NTNU

Norwegian University of
Science and Technology

Realizing Distributed RSA using Secure Multiparty Computations

Atle Mauland

Master of Science in Communication Technology
Submission date: July 2009
Supervisor: Stig Frode Mjølsnes, ITEM
Co-supervisor: Tord I. Reistad, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

A multiparty computation is where three or more parties compute a commonly agreed function with secret input and possibly public output by carrying out some multiparty crypto protocol. Any scenario that involves some kind of both information hiding and sharing between several parties can normally be converted into a multiparty computation problem. A multiparty computation is done without the assistance of a trusted third party. The trusted party responsibility is shared among the participants instead.

Quite a lot of theoretical literature on the topic of multiparty computations and the technique of sharing a secret among several participants exists, but few of the schemes are practical and have been realized.

This master thesis work will focus on understanding the basic theory of multiparty computations, select some interesting multiparty computation problem, and then program and make experiments with a multiparty crypto protocol solution using VIFF (see http://viff.dk).

Assignment given: 15. January 2009
Supervisor: Stig Frode Mjølsnes, ITEM

"The good thing about secrets is that they can be shared."

- Atle Mauland

Abstract

This thesis describes the basic theory of multiparty computation (MPC) in addition to a fully functional distributed Rivest-Shamir-Adleman (RSA) protocol for three players implemented in Virtual Ideal Functionality Framework (VIFF) using secure MPC (SMPC). MPC can be used to solve problems where n players, each with a private input x_i, wants to compute a function with public output $f(x_1, x_2, ..., x_n) = y$, such that the private inputs remains secret for each player, but the output y is public. A cornerstone in MPC is the concept of secret sharing. In secret sharing, a dealer has a secret and gives each participating player a share of the secret in such a way that a certain number of the players are needed in order to reconstruct the secret. The number of players needed to reconstruct the secret is referred to as the threshold of the scheme. VIFF is a high level programming framework that allows programmers to create applications using SMPC for any number of players in an easy, efficient and secure manner. The distributed RSA solution implemented in VIFF includes distributed key generation, decryption and signature, which are the main functions needed for the distributed RSA scheme, and is based on the distributed RSA algorithm proposed by Dan Boneh and Matthew Franklin in 1997.

Four improvements compared to Boneh and Franklin's algorithm are described, two related to the run-time and two related to the security of the algorithm. The run-time improvements are regarding the distributed trial division step and the local trial division on the revealed N, both implemented. The security improvements are related to the way a random number is used to secure a revealed number. The first security improvement is related to the distributed trial division, whereas the second security improvement is regarding the alternative step in the biprimality test. The first security improvement, which is also the more important of the two, has been implemented in this thesis.

At last some benchmarks regarding the key generation, decryption and signature process are presented, which indicates that the current implementation is best suited for scenarios where the distributed RSA keys can be generated in

advance, whereas the decryption and signature process is fast enough for any type of scenario. The key generation process can become much faster with a few adjustments described at the end of the thesis.

Acknowledgements

This master's thesis is the result of a twenty weeks long project conducted during the 10th semester of my masters program at the Department of Telematics at the Norwegian University of Science and Technology, NTNU.

I would like to thank my supervisor, PhD student Tord Ingolf Reistad, who has provided me with valuable inputs, good feedback and helpful assistance whenever needed through meetings, discussions and via e-mail.

Also, I would like to thank Håvard Vegge for the collaboration in the writing process for some parts of the background material in addition to several discussion to clarify the concepts of secret sharing, MPC and VIFF.

In addition, I would also like to thank the following people:

- Professor Stig Frode Mjølsnes for valuable inputs regarding the problem description and good feedback regarding the report writing.
- Marting Geisler and the rest of the VIFF Developer Team for very fast and informative answers on the VIFF mailing list.
- Pål Sturla Sæther, engineer at the Department of Telematics at NTNU, for supplying me with equipment needed to benchmark the application.
- Steffen Tøsse Brun for proofreading this thesis.
- Bengt Jonny Mauland for proofreading parts of this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

2PC	Two-party Computation
CA	Certificate Authority
CACE	Computer Aided Cryptography Engineering
ECC	Error-Correcting Code
GB	Gigabyte
gcd	Greatest Common Divisor
GF	Galois Field
GHz	Gigahertz
GMPY	General Multiprecision PYthon
IP	Internet Protocol
LAN	Local Area Network
Mbit/s	Megabit per second
MHz	Megahertz
MIT	Massachusetts Institute of Technology
MITM	Man-in-the-middle
MPC	Multiparty Computation
NTNU	Norwegian University of Science and Technology
PR	Private key
PRSS	Pseudo-random Secret Sharing
PU	Public key
RSA	Rivest-Shamir-Adleman
SCET	Secure Computing Economy and Trust
SIMAP	Secure Information Management and Processing
SMPC	Secure Multiparty Computation
SP3	Service Pack 3
SSL	Secure Sockets Layer
SSSS	Shamir's Secret Sharing Scheme
TLS	Transport Layer Security
TTP	Trusted Third Pary
VIFF	Virtual Ideal Functionality Framework
VM	Virtual Machine
XOR	Exclusive or

Chapter 1

Introduction

1.1 Motivation

The concepts of secret sharing and multiparty computation were introduced about 3 decades ago. Since then, quite a lot of theoretical contributions have been published on the subjects, but very few of the schemes have been realized and made practical. Today, well known scenarios like certificates issued from certificate authorities (CA) on the Internet, stock trading on all of the world's stock exchanges, auctions and voting schemes of all kinds includes a trusted third party (TTP). Such a TTP is a neutral entity which operates as the communication link between several non-trusting players. The TTP accepts private inputs from players and gives outputs to the players. The need for a TTP can be avoided by using SMPC, in which the responsibility of the TTP is shared in a secure manner among some or all of the participating players instead.

The general problem with SMPC has been that very few schemes have been realized and even fewer have proven practical. In recent years, more frameworks for easy implementation of SMPC have appeared, making it easier to realize practical protocols in larger scale. The most prominent frameworks as of today are *FairplayMP* ([BDNP08]), which is developed in Israel, and *VIFF* ([rGiN09]), which is developed in Denmark. VIFF, which is the alternative chosen for this thesis, has its origin in a former framework called Secure Information Management and Processing (SIMAP). SIMAP has already been used to perform the world's first large scale SMPC in 2008 when the auction between the Danish sugar beets processor, Danisco, and the Danish sugar beets farmers took place to securely find the marked clearing price for sugar beets ([BCD+08]).

As such frameworks are getting faster, more reliable and more usable, the

number of realized practical protocols is likely to increase in the near future. VIFF already has all the functionality that SIMAP had and much more, so if SIMAP could do large scale SMPC, then surely VIFF can as well.

The RSA algorithm is the most widely used general-purpose algorithm for public-key encryption today. It can be used for both encryption and signature of messages, and is used tremendously amount of times on the Internet every day, mostly for exchanging private keys to use in private-key encryption, e.g. in secure sockets layer (SSL) or transport layer security (TLS), in addition to encryption and signing of digital certificates.

1.2 Problem Statement

The focus in this thesis is to get a basic understanding of the theory of MPC and use that knowledge to experiment and realize a cryptographic protocol using the MPC framework VIFF. The chosen protocol is distributed RSA (also referred to as shared RSA), with that bringing two highly relevant and powerful security concepts together in one solution. Such a solution can for example be used to digitally sign certificates by n servers instead of a single server (TTP), which increases the security by increasing the probability that the signed certificate is from a trusted entity.

Distributed RSA protocols have been implemented before, but never in VIFF, which makes this a challenge. The main part of this thesis is to realize a fully functional distributed RSA protocol for three players in VIFF, which will include key generation and possibility for decryption and signature, all conducted in a secure and distributed manner. In addition, a supplementary part is to benchmark and analyze the security of the solution in order to improve both the run-time and the security of the application.

1.3 Methodology

The whole implementation is written in VIFF, which allows a programmer to write high-level code on top of a well-defined structure of already implemented modules. The code written for this thesis is written in the application layer shown in Figure 1.1. Writing VIFF code in the application layer means using overloaded mathematical operators that operates on secret shared values to do MPC, which again uses a secret sharing scheme to secret share values among the players. The secret shared values need to be transferred between the players, which is done by the network communication layer.

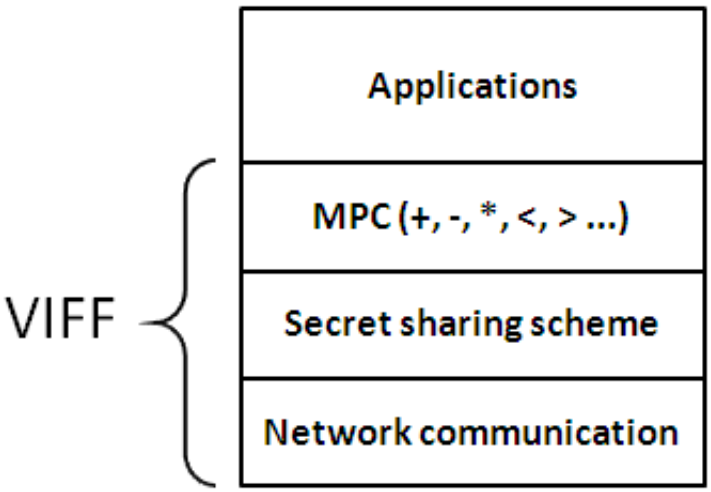

Figure 1.1: VIFF structure: Network Communication, secret sharing scheme and MPC. The programmer writes applications on top of the VIFF structure, using the VIFF modules, and does not need to think about the underlying layers.

The methodology used in this thesis has been to first study the concepts of secret sharing, MPC and RSA in order to acquire a fundamental understanding of these topics. At the same time, experimentation with simple secret sharing and MPC examples have been conducted in VIFF by implementing simple protocols in order to understand the structure of the framework.

The implemented distributed RSA protocol is the result of an iterative process by implementing one step of the algorithm at the time. Once a functional algorithm was in place, work was conducted to improve both the run-time and the security of the algorithm, also in an iterative manner. Time spent implementing and improving the distributed RSA protocol has been the most time-consuming part of this thesis.

1.4 Related Work

Even though a distributed RSA protocol has not been implemented in VIFF before, both the distributed RSA protocol and VIFF are fields in continuous development. The current milestone for distributed RSA is, as mentioned, documented in [BF97], with follow-up work documented in [MWB99]. Another distributed RSA protocol is presented in [ACS02], which is comparable in speed to Boneh and Franklin's protocol for normal distributed RSA, but with the possibility of using safe primes[1] in the public modulus $N = pq$ in a very efficient manner. As for the calculation of the exponents used in

[1] A safe prime is a prime number of the form $2p+1$, where p is also a prime ([Con09f]).

distributed RSA, both [ACS02] and [CGH00] have proposed alternative and efficient methods compared to [BF97].

VIFF is developed by the VIFF Development Team, who works on adding functionality to the framework, speeding it up and making it more secure.

1.5 Report Outline

Section 3.5 and 3.6 (along with Appendix C) has been written in collaboration with Håvard Vegge. The remainder of this thesis is outlined as follows:

Chapter 2 - Secret Sharing
This chapter presents the basic concepts of secret sharing, with the additive scheme and Shamir's scheme being the two most important schemes.

Chapter 3 - Multiparty Computation
This chapter gives an introduction to multiparty computations in general, along with detailed description of how to perform addition and multiplication in MPC.

Chapter 4 - Virtual Ideal Functionality Framework
This chapter gives an introduction to VIFF, the MPC framework used for realizing SMPC in this thesis.

Chapter 5 - RSA
This chapter describes public-key encryption and the RSA scheme, both the standard RSA scheme and the distributed RSA scheme.

Chapter 6 - Distributed RSA Implementation in VIFF
This chapter describes the distributed RSA implementation made in VIFF.

Chapter 7 - Security Analysis and Benchmarking
In this chapter a security analysis of the distributed RSA scheme is given along with benchmark results of the implementation.

Chapter 8 - Conclusions
This chapter summarizes and concludes the thesis.

Chapter 9 - Further Work
In this chapter, some suggestions for further work are given.

Chapter 2

Secret Sharing

Secret sharing was introduced by Adi Shamir and George Blakley independently in 1979. Their two schemes along with a numerous of other schemes can all be used in different cryptographic scenarios where it's desirable that a secret is not in hands of a single player. Secret sharing is very important in MPC and is therefore described in detail in this chapter.

The main idea of secret sharing is to have a dealer distribute a secret s among more than one player. Each player will only have a share of the secret, not the secret itself, see Figure 2.1. The secret can be reconstructed and used for a specified purpose by recombining a certain number of the total shares, depending on the scheme used. The security is based on the fact that each share is useless when used alone, but can be used for its purpose when combined.

This chapter will first give a brief explanation of finite fields, which are used in secret sharing schemes, before presenting secret sharing in general. Secret sharing is mainly divided into *secret splitting schemes* and *threshold schemes*, which will both be described below along with some numeric examples to clarify the concepts.

2.1 Finite Fields

Finite fields are of particular interest in many cryptographic protocols. Every finite field contains a finite number of elements, where the number of elements is referred to as the *order* of the finite field. The order of a finite field must be a power of a prime p, that is, the order is on the form p^n for a prime p and a positive integer n. A finite field of order p^n is generally written GF(p^n), where GF stands for Galois Field (after Évariste Galois, the first one to study finite fields).

The prime p is called the characteristic of the field and is defined to be the smallest number of times one must add the multiplicative identity element (1) to itself to get the additive identity element (0), that is $\overbrace{1 + \ldots + 1}^{p \text{ summands}} = 0$.

For secret sharing, a special case of the finite fields is used, more specifically finite fields with $n = 1$, having the form GF(p), which contains p elements. GF(p) is defined as the set $\mathbb{Z}_p$ of integers $\{0, 1, 2, ..., p-1\}$ and arithmetic operations are performed modulo p. Observe that all nonzero element in $\mathbb{Z}_p$ has a multiplicative inverse, because every element $1, 2, ..., p-1$ is relatively prime to p. Given that every operation is performed *mod* p, this ensures that all values are restricted to the interval $[0,\ p)$. In addition, every element in $\mathbb{Z}_p$ is relative prime to p, which ensures the uniform random distribution of values in the field, making finite fields an important necessity to obtain what is called perfect security (see Definition 3 below).

Normally when doing modular arithmetic modulo an integer n, the operations addition, subtraction and multiplication are defined for any element in the field and can all be performed without leaving the set. The observation above ensures that every nonzero element in the finite field has a multiplicative inverse, which means that finite fields also includes division by any nonzero number.

Example 1. (Finite fields calculations) First, let a finite field be defined to $\mathbb{Z}_p$ = GF(19). Next, two elements in the field are defined: $x = \mathbb{Z}_p(10)$ and $y = \mathbb{Z}_p(15)$. Now, operations can be performed on the elements in the field:

$$
\begin{aligned}
x + y &= 10 + 15 \ mod\ 19 = 6 \\
x - y &= 10 - 15 \ mod\ 19 = 14 \\
x \times y &= 10 \times 15 \ mod\ 19 = 17 \\
x\ /\ y &= x \times y^{-1} = 10 \times -5 \ mod\ 19 = 7
\end{aligned}
$$

2.2 Secret Splitting

This section defines secret splitting in addition to presenting two secret splitting schemes, one insecure and one secure.

2.2.1 Terminology

The secret splitting scheme is the simplest of the secret sharing schemes, and is defined as follows ([GF02]):

Definition 1. ***Secret splitting*** *is done by giving each player a share of the secret in such a way that it takes all the players to reconstruct the secret.*

General secret sharing is illustrated in Figure 2.1. A dealer has a secret s that is to be secret shared among a set of players. The dealer creates shares, s_i, out of s and distributes them (in a secure manner) to a number of players.

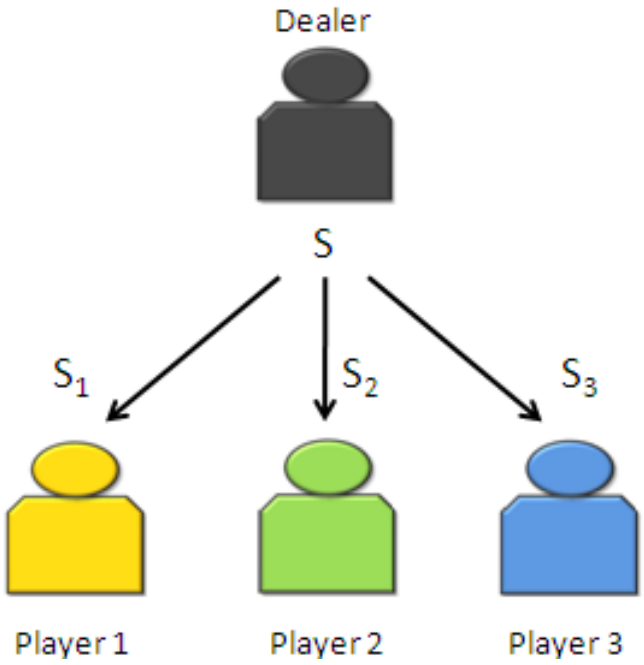

Figure 2.1: Illustration of distribution of shares for three players using secret sharing. The dealer creates the shares and distributes them (through secure channels) to the players.

Further, the *privacy* of a secret sharing protocol is defined as ([BOGW88]):

Definition 2. *A protocol is* ***t-private*** *if any set of at most* $\boldsymbol{t}$ *players cannot compute after the protocol more than they could jointly compute solely from their set of private inputs and outputs.*

Secret splitting is therefore $(n-1)$-private and can be conducted in many ways, all of which have in common that all the players needs to input their share to be able to reconstruct the secret.

2.2.2 Insecure Flawed Scheme

Suppose that the message *"secret"* is to be secret shared among 3 players. The naive approach would be to split the word in three and let each player have one third of the word, that is:

Player 1:"se''
Player 2:"cr''
Player 3:"et''

As can be seen above, all players know a part of the secret, but none of them know the whole secret. The flaw in this scheme is that even though none of the players know the whole secret, they all know something about the *actual* secret. They all know they have $1/n$ part of the actual secret. This makes it easier to reconstruct the secret for a shareholder. A person that holds no share, but knows the secret is 6 letters long (restricted to lower case for this example), would have to guess all the letters, which is equivalent to $26^6 =$ 309 million combinations. For a player with one share, only $26^4 = 457\ 000$ combinations exist for the remaining four letters. This also means that the more of the players that collaborates, the more they know about the *actual* secret.

The insecure scheme described above is illustrative for the secret sharing concept, but it should come as no surprise that this scheme is flawed. The next scheme on the other hand, *is* secure and can be used in many secret sharing scenarios.

2.2.3 Additive Scheme

In any secret sharing scheme it's a necessity that the players' individual shares yield no information about the actual secret, that is, a player holding a share should not give that player any advantage in reconstructing the secret alone.

Additive schemes involve that all players input their share to reconstruct the secret. The procedure of additive secret sharing for n players with a secret s in a finite field $\mathbb{Z}_p$ and a dealer D is conducted in the following way:

D picks $n-1$ random numbers $\{r_1, r_2, \ldots, r_{n-1}\}$ from $\mathbb{Z}_p$. D then computes

$$s_n = s - \sum_{i=1}^{n-1} r_i \ (mod\ p)$$

Then player 1, player 2,..., player $n-1$ receives the shares $s_i = r_i$ from the dealer (through secure channels). Player n receives the share s_n (as calculated above) from the dealer (through a secure channel). The reconstruction of the secret is done simply by adding the shares from all the players in the finite field $\mathbb{Z}_p$:

$$s = \sum_{i=1}^{n} s_i \ (mod\ p)$$

As shown above, no single shareholder knows anything about the actual secret, only a random integer. Suppose an adversary should get hold of $n-1$ shares, this would yield nothing about the actual secret, because the last random integer does only make sure that the secret s is in the range $[0,\ p)$, which is already given by the finite field used.

Another variant of the additive scheme is the exclusive or (XOR) scheme. The XOR scheme also needs all the players to participate to reconstruct the secret, therefore being $(n-1)$-private. The scheme is conducted in the following way: A dealer wants to secret share a message M among several players, where M is of length l-bit. The dealer gives the first $n-1$ players a random l-bit sequence each, and give the last player the l-bit sequence such that the XOR of all bit sequences equal the bit sequence of the message M. More formally it can be written:

$$\begin{aligned} m_i &= \{0,1\}^l \text{ for } i \in [1, n-1] \\ m_n &= M \oplus m_1 \oplus \ldots \oplus m_{n-1} \end{aligned}$$

As can be seen from the equations above, each player has only a random bit sequence, but when they are all XOR-ed together, they will yield the message M. The same privacy applies here as for the additive scheme, $(n-1)$-private, knowing all except one share of the secret yields nothing for an adversary. The adversary can XOR all obtained shares, but this only yields a random bit sequence, and knowing no bits of the actual secret makes every l-bit sequence a possible last share, that is, M can be any l-bit sequence, which is already given. This is a very neat property of the additive schemes, all shares have the same length as the actual secret, but gives no information to adversaries unless all shares are known. This leads to another definition for perfect security in a cryptographic system (from [Sch96]):

Definition 3. ***Perfect security*** *is a cryptographic system in which the ciphertext yields no possible information about the plaintext (except possibly its length).*

Both the additive and Shamir's scheme (explained later) offers perfect security by doing modular arithmetic *mod* p, which means that all values are in the interval $[0,\ p)$, but all values are equally likely, and therefore offers no information to any of the players. Definition 3 was theorized by Claude Shannon such that perfect security is possible only if the number of possible keys is at least as large as the number of possible messages. In other words, the key must be at least as large as the message itself, and no key can be reused, which makes the One-time pad (Vernam cipher) or equivalents the only cryptographic systems that achieves perfect security (for more information about the One-time pad scheme, see [WM05] and [Sta06]).

The XOR secret sharing scheme for two players is equivalent to the One-time pad cryptographic scheme, where the plaintext message (as a bit sequence of length l) is XOR-ed with a random bit sequence, also of length l, to get the ciphertext (encrypted message) of length l. The encrypted message can then be decrypted back to the plaintext by taking the XOR of the ciphertext and the random bit sequence. An example is included below to illustrate the additive secret sharing scheme using the XOR technique for three players.

Example 2. (XOR secret sharing) Dealer D wants to secret share a message M with length 4 bits. $M = 1001$ and is to be secret shared among $n = 3$ players, Alice, Bob and Carol, using the XOR secret sharing scheme.

The dealer is the only one that knows M, and uses M to generate shares for all the players. The dealer needs to generate $n - 1$ random bit-string shares and calculate the last share based on M and the random generated shares. The dealer generates 2 random bit-strings, 0111 and 1100, and gives them (in a secure manner) to Alice and Bob respectively. The dealer then calculates the last share as $1001 \oplus 0111 \oplus 1100 = 0010$ and gives that share (also in a secure manner) to Carol. The players now have the following (and nothing else):

$$\begin{array}{ll} Alice: & 0111 \\ Bob: & 1100 \\ Carol: & 0010 \end{array}$$

Now, if all the players are willing to participate in message decryption, they can find the message M, but one player resisting is enough for M to remain secret. It can easily be seen that this protocol works by using XOR with all the shares:

$$\begin{aligned} M &= m_{Alice} \oplus m_{Bob} \oplus m_{Carol} \\ M &= 0111 \oplus 1100 \oplus 0010 = 1001 \end{aligned}$$

2.3 Threshold Schemes

Eleven scientists are working on a secret project. They wish to lock up the documents in a cabinet so that the cabinet can be opened if and only if six or more of the scientists are present. What is the smallest number of locks needed? What is the smallest number of keys to the locks each scientist must carry? ([Liu68])

In this section, threshold secret sharing schemes are presented, starting with defining threshold schemes in general before presenting the two threshold schemes invented independently by Shamir and Blakley in 1979.

2.3.1 Introduction

The answer to the problem stated above using combinatorics is that the number of locks on the cabinet is $\binom{11}{5} = 462$ and the number of keys each scientist has to carry is $\binom{10}{5} = 252$. This clearly is impossible in reality, but a much more sophisticated solution exists, namely a threshold scheme ([BL90] and [Mor07]):

Definition 4. *Given a finite field $\mathbb{Z}_p$ of possible secret values, a **(t,n)-threshold secret sharing scheme** is a secret sharing scheme that can divide a secret $s \in \mathbb{Z}_p$ into shares $\{s_1, s_2, \ldots, s_n\} \in \mathbb{Z}_p$ so that $t \leq n$ and:*

1. *Given any set of t or more shares s_i, s can be reconstructed.*
2. *Any set of fewer than t shares gives no information about s.*

From Definition 2 it can be verified that threshold schemes are $(t-1)$-private, where t refers to the threshold used.

2.3.2 Shamir's Secret Sharing Scheme

Shamir's secret sharing scheme (SSSS) was introduced by Adi Shamir in 1979 ([Sha79]). Shamir's secret sharing is very suitable with threshold schemes and is a widely used secret sharing scheme. This scheme uses a secret random polynomial to hide the secret and Lagrange's interpolation to reconstruct the secret. The good suitability for threshold schemes is due to the simple fact that two points are needed to uniquely define a straight line (polynomial of degree 1), three points are needed to uniquely define a quadratic function (polynomial of degree 2) and so on, and therefore the degree of the polynomial defines the threshold t.

SSSS has, like the additive scheme, perfect security. All the shares are of the same length as the secret, $[0,\ p)$, and no information about the secret can be found without knowing at least t shares (all values in $[0,\ p)$ has equal probability as a last share).

Creating the Shares

A dealer wants to share a secret s using SSSS. The dealer then constructs a random polynomial $f(x)$ with degree $deg(f) = t - 1$:

$$f(x) = s + r_1 x + r_2 x^2 + \ldots + r_{t-1} x^{t-1} \ mod\ p$$

The following conditions for SSSS must hold:

- The threshold $t \leq n$
- The secret $s \in \mathbb{Z}_p$ for a prime number p.
- The number of players $n < p$
- The coefficients $\{r_1, r_2 \ldots, r_{t-1}\}$ used in the polynomial are randomly and independently chosen from the interval $[0,\ p)$.

Creating shares s_i for the n players is now really simple. The dealer first picks a random polynomial of degree $t-1$, where t is the threshold. Each player has a different id, $P_{id} = x_i$, which is constant for that player in the current scheme and is known by everyone. Normally, this id is given in increasing order for simplicity, such that $x_i = 1, 2, 3, ...$ for player $1, 2, 3, ...$ The dealer now give each player a share $f(x_i)$, such that player 1 gets the share $s_1 = f(1)$, player 2 gets the share $s_2 = f(2)$ and so on. The SSSS share creation for a secret s, n players and threshold $t = 2$ (a straight line) is illustrated in Figure 2.2.

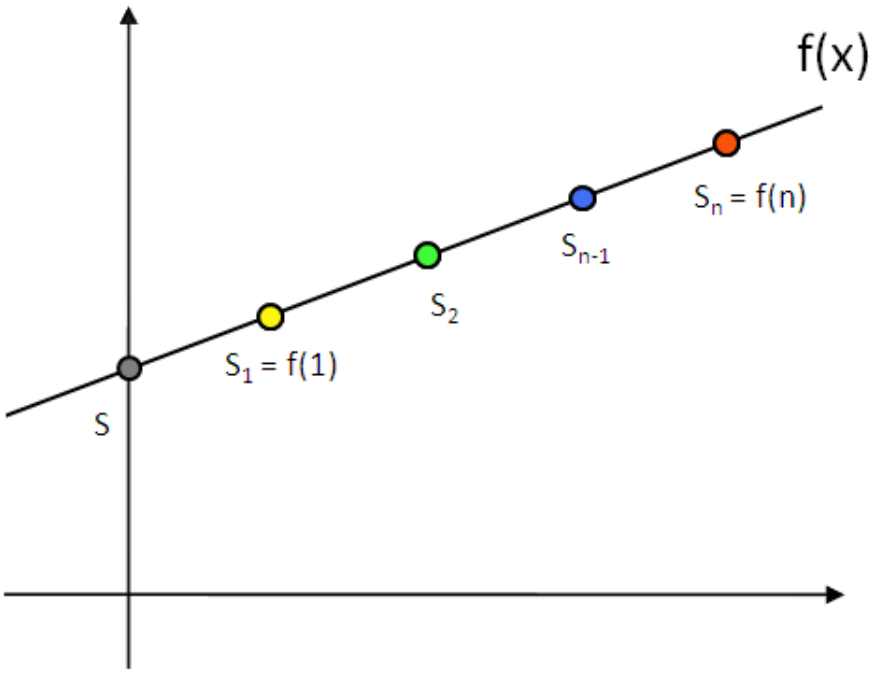

Figure 2.2: Creating the shares with SSSS in a $(2, n)$-threshold scheme: Each player receives from the dealer a share $s_i = f(x_i)$ from the secret polynomial $f(x)$, and it will take at least 2 players to reconstruct the secret s.

In Figure 2.2 it will take at least 2 out of the n players to reconstruct the secret, making it a $(2, n)$-threshold scheme. Another example where it will take at least 3 out of the n players, making it a $(3, n)$-threshold scheme, is

illustrated in Figure 2.3. This function is quadratic (of degree 2), and such a function can be uniquely defined by knowing at least 3 function values.

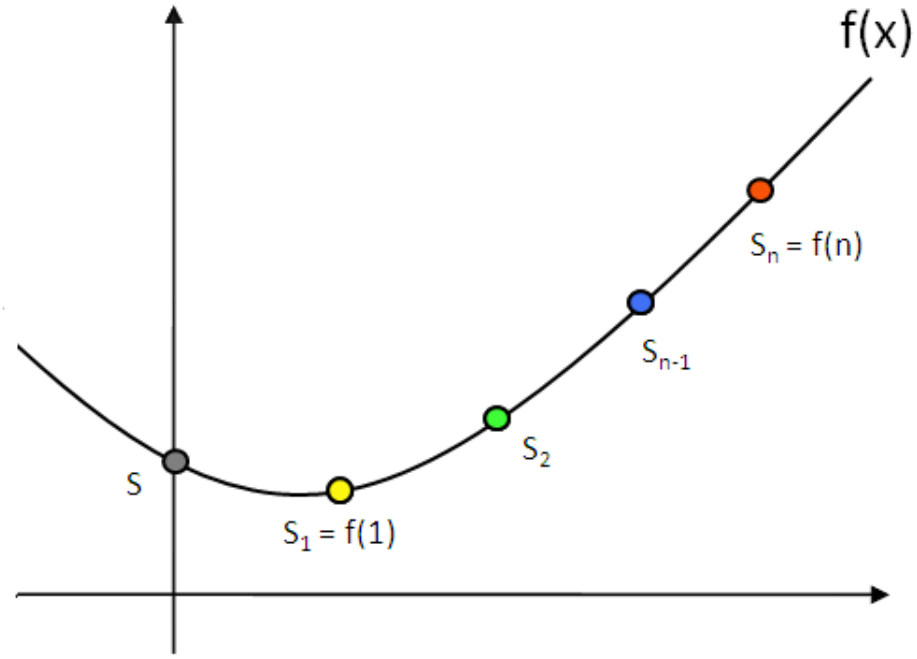

Figure 2.3: Creating the shares with SSSS in a $(3, n)$-threshold scheme: Each player receives from the dealer a share $s_i = f(x_i))$ from the secret polynomial $f(x)$, and it will take at least 3 players to reconstruct the secret s.

A very neat property of the threshold schemes is that the number of players, n, can be increased at any time without the need to create new shares for the former set of players. The dealer just needs to assign a $P_{id} = x_{n+1}$ to the new player and give the share $s_{n+1} = f(x_{n+1})$ to that player. This way the total number of players has increased from n to $n + 1$, but the number of shares needed to reconstruct the secret is still the threshold t since the secret polynomial has not changed.

Example 3. (SSSS share creation) A dealer wants to share a secret $s = 11$ using SSSS among five players with a threshold $t = 3$ using SSSS. The dealer chooses a prime number $p = 23$ and generates two random numbers $r_1 = 5$ and $r_2 = 2$, which yields the following polynomial:

$$f(x) = 11 + 5x + 2x^2 \ mod\ 23$$

The dealer then calculates one share for each player and gives it to that player in a secure manner:

$$
\begin{aligned}
s_1 &= f(1) = 11 + 5 \cdot 1 + 2 \cdot 1^2 \ mod\ 23 = 18 \\
s_2 &= f(2) = 11 + 5 \cdot 2 + 2 \cdot 2^2 \ mod\ 23 = 6 \\
s_3 &= f(3) = 21 \\
s_4 &= f(4) = 17 \\
s_5 &= f(5) = 17
\end{aligned}
$$

The secret s is now secret shared among the 5 players, each of them having their own distinct share of s. It would require at least $t = 3$ players in order to reconstruct the dealers secret, the method is explained next.

Reconstructing the Secret

Reconstructing the secret can be performed by any t number of players using their values x_i and $f(x_i)$. The reconstruction is done by using Lagrange's interpolation on t (or more) shares. Lagrange interpolation is defined as follows ([Kre99]):

$$f(x) = \sum_{i=1}^{n} L_i(x) f_i = \sum_{i=1}^{n} \frac{l_i(x)}{l_i(x_i)} f_i$$

where $l_i(x)$ and $l_i(x_i)$ is defined as follows:

$$
\begin{aligned}
l_i(x) &= \prod_{j=1}^{n, i \neq j} (x - x_j) \\
l_i(x_i) &= \prod_{j=1}^{n, i \neq j} (x_i - x_j)
\end{aligned}
$$

By substituting the expressions for $l_i(x), l_i(x_i)$ and setting $f_i = s_i, f(x)$ can be rewritten as:

$$f(x) = \sum_{i=1}^{n} s_i \cdot \prod_{j=1}^{n, i \neq j} \frac{x - x_j}{x_i - x_j}$$

Since the threshold is t, n can be substituted by t. The secret is found for $x = 0$, therefore the desired expression is found by finding $f(0)$, and all values are calculated $mod\ p$ in secret sharing schemes, which yields the following:

$$f(0) = \sum_{i=1}^{t} s_i \cdot \prod_{j=1}^{t, i \neq j} \frac{0 - x_j}{x_i - x_j} \ mod\ \mathrm{p}$$

By multiplying both the numerator and the denominator by minus one, which interchanges the indexes for i and j in the denominator, and putting

the denominator as a power of minus one instead, the final expression for calculation is:

$$s = f(0) = \sum_{i=1}^{t} s_i \cdot \prod_{j=1}^{t,i\neq j} x_j \cdot (x_j - x_i)^{-1} \ mod\ p \tag{2.1}$$

Notice that the indices i and j refers to the P_{id}'s of the players participating in the reconstruction, but are written as 1 to t here for simplicity.

Example 4. (Reconstructing the secret with SSSS) Continuing from Example 3 the shares of s have now been distributed to the players. Three of the players, player 1, player 3 and player 4 want to reconstruct the secret s. Each of these players has two values each, x_i and $f(x_i)$ that is used in the reconstruction:

$$\begin{array}{ll} \text{Player 1}: & (1, 18) \\ \text{Player 3}: & (3, 21) \\ \text{Player 4}: & (4, 17) \end{array}$$

These values can be used as input to Equation 2.1 to reconstruct the secret as shown below:

$$\begin{aligned}
s &= \sum_{i=1}^{t} s_i \cdot \prod_{j=1}^{t,i\neq j} x_j \cdot (x_j - x_i)^{-1} \ mod\ 23 \\
&= 18 \cdot \prod_{j=1}^{t,i\neq j} x_j \cdot (x_j - x_i)^{-1} \ + \ 21 \cdot \prod_{j=1}^{t,i\neq j} x_j \cdot (x_j - x_i)^{-1} \ + \\
&\quad 17 \cdot \prod_{j=1}^{t,i\neq j} x_j \cdot (x_j - x_i)^{-1} \ mod\ 23 \\
&= 18 \cdot (3 \cdot (3-1)^{-1}) \cdot (4 \cdot (4-1)^{-1}) \ + \\
&\quad 21 \cdot (1 \cdot (1-3)^{-1}) \cdot (4 \cdot (4-3)^{-1}) \ + \\
&\quad 17 \cdot (1 \cdot (1-4)^{-1}) \cdot (4 \cdot (3-4)^{-1}) \ mod\ 23 \\
&= 18 \cdot (3 \cdot 12) \cdot (4 \cdot 8) + 21 \cdot (1 \cdot 11) \cdot (4 \cdot 1) \ + \\
&\quad 17 \cdot (1 \cdot 15) \cdot (3 \cdot 22) \ mod\ 23 \\
&= 20736 + 924 + 16830 \ mod\ 23 \\
&= 38490 \ mod\ 23 \\
&= 11
\end{aligned}$$

Example 4 above shows that using three out of the five shares is enough to obtain the secret $s = 11$ using Lagrange's interpolation. The calculations require finding inverses *mod* p[1].

[1]The parenthesis $(x_j - x_i)^{-1}$ can be calculated by first calculating $(x_j - x_i)$. If this value is negative, add a multiple of p to obtain a value in the interval $[0,\ p)$. Calculating numbers with negative powers *mod* p requires finding inverses *mod* p, and substituting the

2.3.3 Blakley's Scheme

In Blakley's scheme the secret is a point in a t-dimensional space. Each of the n shares constructed and distributed to the players are nonparallel planes in this t-dimensional space which contains the secret point. The secret is reconstructed by finding the intersection of (at least) t shares (planes), as shown in Figure 2.4 (taken from [Con09g]).

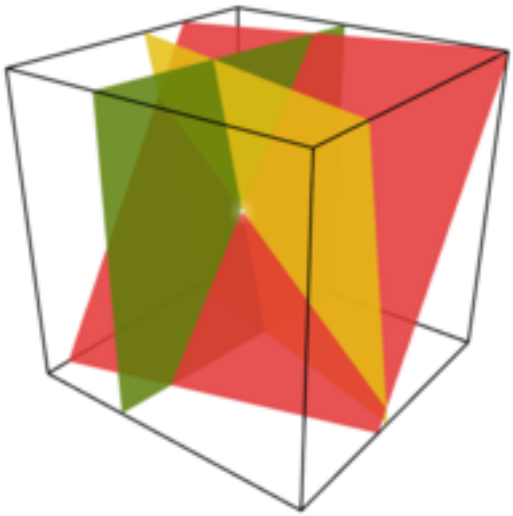

Figure 2.4: Blakley's scheme in three dimensions: Each share is a plane, and the secret is the point at which three shares intersect.

Each share is a plane and the secret itself is just a point in the t-dimensional space. This means that the shares are t times larger than the secret. In both the additive schemes and in SSSS the shares are as large as the secret itself, which is one of the properties of perfect security.

Blakley's scheme also lacks another property of perfect security. An adversary with a share of the secret knows the secret is a point on its plane. Knowing more shares makes an adversary know more about each of the dimensions, therefore also more about the actual secret. The scheme can however be modified to achieve perfect security (see [Sim92] and [Bri90]).

number with the inverse instead. To calculate $a^{-b}\ mod\ p$, find an inverse x of $a\ mod\ p$: $x = a^{-1}\ mod\ p$ and calculate $x^b\ mod\ p$ instead. Inverses can be found using the Extended Euclidean Algorithm (see [CLRS01], [Ros03] or [TW06]).

Chapter 3

Multiparty Computation

Three millionaires that do not trust each other want to rank their fortunes to find out once and for all who's the richest. Each of them wants to know the ranking of the fortunes, but no one wants to reveal their own fortune to any of the other two millionaires. How can this be achieved?

3.1 Introduction

The traditional model for trust between several players involves a TTP which all the players trust. This scheme is widely used in many fields of communications today, including CA on the Internet, stock trading in all the world's largest stock exchanges, auctions and important elections of all sorts. These all relies on that the trusted third party, who takes some inputs from players, does some computations and gives the public output, actually can be trusted. A big problem with this scheme is that the TTP is a single point of failure, which means that corrupting the TTP, corrupts the whole scheme.

The millionaire problem at the beginning of this chapter is a variant of the millionaire problem introduced by Andrew C. Yao in 1982 ([Yao82]). The solution to the problem can be accomplished by using multiparty computations. In fact all problems that uses a TTP can be avoided by using MPC instead.

MPC can be used in situations where a group of players want to calculate the value of a public function $f(x_1, x_2, \ldots, x_n) = y$, where x_i is the private input for player i, and y is the public output, with the restriction that no player should learn more information about the public calculation of f other than what is given from that player's input to the public function and the public information. Put another way, each player wants to keep their private value, x_i, secret, but all players want to know the result from the

public calculation of f. Note that the value y can also be a vector of values, $y = (y_1, y_2, ..., y_n)$, if all players should only learn its part of y.

General MPC is defined to be 3 players or more, while the 2 player case most often is referred to as two-party computation (2PC). The main idea is to let the trusted third party role be distributed among some or all of the players instead. MPC is closely related to secret sharing described in Chapter 2 in the way that every MPC uses a secret sharing scheme as it's cornerstone for calculating the outcome of the function f.

The difference between secret sharing and MPC can be described as follows: In secret sharing a dealer knows a secret and wants to distribute it among more than one player so that a certain number of players are needed to reconstruct the secret. In MPC on the other hand, the secret of each player is to be kept secret at all times, and only the public output from the function f is wanted. Therefore, in MPC each player that has an input to the function f is a dealer and secret share its private value to the other players, such that the players jointly can calculate the public outcome of f without ever knowing each of the private inputs.

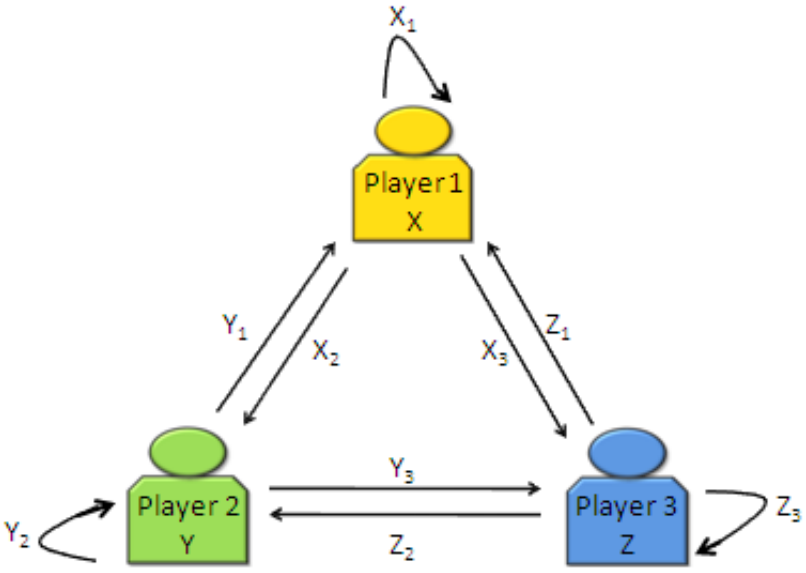

Figure 3.1: Multiple secret sharings: Three players have a private data, x, y and z for player 1, 2 and 3, respectively. They all secret share the values among all three players, resulting in each player i having three shares: x_i, y_i and z_i. This is a possible scenario for the input stage of an MPC (see below).

Figure 3.1 shows three players who each secret share a private value with the other two players. Each player i holds 3 shares, x_i, y_i and z_i, one for each of the private inputs. Once the secret sharing has been performed,

the players can compute their share of the function f by using the received shares from the other players. Notice however, that a round of distribution of the calculated values is needed to obtain the answer that is to be revealed, explained in the example below.

Example 5. (**MPC**) The three millionaires want to find their total fortune without revealing to the others their own fortune. Here, the three millionaires are player 1, 2 and 3, with a private input x, y and z respectively. They want to do an MPC to find $f(x, y, z) = x + y + z$, such that none of them get to know any of the private values except their own, but everyone gets the public output from f.

The solution to this problem is to use a simple MPC addition (explained in more detail in section 3.5). Each millionaire has its own private input x, y and z (the fortune), which is to be kept secret. The millionaires wants to calculate the function $f(x, y, z) = x + y + z$.

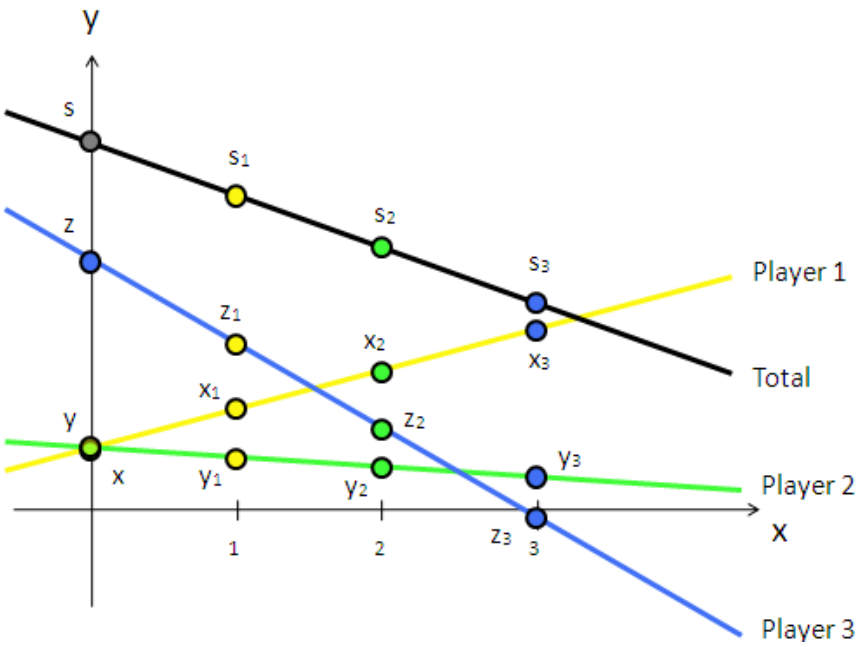

Figure 3.2: Three players calculating the outcome of the function $f(x, y, z) = x+y+z = s$ (where the total line intersects the y-axis). First each player secret share their private input (x, y, z) with the other players, such that each player i obtains x_i, y_i, z_i. Next each player calculates the addition of its shares, by calculating $s_i = x_i + y_i + z_i$, where s_i is player i's share of the total polynomial. Next, every player sends their share s_i to the other players, so that every player locally can interpolate the s_i's and find s, which is the output of f.

Each millionaire secret shares their secret using SSSS with the other two

millionaires as shown in Figure 3.1, resulting in Figure 3.2. Each of the players' lines represents the randomly chosen polynomial where the dot at the y-axis represents that player's fortune (x, y and z). After all players secret share their fortune, each player has a share of all lines (x_i, y_i, z_i). Each player can add its shares from all the players' lines and obtain a share of the total polynomial (dot on the total line), s_i. Now comes the distribution part mentioned above, they now each have a share on the total polynomial, but that is not sufficient for interpolating and reconstructing the public output alone. Each player therefore distributes their share s_i on the total polynomial to the other players, such that each player has all s_i's. The millionaires can now locally use Lagrange's interpolation on these s_i's to obtain the output of f, which is s, (the dot on the total line at the y-axis intersection) without revealing any information about the fortunes of each millionaire.

3.2 Stages in MPC

Generally, MPC consist of three stages: *input stage*, *computation stage* and *final stage*. An important point here is that the group of players that is active in each stage is not defined to be some static subgroup or follow some kind of pattern. This group of players will vary from each computation to be done, and the same group of players need not participate in all stages, all depending on the function f.

3.2.1 Input Stage

The input stage is where the players give input to the function $f(x_1, x_2, \ldots, x_n)$ to be calculated. As mentioned, these x_i's are the private values for player i, and these are to remain private for that player. Every player having an input x_i therefore acts as a dealer for its own private input, and secret share the input among all the players as shown in Figure 3.1. Several inputs per player are possible if the function f requires more than one value from certain players (this would result in more random polynomial picked by that player, and shares of the new polynomial(s) distributed to all the players). When the input stage ends, the players hold shares of the inputs to be used to calculate the shares of f, and can continue to the computation stage.

3.2.2 Computation Stage

The computation stage is where the actual computation on the shared values to obtain a share of the output of f. Several operations can be supported, although the most common ones are the basic addition and multiplication (described in detail later in this chapter). In this stage the calculations are done on shares, not on actual values, which is a requirement because the

inputs values are to be kept secret. When all the calculations are finished, the players have a share of the output from f, which are the s_i's in Figure 3.2, and the protocol moves on to the last stage.

3.2.3 Final Stage

The final stage is where the output values of the function f are revealed to all or some of the players. This is carried out by each player distributing their share of the total polynomial, s_i, to all the other players. That way every player can locally use Lagrange interpolation to find the public output s, which is the intersection of the total line with the y-axis in Figure 3.2. Revealed values are of course not reversible, that is, knowing a revealed value does not make it easier to find the inputs to the function f. This is easily seen from Figure 3.2, each player only knows one point of the other players' polynomial, which is not enough to reconstruct any of the private inputs besides their own. Numeric examples of the procedure of an MPC multiplication is described in Section 3.6.

3.3 Adversary Model

Adversaries in cryptographic protocols are malicious players who aim to prevent the protocol from running correctly. This can be done by corrupting honest players or some part of the system in such a way that the outcome is incorrect or absent. Adversaries can also try to collect helpful information by eavesdropping or act as a man-in-the-middle (MITM) to threaten the privacy of other players and the protocol.

Generally, the adversaries for MPC are divided into two groups, *passive adversaries* and *active adversaries*. *Mobile adversaries* are also mentioned in some literature, but will not be described any further in this thesis. The adversaries can often be seen as one entity, the adversary, to simplify of the concept.

3.3.1 Passive Adversary

The passive adversary is also known as the *honest-but-curious* or the *semi-honest* adversary. The passive adversary follows the protocol seemingly properly, but also does something more on the side. This could be to deliberately not delete some internal data, deviate from the randomness suppose to be used, eavesdropping on traffic or any other action that will not directly harm the execution of the protocol. A passive adversary can either input the correct value to the calculation or no value, but not an incorrect value.

The passive adversary is seen as violator of the privacy constraint. As an example, a passive adversary in an election has respect for the majority's opinion and would thus not alter the results in any way, but would want to know who voted for whom.

3.3.2 Active Adversary

The active adversary is also known as the *malicious* or the *Byzantine* adversary. This coalition of players can deviate arbitrary from the protocol with the intent to disrupt the computations and by doing so, produce incorrect results and/or violate the privacy of the other players.

The active adversary is seen as violator of both the privacy and the correctness constraint. As an example, an active adversary can omit to give an input to the protocol at any time during execution or give an incorrect input to prevent the correctness of the protocol.

3.3.3 Static vs. Adaptive Adversary

Both types of adversaries can be *static* or *adaptive*. A *static adversary* must choose the set of players to corrupt before the execution of the protocol, i.e., the set of corrupted players is fixed (but typically unknown) during the whole computation. An *adaptive adversary* on the other hand, can choose to corrupt players at arbitrary times during the execution of the protocol, depending on the information gathered so far.

Adversaries in MPC are a large field of study, and therefore much is left out of this thesis. For more details about the adversary model, mobile adversaries and general adversary structures for MPC, the reader is referred to [GMW87], [Gol97], [Gol99], [Gol00], [CDM00], [CrN08], [Can95] and [Hir01].

3.4 Secure Multiparty Computation

Secure multiparty computation means that the MPC protocol is declared secure by some measurement. Generally, SMPC is divided into two groups: *computationally secure MPC* and *information-theoretically secure MPC*, both are described below:

- **Computationally secure MPC** is based on some unproven cryptographic primitive (e.g. a mathematical problem, like factoring) which is assumed to be computationally infeasible to solve. The players share an authenticated, but otherwise insecure channel, which means an adversary have access to all the messages sent, but it's computationally infeasible for the adversary to modify the messages.

- **Information-theoretically secure MPC** is secure even if the adversary has unbounded computing power, because the players are assumed to communicate over (somewhat impractical, but fully possible) pairwise secure channels, i.e. the adversary gets no information at all about messages exchanged between honest players (except that something was sent).

Recall from Chapter 2 that both the additive scheme and Shamir's scheme offers perfect security, which also makes them information-theoretically secure. This property is one of the reasons that these schemes are a cornerstone in SMPC.

From [BOGW88] and [CCD88] the following are two important definitions about adversaries in MPC using threshold schemes:

- If the adversary is *passive* and *adaptive*, then every function can be securely computed with perfect security if and only if the adversary corrupts less than n/2 players, i.e. the adversary gets no additional information about the honest players.
- If the adversary is *active* and *adaptive*, then every function can be securely computed with perfect security if and only if the adversary corrupts less than n/3 players, i.e. the adversary gets no additional information about the honest players.

The limits for maximum adversaries vary depending on what security that is to be used, with perfect security having the strictest requirements. See tables in [Mor07] and [CrN08] for limits with other security measures.

The limits are optimal in the sense that they prevent the adversaries to:

1. Collaborate to reconstructing the secret.
2. Collaborate to prevent the honest players from reconstructing the secret.

Since the passive adversaries cannot input incorrect values to the calculation, point 2 for passive adversaries is only regarding whether they input the correct value or nothing. To see that the $n/2$ is the optimal limit, let $n = 6$ and $n = 7$ be the number of players and let t be the threshold.

Table 3.1 shows how many adversaries are needed to accomplish the undesired points 1 and 2 above for $n = 6$ and $n = 7$ players (the same pattern relate to all n). To prevent the adversaries of both point 1 and 2 above, the maximum number of adversaries needs to be less than both the *Reconstruct*

	n = 6		**n = 7**	
Threshold	**Reconstruct**	**Prevent**	**Reconstruct**	**Prevent**
t = 2	2	5	2	6
t = 3	3	4	3	5
t = 4	4	3	4	4
t = 5	5	2	5	3
t = 6	6	1	6	2
t = 7	-	-	7	1

Table 3.1: Explanation of the passive adversaries limit for $n = 6$ and $n = 7$ players. *Threshold* is the number of players needed to reconstruct the secret, *Reconstruct* is the number of adversaries needed to reconstruct the secret (equal to t) and *Prevent* is the number of adversaries needed to prevent the honest players from reconstructing the secret.

and the *Prevent* column at the same time. The row that maximizes this number is the row with $t = \lceil n/2 \rceil$ (two rows are applicable if n is even). The maximum number of adversaries if therefore less than 3 and less than 4 for $n = 6$ and $n = 7$ respectively, which agrees with the limit $< n/2$.

Note that the maximum number of adversaries increase for odd numbers of n only, and therefore, in general, odd number of players are most common in MPC schemes.

The limit for maximum active adversaries is lower than for passive adversaries, namely $n/3$, because the active adversaries can also input incorrect values. An intuitive way to see how to cope with incorrect inputs is to let 3 versions of each share determine the correctness of the share, e.g. if two players say the value is 5, and a third player says the value is 8, then the two players with the same value would be seen as honest players, while the third player would be seen as an adversary. In reality it's not exactly that simple, because the adversaries are not evenly distributed among the players, and therefore error-correcting codes (ECC) are used, see more in [BOGW88].

Property	**Shamir**	**Additive**
Distributed among	n	n
Needed to reconstruct	t	n
Maximum passive adversaries	$<$ n/2	0
Maximum active adversaries	$<$ n/3	0

Table 3.2: Summary of some important properties of secret sharing and MPC.

Table 3.2 summarizes some of the important points for secret sharing and SMPC. As can be seen, in each of the two secret sharing schemes, the secret is shared among all the n players. The reconstruction depends on whether

the secret sharing scheme is threshold or additive, which takes t and n players, respectively, to reconstruct the secret, where t is the threshold. In Shamir's scheme, the number of passive adversaries must be less than $n/2$, while the number of active adversaries must be less than $n/3$ in order for the protocol to be computed securely with perfect security. In the additive scheme, there is no room for any adversaries, because the lack of one or more input values or faulty input values will produce an erroneous output.

3.5 Addition

Let s_f and s_g be two secrets that are shared with Shamir's secret sharing scheme using the polynomials $f(x)$ and $g(x)$, respectively. Every player has a share of both secrets denoted by $s_{i,j}$ where i is the polynomial and j is the player. The addition of s_f and s_g can be done locally by each player simply by adding its own shares of the secrets s_f and s_g resulting in a new share for each player. For three players the calculations are:

$$\begin{aligned} s_{new,1} &= s_{f,1} + s_{g,1} \\ s_{new,2} &= s_{f,2} + s_{g,2} \\ s_{new,3} &= s_{f,3} + s_{g,3} \end{aligned}$$

This is possible due to the following calculations. Let f and g be the two polynomials:

$$\begin{aligned} f(x) &= s_f + r_{1_f}x + r_{2_f}x^2 + \ldots + r_{t-1_f}x^{t-1} \\ g(x) &= s_g + r_{1_g}x + r_{2_g}x^2 + \ldots + r_{t-1_g}x^{t-1} \end{aligned}$$

Let $h(x)$ be the sum of $f(x)$ and $g(x)$:

$$\begin{aligned} h(x) &= f(x) + g(x) \\ h(x) &= (s_f + s_g) + (r_{1_f} + r_{1_g})x + (r_{2_f} + r_{2_g})x^2 + \ldots + (r_{t-1_f} + r_{t-1_g})x^{t-1} \\ h(x) &= (s_f + s_g) + r_1x + r_2x^2 + \ldots + r_{t-1}x^{t-1} \end{aligned}$$

The result is a new polynomial with the same degree as $f(x)$ and $g(x)$, where the coefficients in each term of $h(x)$ is the sum of the coefficients in the corresponding terms of $f(x)$ and $g(x)$. The polynomial $h(x)$ intersects the y-axis in the same point as the addition of $f(x) + g(x)$.

3.6 Multiplication

Multiplication is a bit more complicated than addition. Again, let s_f and s_g be two secrets that are shared using the polynomials $f(x)$ and $g(x)$, respectively, which are of degree $t-1$. The multiplication of two polynomials

of degree $t-1$ will result in a new polynomial $h(x)$ with degree $2t-2$. This would require more points for the interpolation used to reconstruct the secret, meaning that more players have to participate in the reconstruction. Additional multiplications will raise the degree even further, eventually rendering the interpolation impossible due to the lack of participating players. To overcome this problem, $h(x)$ needs to be reduced to the original degree $t-1$. Let f and g be the two polynomials:

$$f(x) = s_f + r_{1_f}x + r_{2_f}x^2 + \ldots + r_{t-1_f}x^{t-1}$$
$$g(x) = s_g + r_{1_g}x + r_{2_g}x^2 + \ldots + r_{t-1_g}x^{t-1}$$

The multiplication of $f(x) \cdot g(x)$ results in a new polynomial $h(x)$:

$$h(x) = f(x) \cdot g(x)$$
$$h(x) = s_f g(x) + r_{1_f}xg(x) + r_{2_f}x^2 g(x) + \ldots + r_{t-1_f}x^{t-1}g(x)$$
$$h(x) = s_f s_g + s_f r_{1_g}x + s_f r_{2_g}x^2 + \ldots + r_{1_f}s_g x + \ldots + r_{t-1_f}x^{t-1}r_{t-1_g}x^{t-1}$$

To clarify $h(x)$ can be written in the following form:

$$h(x) = s_f s_g + r_1 x + r_2 x^2 + \ldots + r_{2t-2}x^{2t-2}$$

Each player now holds a "share" of $h(x)$, a polynomial of degree $2t-2$, which needs to be reduced to a degree $t-1$ polynomial. These $h(x)$ outputs are then used as input to a new round of sharing, which results in a new set of shares in new random polynomials on the form of $i(y)$:

$$i(y) = h(x,y) = h(x) + r_1 y + r_2 y^2 + \ldots + r_{t-1}y^{t-1}$$

3.6.1 Multiplication Example

In order to explain the secret shared multiplication, an example with small numbers is included below. Two secret are defined (3 and 2), and the two polynomials are set to:

$$f(x) = 3 - 2x$$
$$g(x) = 2 + x$$

Each player has a share in $f(x)$ and $g(x)$ where $x = 1, 2, 3$ for player 1, player 2 and player 3, respectively.

Player 1:	$f(1) = 1$	$g(1) = 3$
Player 2:	$f(2) = -1$	$g(2) = 4$
Player 3:	$f(3) = -3$	$g(3) = 5$

By multiplying the shares from $f(x)$ and $g(x)$ each player obtain a "share" in $h(x)$. The players share these values with a new random polynomial as shown in row 4 in Table 3.3. The rest of the table is calculated by each player inputting $x = 1, 2, 3$ in its own polynomial and distributes a share to each of the other players. As an example, player 1 calculates:

$$\begin{aligned} \text{Share 1:} \quad & 3 + 2 \cdot 1 = 5 \\ \text{Share 2:} \quad & 3 + 2 \cdot 2 = 7 \\ \text{Share 3:} \quad & 3 + 2 \cdot 3 = 9 \end{aligned}$$

Player 1 then distributes share 2 to player 2 and share 3 to player 3. Both player 2 and player 3 calculate their column in Table 3.3 and distribute the shares to the other players.

	Player 1	**Player 2**	**Player 3**	
f(x)	1	-1	-3	
g(x)	3	4	5	
h(x)	3	-4	-15	
	3 + 2x	-4 + 3x	-15 + x	S_h
Player 1	5	-1	-14	4
Player 2	7	2	-13	2
Player 3	9	5	-12	0

Table 3.3: Example matrix for secret shared multiplication.

Now each player holds its secret polynomial (player 1 holds $3 + 2x$ etc.) together with a single point from each of the other players' polynomials (player 1 receives -1 from player 2 and -14 from player 3). With this information, each player can calculate its share of the total polynomial using one of two methods:

- Linear system approach (Appendix C.1)
- Vandermonde matrix (Appendix C.2)

These calculations give the players the following values, which can also be found in the s_h column in Table 3.3:

Player	**Share value**
1	4
2	2
3	0

Table 3.4: The players' shares of the total polynomial.

When a subset of at least two players exchanges shares the secret can be reconstructed. By plotting the values as shown in Figure 3.3, the secret is found where the line intersects the y-axis. The revealed number is 6, which corresponds with the multiplication of the initial secrets 3 and 2.

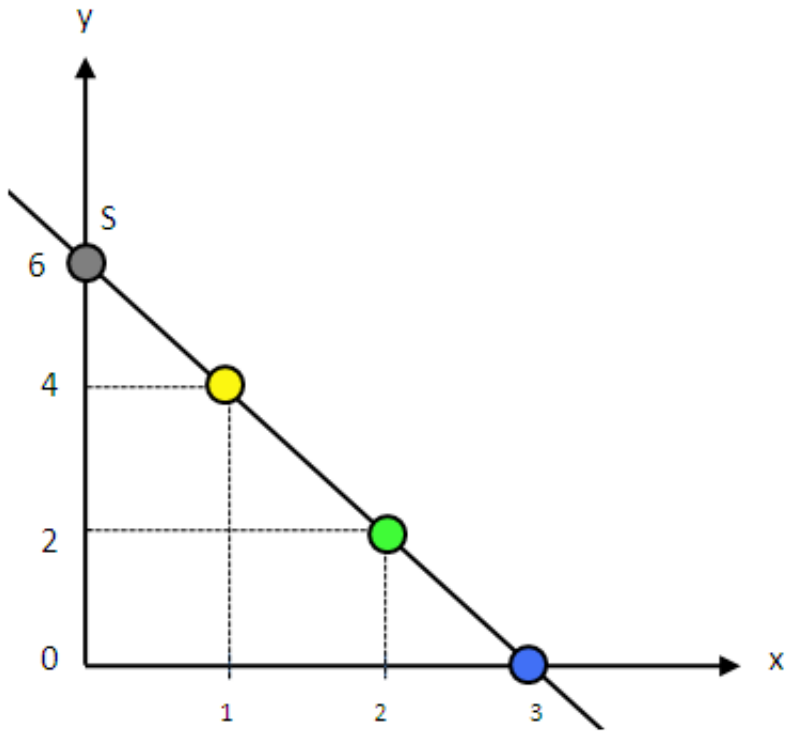

Figure 3.3: Graph with the points (1,4), (2,2) and (3,0) which implies that the secret is 6 (the line intersects the y-axis).

Chapter 4

Virtual Ideal Functionality Framework

VIFF is used as the framework for experimenting and realizing SMPC in this thesis, more specifically, a protocol for distributed RSA has been implemented using this framework. This chapter will present some useful information regarding the background of VIFF and the framework in general. A complete guide for setting up VIFF on a computer using Windows XP can be found in Appendix B.

4.1 Background

VIFF was started by Martin Geisler in March 2007 and is now a part of the Computer Aided Cryptography Engineering (CACE). It grew out of the research project SIMAP, which is the successor of Secure Computing Economy and Trust (SCET). All these projects aim to increase the easiness of using secure protocols when many parties are communicating, regardless of what type of protocol they perform.

VIFF is a library with building blocks for developing secure cryptographic protocols. The goal is to provide a solid basis on which practical applications using SMPC can easily be developed. By hiding most of the difficult mathematics behind MPC, security can be achieved more efficient and with less knowledge, and therefore making it more realistic for use in real life scenarios. Recall from Figure 1.1 that VIFF already contains the modules for doing network communication, secret sharing and MPC operations such that the programmer can focus on developing applications instead of the underlying mathematics and security.

Since the start, the supported MPC mathematical operators have increased

and the VIFF team is continuously working on speeding up the framework. Already a large number of useful applications are available for free use from the VIFF web page ([Tea09]) and the number is growing.

4.2 Model

VIFF is implemented using Python ([vR08]) and Twisted ([Lef09]), it's free to use and is supported on all major platforms (Linux, Windows and Mac OS X). Each module and each application in VIFF is written as a standard Python file.

Python is a very flexible, high level programming language with support for object-oriented programming. Twisted is an event-driven network programming framework written in Python that abstracts the low-level socket communication away for the programmer, which allows the programmer to implement efficient asynchronous (see below) network applications in an easy way.

Figure 4.1 (taken from [Gei09]) shows the language stack for VIFF applications:

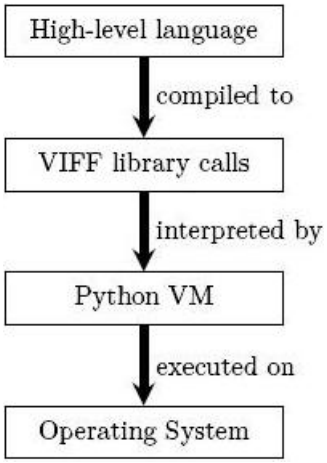

Figure 4.1: The language stack: VIFF is used as an intermediate language between the programmer and the Python virtual machine (VM).

4.3 Security Assumptions

As explained earlier, the standard model for trust used today is the use of a TTP. If a set of players have access to an incorruptible TTP, then this would be an ideal process ([Can01]):

Definition 5. ***Ideal process*** *imply that all players hand their inputs to a trusted party who locally computes the outputs, and hands each player its prescribed outputs.*

VIFF allows programmers to write programs as if they had access to an ideal functionality, which means an incorruptible party in an ideal protocol, making it an ideal process (for more information about the ideal functionality, see [Gei09] and [rGiN09]). This includes both correctness, that is the expected input-output relations of uncorrupted parties, and secrecy, which means the acceptable leakage of information to the adversary. Therefore ideal protocols can be performed using VIFF with the use of a simulated TTP.

In addition to the ideal functionality, VIFF states three security assumptions ([Tea09]):

- The adversary can only corrupt up to a certain threshold of the total number of players. The threshold will normally be 1/2 of the players, so for three players, at most one player may be corrupted (there must be an honest majority).
- The adversary is computationally bounded. The protocols used by VIFF rely on certain computational hardness assumptions, and therefore only polynomial time adversaries are allowed.
- The adversary is passive. Being passive means that the adversary only monitors the network traffic, but still follows the protocol.

4.4 Implementation

This section explains the important concepts of how VIFF is implemented. Some code examples are included to simplify the understanding of the framework.

4.4.1 The Basics

The implementation of VIFF seeks to offer an easy way of writing SMPC programs. In order to do so, there needs to be a well defined foundation that allows the programmer to focus on what to input to the function f of the MPC, instead of the mathematics that are being done in the background in order to maintain security. An example of a simple VIFF code snippet is included in Figure 4.2.

This is a very simple example to show how MPC are easily carried out in VIFF. Remember from Section 3.2 that MPC consist of three stages,

```
# (standard program setup not shown)

def some_function(self):
    input = int(raw_input("Your input: "))
    Zp = GF(1031)
    a, b, c = rt.shamir_share([1, 2, 3], Zp, input)

    total_plus = a + b + c
    total_mult = a * b * c
    open_total_add = rt.open(total_add)
    open_total_mult = rt.open(total_mult)

    results = gather_shares([open_total_add, open_total_mult])
    results.addCallback(self.results_ready)

def results_ready(self, results):
    print "Inputs added = " + str(results[0].value)
    print "Inputs multiplied = " + str(results[1].value)
```

Figure 4.2: Simple VIFF code for sharing three values from three players, calculating the sum and the product of the inputs and revealing them.

the input, computation and final stage. These can clearly be seen in the code snippet shown in Figure 4.2. First every player has an input, which is secret shared among the players, yielding the shares a, b and c for each player. Next, some computations are done with the shares, in this example the sum and product are calculated. At last, the result is computed and revealed by printing it to the screen. To fully understand the whole code, the concept of the Share class, the Runtime class and the Field class needs to be understood. These are the three main layers used for implementing protocols using VIFF. Briefly, the programmer writes program where Python integers or Share objects are manipulated. The runtime deals with shares and normal integers, and the field elements deal with modular arithmetic. An illustration of the relationship between the three layers is illustrated in Figure 4.3.

4.4.2 Deferred and Shares

Central to the Twisted application is the concept of the *Deferred* class. A deferred is a value that has not yet been computed because some of the data required to calculate the value is not ready yet, but it is certain that it will obtain a value sometime in the future. A deferred object can be passed around, just like ordinary objects, but it cannot be asked for its value. The deferred objects works by adding what is called a callback chain to the object. Callbacks are simply function pointers, and each function that depends

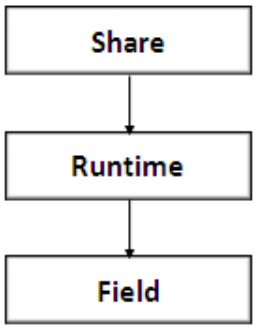

Figure 4.3: The three main layers in VIFF. Applications are written using Share objects, which is interpreted in the runtime, where operator overloading etc. is conducted automatically. At the bottom, the actual values that are being calculated are represented as field objects.

on a deferred, is added in that deferred's callback list. When the deferred obtain its value (typically when some share value is received over the network), these functions will be called, which again can possibly create a chain of callbacks in other deferred objects.

VIFF uses a class called *Share* to represent and do calculations on shares. This Share class is a subclass of the Deferred class found in Twisted. Furthermore, the Share class overloads the arithmetic operators. If a and b are Share objects, then the expression x = a + b will create a new Share object x which will eventually contain the correct share value of a and b added. This overloading of arithmetic operators is delegated to a Runtime instance, which is described below.

4.4.3 Runtime

The Runtime class is located in the runtime.py file and is one of the cornerstones in VIFF. As explained above, once the values are represented using the Share class, code to compute share values can be written using standard operators for plus, minus, multiplication and so on. This is accomplished by overloading the operators as shown in Figure 4.4.

Figure 4.4 only shows the overloaded definitions for addition and multiplication, several others exist, like XOR, less than, less than or equal, equal, just to mention some. These are just the redefinitions of the operators, the actual code is found in the Runtime class (still in the runtime.py file). The code for shared addition is included in Figure 4.5.

The code shown in Figure 4.5 is not very complex. First the field variable is set to the field used for share_a. If this field is undefined, use the field from

```
class Share(Deferred):
    """A shared number.

    # (some code left out)

    def __add__(self, other):
        """Addition."""
        return self.runtime.add(self, other)

    # (several operators between)

    def __mul__(self, other):
        """Multiplication."""
        return self.runtime.mul(self, other)
```

Figure 4.4: Definitions for overloading the basic operators for shares in VIFF (screen capture from runtime.py).

share_b instead, and if none are defined, it's equal to None since neither share_a nor share_b is actually a share. Next, make Share objects of both the variables share_a and share_b (if they are not already Share objects). The *gather_shares* function defines which variables that must contain values before the result is ready. Next comes one of the reasons for using Python to implement VIFF, namely the easy use of the so-called *lambda* functions. Lambda functions are anonymous functions (nameless functions) that are created at runtime, and do not need to be defined elsewhere in the code. In the code for addition on shares, a lambda function is defined at runtime to take two parameters, a and b, and to return the sum of these two. Next, this lambda function is added as a callback on the result variable, which means that when share_a and share_b obtains its values, the lambda function will be calculated with a equal to the value of share_a and b equal to the value of share_b.

4.4.4 Fields

The implementation of Galois fields is modeled in the field.py file, and are used to do all kinds of modular arithmetic. There are two possible field classes to use, *FieldElement* and *GF256*. A field of type FieldElement uses a prime p as the characteristic and the order, whereas GF(256) uses the field GF(2^8), with characteristic of 2 and with an order of 256.

The values contained in FieldElement objects or GF256 objects holds the concrete values on which calculations are performed, and is therefore an ordinary Python integer. Examples of how operations are conducted in finite fields can be found in Section 2.1, and VIFF code examples are found in Figure 4.2, where a field GF(1031) is instantiated such that the shares

```
class Runtime(BasicRuntime):
    """The VIFF runtime.

    # (some code left out)

    def add(self, share_a, share_b):
        """Addition of shares.

        Communication cost: none.
        """
        field = getattr(share_a, "field", getattr(share_b, "field", None))
        if not isinstance(share_a, Share):
            share_a = Share(self, field, share_a)
        if not isinstance(share_b, Share):
            share_b = Share(self, field, share_b)

        result = gather_shares([share_a, share_b])
        result.addCallback(lambda (a, b): a + b)
        return result
```

Figure 4.5: VIFF code for adding Share objects.

a, b, c are FieldElement objects with modulus 1031.

4.4.5 Asynchronous Communication

The communication model used on the Internet today is asynchronous, which means that communication can be initiated at arbitrary points in time and there's no guarantee that a message is delivered before a certain time (as opposed to synchronous communication) or delivered at all. VIFF implements asynchronous MPC with the use of Twisted explained earlier. One big benefit of using asynchronous communication is that there is no need to adapt the communication into time slots, where each time slot must be long enough for all honest players to deliver their communication to every other player, and if the time slot is not sufficient for one or more players, they will marked as corrupted players. This increases the efficiency of the framework since all the calculations are performed as soon as they can be performed for all players individually. The Share objects and the lambda functions explained earlier is necessary in order to be able for VIFF to function in this way, because all required values does not necessarily arrive at the same time given that the communication model is asynchronous.

Although VIFF is an asynchronous framework, the framework has the opportunity to have a single synchronization point, but this is not necessary. If such a synchronization point is chosen, the players have an asynchronous communication model all the time up to the synchronization point and all

the time after. This requires two modifications of the asynchronous model as follows ([Tea09]):

- The protocol is allowed to have one synchronization point. More precisely, the assumption is that a certain time-out is set, and all messages sent by honest players before the deadline will also be delivered before the deadline.
- There is no guarantee that the protocol always terminates and gives output to all honest players. Instead, the following is required: The preprocessing phase of the protocol, up to the synchronization point, never releases any new information to the adversary. The adversary may cause the preprocessing to fail, but if it terminates successfully, the protocol is guaranteed to terminate with output to all honest players.

The first point is simply to ensure that every honest player have the possibility to contribute input as long as they reach the deadline set. The second point of course gives an adversary extra power to stop the protocol, but if the adversary stops the protocol at a point where no information is released, why bother wasting time on it?

4.4.6 Parallel Execution

As described in Chapter 3, addition and multiplication in MPC are quite different. Addition is very easy, and requires no communication between players (done locally), while multiplication on the other hand is much more complex and involve communication between players. It is therefore desirable that many sequential computations are not conducted in a serial manner if these can actually be conducted in parallel. VIFF solves this problem by building a tree structure of computations. With the use of the Share class, all computations can be scheduled before the actual values are ready, which speeds up the performance. The concept for tree structure computations in VIFF is shown in Figure 4.6 (taken from [rGiN09]).

As shown in Figure 4.6, x and y are independent and can therefore be calculated in parallel and since all shares are Share objects, z can also be scheduled and calculated as fast as x and y has obtained their values. The concept of time-saving by scheduling and calculating in parallel is shown in Figure 4.7 (taken from [Gei08c]).

Given that network latency is the dominant factor in benchmark of MPC conducted in VIFF ([Gei08b]), efficient use of network resources is a key to speeding up the framework. The property of scheduling and calculating many operations in parallel greatly reduces the average cost of computations

```
# (Standard program setup not shown.)
input = int(raw_input("Your input: "))
Zp = GF(1031)

a, b, c = rt.shamir_share([1, 2, 3], Zp, input)
d = rt.prss_share_random(Zp)

x = a * b
y = c * d
z = x + y
```

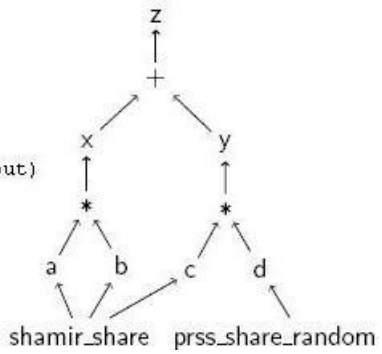

Figure 4.6: Left: A small VIFF code snippet for secret sharing values a, b, c and d using the shamir_share and the prss_share_random functions respectively (pseudo-random secret sharing). Right: A tree structure of how the calculations are parallelized and calculated separately to increase efficiency.

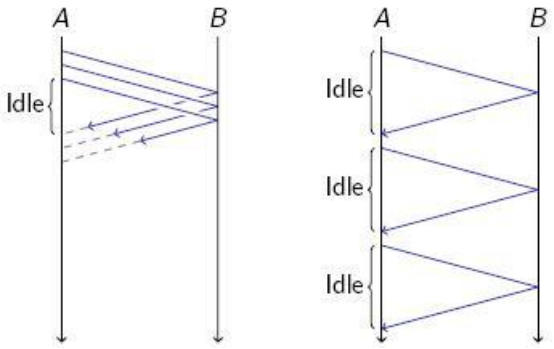

Figure 4.7: Three multiplications scheduled in parallel (to the left) and in serial (to the right) between two parties A and B. The difference amount of idle time spent for the two approaches is shown. Every operator that requires communication between players will follow the same scheme.

in VIFF according to benchmarks for multiplications in Figure 4.8 (taken from [Gei08c]).

From Figure 4.8 it can be seen that from 50 multiplications scheduled in parallel to 12000 multiplications scheduled in parallel, the average used per multiplications decreases from approximately 4.5 milliseconds per multiplication to approximately 1.3 milliseconds per multiplication. As for the serial scheduling case, the average is, not surprisingly, almost constant and approximately 190 milliseconds per multiplication. This tremendous decrease in average time per multiplication makes VIFF a lot more efficient.

The same decrease in average time can be found in all operators that requires communication between players, such as comparison operators. For more

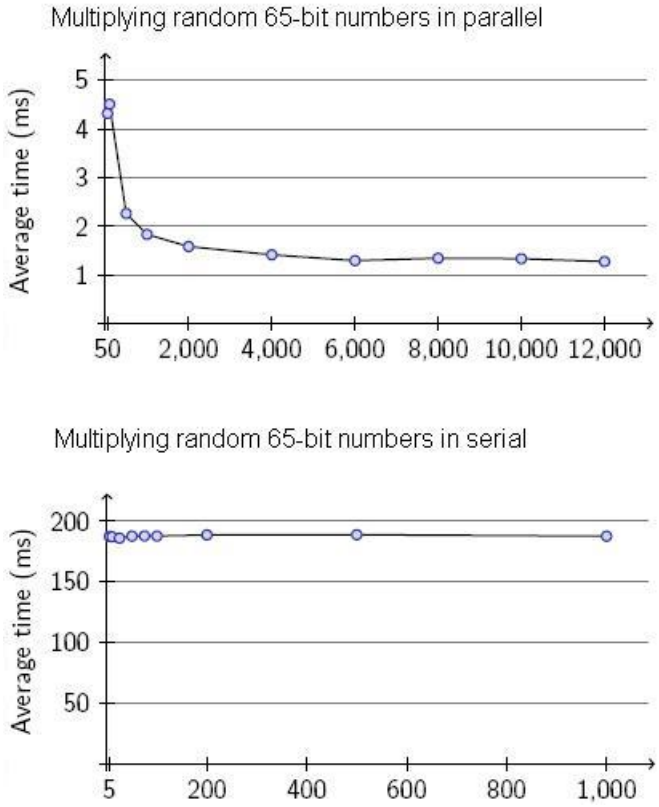

Figure 4.8: VIFF benchmarking: Multiplying random 65-bit numbers in parallel and in serial.

details on VIFF benchmarks see [rGiN09], [Gei08a], [Gei09] and [Gei08c].

Chapter 5

RSA

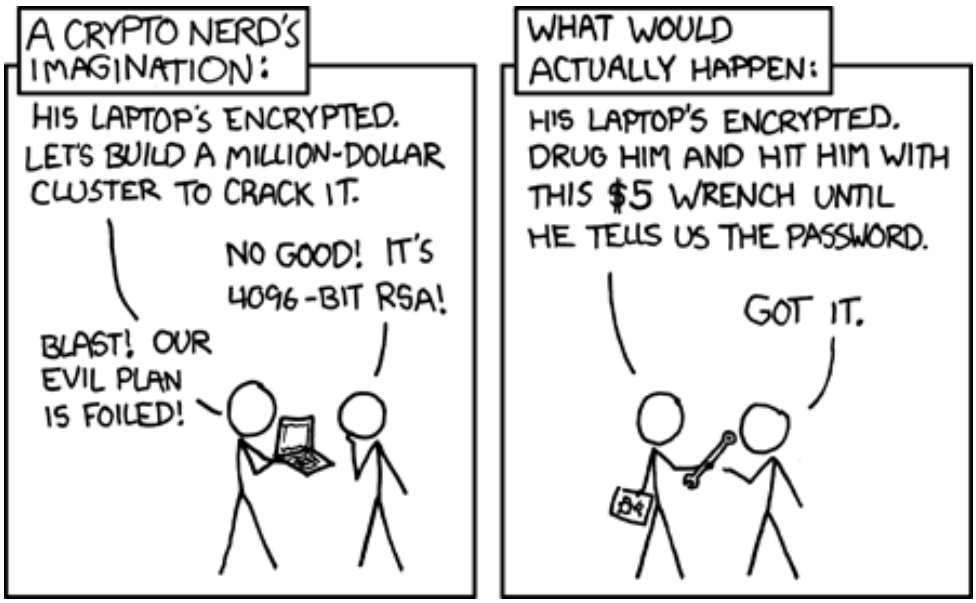

Figure 5.1: Comic strip from xkcd regarding RSA security ([XKC09]).

This chapter will present public-key encryption in general before describing the RSA scheme. Both the standard RSA scheme and a distributed RSA scheme will be described.

5.1 Public-key Encryption

Public-key encryption (also known as asymmetric encryption) is a form of cryptosystem that uses different keys for the encryption and decryption procedures, one *public key* (PU), which is known by all, and one *private key*

(PR), which is only known by the one generating it. One of the keys is used along with an encryption algorithm to transform a plaintext into a ciphertext, while the paired key used along with a decryption algorithm recovers the plaintext from the ciphertext again. Public-key encryption can be used for confidentiality[1] (encryption), authentication[2] (digital signature), or both.

The opposite of public-key encryption is secret-key encryption (also known as symmetric encryption) where the same key is used for both encryption and decryption.

Figure 5.2 shows how public-key encryption is conducted when Alice wants to send a message to Bob in such a way that only Bob can read the message.

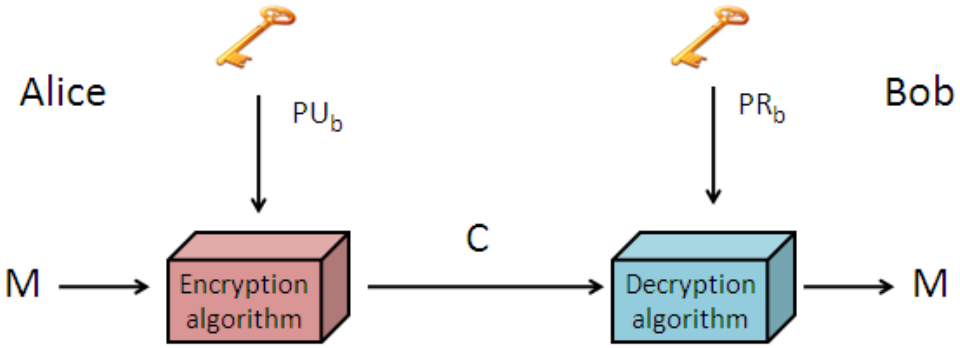

Figure 5.2: Public-key encryption: Alice sends a confidential (encrypted) message to Bob using Bob's public key, PU_b. Bob is the only one with the paired key, PR_b, and therefore the only one who can decrypt and read the message.

First Alice downloads Bob's public key (PU_b) which is publicly available. Alice then inputs the message M and Bob's public key to the encryption algorithm and sends the output from the encryption (ciphertext C) to Bob. Once M is encrypted with Bob's public key, only the paired key (Bob's private key, PR_b) can obtain M again. This is done by Bob inputting the ciphertext C and his private key to the decryption algorithm which outputs M for Bob to read.

Figure 5.3 shows how public-key encryption is conducted when Bob wants to authenticate that a message actually is sent by him (referred to as signing a message). Notice that in this figure, the arrows are from Bob, while in Figure 5.2 they are towards Bob. This is because now the authentication is the important property, Bob wants to prove that he actually sent the

[1] Confidentiality: Protection of data from unauthorized disclosure ([Sta06]).

[2] Authentication: Assurance that the communicating entity is the one that it claims to be ([Sta06]).

message, but the message is not secret. The procedure is as follows: Bob uses his private key to encrypt a message M and sends it to Alice, which in turn uses Bob's public key to assure herself that this message actually is from Bob. Anyone who obtains the ciphertext C sent from Bob can decrypt it by using Bob's public key.

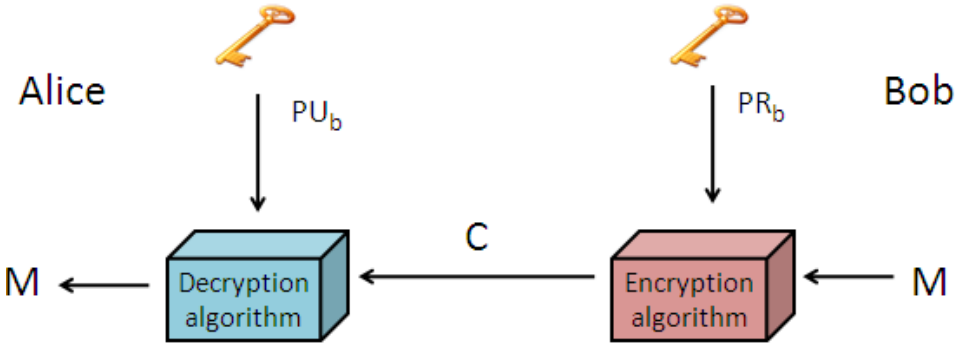

Figure 5.3: Public-key encryption: Bob sends an authenticated (signed) message to Alice by using his private key, PR_b to encrypt a message M into the ciphertext C. Alice receives C and uses Bob's public key, PU_b to decrypt the ciphertext C into the message M.

Notice that if Bob wants to send a confidential and authenticated message to Alice, he first needs to sign the message using his own private key and the message M as input to the encryption algorithm, obtaining C_1. Next, inputting this C_1 along with Alice's public key to the encryption algorithm, results in C_2, which he sends to Alice. Alice would now need to use her own private key and C_2 as input to the decryption algorithm to obtain C_1. Lastly, Alice inputs C_1 and Bob's public key to the decryption algorithm and obtains the message M. A figure for this scheme is omitted here.

A common misconception about public-key schemes is that one pair of keys is enough to send messages back and forth between two or more players. But as can be seen in Figure 5.2 and 5.3 this would only yield confidentiality one way and authentication the other way. Therefore, each player needs a own key pair in order to be able to maintain both confidentiality and authentication both ways.

Well-known public-key algorithms include RSA ([RSA78]), Diffie-Hellman key exchange ([DH76]) and ElGamal encryption system ([EG85]), where RSA is based on the difficulty of factoring large numbers, whereas both Diffie-Hellman and ElGamal relies on the difficulty of computing discrete logarithms.

5.2 RSA Scheme

RSA was developed by Ron Rivest, Adi Shamir and Leonard Adleman at Massachusetts Institute of Technology (MIT) in 1977 and published in 1978 in the article [RSA78]. Since then, it has become the most widely used general-purpose algorithm for public-key encryption. The security of RSA relies on the difficulty of factoring large numbers, more specifically the factoring of the public modulus N.

The RSA algorithm consists of four separate parts, namely: *key generation*, *encryption*, *decryption* and *signature*, all described in full detail below.

Key generation:

Select $p,\ q$	p and q are both prime, $p \neq q$
Calculate $N = p \cdot q$	
Calculate $\varphi(N) = (p-1)(q-1)$	
Select an integer e	$gcd(\varphi(N),\ e) = 1,\ 1 < e < \varphi(N)$
Calculate d	$d \equiv e^{-1}\ (mod\ \varphi(N))$
Public key	$PU = \{e,\ N\}$
Private key	$PR = \{d,\ N\}$

Encryption:

Plaintext	$M < N$
Ciphertext	$C = M^e\ mod\ N$

Decryption:

Ciphertext	C
Plaintext	$M = C^d\ mod\ N$

Signature:

Plaintext	$M < N$
Ciphertext	$C = M^d\ mod\ N$
Verification	$M = C^e\ mod\ N$

The protocol for key generation must be done first. It starts by finding two distinct prime numbers p and q. From p and q, N and $\varphi(N)$ are calculated as $N = p \cdot q$ and $\varphi(N) = (p-1) \cdot (q-1)$ where $\varphi(N)$ is the Euler totient function[3]. Next, an integer e is selected such that the greatest common divisor (gcd) of $\varphi(N)$ and e is equal to 1, and e is a number larger than 1 but less than $\varphi(N)$. The reason why e is selected with these restrictions is that

[3]Euler's totient function $\varphi(N)$ is defined to be the number of positive integers less than N and relatively prime to N. This means that if p is prime, then $\varphi(p) = p-1$ by definition of prime numbers. It can also be shown that if p and q are distinct primes, with $N = p \cdot q$, then $\varphi(N) = \varphi(pq) = \varphi(p) \cdot \varphi(q) = (p-1) \cdot (q-1)$ ([Fra03]).

there must exist an inverse to $e \ mod \ \varphi(N)$. Such an inverse can be found by using the Extended Euclidean Algorithm (see [CLRS01], [Ros03] or [TW06]) if and only if $\gcd(\varphi(N), e) = 1$. Lastly, the inverse $d \equiv e^{-1} \ mod \ \varphi(N)$ is calculated, yielding the public key PU $= \{e, N\}$ and the private key PR $= \{d, N\}$. The size of a RSA key is the length of the modulus N in bits, meaning that p and q preferably are approximately half the key size in length each.

Encryption and decryption are closely related, and encryption must be conducted first. Say Alice wants to send a message M to Bob such that no other person than Bob can read the message as shown in Figure 5.2. Alice downloads Bob's public key $\{e, N\}$ and calculate the ciphertext as $C = M^e \ mod \ N$, before sending C to Bob. Notice if M is larger than N, Alice needs to break the message into smaller pieces before encrypting it, $M_1, M_2 \ldots$, where each $M_i < N$, resulting in $C_1, C_2 \ldots$.

When Bob receives C from Alice, he can use his private key $\{d, N\}$ to obtain the plaintext M. This is done by Bob calculating $M = C^d \ mod \ N$. It can easily be seen that this protocol is correct:

$$\begin{aligned} C &= M^e \ mod \ N \\ M &= C^d \ mod \ N \ = \ (M^e)^d \ mod \ N \ = \ M^{ed} \ mod \ N \end{aligned}$$

Since d is the inverse of e $(mod \ N)$, this means that $ed \equiv 1 \ mod \ N$, which in turn makes the last expression equal to $M^1 \ mod \ N = M$.

The signature protocol is very similar to encryption and decryption, the only difference is that the private key is used for encryption and the public key is used for decryption, the opposite of how the keys are used in standard encryption and decryption. An example with actual values for encryption and decryption are included in Example 6 below.

Example 6. (**RSA encryption/decryption**) Bob has generated a valid RSA key as explained above obtaining the values:

$$\begin{aligned} p &= 19 \\ q &= 23 \\ N &= p \cdot \ q \ = 437 \\ \varphi(N) &= (p-1) \cdot (q-1) \ = 396 \\ e &= 17 \\ d &= e^{-1} \ mod \ 396 \ = \ 233 \end{aligned}$$

This means that the Bob's private and public keys are $\{233, 437\}$ and $\{17, 437\}$, respectively. Alice wants to send a message $M = 2$ to Bob. She encrypts the message using Bob's public key to obtain the ciphertext C:

$$C \ = \ M^e \ mod \ N \ = \ 2^{17} \ mod \ 437 \ = \ 409$$

Alice then sends C to Bob, which uses his private key to obtain the message M:

$$M \;=\; C^d \; mod \; N \;=\; 409^{233} \; mod \; 437 \;=\; 2$$

Bob yields the message $M = 2$, which is correct. Signature is very similar, as explained earlier, therefore an example is omitted here.

5.3 Distributed RSA scheme

The difference between a standard RSA protocol and a distributed RSA protocol is that no single player can have complete knowledge of the private key, meaning that the private key needs to be secret shared among all the players. Distributed RSA can be performed in a number of different ways, although the method which has gained most acceptance is the one proposed by Dan Boneh and Matthew Franklin in their article *Efficient Generation of Shared RSA keys* ([BF97]), and in the updated and more detailed version *Experimenting with Shared Generation of RSA keys* ([MWB99]) by Michael Malkin, Thomas Wu and Dan Boneh. This method is the current milestone for generating distributed RSA keys, and will be described here. The generation of distributed RSA keys using this method consists of 4 steps: *Picking candidates*, *trial division on N*, *distributed biprimality test* and *calculate exponents*, shown in Figure 5.4. These steps, in addition to how to perform distributed decryption and distributed signature are all described in detail in the following.

5.3.1 Pick Candidates

This step is where candidates for p and q are chosen, but since the character p is also used as the order of the finite field, q will be used here for the candidates. Each player i picks a secret integer q_i and keeps it secret. For the protocol to work, N needs to be a Blum integer[4], therefore player 1 picks a random q_1 which is congruent to 3 *mod* 4, while the rest of the players picks q_i's which are congruent to 0 *mod* 4, such that the total $q = q_1 + q_2 + ... + q_k$ (where k is the number of players), is congruent to 3 *mod* 4, making N a Blum integer.

Next, the parties performs distributed trial division to determine that $q = q_1 + q_2 + ... + q_k$ is not divisible by any small prime less than a boundary $B1$ by using MPC. The distributed trial division is conducted as follow: Let q be as defined above, and let l be a small prime. To test if l divides q each player picks a random $r_i \in \mathbb{Z}_p$. Next, the players compute

[4] N is a Blum integer if $N = p \cdot q$ where p and q are distinct prime numbers congruent to 3 *mod* 4. That is, p and q must be of the form $4t + 3$, for some integer t ([Con09a]).

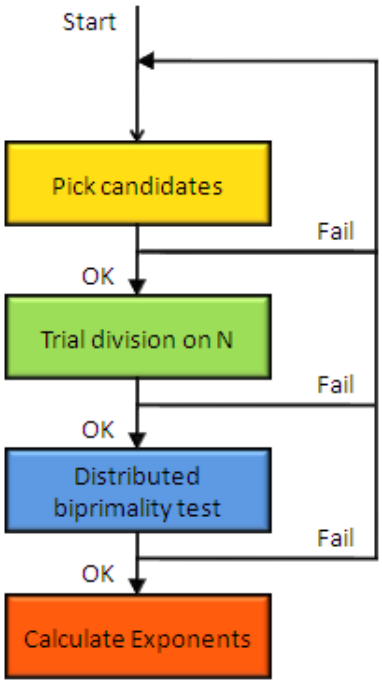

Figure 5.4: The distributed RSA protocol consists of 4 steps: Picking candidates, trial division on N, distributed biprimality test and calculate exponents. The protocol can fail at any of the 3 first steps, which means that new candidates must be picked.

$$qr = \left(\sum_{i=1}^{k} q_i\right)\left(\sum_{i=1}^{k} r_i\right) \ mod\ l$$

If $qr \neq 0$, then l does not divide q. By using this method a bad candidate is always rejected, but a good candidate can also be rejected if l divides $r = r_1 + r_2 + ... + r_k$. To decrease the probability of discarding a good candidate, do the test with two different picked r for each l such that $r1 = r_{11} + r_{12} + ... + r_{1k}$ and $r2 = r_{21} + r_{22} + ... + r_{2k}$, and therefore computing

$$qr1 = \left(\sum_{i=1}^{k} q_i\right)\left(\sum_{i=1}^{k} r_{1i}\right) \ mod\ l$$
$$qr2 = \left(\sum_{i=1}^{k} q_i\right)\left(\sum_{i=1}^{k} r_{2i}\right) \ mod\ l$$

The test for l is passed if at least one of the values is different from zero, that is if $qr1 + qr2 \neq 0$. If the test is passed, then set l to the next prime and redo the test until the boundary $B1$ is reached. If the test fails at any of the l's, then a new q needs to be picked as described above. If $B1$ is reached,

then q has passed the distributed trial division, and the other prime q is picked and tested in the same manner.

5.3.2 Trial Division on N

When both the candidates p and q have passed the distributed trial division, an MPC is conducted to compute $N = (p_1 + p_2 + ... + p_k) \cdot (q_1 + q_2 + ... + q_k)$, which is revealed to all players. As N is the product of two large candidate primes p and q, it should not be divisible by any other primes. The players therefore do a more comprehensive trial division on the revealed N locally to check that N is not divisible by any small prime in the range $[B1, B2]$ for some boundary $B2$ (typically much larger than $B1$). If it turns out that N is indeed divisible by a small prime up to $B2$, this test is declared a failure and the whole key generation protocol restarts by the players picking new values for the candidates p and q.

5.3.3 Distributed Biprimality Test

After the two trial division tests already conducted, it is clear that N is not divisible by any small prime numbers up to the boundary $B2$. The next test is a distributed test and also a probabilistic test since it's infeasible to check all prime number up to the square root[5] of p and q to be absolutely sure that p and q actually are prime numbers.

The distributed biprimality test consists of 4 steps (for proof of the correctness of the protocol the reader is referred to [BF97] due to the length of the proof).

Step 1: The players agree on a random $g \in \mathbb{Z}_N^*$.[6] This can be done by one of the players picking a random g and revealing it to all the other players.

Step 2: The players compute the Jacobi symbol[7] g over N. If $(\frac{g}{N}) \neq 1$ the protocol is restarted at step 1 by choosing a new g.

Step 3: Otherwise, the players computes $v = g^{\varphi(n)/4}\ mod\ N$ as an MPC. Note that $\varphi(n) = (p-1)(q-1) = N - p - q + 1$, therefore player 1 computes $v_1 = g^{(N-p_1-q_1+1)/4}\ mod\ N$. The rest of the players compute $v_i = g^{-(p_i+q_i)/4}\ mod\ N$. Next, all players secret share their values of

[5] If n is a composite integer, then n has a prime divisor less than or equal to $\sqrt{n}$. ([Ros03]).

[6] $\mathbb{Z}_N^*$ is the set of nonzero members of $\mathbb{Z}_N$ ([Fra03]).

[7] The Jacobi symbol is a generalization of the Legendre symbol ([Con09c]) and defined as follows: For any integer a and any positive odd integer n the Jacobi symbol is defined as the product of the Legendre symbols corresponding to the prime factors of n: $\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1}\left(\frac{a}{p_2}\right)^{\alpha_2}\ldots\left(\frac{a}{p_k}\right)^{\alpha_k}$ where $n = p_1^{\alpha_1}p_2^{\alpha_2}\ldots p_k^{\alpha_k}$ ([Con09b]).

v_i such that v can be calculated and revealed, $v = \prod_{i=1}^{k} v_i \ mod\ N$. Once v is revealed, the players check if:

$$v = \prod_{i=1}^{k} v_i \stackrel{?}{=} \pm 1 \ mod\ N$$

If the test fails the parties declare that N is not a product of two distinct primes, and the protocol is restarted from the beginning by picking new values for p and q.

Step 4: There are two ways of implementing step 4, and only the alternative step is shown here. This alternative step requires very little calculations, although there is a bit more communication between the players. The step tests if $\gcd(N, p + q - 1) > 1$. The players cannot reveal their private p_i and q_i, therefore each player picks a random number $r_i \in \mathbb{Z}_N$ and keeps it secret. Then they do an MPC by calculating z such that p and q are not revealed:

$$z = \left(\sum_{i=1}^{k} r_i \right) \cdot \left(-1 + \sum_{i=1}^{k} (p_i + q_i) \right) \ mod\ N$$

Next, z is revealed, and the players check if $\gcd(z, N) > 1$. If so, N is rejected, and the protocol is restarted from the beginning by picking new values for p and q. If N is actually a product of two distinct prime numbers, it will pass this test with overwhelmingly high probability. If N passes this test, then N is declared to be the product of two distinct primes, and the calculation of the public and private exponent can start.

5.3.4 Calculate Exponents

When p and q have been found and N has been calculated, the next step is to find e and d that form the public key and private key respectively together with N. There are two options regarding the public exponent e, it can be set to a standard (small) RSA exponent such that no calculations are required, or it can be calculated, and therefore vary from key to key. In this description, only the static e approach is outlined, which can use a chosen e less than approximately 2^{20} ([MWB99]). In the following, it's given that $\varphi = \varphi(N)$. Since e is an RSA exponent, it is given that $\gcd(e, \varphi) = 1$.

The calculation of $d = \sum_{i=1}^{k} d_i$ needs to be computed in a distributed manner, where at the end of the computation, each player only knows its own d_i. Traditionally, the gcd algorithm is used to find an inverse of $e \ mod\ \varphi$, but

that would involve computing modular arithmetic when the modulus is secret shared, which is possible, but really slow. The value of $\varphi = N - p - q + 1$ is not known by any of the players, but all players know their part, φ_i, where $\varphi = \sum_{i=1}^{k} \varphi_i$. Knowing this, fortunately there is a trick for computing $e^{-1}\ mod\ \varphi$ without using any reductions modulo φ. The trick involves three steps:

1. Compute $\varsigma = \varphi^{-1}\ mod\ e$.
2. Set $T = -\varsigma \cdot \varphi + 1$. Observe that $T \equiv 0\ mod\ e$.
3. Set $d = T/e$. It can be verified that $d = e^{-1}\ mod\ \varphi$ since $d \cdot e = T \equiv 1\ mod\ \varphi$. Using this method, the need for reductions modulo φ is avoided.

The protocol is performed as follows:

Step 1: The players compute the value of $l = \varphi\ mod\ e$. This can be done by each player calculating $l_i = \sum \varphi_i\ mod\ e$ locally, before doing a joint MPC to obtain the value $l = \sum l_i\ mod\ e$.

Step 2: Each player now calculates $\varsigma = l^{-1}\ mod\ e$ locally. As shown above $d = T/e = (-\varsigma \cdot \varphi + 1)/e$, therefore each player also locally calculates

$$d_i \ = \ \left\lfloor \frac{-\varsigma \cdot \varphi_i}{e} \right\rfloor$$

After each player has successfully calculated d_i, the RSA private exponent $d = \sum d_i + r$ where $1 \leq r < k$.

Step 3: Once each player has obtained its d_i, a final computation needs to be done in order to determine the value of r. Note that for a encrypted message c, the decryption would be

$$M \equiv c^d \equiv c^r \prod c^{d_i}\ mod\ N$$

Therefore, one of the players can determine the value of r simply by trying all possible r's in a trial decryption. Say player 1 is the one doing the trial decryption, it picks a random message $m \in \mathbb{Z}_N$ and computes $c = m^e\ mod\ N$. Then every player participates in a decryption of c. Each player calculates $m_i = c^{d_i}\ mod\ N$ locally, and sends the result to player 1. Player 1 knows that the value of r is in the range $1 \leq r < k$, and tries all of them to see which one satisfies

$$m \ \stackrel{?}{=} \ \Big(\prod m_i\Big) c^r\ mod\ N$$

At last player 1 updates d_1 by setting $d_1 = d_1 + r$. The distributed RSA protocol is now complete with the correct value of d secret shared among the k players.

Note that using a static e makes the protocol very efficient, but some bits of the key is leaked to all the players. The leakage happens when calculating $l = \varphi \ mod \ e$ and the trial decryption process where r is determined. This is a total of $log_2 \ e + log_2 \ k$ bits. This step can however be conducted such that no bits are leaked by using an arbitrary public exponent (calculated each time a key is generated), but this makes the protocol somewhat less efficient (see [BF97]). Another approach is to just increase the total number of bits in N to compensate for the leaked bits.

5.3.5 Decryption

Once a distributed RSA key is generated, the players can participate in a joint decryption of a ciphertext C that has been encrypted using the public key. In order to do so, the players do almost the same as are done when the trial decryption is conducted, only this time r is already found, therefore they do an MPC to find M directly:

$$M = \prod_{i=1}^{k} m_i \ mod \ N = \prod_{i=1}^{k} C^{d_i} \ mod \ N$$

The decryption process is conducted by each player locally calculating its part of M, $m_i = C^{d_i} \ mod \ N$, which in turn is secret shared among the players. Next, the players perform an MPC on the shared m_i's to obtain the total m

$$m = \prod_{i=1}^{k} m_i$$

The value of m is revealed and the message M is found by calculating

$$M = m \ mod \ N$$

5.3.6 Signature

The players can also sign a message, and therefore provide authentication. The signature process is performed as follows: A message M is to be signed using the secret shared part of the private key, d. The message M is chosen by one of the players and sent securely to all the other players. Then each player calculates its part, c_i, of the signature C:

$$c_i = M^{d_i} \ mod \ N$$

The values of the c_i's are then secret shared among all the players, and the total c is obtained by conducting an MPC

$$c = \prod_{i=1}^{k} c_i$$

The value of c is revealed and the signature C is found by calculating

$$C = c \ mod \ N$$

Chapter 6

Distributed RSA Implementation in VIFF

This chapter describes the implementation of a distributed RSA protocol for three players in VIFF, programmed as the main part of this thesis (the code can easily be altered to support more players). The implemented code is included in Appendix D, and a description of the general multiprecision Python (GMPY) module, which is used extensively throughout the implementation, is found in Appendix E.

The distributed RSA implementation is based on the protocol proposed in [BF97], which is described in detail in Section 5.3. Two changes have been made to the protocol to speed up the time needed to generate valid keys, regarding the distributed trial division and trial division on N, respectively. The implemented algorithm can generate arbitrary large key sizes (validated up to 4096 bits in VIFF) with a success rate of 100%. At the end of the key generation process, the players are convinced that the public N is the product of two unknown distinct primes, p and q, and that they share a valid key.

6.1 Coding Style

Recall from Section 3.2 that an MPC consists of three stages: Input stage, computation stage and final stage. Further, as explained in Section 4.4.1, these three stages are implemented using two functions in VIFF. The first function is used for the two first stages, inputting the values and do the computations. The second function is used for the final stage, to reveal the answer. Throughout the distributed RSA code, this procedure is used thoroughly, the first function having a descriptive name *function_name()* and the reveal function having the paired name *check_function_name()*.

The first step in the distributed RSA algorithm, involving picking candidates for p and q, have distinct functions for p and q, although the code contained in these functions are virtually alike, the only difference found is when the protocol moves to the next step (trial division on N). This choice is purely for simplifying the readability of the code, and does not make the code slower in any way, although more code is needed.

6.2 Initialization

The initialization process is a standard code used in all VIFF applications to setup the program with the right parameters and is written by the VIFF Developer Team. Basically, this code creates a runtime instance, parses the command line arguments into the application and starts the Twisted event loop. Next, the application is initialized by making an instance of the Protocol class, which is the main application. The code for the setup process is of course included in the distributed RSA code in Appendix D, but will not be described in any more detail in this thesis.

The main application starts at the __init__ function in the class Protocol, where variables marked as *changeable variables* must be set to the desired output from the protocol. These include the number of rounds to be conducted (both for key generation and for decryption), the key size for each round, and the boundaries for trial division.

6.3 Key Generation

The step to generating valid keys is by far the most time consuming step in the protocol. As shown in Figure 5.4, the algorithm for generating distributed RSA keys involve 4 steps, which will be described in the following.

6.3.1 Pick Candidates

An overview of the implemented function in VIFF for picking candidates is shown in Figure 6.1. To avoid misunderstandings with the candidate p and the size of the finite field $\mathbb{Z}_p$, q is used to represent both the candidates, p and q, in the following.

The implementation starts by each player picking a private q_i such that $q = q_1 + q_2 + q_3$. Next, the players perform a distributed trial division on q, which assures that q is not divisible by any small prime number up to the boundary $B1$.

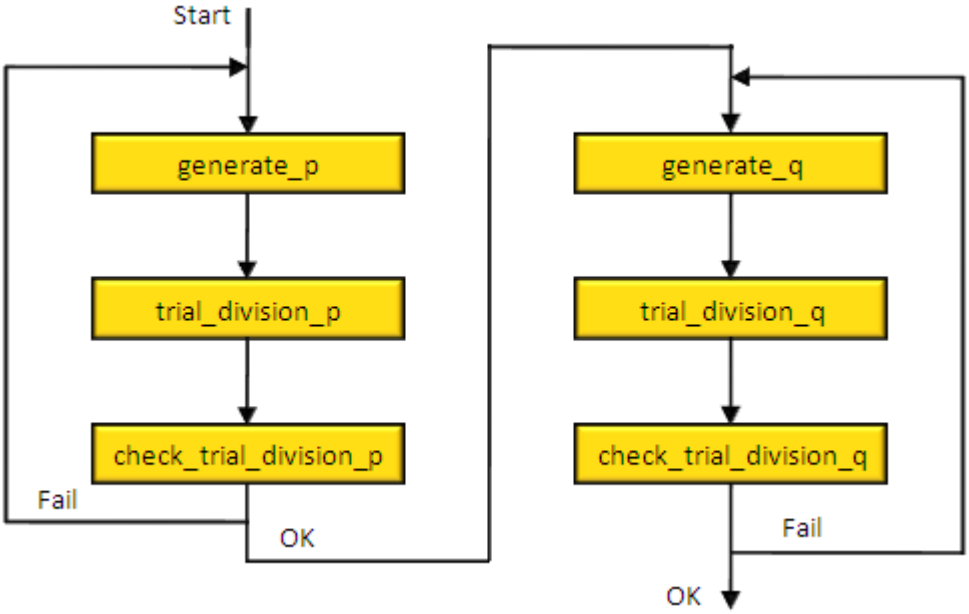

Figure 6.1: Flow chart for the implemented functions for picking candidates p and q and doing distributed trial division. Parameters to the functions are omitted.

The trial division in this thesis is implemented in a more optimized manner compared to the method described in Section 5.3.1, and is the first of the two improvements to the algorithm. Recall that for the distributed trial division in [BF97], the distributed trial test is conducted two times for each small prime $l < B1$ to decrease the probability of discarding a good candidate. By doing this the probability of discarding a good candidate is equal to

$$1 - \prod_{l<B1} (1 - \frac{1}{l^2}) < \frac{1}{2}$$

In the VIFF implementation, a boundary $B1 = 12$ is used, meaning that the probability of discarding a good candidate p and q, independently, would have been approximately 17.1%. Consequently nearly 1/5 of the good candidates would have been discarded for both p and q. The implemented method for the distributed trial division in this thesis has zero probability of discarding a good candidate, and is specially constructed for three players, although it can easily be expanded to support an arbitrary number of players. The method is as follow: Let $l < B1$ be a small prime number and q_i be player i's part of q. Each player now locally calculates $q\_trial_i = q_i \ mod \ l$ and picks a random integer $r_i \in \mathbb{Z}_p^*$. Then the players secret share the values of q_trial_i and r_i and computes

$$q\_trial\_tot = (q\_trial_1 + q\_trial_2 + q\_trial_3)$$
$$r\_trial\_tot = (r_1 + r_2 + r_3)$$

Next, for simplicity, set $t = q\_trial\_tot$, and the players computes $trial\_reveal$ as follows

$$trial\_reveal \;=\; t \cdot (t - l) \cdot (t - 2 \cdot l) \cdot r\_trial\_tot$$

and the answer $trial\_reveal$ is revealed to all players. The beauty of this method is that the revealed answer is now zero if and only if q is a bad candidate, else it's just a random number. The correctness of this method is because when summing up $t = \sum (q\_trial_i \; mod \; l)$, for three players, there are only three illegal t's, namely: 0, l and $2 \cdot l$ (for four players an additional multiplication of $(t - 3 \cdot l)$ would have been needed and so forth). If the revealed answer is not zero, it's just a random number that does not yield any information about q.

Notice from Figure 6.1 that the distributed trial division is performed on the primes p and q separately, which means failing either p or q, it's only necessary to generate the failed prime again.

6.3.2 Trial Division on N

The next step in the algorithm is to reveal N and continue by doing local trial division on this value. The second improvement in comparison to [BF97] is implemented for this step. An overview for the implementation functions regarding trial division on N is shown in Figure 6.2.

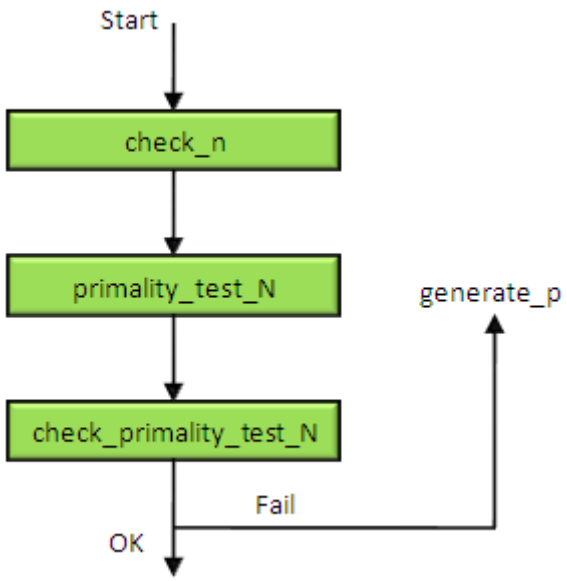

Figure 6.2: Flow chart for the implemented functions for doing trial division on the revealed N. Parameters to the functions are omitted.

In [BF97], once p and q have passed the distributed trial division and N has been computed using an MPC, a more comprehensive trial division on N in the range $[B1, B2]$ is conducted locally. In the VIFF implementation however, the same range $[B1, B2]$ is not checked by each player. Instead each player checks a different range of small primes, and when finished, all players agree whether any of the players have found an illegal factor of N.

The VIFF code is written such that the players collaborate to check that N is not divisible by any prime number less than 20000. Player 1 checks all the primes from $B1(= 12)$ to 15000 (1749 in total). Player 2 checks all primes from 15001 to 17500 (260 in total), and finally, player 3 checks all primes from 17501 to 20000 (248 in total). The code for the trial division on N is included in Figure 6.3 (comments omitted):

```
def primality_test_N(self):
    test_fail = 0
    for i in self.prime_list_b2:
        if self.n_revealed % i == 0:
            test_fail = 1
            break

    fail1, fail2, fail3 = self.runtime.shamir_share([1, 2, 3], self.Zp, test_fail)

    failed_tot = fail1 + fail2 + fail3
    open_failed_tot = self.runtime.open(failed_tot)

    results = gather_shares([open_failed_tot])
    results.addCallback(self.check_primality_test_N)
```

Figure 6.3: VIFF code for the local trial division on N. Each player checks a range of small primes before all players agree whether N has an illegal factor not equal to p or q.

As can be seen in Figure 6.3 each player runs through its list of prime numbers and checks that N is not divisible by any of them. If N turns out to be divisible by a prime number, the loop for that player will break. Next, the value from each player is secret shared and summed up by doing an MPC before it is revealed (the reveal function *check_primality_test_N* is not shown here). If the revealed answer is not equal to zero, one or more of the players have failed N as a candidate, and the whole process have to start over again with picking p and q. One interesting thing to notice here is why player 1 can check a much larger span of prime numbers. This is simply because of the **break** command in the code. The probability of a number N being divisible by any prime number is higher for small primes, and therefore player 1 will fail in the loop most often. If all players had checked equal amounts of prime numbers, then on average player 1 would fail rather quickly and would just be waiting, while player 2 and player 3 would most often have to finish their whole lists.

Both [BF97] and [MWB99] have proposed similar and quite different methods for optimizing this step, see more in Further Work (Chapter 9).

6.3.3 Distributed Biprimality Test

No changes have been done to the distributed biprimality test as it is explained in Section 5.3.3. The implemented functions for this step are shown in Figure 6.4. This test can fail when checking the v and when checking the z, both of which will result in picking new candidate primes p and q from the beginning. Failing the test on g is solved by picking a new g and does not have to start the whole protocol from the beginning.

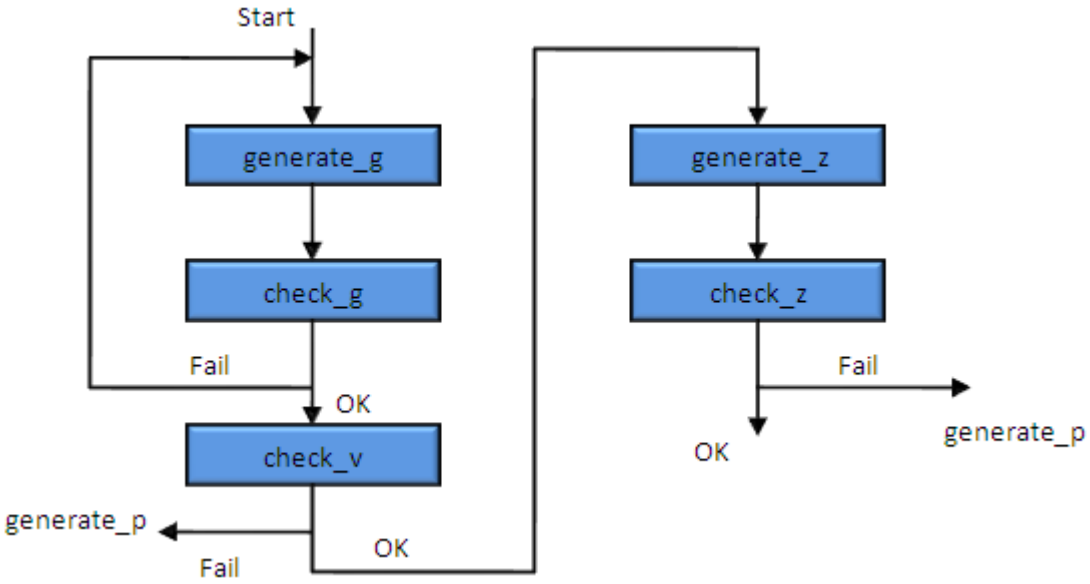

Figure 6.4: Flow chart for the implemented functions for performing the distributed biprimality test. Parameters to the functions are omitted.

One thing to notice in the implemented code is when generating the g (shown in Figure 6.5). The g is picked by player 1, but everyone needs to get hold of the value. This is done by a simple secret sharing where player 1 inputs its g-value, and the other players participate in the sharing, but does not input anything to the *shamir_share*-function. This way, the players can efficiently agree on the value of g.

6.3.4 Calculate Exponents

After the first three steps in generating a distributed RSA key, the rest of the protocol, calculating the exponents, cannot fail, and is therefore done

```
def generate_g(self):
    if self.runtime.id == 1:
        self.g = random.randint(1, self.n_revealed - 1)
        self.g = self.runtime.shamir_share([1], self.Zp, self.g)
    else:
        self.g = self.runtime.shamir_share([1], self.Zp)

    self.open_g = self.runtime.open(self.g)
    results = gather_shares([self.open_g])
    results.addCallback(self.check_g)


def check_g(self, results):
    self.g = results[0].value
    ...
```

Figure 6.5: VIFF code for sharing the g picked by player 1 among all players before revealing the value.

quickly in order of time consumption. The flow chart for implementing this step in VIFF is shown in Figure 6.6.

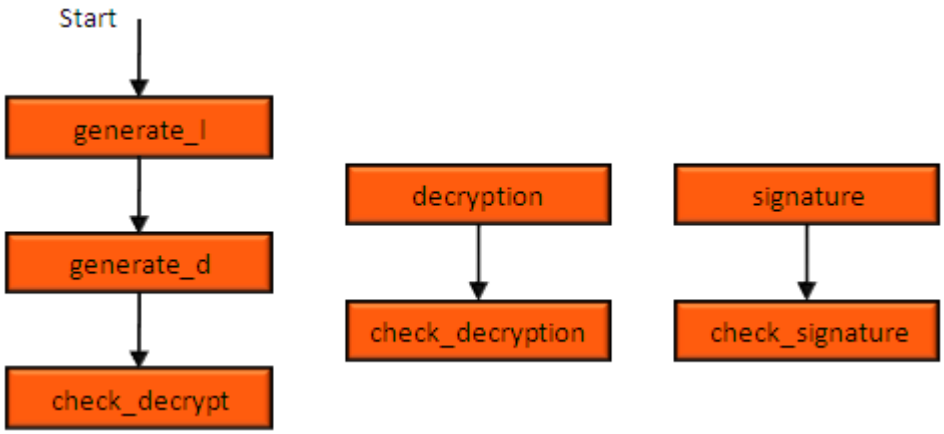

Figure 6.6: Flow chart for the implemented functions for calculating the public and private exponents. In addition, decryption and signature are shown as seperate functions, which can only be performed after the whole key generation is already done. Parameters to the functions are omitted.

This step is carried out precisely as described in Section 5.3.4, where the public exponent used is the standard RSA exponent $e = 2^{16} + 1 = 65537$ and player 3 is the one adjusting $d_3 = d_3 + r$ in the implemented code when trial decryption is conducted. Note that it's very common to use a static, small RSA exponent as the public exponent e, like the one chosen in this implementation. In practice, the most common static e's are the *Fermat*

primes[1]. The reason why these numbers are convenient to use is because they make the modular exponentiation operations faster. When representing a Fermat prime in bits, there are only two 1's (the most significant bit and the least significant bit), the rest of the bits are zeros, meaning that the needed calculations are minimal. This approach however, can suffer from some powerful attacks, see [FKJM+06] and [AA07].

6.4 Decryption and Signature

The functions for decryption and signature are separate functions that can be executed after a valid distributed key has been generated. The flow chart for the implemented VIFF decryption and signature functions are included in Figure 6.6. The trial decryption when adjusting the private exponent d, decryption and signature are all very similar, and the code for normal decryption is shown in Figure 6.7.

```
def decryption(self, ciphertext):
    # since player 1's d is negative, find the inverse
    if self.runtime.id == 1:
        ciphertext = gmpy.divm(1, ciphertext, self.n_revealed)
    base = gmpy.mpz(ciphertext)

    if self.runtime.id == 1:
        power = gmpy.mpz(-self.d)
    else:
        power = gmpy.mpz(self.d)

    modulus = gmpy.mpz(self.n_revealed)
    m_i = int(pow(base, power, modulus))

    m1, m2, m3 = self.runtime.shamir_share([1, 2, 3], self.Zp, m_i)
    m_tot = m1 * m2 * m3
    open_m_tot = self.runtime.open(m_tot)

    results = gather_shares([open_m_tot])
    results.addCallback(self.check_decryption)


def check_decryption(self, results):
    message = results[0].value % self.n_revealed
    print "\nDecryption of ciphertext yields M = " + str(message)
```

Figure 6.7: VIFF code for performing a distributed RSA decryption for three players.

[1]**Fermat Primes**: Prime numbers Fx, that have the form $Fx = 2^{2^x} + 1$. The first three Fermat primes are 3, 17 and 65537, referring to $x = 0, 2, 4$ respectively ([Lim09]).

The code consists of two functions, *decryption* and *check_decryption*, where *decryption* does the computations and *check_decryption* reveals the answer. As can be seen, first each player locally calculates its m_i, which in turn is secret shared among the players, obtaining the variables $m1$, $m2$ and $m3$ with shared values. Next, an MPC is conducted yielding $m\_tot = m1 \cdot m2 \cdot m3$, before the result of m_tot is opened and revealed when the result is ready. The final answer, message M, is found by calculating the revealed value modulus N.

6.5 Code for Benchmarking

The implemented VIFF distributed RSA code also contains functions for benchmarking. Benchmarking of the key generation process is incorporated in the trial decryption at the end of the step for calculating the RSA exponents. In addition, the decryption process can also be benchmarked, which is optional. If this benchmark is activated, several decryptions of different ciphertexts are perform in a serial manner. The number of rounds for both key generation benchmark and decryption benchmark is set as described in the initialization above, and needs to be set before the protocol is executed.

6.6 Running the Program

To run the distributed RSA protocol, VIFF needs to be installed, see how to do so in Appendix B. As explained above, in the function *__init__* a list of alterable variables is found, that are the preferences for the output of the protocol, and needs to be set. Do not however alter the variables that are outlined as unalterable variables, as this may result in failure of the protocol. This guide to run the program is described for running all three players locally on one machine using SSL between the players.

Start by opening three Windows Command Prompts, then create the configuration files and the certificates files before starting the program for each player in separate windows, the procedure is as follows:

Window 1 : python generate-config-files.py -n 3 -t 1 localhost:9001 localhost:9002 localhost:9003

Window 1 : python generate-certificates.py

Window 1 : python RSA.py player-1.ini

Window 2 : python RSA.py player-2.ini

Window 3 : python RSA.py player-3.ini

The time required to finish the protocol will vary, depending on the size of the key and the number of rounds chosen, and of course the speed of the computer used. In Figure 6.8, Figure 6.9 and Figure 6.10 the output from player 1, player 2 and player 3 respectively are shown when a 128-bit distributed RSA key is generated.

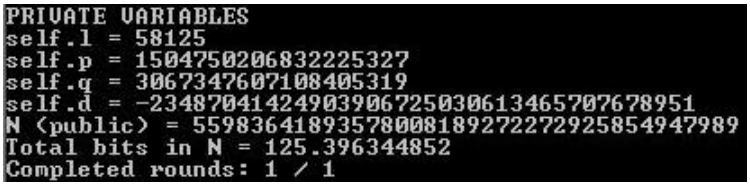

Figure 6.8: Player 1's output when generating a distributed 128-bit RSA key.

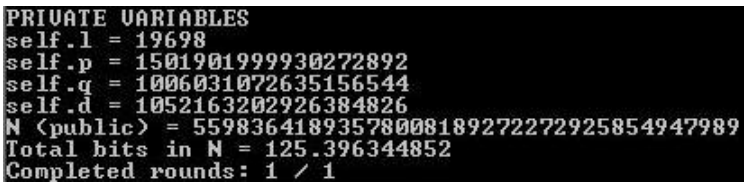

Figure 6.9: Player 2's output when generating a distributed 128-bit RSA key.

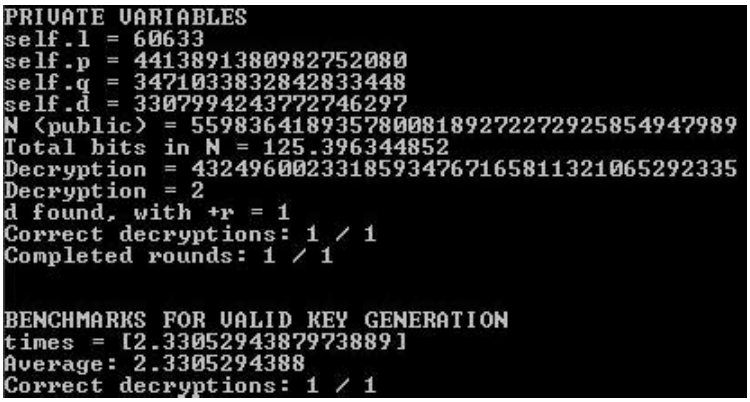

Figure 6.10: Player 3's output when generating a distributed 128-bit RSA key. Notice that player 3 is the one doing trial decryption and therefore has more output.

Even though a 128-bit key is not very hard to attack and considered totally insecure, it's just as an example, larger keys would produce multi lines output for some or all of the variables, making it harder to read.

Chapter 7

Security Analysis and Benchmarking

This chapter starts by describing two security weaknesses found in the protocol in [BF97], followed by some guidelines for required key size in RSA. Finally, the benchmark results of the implemented distributed RSA algorithm in VIFF are presented and discussed.

7.1 Security Weaknesses

Two weaknesses are found in the article [BF97], both with respect to the way a random number is used to secure the revealed answer. Both weaknesses could possibly reveal p and q and therefore also the private key $\{d, N\}$.

7.1.1 Weakness 1: Distributed Trial Division

The first weakness is found in the distributed trial division, one of the very first steps in the protocol, but can be avoided by implementing the improvement described in Section 6.3.1.

As explained in Section 5.3.1, let $q = q_1 + q_2 + q_3$ be the candidate, l be a small prime and r_i a randomly picked integer by player i from the field $\mathbb{Z}_p$. In the proposed algorithm in [BF97], the players now compute

$$qr \;=\; (\sum q_i)(\sum r_i) \; mod \; l$$

The problem here is that $\sum r_i$ is a random number from a large field $\mathbb{Z}_p$, meaning that the probability for r to be prime is approximately $1/ln(p)$[1]

[1] The **Prime Number Theorem**: The ratio of the number of primes not exceeding x

(where $ln(p)$ represents the natural logarithm of p). If r is not a prime, it must be composite, meaning that it can be factorized into prime numbers smaller than r. Any number, a, can be written as a product of prime numbers, $a = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \dots p_n^{\alpha_n}$ for prime numbers p_i and positive integers α_i. Therefore the security of q in this scheme is dependent of the size of the biggest factor of r. If the biggest factor of r is small enough, r can be successfully factorized and in turn, q can be found.

The security can however be increased by letting each player locally calculate $s_i = q_i \ mod \ l$, then jointly compute

$$qr \ = \ (\sum s_i)(\sum r_i) \ mod \ l$$

This way, the calculation of $q \ mod \ l$ is correct, and since the actual q is never included in the calculation it can therefore not be found even if the factorization of r is found. The problem however, is that the speed improvement proposed for this step in Section 6.3.1 would then not apply. On the other hand, by using the distributed trial division improvement proposed in this thesis, both the speed and the security is improved and is therefore the preferred method.

This weakness can be exposed after the protocol has accepted a pair of candidates p and q. Knowing that p and q are actually prime numbers, by the fact that the trial decryption for d was correct, a player can go back to this step and find all small factors of r. When the remainder of qr, after successfully dividing it by small prime numbers, is close to $ln_2(N)/2$ bits, a statistical prime test[2] can be used to check a range of primes around the remainder of qr, then there's a chance that the correct prime is found. The reason is that the number of bits in the remaining part of qr, after successfully dividing it by small prime numbers, will decrease with $ln_2(x)$ bits (where x is a prime factor of qr) for each successful factor found of qr, and both p and q are approximately $ln_2(N/2)$ bits. Knowing that p and q are valid candidates and knowing the public exponent N makes it easy to check if a valid prime has been found since N has exactly two factors, and guessing one of them, reveals the other one.

---

and $x/ln(x)$ approaches 1 as x grows without bound. This consequently means that the probability for a randomly picked integer x to be prime is approximately $(x/ln(x))/x = 1/ln(x)$ ([Ros03]).

[2] A statistical prime test (actually a compositeness test, since the test only outputs *probably prime*, or *not prime*) is most often a simple fast test that is performed many times to achieve a certain probability of a correct answer. The most popular prime tests are the Miller-Rabin primality test, Solovay-Strassen primality test and the Fermat's primality test, see [TW06] for detailed information about each of them.

7.1.2 Weakness 2: Alternative Step in Distributed Biprimality Test

The second weakness is found in the alternative step to step 4 in the biprimality test. The step checks the integers that fall into case 4 in the proof in [BF97], meaning whether $gcd(N, p+q-1) > 1$. If the test is true, then N is rejected and the whole protocol will have to restart by picking candidates for p and q. Each player picks a random $r_i \in \mathbb{Z}_N$ and keeps it secret. Next, the players jointly computes

$$z \;=\; \Big(\sum_{i=1}^{3} r_i\Big)\Big(-1+\sum_{i=1}^{3}(p_i+q_i)\Big)\ (mod\ N)$$

where the *mod N* part must be done after z is revealed. Because the *mod N* part needs to be done after revealing z, this step suffers from the same weakness as the weakness described for the distributed trial division. The security relies on the random number r, and more specifically, on the largest prime factor of r. If r does not consist of large enough factors, both p and q can be found by any of the players by finding the relation of $N = pq$ and part of the expression used to calculate z, namely $p + q$.

It can be seen that the expression $(-1 + p + q)$ is approximately $b = (log_2(N/2) + 1)$ bits long. Now consider if r actually consist of very many small factors, such that it's computationally feasible to find all factors of z except the factor $(-1 + p + q)$. The search for factors ends when z is divided down to approximately b bits. If such is the case, then the following relationship between N and p, q can be found:

$$\begin{aligned} N &= pq \quad \Rightarrow\ q = \tfrac{N}{p} \\ z &= r(-1+p+q) \end{aligned}$$

Rearranging variables and inserting the new expression for q yields

$$\begin{aligned} \tfrac{z}{r} &= -1+p+q \\ \tfrac{z}{r}+1 &= p+\tfrac{N}{p} \\ \tfrac{z}{r}+1-p &= \tfrac{N}{p} \\ \tfrac{zp}{r}+p-p^2 &= N \end{aligned}$$

Moving all values on the left side yields

$$-p^2+\frac{zp}{r}+p-N=0$$

This is a quadratic equation[3] where p is the only unknown value. Solving this equation yields two possible p's, where the right one is found by dividing

[3] A quadratic equation is given on the form $ax^2+bx+c=0$, where $a \neq 0$. The solution to this equation (if any) is found by calculating $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$.

N/p for both p's and see which one yields an integer as the answer, which in fact is the other factor of N.

Given that $(-1+p+q)$ is not necessarily a prime, the probability of a successful attack is higher for the trial division weakness (weakness 1), because there it is given that one of the factors are around half the length in bits. If $(-1+p+q)$ also turns out to be a composite number with many small factors, every combination of small factors is possible for the factorization of r, including those factors found for $(-1+p+q)$.

In both weaknesses, the security relies on the biggest factor of the random number r. One way of securing both the weaknesses is to pick random numbers r that are guaranteed to have a large factor or ensure that r is prime, but this is of course bothersome, and in fact the same problem that is to be solved by picking the prime numbers p and q. Another way for securing this weakness is to do the normal step in the distributed biprimality test instead (see [BF97] and [MWB99]).

7.2 RSA Key Size Recommendation

The security of RSA relies on the difficulty of factoring large prime numbers and therefore the size of a RSA key, both a standard key and a distributed key, should be large enough such that is would be computationally infeasible to factor the key and find p and q in reasonable amount of time. The size of the key should therefore take into consideration how long the key will be in use, and what it's supposed to protect.

The RSA Laboratories started a challenge in 1991 with the name *The RSA Factoring Challenge*, where RSA keys of different sizes where generated and the modulus N was published for each of them. The aim of the challenge was to be the first to find p and q given N, where finding the solution involved collection of a prize money reward (for some key sizes only). The challenge ended in 2007, and the highest factored key so far is the RSA-200 (factored in 2005), which contains 200 decimals, equal to 663 bits. For more about the challenge, see [Sec07], [Con09d] and [Con09e].

Although the factoring of a 663-bit key was conducted using 80 powerful computers and took several months to finish, keys less than 1024 bits are considered insecure and are advised not to be used in any circumstances. 1024-bit RSA keys are used in many applications today, and for many of those applications very high security is not required, typically in scenarios where a key is used only once to send some data, e.g. form data, over the Internet. A 1024-bit RSA key is not expected to be broken in the very near

future, although it's the next of the main keys sizes that will fall, since the 512-bit RSA key was broken in 1999.

For other applications that are more dependent on high security in order to maintain their reputation, such as banks, a key size of at least 2048 bits is recommended, in some cases also 3072 bits or 4096 bits, all of which are expected not to be breakable in decades to come. In the case of a bank, typically the RSA key is used to encrypt a certificate that is used to communicate securely with the bank. Such a certificate often has long operating time, ranging from several months to several year, which strengthen the recommended need of a very secure key.

7.3 Benchmarking the Implementation

Benchmarking for the implemented distributed RSA protocol in VIFF has been conducted both with three players on three distinct computers on a local area network (LAN) in addition to all players performing the protocol on one computer (using different port numbers). Both key generation and decryption have been benchmarked. The results are discussed in this section.

7.3.1 Benchmark Equipment

The benchmark equipment used is three computers which are connected via a 10 Mbit/s wired LAN. The specifications on the three computers are as follows:

- **HP Compaq DC7900**, Intel Core 2 Duo processor clocked at 3 GHz, 3.5 GB memory, Windows XP SP3.
- **Dell Optiplex GX270**, Intel Pentium 4 processor clocked at 2.6 GHz, 1 GB memory, Windows XP SP3.
- **Dell Optiplex GX270**, Intel Pentium 4 processor clocked at 2.6 GHz, 1 GB memory, Windows XP SP3.

All three computers have been used when benchmarking over LAN, while the HP Compaq DC7900 computer has been used to benchmark locally with all player on the same computer.

7.3.2 Key Generation

The key generation part measures the average time needed to generate a valid key. In this thesis the average is found by performing the key generation protocol 100 times and take the average of all rounds. The benchmarking is conducted for key sizes 32-bit to 4096-bit using SSL on all tests. The

# bits	Rounds	Avg.time		Ratio	Min (s)	Max (s)
32	100	1.75 s	0.03 min	N/A	0.30	6.54
64	100	3.08 s	0.05 min	1.76	0.48	9.67
128	100	15.20 s	0.25 min	4.94	0.77	87.17
256	100	58.28 s	0.97 min	3.83	0.67	294.77
512	100	226.55 s	3.78 min	3.89	1.04	1326.16
1024	100	1956.69 s	32.61 min	8.64	7.04	8861.80
2048	10	7252.28 s	120.87 min	3.71	9.51	20713.43
4096	1	132603.92 s	2210.07 min	18.28	-	-

Table 7.1: Benchmark for generating valid distributed RSA keys on LAN. Ratio is the current average divided by the previous average.

benchmarking of the largest keys is very time consuming, and has therefore been benchmarked less rounds (10 rounds and 1 round, respectively), which means that they are not very statistical accurate. Key sizes less than 1024 bits are generally considered insecure, and benchmarking these are purely to get an overview of the increase in time needed to generate valid keys as the keys get larger. The results from the LAN benchmark are presented in Table 7.1.

The first thing to notice from Table 7.1 is that the average time for generating a 1024-bit distributed RSA key over LAN is 32.6 minutes, ranging from 7 seconds as the fastest to 8861 seconds (∼148 minutes) as the slowest. Half an hour is quite a lot of time, and it excludes several scenarios for use of a distributed RSA key. It can also be seen that based on 10 rounds, it takes on average roughly 2 hours to create a 2048-bit distributed RSA key, and a stunning 37 hours to create a single 4096-bit distributed RSA key. Even though the last is based on one round only, it reveals that creating such a large key can be very time consuming. On the other hand, for scenarios where the distributed key is not needed instantly and is going to be used for a long while, half an hour or more does not seem to be impractical.

As for the ratio measurement, notice that a steady ratio of approximately 4 applies to the low length keys (up to 512 bits), which is also the ratio to expect when generating a distributed RSA key using the algorithm in [BF97]. Recall that the probability of a randomly picked number near N being prime is approximately $1/ln(N)$. Doubling the number of bits in N means to square the maximum value of both p and q, e.g. $max(p, q) = (2^{256})^2 = 2^{512}$, which in turn decrease the probability of picking a prime to half.

$$1/ln(2^{256}) = 1/(256 \cdot ln(2)) = 0.56\%$$

$$1/ln(2^{512}) = 1/(512 \cdot ln(2)) = 0.28\%$$

Consequently, a steady decrease factor of 2 applies for finding primes when doubling the key size. Recall also that the distributed RSA protocol picks both p and q before N is calculated and revealed, meaning that the probability of both being prime at the same time is

$$\frac{1}{ln(p)} \cdot \frac{1}{ln(q)} \approx 1/(ln(p)^2)$$

which means that the probability is decreased as a consequence of squaring (in reality a little less because distributed trial division is conducted when choosing p and q, before calculating N). The total ratio is therefore expected to be approximately the decrease factor squared, that is $2^2 = 4$.

The ratio from a 512-bit key to a 1024-bit key is 8.64, which is not according to the expected ratio. The average time used increases from an average of 3.78 minutes on the 512-bit keys to an average of 32.61 minutes on the 1024-bit keys, which indicates that 1024 is the first key that is less efficient to generate. The cause is not solely one reason, but rather several reasons is of significance. As the key sizes increases, the calculations must be performed on larger numbers, requiring more memory, more network traffic is generated (potentially exceeding the limit of maximum packet size) and a larger $\mathbb{Z}_p$ must be used to be able to represent all the shared values.

The results from the local benchmark are presented in Table 7.2. The first thing to notice is that these results are greatly improved compared to the LAN benchmarks. The reason is mainly because the network traffic can be sent locally on different ports instead of via the wired LAN. Another reason is that the computer benchmarking locally is slightly faster than the two other computers. This last point does not however contribute that much, given that this computer only have 2 cores, therefore only 2 players can do calculations at the same time, while in the LAN benchmark, all players have calculation power whenever needed.

Compared to the LAN benchmark, all key sizes takes approximately half the time to conduct locally instead of over the LAN. The 1024-bit key size is actually even better, performed in approximately 43% of the time needed over LAN. The ratios are a bit more fluctuating for this benchmark, although reasonable near a factor of 4 up to key size of 1024 bits. An interesting thing is the one 4096-bit key generated, using less than 3 hours compared to the one using 37 hours to generate over LAN, which means, as mentioned above, that such large keys can vary a lot in time consumption.

# bits	Rounds	Avg.time		Ratio	Min (s)	Max (s)
32	100	0.66 s	0.01 min	N/A	0.07	3.23
64	100	1.54 s	0.03 min	2.33	0.09	10.47
128	100	7.90 s	0.13 min	5.13	0.61	47.75
256	100	26.72 s	0.45 min	3.38	1.10	139.88
512	100	124.89 s	2.08 min	4.67	3.07	703.02
1024	100	835.48 s	13.92 min	6.69	4.94	3753.72
2048	10	6165.49 s	102.76 min	7.38	19.46	13128.69
4096	1	10431.07 s	173.85 min	1.69	-	-

Table 7.2: Benchmark for generating valid distributed RSA keys locally. Ratio is the current average divided by the previous average.

The range between minimum time and maximum time in the key generation benchmarks is quite big for all key sizes and for both LAN and locally. This is as expected because of the distribution of primes and the fact that both p and q must be prime at the same time, which increases the variance in these results.

From [BF97] and [MWB99] it can be read that 1024-bit keys are generated in approximately 90 seconds on much slower computers (clocked at 300 MHz). The reasons are many, the algorithm can be optimized in many ways, see Chapter 9 for more details. As a comparison, standard RSA protocols that are implemented efficiently typically uses milliseconds to generate a 1024-bit key on a standard desktop computer, whereas 4096-bit keys typically ranges from milliseconds to hundreds of milliseconds.

The fact that the most time-consuming step in the distributed RSA protocol is the key generation becomes clear from Table 7.3. The important thing to notice here is that the steps for key generation, up to the step for generating l, is very time consuming, and is conducted numerous times. On the other hand, once some candidates p and q have passed all the test up to the step for checking v, the rest of the steps are only conducted 1 time. This means that improvements on the run-time of the protocol should focus on the key generation step, and not so much on the step for calculating the exponents and doing decryption and signature.

7.3.3 Decryption

The benchmark results for decryption are shown in Table 7.4. Note that these results will also be valid as signature benchmarks, because basically the same code is executed.

The number of rounds for all key sizes is 20, which is enough to give a very

Function name	LAN	Local
generate_p	15049	13972
generate_q	15051	13970
trial_division_p	39997	37152
check_trial_division_p	39997	37152
trial_division_q	39999	37137
check_trial_division_q	39999	37137
check_n	6256	5812
primality_test_N	6256	5812
check_primality_test_N	6256	5812
generate_g	934	861
check_g	934	861
check_v	467	431
generate_z	1	1
check_z	1	1
generate_l	1	1
generate_d	1	1
check_decrypt	1	1

Table 7.3: The average number of times each of the functions in the implementation is run when generating a valid 1024-bit key (divided into the 4 steps for the distributed RSA protocol).

		LAN		Local	
# bits	Rounds	Avg. time	Ratio	Avg. time	Ratio
32	20	6.4 ms	N/A	3.3 ms	N/A
64	20	6.6 ms	1.03	3.4 ms	1.03
128	20	6.7 ms	1.02	3.4 ms	1.00
256	20	7.6 ms	1.15	4.1 ms	1.21
512	20	9.7 ms	1.28	5.0 ms	1.22
1024	20	20.2 ms	2.08	12.8 ms	2.56
2048	20	69.1 ms	3.42	53.2 ms	4.16
4096	20	560.7 ms	8.11	263.6 ms	4.95

Table 7.4: Benchmark results for decrypting a message once a valid key is found. Ratio is the current average divided by the previous average.

good estimate for this benchmark because the variance in each set of results is very small. As can be seen, the times are measured in milliseconds, which means that once the key is generated, both decryption and signature can be used to more or less all possible scenarios by virtually not using any time at all, even for large keys.

The ratio is increasing very slowly up to 1024 bits, using almost the same amount of time for 32-bit keys as for 512-bit keys. Again, the first leap is from 512 bits to 1024 bits, however this leap is not as big for decryption as for key generation. One reason for the lesser leap is that doing decryption and signature code is conducted one time only in any case, while for key generation the leap is affected by the accumulated value of many failed tries. The ratio leaps further to 2048 bits and 4096 bits increase even a bit more, but the overall time needed is fairly low for all key sizes. It can also be seen that the time needed to locally compute decryption and signature is about half the time needed over LAN, which is essentially the same as was found for key generation.

The results from each benchmark can be found in the electronic appendix along with a valid generated distributed 4096-bit RSA key for 3 players.

Chapter 8

Conclusions

The main goal of this thesis was to understand the basic theory of multiparty computations and implement a fully functional distributed RSA protocol using secure multiparty computations in VIFF. The theory of MPC is covered in Chapter 2 and 3, with the main focus on additive secret sharing, Shamir's secret sharing, the three stages used in every MPC, MPC adversaries and the two most basic mathematical operations used in MPC, addition and multiplication. Next, a distributed RSA protocol has been successfully implemented for three players in VIFF, which includes distributed key generation, decryption and signature, which are the important features of a distributed RSA protocol. The implemented protocol allows three players to generate and use a distributed RSA key of arbitrary size in a secure manner.

A supplementary goal of this thesis was to benchmark the solution in order to find ways to speed up the implementation. Benchmarks have shown that generating keys sufficiently large for use in common scenarios, having at least 1024 bits, varies from seconds to days, averaging from tens of minutes to several hours, which indicates that the current implementation is best suited for scenarios that allow the key to be generated in advance. The benchmark results also show that once a key is generated, both the decryption and signature process can be conducted very fast even for large key sizes and could be used to perform immediate tasks. Two run-time improvements are implemented compared to the original protocol, the first at the step for distributed trial division and the second at the step for local trial division on the revealed N. The distributed trial division improvement is the more important of the two when it comes to increasing the efficiency of the protocol because this step involves communication between the players, which require much more time than local computations.

Another supplementary goal of this thesis was to analyze the security of the protocol. Two security weaknesses was found, both of which relates to the way a random number is used to secure a revealed answer. Both weaknesses could possibly reveal the private key to any of the players. The first weakness relates to the distributed trial division step, whereas the second weakness is regarding the alternative step in the biprimality test. Methods for avoiding both the weaknesses are described, and the distributed trial division weakness is also repaired in the implemented protocol, a repair that fixes the security weakness and speeds up the protocol at the same time.

Chapter 9

Further Work

In this chapter, some suggestions for further work for the distributed RSA VIFF implementation are presented. In general, many changes can be done in order to enhance the efficiency of the implementation. These changes have not been carried out in this thesis due to the lack of time. Most of the proposed changes are inspired by [MWB99], where a lot of experimentation has been conducted. Implementing some or all of these changes will definitely make the key generation process a lot faster, and therefore making it more useful in any type of scenario.

- GMPY should be used to represent all the values in the VIFF program. Using GMPY instead of standard Python integers on all values in the program will greatly increase the efficiency of the protocol. Sigurd Meldgaard from the VIFF Developer Team estimates a 10-20% speed up in key generation with this rather simple fix alone.

- Apply *distributed sieving* to improve the distributed trial division step. The players can pick their p_i and q_i in such a way that it is guaranteed that $\sum p_i$ and $\sum q_i$ is not divisible by any prime less than a *sieving bound*. The experimentation in [MWB99] reports on a 10-fold improvement in running time for this step alone when generating a 1024-bit key.

- Test several candidates in parallel by testing several values for p and q simultaneously. The nature of MPC is not very efficient, given that the players are waiting at several synchronization points to receive shares from each other. By testing several candidates in parallel, each player normally have some calculations that can be done for at least one of the candidates, which decreases the idle time for each player, and thus improving the efficiency of the protocol.

- Perform parallel trial division on N, which is the idea of trying many primes in each division conducted. The idea is that instead of checking that N is not divisible by any prime number $[B1, B2]$ for some bounds $B1$ and $B2$, instead a more efficient check is to multiply several primes from this range, $a = p_1 \cdot p_2 \cdot ...$, and check that $\gcd(a, N) = 1$. If any of the primes divide N, then $\gcd(a, N)$ will not output 1, and the test consequently fails.

- Apply load balancing, which is the idea of balancing the calculations done for each player. Recall that the protocol at several places let a specific player do some calculation, such as the calculation of the Jacobi symbol in the distributed biprimality test, which is always conducted by player 1, or the trial decryption process which is always calculated by player 3 in the implementation. The responsibility should rotate between all players, such that player i does the calculations every k time, where k is the total number of players. Applied together with testing several candidates in parallel, makes the workload for each player very uniform.

- The step for calculating the private exponent d should be implemented for arbitrary e's, either using the method described in [BF97] or the method described in [CGH00]. This step will hardly affect the runtime for generating a valid key, but will increase the security of the protocol.

References

[AA07] P. Antonov and V. Antonova. Development of the attack against rsa with low public exponent and related messages. In *CompSysTech '07: Proceedings of the 2007 international conference on Computer systems and technologies*, pages 1–8, New York, NY, USA, 2007. ACM.

[ACS02] Joy Algesheimer, Jan Camenisch, and Victor Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In *In Advances in Cryptology - Proceedings of CRYPTO 2002*, pages 417–432. Springer-Verlag, 2002.

[BCD+08] Peter Bogetoft, Dan Lund Christensen, Ivan Damgard, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. *Cryptology ePrint Archive, Report 2008/068*, 2008.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266, New York, NY, USA, 2008. ACM.

[BF97] Dan Boneh and Matthew Franklin. Efficient generation of shared rsa keys. In *Advances in Cryptology – CRYPTO 97*, pages 425–439. Springer-Verlag, 1997.

[BL90] Josh Cohen Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In *CRYPTO '88: Proceedings of the 8th Annual International Cryptology Conference*

on Advances in Cryptology, pages 27–35, London, UK, 1990. Springer-Verlag.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1988. ACM Press.

[Bri90] Ernest F. Brickell. Some ideal secret sharing schemes. In *EUROCRYPT '89: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 468–475, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[Can95] Ran Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Weizmann Institute of Science, Department of Computer Science and Applied Mathematics, 1995.

[Can01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS '01: Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, page 136, Washington, DC, USA, 2001. IEEE Computer Society.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, New York, NY, USA, 1988. ACM.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli Maurer. General secure multi-party computation from any linear secret sharing scheme. Cryptology ePrint Archive, Report 2000/037, 2000. `http://eprint.iacr.org/`.

[CGH00] Dario Catalano, Rosario Gennaro, and Shai Halevi. Computing inverses over a shared secret modulus. In *EUROCRYPT*, pages 190–206, 2000.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[Con09a] Wikipedia Contributors. Blum integer. `http://en.wikipedia.org/wiki/Blum_integer`, 2009.

[Con09b] Wikipedia Contributors. Jacobi symbol. `http://en.wikipedia.org/wiki/Jacobi_symbol`, 2009.

[Con09c] Wikipedia Contributors. Legendre symbol. `http://en.wikipedia.org/wiki/Legendre_symbol`, 2009.

[Con09d] Wikipedia Contributors. Rsa factoring challenge. `http://en.wikipedia.org/wiki/RSA_Factoring_Challenge`, 2009.

[Con09e] Wikipedia Contributors. Rsa numbers. `http://en.wikipedia.org/wiki/RSA_numbers`, 2009.

[Con09f] Wikipedia Contributors. Safe prime. `http://en.wikipedia.org/wiki/Safe_prime`, 2009.

[Con09g] Wikipedia Contributors. Secret sharing. `http://en.wikipedia.org/wiki/Secret_sharing`, 2009.

[CrN08] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation, an introduction. `http://www.daimi.au.dk/~ivan/mpc.pdf`, 2008.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[EG85] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[EP87] C. H. Edwards and David E. Penney. *Elementary Linear Algebra*. Prentice Hall, 1987.

[FKJM+06] Pierre-Alain Fouque, Sébastien Kunz-Jacques, Gwenaëlle Martinet, Frédéric Muller, and Frédéric Valette. Power attack on small rsa public exponent. In *CHES*, pages 339–353, 2006.

[Fra03] John B. Fraleigh. *A First Course In Abstract Algebra, Seventh Edition*. Addison Wesley, 2003.

[Gei08a] Martin Geisler. Implementing asynchronous multiparty computation. `http://viff.dk/files/mg-progress-report-talk.pdf`, February 21st, 2008.

[Gei08b] Martin Geisler. Implementing asynchronous multi-party computation, phd progress report. `http://viff.dk/files/mg-progress-report.pdf`, January 2008.

[Gei08c] Martin Geisler. Multiparty computation made practical using the virtual ideal functionality framework. In *ECRYPT Research Meeting*, June 23 2008.

[Gei09] Martin Geisler. Mpc virtual machine specification. `http://www.cace-project.eu/downloads/deliverables-y1/CACE_D4.3_MPC_Virtual_Machine_Specification.pdf`, 2009.

[GF02] Laurence Grant and Brian Fleming. *Secret Sharing and Splitting*. University of Notre Dame, Indiana, USA, 2002.

[GMP03] GMPY. General multiprecision python. `http://gmpy.sourceforge.net/`, 2003.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229, New York, NY, USA, 1987. ACM.

[Gol97] Shafi Goldwasser. Multi party computations: past and present. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 1–6, New York, NY, USA, 1997. ACM.

[Gol99] Oded Goldreich. Preface to special issue on general secure multi-party computation. `http://citeseerx.ist.psu.edu/`, 1999.

[Gol00] Oded Goldreich. Secure multi-party computation. Working Draft, 2000.

[Hir01] Martin Hirt. *Multi-Party Computation: Efficient Protocols, General Adversaries, and Voting*. PhD thesis, ETH Zurich, 2001.

[Kre99] Erwin Kreyszig. *Advanced Engineering Mathematics, 8th Edition*. John Wiley & Sons, Inc., 1999.

[Lef09] Glyph Lefkowitz. Twisted matrix labs. `http://twistedmatrix.com/trac/`, 2009.

[Lim09] DI Management Services Pty Limited. Rsa algorithm. `http://www.di-mgt.com.au/rsa_alg.html`, 2009.

[Liu68] C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.

[Mor07] Michael Mortensen. Secret sharing and secure multi-party computation. Master's thesis, Department of Informatics, University of Bergen, 2007.

[MWB99] Michael Malkin, Thomas Wu, and Dan Boneh. Experimenting with shared generation of rsa keys. In *In Proceedings of the Internet SocietyŠs 1999 Symposium on Network and Distributed System Security (SNDSS)*, pages 43–56, 1999.

[rGiN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[Ros03] Kenneth H. Rosen. *Discrete Mathematics and Its Applications, Fifth Edition*. McGraw-Hill, 2003.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[Sch96] Bruce Schneier. *Applied Cryptography - Protocols, Algorithms, and Source Code in C, Second Edition*. John Wiley & Sons, Inc., 1996.

[Sec07] RSA Security. The rsa factoring challenge. `http://www.rsa.com/rsalabs/node.asp?id=2092`, 2007.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[Sim92] Gustavus J. Simmons. *Contemporary Cryptology: The Science of Information Integrity*. IEEE Press, 1992.

[Sta06] William Stallings. *Cryptography and Network Security - Principles and Practices, Fourth Edition*. Pearson Prentice Hall, 2006.

[Tea09] VIFF Development Team. Viff, the virtual ideal functionality framework. `http://viff.dk/`, 2009.

[Tur66] L. Richard Turner. *Inverse of the Vandermonde Matrix With Applications*. Lewis Research Center, NASA, Cleveland, Ohio, 1966.

[TW06] Wade Trappe and Lawrence Washington. *Introduction to Cryptography with Coding Theory, Second Edition*. Pearson Prentice Hall, 2006.

[vR08] Guido van Rossum. Python. `http://www.python.org/`, December 23, 2008.

[WM05] Michael E. Whitman and Herbert J. Mattord. *Principles of Information Security, Second Edition*. Thomson Course Technology, 2005.

[XKC09] XKCD. Security. `http://xkcd.com/538/`, 2009.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

Appendix A

Electronic Appendix

A compressed zip-file is attached to this thesis and contains the following:

- The distributed RSA code implemented in VIFF for three players.
- The references used for this thesis (articles only).
- The benchmark results for both key generation and decryption. The results are divided into two folders, *LAN* and *Local*, each having results for all key sizes, 32 bits to 4096 bits.
- A valid 4096-bit distributed RSA key for three players as a proof of concept of the implemented protocol in VIFF.

Appendix B

Install VIFF

This chapter will describe step by step how to install VIFF on a Windows XP machine. An installation guide can also be found by choosing your preferred operating system from `http://viff.dk/doc/index.html` (although some steps are missing at the time).

B.1 Download and Install all the Necessary Files

A number of programs and modules need to be downloaded and installed in order to successfully run VIFF programs. The steps below needs to be done in this particular order:

- From the web page `http://python.org/download/` download and install Python version 2.5.4 for Windows (python-2.5.4.msi)
- Update the environment variable Path (see below).
- Download and install Twisted for Python 2.5, `http://twistedmatrix.com/trac/`.
- Download and install GMPY from `http://code.google.com/p/gmpy/` (press *Show all* to find GMPY for python 2.5).
- Download and install Win32OpenSSL for Windows (newest version) at `http://www.slproweb.com/products/Win32OpenSSL.html`. If installation requires *Visual C++ 2008 Redistributables*, it can be found at the same web page, and have to be installed before Win32OpenSSL.
- Download and install PyOpenSSL for Python 2.5, found at `http://pyopenssl.sourceforge.net/`.

- Download and install Python for Windows extensions for Python 2.5, found at `http://sourceforge.net/project/showfiles.php?group_id=78018&package_id=79063`.

- Download and install VIFF executable file from `http://viff.dk/#releases`.

- Download and unpack the VIFF zip file from the same web page, copy the *apps* folder into your *viff* folder.

The Windows PATH environment variable needs to be updated in order to be able to execute Python code outside the Python folder itself. Follow these steps to update the PATH environment variable in Windows XP.

Right-click on My Computer on the desktop and choose Properties from the menu (Figure B.1).

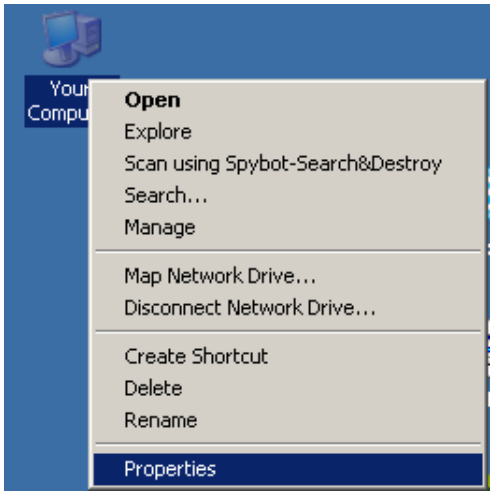

Figure B.1: First step to update Windows XP's environment variable: Go to the computers properties.

From there go to the Advanced tab and press the Environment Variables button (Figure B.2).

Next, choose Path in the System Variables view and press the Edit button (Figure B.3).

Lastly, input your Python install folder in the Variable Value text field, remember to separate with ; from the last entry in the text field (Figure B.4).

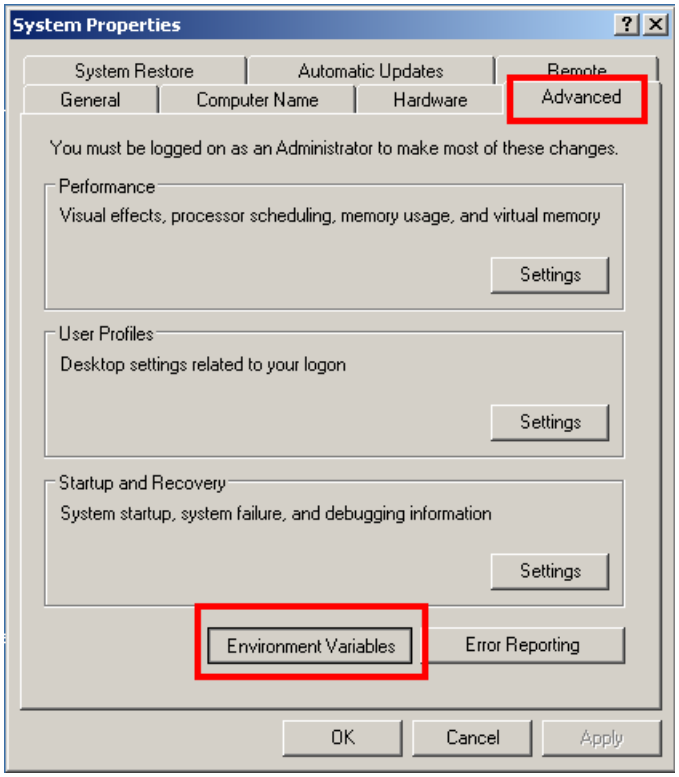

Figure B.2: Second step to update Windows XP's environment variable: Go to the Environment Variables.

B.2 Run Test Application

To test if the installation is working, try to run the millionaire example included in /viff/apps/ as follows:

- Start three Windows Command Prompts by pressing Start menu $--> $ Run... and write *cmd*.
- Browse to your /apps/ folder, found in /Python25/lib/site-packages/Viff/apps/
- In the first window, execute the following command: *python generate-config-files.py -n 3 -t 1 localhost:9001 localhost:9002 localhost:9003*. The configuration files for three players are now created with a random seed value.
- In the first window, execute the following command: *python millionaires.py –no-ssl player-3.ini*.
- In the second window, execute the following command: *python millionaires.py –no-ssl player-2.ini*.

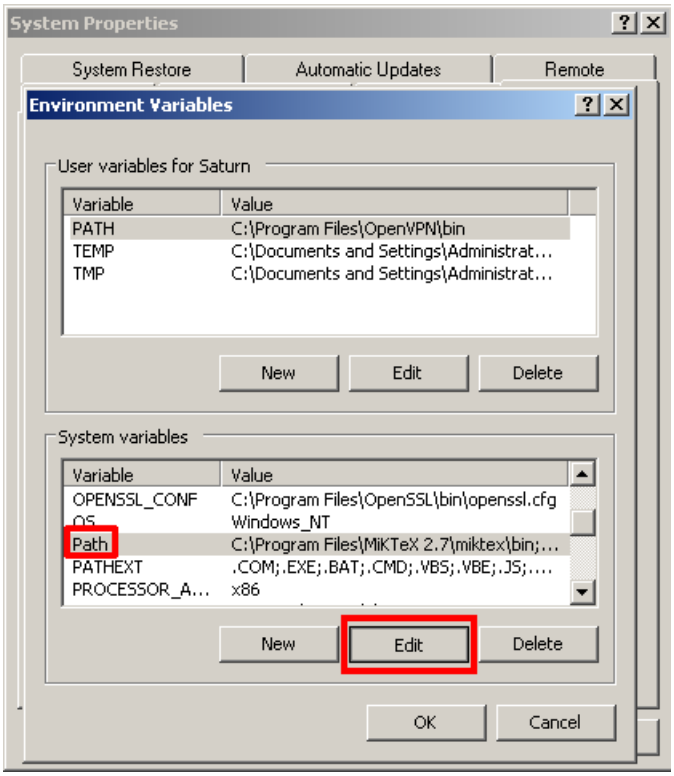

Figure B.3: Third step to update Windows XP's environment variable: Open the System Variable Path.

- In the last window, execute the following command: *python millionaires.py –no-ssl player-1.ini*.

You should now get the correct ranking of the three millionaires, but each window should only reveal their own amount of money (Figure B.5, Figure B.6 and Figure B.7).

The option of running protocols with SSL is also an option. This will require running the following command in any of the windows after running the *generate-config-files.py* command: *generate-certificates.py*. This will automatically create certificates for three players.

In order to run the program on distinct computers, and not all players locally on one computer, both the configuration files and the certificates (if used) needs to be distributed to the other computers, and the Internet Protocol (IP) addresses of all the computers must be types in when running the configuration files command. So instead of writing the addresses as localhost:port, the command would be IPaddress:port.

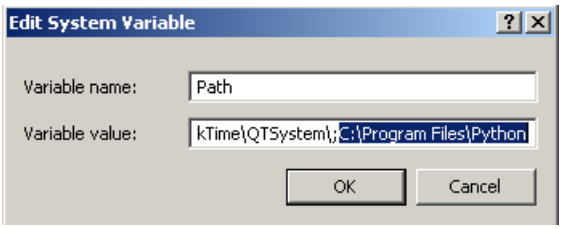

Figure B.4: Fourth step to update Windows XP's environment variable: Append a path for the System Variable.

```
I am Millionaire 1 and I am worth 40 millions.
From poorest to richest:
  Millionaire 1 (40 millions)
  Millionaire 3
  Millionaire 2
```

Figure B.5: Player 1's output when the test application finishes.

```
I am Millionaire 2 and I am worth 158 millions.
From poorest to richest:
  Millionaire 1
  Millionaire 3
  Millionaire 2 (158 millions)
```

Figure B.6: Player 2's output when the test application finishes.

```
I am Millionaire 3 and I am worth 134 millions.
From poorest to richest:
  Millionaire 1
  Millionaire 3 (134 millions)
  Millionaire 2
```

Figure B.7: Player 3's output when the test application finishes.

Appendix C

Mathematics

C.1 Linear System Approach

Continuing from Section 3.6.1 the players can solve a linear system of equations. Each player can establish three equations using the formula as shown in Equation (C.1).

$$fg(i,j) = s_{i,j} = s_h + r_1 j + r_2 j^2 \tag{C.1}$$

In Equation (C.1) i refers to the player holding the share and j refers to the player that created the share. Player 1 can do the following calculations:

$$\begin{aligned} fg(1,1) &= s_{1,1} = 5 \\ fg(1,2) &= s_{1,2} = -1 \\ fg(1,3) &= s_{1,3} = -14 \end{aligned}$$

Organizing these values into a matrix yields:

$$\begin{bmatrix} s_h & r_1 & r_2 & 5 \\ s_h & 2r_1 & 4r_2 & -1 \\ s_h & 3r_1 & 9r_2 & -14 \end{bmatrix}$$

Player 1 wants to solve the equations with respect to s_h, which is player 1's share of the total polynomial. Solving the linear system can be done using Gaussian elimination [EP87] as shown below:

$$\begin{bmatrix} 1 & 1 & 1 & 5 \\ 1 & 2 & 4 & -1 \\ 1 & 3 & 9 & -14 \end{bmatrix} \begin{matrix} R_2 - 1 \cdot R_1 \\ \Longrightarrow \\ R_3 - 1 \cdot R_1 \end{matrix} \begin{bmatrix} 1 & 1 & 1 & 5 \\ 0 & 1 & 3 & -6 \\ 0 & 2 & 8 & -19 \end{bmatrix} \begin{matrix} R_3 - 2 \cdot R_2 \\ \Longrightarrow \end{matrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 5 \\ 0 & 1 & 3 & -6 \\ 0 & 0 & 2 & -7 \end{bmatrix} \begin{matrix} \frac{1}{2} \cdot R_3 \\ \Longrightarrow \end{matrix} \begin{bmatrix} 1 & 1 & 1 & 5 \\ 0 & 1 & 3 & -6 \\ 0 & 0 & 1 & -3.5 \end{bmatrix}$$

Going backwards from row three, all the unknown variables can be calculated:

$$
\begin{aligned}
r_2 &= -3.5 \\
r_1 &= -6 - 3 \cdot r_2 = -6 - 3 \cdot (-3.5) = 4.5 \\
s_h &= 5 - r_1 - r_2 = 5 - 4.5 - (-3.5) = 4
\end{aligned}
$$

The value $s_h = 4$ can be located in Table 3.4. Player 2 and player 3 have to use their values in order to calculate their value of s_h.

C.2 Vandermonde Matrix

Continuing from Section 3.6.1 the players do not need to solve the linear system. By using the inverse of a Vandermonde matrix each player can obtain its share of the total polynomial by means of a less complex calculation. The Vandermonde matrix is defined as [Tur66]:

$$
V = \begin{bmatrix}
1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\
1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\
1 & x_3 & x_3^2 & \cdots & x_3^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_n & x_n^2 & \cdots & x_n^{n-1}
\end{bmatrix}
$$

V is the Vandermonde matrix and I is the identity matrix, both of size 3x3. For three players where $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$ the two matrices are defined as follows:

$$
V = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{bmatrix} \qquad I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
$$

Gauss-Jordan elimination is used to transform $[V|I]$ into $[I|V^{-1}]$.

$$
\left[\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 2 & 4 & 0 & 1 & 0 \\ 1 & 3 & 9 & 0 & 0 & 1 \end{array}\right]
\begin{array}{c} R_2 - 1 \cdot R_1 \\ \Longrightarrow \\ R_3 - 1 \cdot R_1 \end{array}
\left[\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 3 & -1 & 1 & 0 \\ 0 & 2 & 8 & -1 & 0 & 1 \end{array}\right]
\begin{array}{c} R_3 - 2 \cdot R_2 \\ \Longrightarrow \end{array}
$$

$$
\left[\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 3 & -1 & 1 & 0 \\ 0 & 0 & 2 & 1 & -2 & 1 \end{array}\right]
\begin{array}{c} \frac{1}{2} \cdot R_3 \\ \Longrightarrow \end{array}
\left[\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 3 & -1 & 1 & 0 \\ 0 & 0 & 1 & \frac{1}{2} & -1 & \frac{1}{2} \end{array}\right]
\begin{array}{c} R_2 - 3 \cdot R_3 \\ \Longrightarrow \end{array}
$$

$$
\left[\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & -\frac{5}{2} & 4 & -\frac{3}{2} \\ 0 & 0 & 1 & \frac{1}{2} & -1 & \frac{1}{2} \end{array}\right]
\begin{array}{c} R_1 - 1 \cdot R_2 \\ \Longrightarrow \\ R_1 - 1 \cdot R_3 \end{array}
\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 3 & -3 & 1 \\ 0 & 1 & 0 & -\frac{5}{2} & 4 & \frac{3}{2} \\ 0 & 0 & 1 & \frac{1}{2} & -1 & \frac{1}{2} \end{array}\right]
$$

The tuple $(3, -3, 1)$ in the first row of the inverse Vandermonde matrix will always contain these values when three players are participating and they use the indexes 1, 2 and 3. This gives an advantage for solving the linear systems since no computation on solving the unknown variables needs to be done.

From Table 3.3 player 1 has received the tuple of share values $(5, -1, -14)$ from player 1, 2 and 3, respectively. In order for player 1 to find its share on the total polynomial, the matrix multiplication of the two tuples is calculated:

$$\begin{bmatrix} 3 & -3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ -1 \\ -14 \end{bmatrix} = \begin{bmatrix} 15 + 3 - 14 \end{bmatrix} = \begin{bmatrix} 4 \end{bmatrix}$$

The value 4 can be located in Table 3.4. Player 2 and player 3 will have to calculate the Vandermonde tuple $(3, -3, 1)$ with their own tuple from Table 3.3 in order to find their shares on the total polynomial.

Appendix D

VIFF Distributed RSA Code

```
1  #!/usr/bin/python
3  # Copyright 2007, 2008 VIFF Development Team.
   #
5  # This file is part of VIFF, the Virtual Ideal Functionality
       Framework.
   #
7  # VIFF is free software: you can redistribute it and/or modify
       it
   # under the terms of the GNU Lesser General Public License (LGPL
       ) as
9  # published by the Free Software Foundation, either version 3 of
       the
   # License, or (at your option) any later version.
11 #
   # VIFF is distributed in the hope that it will be useful, but
       WITHOUT
13 # ANY WARRANTY; without even the implied warranty of
       MERCHANTABILITY
   # or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
       General
15 # Public License for more details.
   #
17 # You should have received a copy of the GNU Lesser General
       Public
   # License along with VIFF. If not, see <http://www.gnu.org/
       licenses/>.
19
   # This code can be used to generate shared RSA keys of any
       desired
21 # length. The implementation is based on the algorithm described
   # in "Efficient Generation of Shared RSA keys" written by
23 # Dan Boneh and Matthew Franklin in 1997.
   #
25 # Some adjustments have been made, the first one found in the
       step
```

```
# "Trial division", which is specially implemented for 3 players,
# although it can be be extended to arbitrary number of players.
# The second change is that the trial division for N is for a larger
# span than used in the article and also that each player checks
# different spans instead of all players check the same ones.
#
# Give a player configuration file as a command line argument or run
# the example with '--help' for help with the command line options.

# import the necessary modules
import random
import math
import gmpy
import time

from optparse import OptionParser
from twisted.internet import reactor

from viff.field import GF
from viff.runtime import Runtime, create_runtime, gather_shares, make_runtime_class, Share
from viff.comparison import ComparisonToft07Mixin, Toft05Runtime
from viff.config import load_config
from viff.util import rand, find_prime
from viff.equality import ProbabilisticEqualityMixin

# We start by defining the protocol, it will be started at the bottom
# of the file.

class Protocol:

    # returns the list of primes larger than min and less or equal to max
    def get_primes(self, min, max):
        primes = []
        while True:
            prime = int(gmpy.next_prime(min))
            if prime <= max:
                primes += [prime]
                min = prime
            else:
                return primes


    # the function for generating a private part of p for each player
    def generate_p(self):
        self.function_count[0] += 1
```

```
71          # player 1 needs to obtain its share of p as congruent
                to 3 mod 4
            if self.runtime.id == 1:
73              self.p = 4*random.randint(1, self.numeric_length -
                    1) + 3
            # every other player needs to obtain its share of p as
                congruent to 0 mod 4
75          else:
                self.p = 4*random.randint(1, self.numeric_length -
                    1)
77
            #print "my p = " + str(self.p)
79          self.trial_division_p()

81
        # the function for generating a private part of q for each
            player, equal to the corresponding function for p
83      def generate_q(self):
            self.function_count[1] += 1
85          if self.runtime.id == 1:
                self.q = 4*random.randint(1, self.numeric_length -
                    1) + 3
87          else:
                self.q = 4*random.randint(1, self.numeric_length -
                    1)
89
            #print "my q = " + str(self.q)
91          self.trial_division_q()

93
        # function for doing shared trial division for small primes
            on the choosen p
95      # alternative step to the step described in the article,
            with this solution nothing is revealed
        # check if p is composite for small primes (done secret
            shared)
97      # each player choose a random number from Zp and this number
             along with its private p (mod the current prime number
            to be tested)
        def trial_division_p(self):
99          self.function_count[2] += 1
            # the function is done iterative, therefore the next
                prime to be checked needs to be choosen
101         prime_num = self.prime_list_b1[self.prime_pointer]
            # calculate the remainder of self.p modulus the current
                prime number in the list
103         p_trial = self.p % prime_num
            #print "my p_trial = " + str(p_trial) + " for prime_num
                = " + str(prime_num)
105         r_trial = random.randint(1, self.Zp.modulus - 1)
            #print "my random r_trial = " + str(r_trial)
107
            # share the values
```

```
109         p_trial1, p_trial2, p_trial3 = self.runtime.shamir_share
                ([1, 2, 3], self.Zp, p_trial)
            p_r_trial1, p_r_trial2, p_r_trial3 = self.runtime.
                shamir_share([1, 2, 3], self.Zp, r_trial)
111
            # calculate the needed values
113         p_trial_tot = (p_trial1 + p_trial2 + p_trial3)
            r_trial_tot = (p_r_trial1 + p_r_trial2 + p_r_trial3)
115         # the value to reveal, p_trial_tot is the sum of each
                players' private p, r_trial_tot is the sum of a
                random number from each player and prime_num is the
                current prime number to check
            trial_reveal = p_trial_tot * (p_trial_tot - prime_num) *
                 (p_trial_tot - 2 * prime_num) * r_trial_tot
117
            # open the value of the open_trial_reveal share
119         open_trial_reveal = self.runtime.open(trial_reveal)
            results = gather_shares([open_trial_reveal])
121         # addCallback lets the program wait for the results to
                be ready, then call the function given as the
                argument
            results.addCallback(self.check_trial_division_p)
123

125     # reveal-function that are called from trial_division_p()
            when the results are ready
        # from the equation in trial_division_p() trial_reveal = p(p
             - prime)(p - 2*prime)*r, if prime divides p, then surely
             this expression will be zero for 3 players
127     # if prime does NOT divide p, then the result re_trial will
            be nothing but a random number, and reveals no
            information about the players' private p
        def check_trial_division_p(self, results):
129         self.function_count[3] += 1
            rev_trial = results[0].value
131         #print "rev_trial = " + str(rev_trial)

133         # if prime divides p, generate a new p and start over
            if rev_trial == 0:
135             self.prime_pointer = 0
                #print "generating p again"
137             self.generate_p()
            # if not, check if more primes are to be tested, if yes,
                 go back to trial_division_p(), if no, generate q
139         else:
                self.prime_pointer += 1
141             # if all the primes in the prime_list_b1 is tested,
                    generate q
                if self.prime_pointer >= len(self.prime_list_b1):
143                 self.prime_pointer = 0
                    self.generate_q()
145             # else, check for next prime in the list
                else:
147                 self.trial_division_p()
```

```
149
    # this function is equal to the corresponding function for p
151 def trial_division_q(self):
        self.function_count[4] += 1
153     prime_num = self.prime_list_b1[self.prime_pointer]
        q_trial = self.q % prime_num
155     #print "my q_trial = " + str(q_trial) + " for prime_num
            = " + str(prime_num)
        r_trial = random.randint(1, self.Zp.modulus - 1)
157     #print "my random r_trial = " + str(r_trial)

159     q_trial1, q_trial2, q_trial3 = self.runtime.shamir_share
            ([1, 2, 3], self.Zp, q_trial)
        q_r_trial1, q_r_trial2, q_r_trial3 = self.runtime.
            shamir_share([1, 2, 3], self.Zp, r_trial)
161
        q_trial_tot = (q_trial1 + q_trial2 + q_trial3)
163     r_trial_tot = (q_r_trial1 + q_r_trial2 + q_r_trial3)
        trial_reveal = q_trial_tot * (q_trial_tot - prime_num) *
             (q_trial_tot - 2 * prime_num) * r_trial_tot
165
        open_trial_reveal = self.runtime.open(trial_reveal)
167     results = gather_shares([open_trial_reveal])
        results.addCallback(self.check_trial_division_q)
169

171 # this function is equal to the corresponding function for p
         until a q is accepted so far
    def check_trial_division_q(self, results):
173     self.function_count[5] += 1
        rev_trial = results[0].value
175     #print "rev_trial = " + str(rev_trial)

177     if rev_trial == 0:
            self.prime_pointer = 0
179         #print "generating q again"
            self.generate_q()
181     else:
            self.prime_pointer += 1
183         # if all the primes in the prime_list_b1 is tested,
                reveal n
            if self.prime_pointer >= len(self.prime_list_b1):
185             self.prime_pointer = 0

187             p1, p2, p3 = self.runtime.shamir_share([1, 2,
                    3], self.Zp, self.p)
                # calculate the total p as a share
189             self.ptot = (p1 + p2 + p3)

191             q1, q2, q3 = self.runtime.shamir_share([1, 2,
                    3], self.Zp, self.q)
                # calculate the total q as a share
193             self.qtot = (q1+ q2 + q3)
```

```
195                 # calculate and open the RSA-modulus N
                    n = self.ptot * self.qtot
197                 open_n = self.runtime.open(n)

199                 # FOR DEBUGGING ONLY
                    #open_ptot = self.runtime.open(self.ptot)
201                 #open_qtot = self.runtime.open(self.qtot)
                    # END DEBUGGING ONLY
203
                    results = gather_shares([open_n]) #, open_ptot,
                        open_qtot]) # LAST TWO FOR DEBUGGING ONLY
205                 results.addCallback(self.check_n)
                # else, check for next prime in the list
207             else:
                    self.trial_division_q()
209


211
        # function to save the revealed N and the shared value of
            phi, plus do useful debugging printouts
213     def check_n(self, results):
            self.function_count[6] += 1
215         #print "n = " + str(results[0])

217         self.n_revealed = results[0].value
            self.phi = (self.ptot - 1) * (self.qtot - 1)
219         #print "completed rounds: " + str(self.completed_rounds)
                + " / " + str(self.rounds)
            #print "\nn_revealed = " + str(self.n_revealed)
221
            # FOR DEBUGGING ONLY
223         #print "p_revealed = " + str(results[1].value)
            #print "q_revealed = " + str(results[2].value)
225         # END DEBUGGING ONLY

227         #print "#bits in N = " + str(math.ceil(math.log(self.
                n_revealed, 2)))

229         self.primality_test_N()

231     # function for more primality testing on p and q
        # the primality testing for N can be done very quickly
            locally for each player since N is a revealed value
233     # each player checks N for different intervals (in
            prime_list_b2) for program speed up
        def primality_test_N(self):
235         self.function_count[7] += 1
            # assume that the primality test will not fail
237         test_failed = 0
            for i in self.prime_list_b2:
239             #print "N mod " + str(i) + " = " + str(self.
                    n_revealed % i)
```

```
            # if the current prime in the list divides N, this
                means that N has a factor equal to this prime,
                since this factor is small (in comparison to the
                value of p, q and N), this means that N is not
                the product of two large primes p and q
241         if self.n_revealed % i == 0:
                #print "failed... " + str(i) + " divides " + str
                    (self.n_revealed)
243             test_failed = 1
                break
245
        # share the values
247     failed1, failed2, failed3 = self.runtime.shamir_share
            ([1, 2, 3], self.Zp, test_failed)

249     # calculate and open the sum of failed values
        failed_tot = failed1 + failed2 + failed3
251     open_failed_tot = self.runtime.open(failed_tot)

253     results = gather_shares([open_failed_tot])
        results.addCallback(self.check_primality_test_N)
255

257 # function for checking the primality test for N
    def check_primality_test_N(self, results):
259     self.function_count[8] += 1
        # if each player has checked through its whole list of
            primes, but none divides N, p and q are so far
            accepted
261     if results[0].value == 0:
            #print "primality test for N is OK, generate g"
263         self.generate_g()
        # if the results are not 0, then or or more of the
            players have discovered a factor for N that is not p
            or q, start the whole process from start with
            generating p
265     else:
            #print "primality test for N failed, start
                generating p"
267         self.generate_p()

269
    # function for agreeing on a random chosen g
271 def generate_g(self):
        self.function_count[9] += 1
273     # player 1 chooses a random number in the interval [1, N
            -1] and shares it with the other players
        if self.runtime.id == 1:
275         self.g = random.randint(1, self.n_revealed - 1)
            #print "g = " + str(self.g)
277         self.g = self.runtime.shamir_share([1], self.Zp,
                self.g)
        else:
```

```
279             # no input to the shamir share means that this
                    player has no value to share, but gets a value of
                     what is shared (by player 1)
                self.g = self.runtime.shamir_share([1], self.Zp)
281
            self.open_g = self.runtime.open(self.g)
283         results = gather_shares([self.open_g])
            results.addCallback(self.check_g)
285

287     # function for distributed biprimality test, check that the
            jacobi symbol of g is equal to 1, if yes, calculate v
        def check_g(self, results):
289         self.function_count[10] += 1
            #print "g = " + str(results[0].value)
291         self.g = results[0].value
            # calculate the jacobi symbol of (g/N)
293         jacobi = gmpy.jacobi(self.g, self.n_revealed) % self.
                n_revealed
            #print "jacobi = " + str(jacobi)
295         # if the jacobi value is equal to 1, then calculate v
            if jacobi == 1:
297             # calculate the v's
                if self.runtime.id == 1:
299                 # calculate player 1's private part of phi (N -
                        p1 - q1 + 1)
                    self.phi_i = self.n_revealed - self.p - self.q +
                         1
301                 #self.v = self.g**((self.n_revealed - self.p -
                        self.q + 1) / 4) % self.n_revealed
                    base = gmpy.mpz(self.g)
303                 power = gmpy.mpz(self.phi_i / 4)
                    modulus = gmpy.mpz(self.n_revealed)
305                 self.v = int(pow(base, power, modulus))
                    #self.v = self.powermod(self.g, (self.n_revealed
                         - self.p - self.q + 1) / 4, self.n_revealed)
307             else:
                    # calculate every other players' private part of
                         phi -(pi + qi) for player i
309                 self.phi_i = -(self.p + self.q)
                    # the function gmpy.divm(1, a, b) calculates the
                         inverse of a mod b
311                 self.inverse_v = int(gmpy.divm(1, self.g, self.
                        n_revealed))

313                 base = gmpy.mpz(self.inverse_v)
                    power = gmpy.mpz(-self.phi_i / 4)
315                 modulus = gmpy.mpz(self.n_revealed)
                    self.v = int(pow(base, power, modulus))
317
                #print "self.phi_i = " + str(self.phi_i)
319         # if the jacobi value is not 1, then choose generate a
                new g
            else:
```

```
321             self.generate_g()
                return
323
            #print "self.v = " + str(self.v)
325
            # share the v's (already mod N)
327         v1, v2, v3 = self.runtime.shamir_share([1, 2, 3], self.
                Zp, self.v)

329         # calculate the total v
            v_tot = v1 * v2 * v3
331         self.open_v = self.runtime.open(v_tot)
            results = gather_shares([self.open_v])
333         #print "GIKK GREIT MED GATHER SHARES"
            results.addCallback(self.check_v)
335

337     # function for checking for a valid v
        def check_v(self, results):
339         self.function_count[11] += 1
            # the resulting v is also calculated mod N
341         v = results[0].value % self.n_revealed
            #print "v = " + str(v)
343
            # if v is equal to 1/-1 mod N, go to the next step,
                generating z
345         if v == 1 or v == self.n_revealed - 1:
                self.generate_z()
347         # else, the distributed biprimality test failed, start
                all over with generating p
            else:
349             self.prime_pointer = 0
                self.generate_p()
351

353     # function for the 4th step in the distributed biprimality
            test --> the alternative step described
        def generate_z(self):
355         self.function_count[12] += 1
            # each player generate a random number
357         self.r_z = random.randint(1, self.n_revealed - 1)
            # the random numbers are shared
359         r1, r2, r3 = self.runtime.shamir_share([1, 2, 3], self.
                Zp, self.r_z)
            z = (r1 + r2 + r3) * (-1 + (self.ptot + self.qtot))
361
            self.open_z = self.runtime.open(z)
363         results = gather_shares([self.open_z])
            results.addCallback(self.check_z)
365

367     # function for checking that gcd(z, N) is equal to 1
        def check_z(self, results):
369         self.function_count[13] += 1
```

```
        z = results[0].value % self.n_revealed
371     #print "z = " + str(z)

373     # calculate the gcd of z and N
        z_n = gmpy.gcd(z, self.n_revealed)
375     # if the gcd is equal to 1, then the distributed
            biprimality test is passed
        if z_n == 1:
377         #print "gcd(z, N) = 1, start generating e,d"
            # choosing the RSA public exponent e, a prime close
                to a power of two is often chosen, 2^16 + 1 =
                65537 is very often used
379         self.e = 2**16 + 1
            #self.e = 17
381         #print "e = " + str(self.e)
            self.generate_l()
383         #self.generate_psi()

385     # else the distributed biprimality test has failed, and
            the whole protocol is started again by generating new
             p and q's
        else:
387         #print "gcd(z, N) != 1, restart with generating p"
            self.prime_pointer = 0
389         self.generate_p()

391
    # function for generating l, used to finding the private
        exponent d
393 # by arriving at this function p and q are found to be
        primes, and only a shared d is needed
    def generate_l(self):
395     self.function_count[14] += 1
        # every player calculates his/her private phi_i mod e (
            public exponent)
397     self.l = self.phi_i % self.e
        print "\n\nPRIVATE VARIABLES"
399     print "self.l = " + str(self.l)
        # share the l's and calculate the total l
401     l1, l2, l3 = self.runtime.shamir_share([1, 2, 3], self.
            Zp, self.l)
        l_tot = l1 + l2 + l3
403
        open_l_tot = self.runtime.open(l_tot)
405     results = gather_shares([open_l_tot])
        results.addCallback(self.generate_d)
407

409 # function for generating the private exponent d, each
        player end up with a private part of the total d
    def generate_d(self, results):
411     self.function_count[15] += 1
        # calculate the total l mod e
413     l_tot = results[0].value % self.e
```

```
        #print "l_tot = " + str(l_tot)
415
        # check that total l is invertable mod e
417     try:
            zeta = gmpy.divm(1, l_tot, self.e) # CHECK IF
                INVERTABLE
419     except:
            # if not invertable, the protocol needs to be
                started all over
421         # not invertable often means badly chosen 'e'
            print "not invertable mod e"
423         self.generate_p()

425     #print "zeta (inv) = " + str(zeta)

427     # calculate this player's private d, rounded down, this
            means it's not entirely correct, but corrected later
        self.d = int( - (zeta*self.phi_i)/self.e)
429     print "self.p = " + str(self.p)
        print "self.q = " + str(self.q)
431     print "self.d = " + str(self.d)
        print "N (public) = " + str(self.n_revealed)
433     print "Total bits in N = " +str(math.log(self.n_revealed
            , 2))

435     # calculate this player's c, which is used to correct
            the d with a trial decryption
        base = gmpy.mpz(self.m)
437     power = gmpy.mpz(self.e)
        modulus = gmpy.mpz(self.n_revealed)
439     self.c = int(pow(base, power, modulus))

441     # the wanted value to calculate is this player's c^di
            mod N, but player 1's 'd' is negative, therefore find
             the inverse of player 1's c mod N, and use that
            instead
        if self.runtime.id == 1:
443         self.c = gmpy.divm(1, self.c, self.n_revealed)
        base = gmpy.mpz(self.c)
445     if self.runtime.id == 1:
            power = gmpy.mpz(-self.d)
447     else:
            power = gmpy.mpz(self.d)
449     modulus = gmpy.mpz(self.n_revealed)
        # decrypt = c^di mod N
451     self.decrypt = int(pow(base, power, modulus))
        #print "self.decrypt (c^di mod N) = " + str(self.decrypt
            )
453
        # each player share its c = self.decrypt
455     c1, c2, c3 = self.runtime.shamir_share([1, 2, 3], self.
            Zp, self.decrypt)

457     open_c1 = self.runtime.open(c1)
```

```
        open_c2 = self.runtime.open(c2)
459     open_c3 = self.runtime.open(c3)

461     results = gather_shares([open_c1, open_c2, open_c3])
        results.addCallback(self.check_decrypt)
463
    def check_decrypt(self, results):
465     self.function_count[16] += 1
        # player 3 is responsible for the trial decryption,
            mostly because player 1 has a negative d and that
            means more calculations if player 1 is suppose to do
            the task
467     if self.runtime.id == 3:
            c1 = results[0].value
469         c2 = results[1].value
            c3 = results[2].value
471
            # the adjustment is at most n-1, for three players
                this means max 2
473         for i in range(0,3):
                # calculate the temp_decrypt
475             tmp_decrypt = c1 * c2 * c3 % self.n_revealed #
                    self.c**self.r * c1 * c2 * c3 % self.
                    n_revealed
                print "Decryption = " + str(tmp_decrypt)
477             # check if this value is the correct value
                if (tmp_decrypt == self.m):
479                 print "d found, with +r = " + str(i)
                    # if it is, correct_decryptions is increased
481                 self.correct_decryptions += 1
                    print "Correct decryptions: " + str(self.
                        correct_decryptions) + " / " + str(self.
                        rounds)
483                 break
                else:
485                 # if not, player 3's d is increased by 1 and
                        c3 is recalculated before the next
                        iteration of the for-loop is done
                    self.d += 1
487                 base = gmpy.mpz(self.c)
                    power = gmpy.mpz(self.d)
489                 modulus = gmpy.mpz(self.n_revealed)
                    c3 = int(pow(base, power, modulus))
491
        # time2 is set to calculate the total time for the
            generation of this valid key
493     self.time2 = time.clock()
        # completed_rounds is increased in case of more rounds
495     self.completed_rounds += 1
        print "Completed rounds: " + str(self.completed_rounds)
            + " / " + str(self.rounds)
497     # the time for finding the current key is saved in the
            times variable
        self.times += [self.time2 - self.time1]
```

```
499             # check if all the key generation rounds are finished
                if self.completed_rounds == self.rounds:
501                 # if so, print the datas from the generations
                    print "\n\nBENCHMARKS FOR VALID KEY GENERATION"
503                 print "times = " + str(self.times)
                    print "Average: " + str(sum(self.times) / (self.
                        rounds))
505                 print "Correct decryptions: " + str(self.
                        correct_decryptions) + " / " + str(self.rounds)
                    print "\n"
507                 for i in range(len(self.function_count)):
                        print str(self.function_count_names[i]) + ": " +
                             str(self.function_count[i]) + ", avg: " +
                            str(int(self.function_count[i] / self.rounds)
                            )
509                 # test if the program is suppose to do
                        decryption_benchmark as well
                    if self.decrypt_benchmark_active == True:
511                     self.decrypt_benchmark()
                        return
513                 else:
                        # the protocol is finished, synchronize the
                            shutdown
515                     self.runtime.shutdown()
            else:
517             # more key generation shall be done, reset the
                    parameters for a new round and start the protocol
                     again from generate_p()
                self.prime_pointer = 0
519             self.decrypt_tries = 0
                self.time1 = time.clock()
521             self.generate_p()

523
        # function for benchmarking the decryption time for a valid
            key
525     # the method is to choose a message 'm', calculate the
            cipher c = m^e mod N, then find each player's part of the
             message mi = c^di mod N
        def decrypt_benchmark(self):
527         # start the clock for time benchmark
            self.decrypt_time1 = time.clock()
529
            # calculate this player's cipher c
531         base = gmpy.mpz(self.m)
            power = gmpy.mpz(self.e)
533         modulus = gmpy.mpz(self.n_revealed)
            self.c = int(pow(base, power, modulus))
535
            # since player 1's d is negative, find the inverse
537         if self.runtime.id == 1:
                self.c = gmpy.divm(1, self.c, self.n_revealed)
539         base = gmpy.mpz(self.c)
            if self.runtime.id == 1:
```

```
541             power = gmpy.mpz(-self.d)
            else:
543             power = gmpy.mpz(self.d)

545         modulus = gmpy.mpz(self.n_revealed)
            # calculate this player's mi = c^di mod N
547         self.decrypt = int(pow(base, power, modulus))

549         # share the values
            c1, c2, c3 = self.runtime.shamir_share([1, 2, 3], self.
                Zp, self.decrypt)
551
            # calculate the total c and open
553         c_tot = c1 * c2 * c3
            open_c_tot = self.runtime.open(c_tot)
555
            results = gather_shares([open_c_tot])
557         results.addCallback(self.check_decrypt_benchmark)

559
    # function for checking the results from the decryption
        benchmark
561 def check_decrypt_benchmark(self, results):
        # the offset of the total d is off by at most n-1,
            iterate through all possible values
563     for i in range(0,3):
            # calculate a tmp_decrypt
565         tmp_decrypt = results[0].value % self.n_revealed
            # check if this is equal to the original message
567         if tmp_decrypt == self.m:
                # if so, stop the clock
569             self.decrypt_time2 = time.clock()
                # update the number of decrypt tries and save
                    the time used for the current decryption
571             self.decrypt_tries += 1
                self.decrypt_times += [self.decrypt_time2 - self
                    .decrypt_time1]
573             #print "correct decryption for m = " + str(self.
                    m)

575             # check if more decryption benchmarks is suppose
                     to be done
                if self.decrypt_tries < self.decrypt_rounds:
577                 # if yes, update the original message to not
                         repeat decryption for the same message '
                        m' every time
                    self.m += 1
579                 # go back to the decrypt_benchmark() for a
                        new round
                    self.decrypt_benchmark()
581                 return
                else:
583                 # print some useful output from the
                        benchmark
```

```
                    print "\n\nBENCHMARK FOR DECRYPTION"
585                 print "times = " + str(self.decrypt_times)
                    print "average decrypt time = " + str(sum(
                        self.decrypt_times) / self.decrypt_rounds
                        )
587                 # the protocol is finished, synchronize the
                        shutdown
                    self.runtime.shutdown()
589                 return

591
            # function for distributed decryption of an arbitrary
                ciphertext
593         # the players needs to have a shared key for this function
                to work
            # each player calculates m_i and shares the values to obtain
                 the message M
595         def decryption(self, ciphertext):
                # since player 1's d is negative, find the inverse
597             if self.runtime.id == 1:
                    ciphertext = gmpy.divm(1, ciphertext, self.
                        n_revealed)
599             base = gmpy.mpz(ciphertext)

601             if self.runtime.id == 1:
                    power = gmpy.mpz(-self.d)
603             else:
                    power = gmpy.mpz(self.d)
605
                modulus = gmpy.mpz(self.n_revealed)
607             m_i = int(pow(base, power, modulus))

609             m1, m2, m3 = self.runtime.shamir_share([1, 2, 3], self.
                    Zp, m_i)
                m_tot = m1 * m2 * m3
611             open_m_tot = self.runtime.open(m_tot)

613             results = gather_shares([open_m_tot])
                results.addCallback(self.check_decryption)
615

617         # function for revealing the plaintext from decrypting the
                ciphertext
            def check_decryption(self, results):
619             message = results[0].value % self.n_revealed
                print "\nDecryption of ciphertext yields M = " + str(
                    message)
621

623         # function for distributed signature of an arbitrary message
            # the players needs to have a shared key for this function
                to work
625         # signature is carried out by using the shared 'd' to
                encrypt a message
```

```
    # each player calculates c_i and shares the values to obtain
        the signature C
627 def signature(self, message):
        # since player 1's d is negative, find the inverse
629     if self.runtime.id == 1:
            message = gmpy.divm(1, message, self.n_revealed)
631     base = gmpy.mpz(message)

633     if self.runtime.id == 1:
            power = gmpy.mpz(-self.d)
635     else:
            power = gmpy.mpz(self.d)
637
        modulus = gmpy.mpz(self.n_revealed)
639     c_i = int(pow(base, power, modulus))

641     c1, c2, c3 = self.runtime.shamir_share([1, 2, 3], self.
            Zp, c_i)
        c_tot = c1 * c2 * c3
643     open_c_tot = self.runtime.open(c_tot)

645     results = gather_shares([open_c_tot])
        results.addCallback(self.check_signature)
647

649 # function for revealing the calculated signature C of a
        given message M
    def check_signature(self, results):
651     signature = results[0].value % self.n_revealed
        print "\nSignature for message M is C = " + str(
            signature)
653

655 # function that starts the shared RSA protocol
    def __init__(self, runtime):
657
        # CHANGEABLE VARIABLES
659     #**********************

661     # rounds are the total number of rounds to be run for
            benchmark
        self.rounds = 1
663     # set True to do decryption benchmark, False to drop
            this benchmark
        self.decrypt_benchmark_active = True
665     # The number of decryption rounds to be performed if
            active
        self.decrypt_rounds = 20
667     # the number of bits in N (meaning p and q are bits_N /
            2 each)
        self.bits_N = 64
669
        # m is the message used to check for correct decryption
671     self.m = 2
```

```
673         # the lower limit for primality testing, testing done
                secret shared
            self.bound1 = 12
675         # the limits for primality testing of N, done locally
                with different boundaries for each player
            # more efficient to let player 1 check larger span,
                statistically player 1 will fail most often
677         self.bound2_p1 = 15000 # 12-15000 = 1749 primes
            self.bound2_p2 = 17500 # 15000-17500 = 260 primes
679         self.bound2_p3 = 20000 # 17500-20000 = 253 primes

681
            # VARIABLES NOT TO BE CHANGED
683         #****************************

685         # time1 and time2 is used to measure the total time of
                generating a key
            self.time1 = time.clock()
687         self.time2 = 0
            # completed_rounds are used when running keygeneration
                several times for benchmarking
689         self.completed_rounds = 0
            # times are the times from each round in key generation
691         self.times = []
            # correct_decryptions are used to sum up the total
                number of correct decryptions when benchmarking key
                generation
693         # if printout show that correct_decryptions is not equal
                 to the total number of rounds, the protocol is
                flawed
            self.correct_decryptions = 0
695
            # decrypt_time1/2 is used to measure the time for
                decryption benchmark
697         self.decrypt_time1 = 0
            self.decrypt_time2 = 0
699         # decrypt_times are the times from each round in the
                decrypt benchmark
            self.decrypt_times = []
701
            #self.completed_decrypt = 0
703
            # completed_decrypt is used to count the number of
                decryptions done until now in decryption benchmark
705         self.decrypt_tries = 0

707         # Save the Runtime for later use
            self.runtime = runtime
709
            # bit_length is the number of bits in p and q (correct
                for 3 players)
711         self.bit_length = int(self.bits_N / 2) - 2
```

```
        # numeric_length is the used to generate a numeric value
            based on a certain number of bits and is divided by
            4 because of the way p and q are choosen later
713     self.numeric_length = int((2**self.bit_length) / 4)

715     # prime_list_b1 is the list of primes that are checked
            secret shared
        self.prime_list_b1 = self.get_primes(2, self.bound1)
717
        # prime_list_b2 is the list of primes that are checked
            locally by each player, and is therefore different
            for each player
719     if self.runtime.id == 1:
            self.prime_list_b2 = self.get_primes(self.bound1,
                self.bound2_p1)
721     elif self.runtime.id == 2:
            self.prime_list_b2 = self.get_primes(self.bound2_p1,
                self.bound2_p2)
723     else:
            self.prime_list_b2 = self.get_primes(self.bound2_p2,
                self.bound2_p3)
725
        #print self.prime_list_b1
727     print "length of list b2 = " + str(len(self.
            prime_list_b2))

729     # prime_pointer is used to point to the right prime
            number in the list at all times
        self.prime_pointer = 0
731
        # list used for debugging how many times each function
            is run
733     #
        self.function_count =
            [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
735     self.function_count_names = ["generate_p", "generate_q",
            "trial_division_p", "check_trial_division_p", "
            trial_division_q", "check_trial_division_q", "check_n
            ", "primality_test_N", "check_primality_test_N", "
            generate_g", "check_g", "check_v", "generate_z", "
            check_z", "generate_l", "generate_d", "check_decrypt
            "]

737     # l needs to be large enough to cope with all possible
            numbers that appear in the program during execution
        # if this value is too small, the values could wrap
            around the value of Zp.modulus and give bogus outputs
739     l = int(self.bits_N * 3.5)
        k = runtime.options.security_parameter
741
        # For the comparison protocol to work, we need a field
            modulus
743     # bigger than 2**(l+1) + 2**(l+k+1), where the bit
            length of
```

```
            # the input numbers is l and k is the security parameter
                .
745         # Further more, the prime must be a Blum prime (a prime
                p such
            # that p % 4 == 3 holds). The find_prime function lets
                us find
747         # a suitable prime.
            self.Zp = GF(find_prime(2**(l + 1) + 2**(l + k + 1),
                blum=True))
749
            #print self.Zp.modulus
751
            # start the protocol by each player generating its own
                private value for p
753         self.generate_p()

755
    # Parse command line arguments.
757 parser = OptionParser()
    Runtime.add_options(parser)
759 options, args = parser.parse_args()

761 if len(args) == 0:
        parser.error("you must specify a config file")
763 else:
        id, players = load_config(args[0])
765
    # Create a deferred Runtime and ask it to run our protocol when
        ready.
767 #pre_runtime = create_runtime(id, players, 1, options,
        Toft05Runtime)
    runtime_class = make_runtime_class(mixins=[ComparisonToft07Mixin
        ])
769 pre_runtime = create_runtime(id, players, 1, options,
        runtime_class)
    pre_runtime.addCallback(Protocol)
771
    # Start the Twisted event loop.
773 reactor.run()
```

Appendix E

GMPY

The **G**eneral **M**ultiprecision **PY**thon project focuses on Python-usable modules providing multiprecision arithmetic functionality to Python programmers. GMPY supports all kinds of mathematical functions, written in the programming language C for optimization, in an easy-to-use fashion, and GMPY has been used extensively throughout the implementation of distributed RSA in VIFF. For more information about GMPY see [GMP03], and for where to download GMPY see Appendix B.

All the functions used in the distributed RSA implementation will briefly be described below.

E.1 find_prime

The function *next_prime(x)* returns the smallest prime number > x and does so in a really fast manner even for very large x's. Note that this function uses a probabilistic definition of prime.

E.2 jacobi

The function *jacobi(x, y)* returns the Jacobi symbol $\left(\frac{x}{y}\right)$, and is used in the distributed biprimality test.

E.3 pow

The standard power operator in Python, **, is not very optimized, and takes from seconds to minutes to calculate typical large exponents like the ones used in RSA. The function *pow(a, b, c)* returns the number $a^b \ mod \ c$ in matter of milliseconds for arbitrary large numbers because it's based

on the exponentiation by squaring method (also called square-and-multiply method).

E.4 divm

The function *divm(a, b, m)* returns x such that $b \cdot x \equiv a \ mod \ m$, therefore being an easy way of finding modular inverses by setting $a = 1$.

E.5 gcd

The function *gcd(a, b)* returns the greatest common denominator of the numbers a and b.