePrint Archive, Report 2012/144. `https://eprint.iacr.org/2012/144`. 2012.

[Gen09] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing.* STOC '09. Bethesda, MD, USA: Association for Computing Machinery, 2009, pp. 169–178. ISBN: 9781605585062. DOI: `10.1145/1536414.1536440`. URL: `https://doi.org/10.1145/1536414.1536440`.

[GJR10] Elena Grigorescu, Kyomin Jung, and Ronitt Rubinfeld. "A local decision test for sparse polynomials". In: *Information Processing Letters* 110.20 (2010), pp. 898–901. ISSN: 0020-0190. DOI: `https://doi.org/10.1016/j.ipl.2010.07.012`. URL: `http://www.sciencedirect.com/science/article/pii/S0020019010002243`.

[GM84] Shafi Goldwasser and Silvio Micali. "Probabilistic encryption". In: *Journal of Computer and System Sciences* 28.2 (1984), pp. 270–299. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(84)90070-9`. URL: `http://www.sciencedirect.com/science/article/pii/0022000084900709`.

[Go] Go. *Frequently Asked Questions (FAQ).* URL: `https://golang.org/doc/faq` (visited on 01/11/2021).

[Gon] Gonum. *Gonum matrix GoDoc.* URL: `https://pkg.go.dev/gonum.org/v1/gonum/mat` (visited on 01/11/2021).

[hom] homenc. *HElib.* Version `060d36d4f0b125002ddb51ff9594c930382b3232`. URL: `https://github.com/homenc/HElib`.

[Ion+17] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. *Private Intersection-Sum Protocol with Applications to Attributing Aggregate Ad Conversions.* Cryptology ePrint Archive, Report 2017/738. `https://eprint.iacr.org/2017/738`. 2017.

[Jepa] Anton Jeppsson. *Generic Matrix.* Version 12bed0b60f47db5c8099e8bd65b5cdbb3cd7caf9. URL: `https://github.com/ontanj/generic-matrix`.

[Jepb] Anton Jeppsson. *Threshold Private Set Intersection.* Version 1af08d8ccab5feb59fabf66f0486c634e6085f59. URL: `https://github.com/ontanj/tpsi`.

[jia] jianyu-m. *Damgård–Jurik Cryptosystem.* Version `0d0ab2aacfebd4791ca61af5c1633c64dd181dc4`. URL: `https://github.com/jianyu-m/damgard_jurik`.

[KG] Murat Kantarcioglu and James Garrity. *Paillier Threshold Encryption Toolbox.* URL: `http://www.cs.utdallas.edu/dspl/cgi-bin/pailliertoolbox/index.php` (visited on 01/11/2021).

[Kil+07] Eike Kiltz, Payman Mohassel, Enav Weinreb, and Matthew Franklin. "Secure Linear Algebra Using Linearly Recurrent Sequences". In: *Theory of Cryptography.* Ed. by Salil P. Vadhan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 291–310. ISBN: 978-3-540-70936-7.

[KY02] Jonathan Katz and Moti Yung. "Threshold Cryptosystems Based on Factoring". In: *Advances in Cryptology — ASIACRYPT 2002.* Ed. by Yuliang Zheng. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 192–205. ISBN: 978-3-540-36178-7.

[Lab] NIC Chile Research Labs. *Paillier Threshold Encryption Scheme Implementation.* Version `57365a998b291779975673dd0676d2f8a8717740`. URL: `https://github.com/niclabs/tcpaillier`.

[Mal] Nasim Maleki. *Cryptography.* Version `f890d6933a8fe7b7742266ef51e36f5513f93df4`. URL: `https://github.com/nasimmaleki/Cryptography`.

[Mar14] Moxie Marlinspike. *The Difficulty Of Private Contact Discovery.* 2014. URL: `https://signal.org/blog/contact-discovery/` (visited on 01/11/2021).

[Mou+20] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. *Multiparty Homomorphic Encryption from Ring-Learning-With-Errors.* Cryptology ePrint Archive, Report 2020/304. `https://eprint.iacr.org/2020/304`. 2020.

[Nag+10] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. "BotGrep: Finding P2P Bots with Structured Graph Analysis". In: *Proceedings of the 19th USENIX Conference on Security*. USENIX Security'10. Washington, DC: USENIX Association, 2010, p. 7. ISBN: 8887666655554.

[NLV11] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. "Can Homomorphic Encryption Be Practical?" In: *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*. CCSW '11. Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 113–124. ISBN: 9781450310048. DOI: 10.1145/2046660.2046682. URL: https://doi.org/10.1145/2046660.2046682.

[NS11] Takashi Nishide and Kouichi Sakurai. "Distributed Paillier Cryptosystem without Trusted Dealer". In: *Information Security Applications*. Ed. by Yongwha Chung and Moti Yung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 44–60. ISBN: 978-3-642-17955-6.

[NuC] NuCypher. *A GPU implementation of fully homomorphic encryption on torus*. Version 638e12e931fe39c0a5cd5ee5d271d40084e29b31. URL: https://github.com/nucypher/nufhe.

[Pai99] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238. ISBN: 978-3-540-48910-8.

[PAL] PALISADE. *PALISADE Release*. Version 7eec669e1faa4aca0434021b7bdd5fd43695c075. URL: https://gitlab.com/palisade/palisade-release.

[Pet] Tom Petersen. *Threshold cryptography library*. Version 4505fac4873acb42414f6eefbb720b2ecc562dd3. URL: https://github.com/tompetersen/threshold-crypto.

[Poo] Anna Poon. *BGN*. Version 79878ce25e815cc8355679f7307e5346a3a89f4e. URL: https://github.com/anna138/BGN.

[RD78] Ronald L. Rivest and M. Dertouzos. "ON DATA BANKS AND PRIVACY HOMOMORPHISMS". In: 1978.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342.

[Seo] Cryptography LAB in Seoul National University. *HEAAN*. Version 131d275b2ed071a263fce4d367d418bb23b9bf53. URL: https://github.com/snucrypto/HEAAN.

[Sera] Sacha Servan-Schreiber. *BGN*. Version 0206925573bd96c94be04da000baa54f2dd8416c. URL: https://github.com/sachaservan/bgn.

[Serb] Sacha Servan-Schreiber. *paillier*. Version 30237183ba29b1b833f964976a1591af397a529b. URL: https://github.com/sachaservan/paillier.

[Sho00] Victor Shoup. “Practical Threshold Signatures”. In: *Advances in Cryptology — EUROCRYPT 2000*. Ed. by Bart Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 207–220. ISBN: 978-3-540-45539-4.

[tfh] tfhe. *THFE*. Version a085efe91314f994285fcb06ab8bdae3d55e4505. URL: https://github.com/tfhe/tfhe.

[Zam] Zama. *Concrete Operates oN Ciphertexts Rapidly by Extending TfhE*. Version fff309ec8df99ed5a02e754fbc382da89691c918. URL: https://github.com/zama-ai/concrete.

[ziya] ziyao002. *Threshold-Paillier-with-ZKP*. Version dd66e22b2c88c82412fe957d83cc970f83a675cd. URL: https://github.com/ziyao002/Threshold-Paillier-with-ZKP.

[ziyb] ziyao002. *Threshold-Paillier-without-Trust-Dealer*. Version 16d12e2363da8a53fa0158744f1477ed7e8c01e3. URL: https://github.com/ziyao002/Threshold-Paillier-without-Trust-Dealer.

[ziyc] ziyao002. *Threshold-Paillier-without-Trust-Dealer-with-TCP-IP*. Version ebc2d03ac43fe19795fa950a82729c43719d1d00. URL: https://github.com/ziyao002/Threshold-Paillier-without-Trust-Dealer-with-TCP-IP.