TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

MASTER'S THESIS

Fine-tuned implementation of an efficient secure profile matching protocol

by
N. Mavrogiannopoulos

Supervisors:

dr. ir. B. Schoenmakers
dr. P. Tuyls

Eindhoven, July 2006

Abstract

In this report we study and try to minimize the total execution time of a Profile Matching protocol, a secure computations protocol based on an operation called conditional gate. We focus on reducing the complexities and study the interplay between them, mainly the round, communication, computational, space and random complexities. We propose several optimizations and study their effect, both the theoretical limits and their effect on a real world implementation. The involved protocols were implemented in an application that performed an *Approximate matcher* protocol. In addition to this report, parts of this project are also the "Profile Matcher" application and the "Profile Matcher API Reference Manual".

What we show in this report is that the expected high round complexity can be reduced for this protocol and give ways to utilize the idle time caused by the need for interaction. Furthermore we show that the effects of scheduling of operations with high round complexity can be impressive compared to a straightforward implementation.

This project was carried out within Philips Research, Eindhoven.

Acknowledgements

It was a pleasure working on this project, not only because I had the opportunity to explore new for me concepts and algorithms, but mainly because of the people I cooperated with. I would like to thank Berry Schoenmakers for encouraging me with the interest he showed in the project and his invaluable advices, as well as Pim Tuyls for his support throughout the project's duration and for providing terse and constructive advices and comments. Finally I would like to thank Philips Research for supporting and funding this project.

Contents

Chapter 1

Introduction

1.1 Profile Matching

Given the large amount of personal information stored today's databases and the need for privacy protection, it is important for the owners of such databases to prevent abuse of the contents. Two ways to protect this information is by use of strict access control, or by active protection in a way that information is used but not given away. Profile matching is a protocol of the latter kind. Data, such as biometrics, fingerprints, etc., can be compared against other values without revealing any of them to the parties performing the computations. This protocol would also allow scenarios of authentication where the one performing the authentication has no access to the actual data being verified, thus ensuring the security of the stored data.

The profile matching protocol we base our study on is described in [32]. What the protocol does is, given a set of parties, allows them to compare the Hamming distance[1] of two encrypted binary strings against a given threshold. During the protocol execution no-one learns anything about the two encrypted binary strings, except for the output of the comparison. The protocol is based on algorithms for secure computations and relies on a homomorphic threshold cryptosystem, such as the homomorphic El-Gamal system. It is called Secure Approximate Matcher, but from now on we refer on it simply as *Approximate matcher*.

1.2 Goals of this study

An important question [40] for secure multi-party computations protocols, is whether they are efficient enough to be used in practical applications, i.e., run in acceptable time, exchange a reasonable amount of data, rounds of interaction are realistic, scale well with the size of the input, etc. In this study we try to

[1] The Hamming distance of two bitstrings gives the number of positions for which the strings are different.

give an initial answer to this question for the *Approximate matcher* protocol from [32] without getting restricted in specific system requirements. Our main target is to minimize the total time of execution of the protocol as far as possible with the given algorithms and provide an efficient implementation of it.

For the proposed optimizations we focused on the interplay between complexity measures such as

- round complexity;
- communication complexity;
- computational complexity;
- randomness complexity;
- space complexity.

1.3 Roadmap

The organization of this document is as follows. In the next chapter we discuss the protocols used by the *Approximate matcher* and give brief descriptions and complexities. Those are the building blocks in order to understand the protocol. In the description of these algorithms the knowledge of algebra is assumed. The *Approximate matcher* protocol is explained in chapter 3. The next two chapters discuss optimizations on the described protocols. Generic optimizations that could apply to other protocols except for the *Profile matching* are given in chapter 4. Specific optimizations for the *Approximate matcher* are discussed in chapter 5. Implementation requirements and architecture of the developed software are discussed in chapter 6. Chapter 7 summarizes important issues that were considered throughout the implementation and design phase. Chapter 8 discusses the results obtained by the implementation and Chapter 9 gives a summary of the results of this report.

Chapter 2

Building blocks

2.1 Definitions and terminology

In order for all discussed topics to be clear to the reader, we quote here the definitions of important aspects used throughout this document. All the acronyms used in this document are included in Appendix A.

When one of the following terms are used to describe complexities in multi-party computations, we limit their scope to each party and not as a total.

Communication complexity We quote the definition from [19, p. 84] .

> *The cost of a protocol is the number of bits written on the board*[1] *for the worst case input. The multiparty (deterministic) communication complexity of f, D(f), is the minimal cost of a protocol that computes f.*

Space complexity We will use the apparent meaning for space complexity. That is

> *Space complexity of a protocol is the amount of memory space required to solve an instance of the problem as a function of the size of the input.*

Computational complexity and Running time Usually the term computational complexity is used to describe how many steps are needed for an algorithm to solve a problem, using some model computing device such as the Turing machine. In this document we will not make such a strict use of the term. We will also make a distinction between running time and computational complexity because protocols that involve interaction with other peers have running time that is not depending only on the computations performed

[1] broadcasted.

locally. With this distinction we have a measure of both concepts. We define the computational complexity term as

> *The computational complexity of a problem with respect to a set of operations is the amount of those operations needed to solve an instance of the problem as a function of the size of the input.*

It can be clearly seen that by selecting complex operations to measure the computational complexity, we could get a measurement of the running time. In those cases we will explicitly make use of the term "running time". A more concrete definition of running time is then[2]

> *The running time of a protocol is its computational complexity with respect to a set of operations that if each one of them accounts for a number of time slots[3], in total they account for the total number of time slots used by the protocol.*

In the rest of this document when we simply reference computational complexity, we imply the complexity with respect to modular exponentiations, which is the most expensive operation in the described algorithms.

Round complexity A simplistic approach to define the round complexity would be to set it to be the number of messages exchanged, or the number of broadcasts. However there are cases where parties do independent broadcasts of messages in parallel without any kind of interaction between them. This if studied with the definition above would result to, for n parties, a round complexity of n rounds. However it can be argued that since those broadcasts can be executed in a single time slot (there is no dependence between them), this is a single round.

For this reason here we will define the round complexity to be

> *The number of messages exchanged during interactive computations, when the messages depend on each other and 1 when the messages are independently broadcasted or transmitted[4].*

Our definition is consistent with the usage of round complexity in [11] and [31].

Randomness complexity Here we will use a definition of randomness complexity that is in accordance with [18]. The reason we decided to include this measurement unit is that even in modern operating systems obtaining real random bits is not only expensive but also difficult to do using real world computers [39], [8, Generating randomness]. We define it as

> *The amount of random bits required by a protocol, as a function of the size of the input.*

[2] No parallelism of operations is taking into account by these definitions.
[3] Here we do not define the time slot but assume its obvious meaning.
[4] That is in a fictional protocol where everybody is transmitting random and independent messages to each other, the round complexity is 1.

2.2 El-Gamal cryptosystem

The basis of our protocols is the homomorphic El-Gamal cryptosystem. Initially we give a description of the normal El-Gamal cryptosystem and afterwards discuss the differences.

The El-Gamal cryptosystem is a public key system and works in a (multiplicative) subgroup $\mathbb{S}$ of order q of a group $\mathbb{G}$, where q a prime[5]. This subgroup is generated by an element $g \in \mathbb{G}$ and it is assumed that it is large enough so Decisional Diffie Hellman (DDH) and Discrete Logarithm (DL) problems are infeasible. The basis for the algorithm are a public key, h, and private key, x, pair such that $h = g^x \in \mathbb{S}$.

For this study we selected $\mathbb{Z}_p^*$ as the group, where p is a prime. The subgroup $\mathbb{S}$ will then be generated by an element $g \in \mathbb{Z}_p^*$ of order q and thus its elements will be of the form g^u, $u \in \mathbb{Z}_q$. Whenever possible, we will try to be abstract to allow for versions of the algorithm that work in other groups than $\mathbb{Z}_p^*$.

2.2.1 Encryption

Messages to be encrypted are represented by elements of the subgroup $\mathbb{S}$. The procedure to encrypt a message m is done as follows

- A random element r is generated in $\mathbb{Z}_q$
- The encrypted value is then a pair in $\mathbb{S} \times \mathbb{S}$ calculated as

$$(a, b) = (g^r, mh^r),\ a, b \in \mathbb{S}$$

2.2.2 Decryption

By taking

$$\begin{aligned} b \cdot (a^x)^{-1} &= \\ mh^r \cdot g^{-rx} &= \\ mg^{xr} \cdot g^{-rx} &= \\ m & \end{aligned}$$

we retrieve m if the private exponent x is known.

2.3 Homomorphic El-Gamal cryptosystem

A property that the El-Gamal cryptosystem [30] has is that given two encrypted values (a_0, b_0) and (a_1, b_1) of m_0 and m_1 respectively we can compute an encryption of $m_0 m_1$ in the group without the need for the private key x. Note

[5] Actually El-Gamal can work using any subgroup, but in our study we use a subgroup of prime order q for reasons that will be explained later on.

that we use the multiplication $(\cdot)$ between two encrypted values a bit liberally.

$$\begin{aligned} a_0 &= g^{r_0}, & b_0 &= m_0 h^{r_0} \\ a_1 &= g^{r_1}, & b_1 &= m_1 h^{r_1} \\ (a_0, b_0) \cdot (a_1, b_1) &= & (g^{r_0+r_1}, m_0 m_1 h^{r_0+r_1}) &= & (g^{r_2}, m_0 m_1 h^{r_2}) \end{aligned}$$

The homomorphic El-Gamal is a modification of the El-Gamal system that has the property that given two encryptions of m_0 and m_1 the encryption of m_0+m_1 can be calculated. That can be achieved if encryption of an element m is defined as

$$(a, b) = (g^r, g^m h^r),\ a, b \in \mathbb{S}$$

Then it can be easily seen that by multiplying two encrypted values of m_0 and m_1 we get the encryption of $m_0 + m_1$. It should be noted though that decryption is not easy in this cryptosystem since with the decryption algorithm shown above we get g^m as the output value. From this value since the DL problem holds we cannot retrieve m in general. In special cases where m is restricted to small set $\mathcal{A}$ of values the value of m can be retrieved by testing equality with a precomputed set of g^k, $k \in \mathcal{A}$.

We can now see that with this scheme the message m needs only to be expressed as an element in $\mathbb{Z}_q$, because then g^m is an element of the subgroup.

2.3.1 Equivalence of encrypted values

We define an equivalence relation in $\mathbb{S} \times \mathbb{S}$ as

$$(a, b) \sim (c, d) \text{ iff } b \cdot a^{-x} = d \cdot c^{-x}$$

This relation is a group congruence since it is compatible with the group structure. The cosets (equivalence classes) are[6] q and are denoted as $[\![y]\!]$, $y \in \mathbb{Z}_q$.

In other words so far we have defined an encryption scheme, the form of the encrypted values and when we consider them equal[7]. In the next section we also define what we can do with the encrypted values.

2.3.2 Operations and symbolism

Let's now examine some operations that are possible with the homomorphic El-Gamal cryptosystem and give some symbols that will be used throught this document to represent them.

- Addition: we can obtain the encrypted sum of the plaintext of two encrypted values $[\![x]\!] \oplus [\![y]\!]$ as

[6] Because $b \cdot a^{-x} = g^m \cdot h^r \cdot g^{-rx} = g^m \in \mathbb{S}$, and $\mathbb{S}$ is of order q.

[7] Note that equality of two encrypted values cannot be checked if the private key is absent.

$$\begin{aligned}
[\![x]\!] \oplus [\![y]\!] &= (g^r, g^x h^r) \cdot (g^{r_1}, g^y h^{r_1}) \\
&= (g^{r+r_1}, g^{x+y} h^{r+r_1}) \\
&= (g^{r_2}, g^{x+y} h^{r_2}) \\
&= [\![x+y]\!]
\end{aligned}$$

- Subtraction: similarly the subtraction of two encrypted values $[\![x]\!] \ominus [\![y]\!]$ as

$$\begin{aligned}
[\![x]\!] \ominus [\![y]\!] &= (g^r, g^x h^r) \cdot (g^{r_1}, g^y h^{r_1})^{-1} \\
&= (g^{r-r_1}, g^{x-y} h^{r-r_1}) \\
&= (g^{r_2}, g^{x-y} h^{r_2}) \\
&= [\![x-y]\!]
\end{aligned}$$

- Multiplication of the plaintext value with an element from $\mathbb{Z}_q$: $c \otimes [\![x]\!]$ can be performed as

$$\begin{aligned}
[\![x]\!] \otimes c &= (g^r, g^x h^r)^c \\
&= (g^{cr}, g^{cx} h^{cr}) \\
&= (g^{r_2}, g^{cx} h^{r_2}) \\
&= [\![c \cdot x]\!]
\end{aligned}$$

  Division is possible by multiplying with the inverse of the element in question. This is possible since $\mathbb{Z}_q$ is a field[8].

- Addition (and subtraction) with an element from $\mathbb{Z}_q$: $c \oplus [\![x]\!]$ as

$$\begin{aligned}
[\![x]\!] \oplus c &= (g^r, g^x h^r \cdot g^c) \\
&= (g^r, g^{c+x} h^r) \\
&= [\![x+c]\!] \\
[\![x]\!] \ominus c &= (g^r, g^x h^r \cdot g^{-c}) \\
&= (g^r, g^{x-c} h^r) \\
&= [\![x-c]\!]
\end{aligned}$$

- Random re-encryption is defined as

$$[\![x]\!]_{\mathrm{R}} = [\![x]\!] \cdot (g^{r_1}, h^{r_1}),\ r_1 \in_R \mathbb{Z}_q$$

  This computes an encrypted value for the same message that is distinguishable from the previous value.

[8] That also explains our choice of taking g to be a generator of a subgroup of prime order q rather than being a generator for the group or a random subgroup.

What we will add for the purposes of the *Approximate matcher* to this list of operations is the following

- Binary Multiplication: the multiplication of two encrypted values, one of x or y must be in the $\{0, 1\}$ range;

$$\begin{aligned} [\![x]\!] \otimes_b [\![y]\!] &= \text{ConditionalGate}([\![x]\!], [\![y]\!]) \\ &= [\![x \cdot y]\!] \end{aligned}$$

The Conditional Gate will be discussed in later sections of this chapter.

2.4 Threshold El-Gamal cryptosystem

We will describe a (t, l)-threshold cryptosystem [30] based on El-Gamal. That is a cryptosystem that divides the knowledge of the private key to l parties, such that at least t of them are required for decryption. As with other public key cryptosystems the public key is available for anyone to perform encryptions.

2.4.1 El-Gamal threshold key generation

In the following paragraphs we give a private key generation process based on Feldman's Verifiable Secret Sharing (VSS) protocol, as described in [30]. We give the general case where $t \leq l$. The special case where $t = l$ which may be interesting for the two party case is described in Appendix B.

General case $t \leq l$

1. Each party i selects a random polynomial $p_i(x) = \sum_{z=0}^{t} a_{i,z} x^z \in \mathbb{Z}_q[x]$[9] of degree less than t and broadcasts a commitment to the value of g^{s_i}, where $s_i = p_i(0)$;
2. The public key h is then set as $h = g^{\sum_{i=1}^{l} s_i}$;
3. Each party i sends shares $s_{ij} = p_i(j)$ to participant P_j in private, for $1 \leq j \leq l$. In addition party i broadcasts commitments $B_{i,z} = g^{a_{i,z}}$, $0 \leq z < t$. Upon receipt of share s_{ij}, participant P_j verifies its validity by the equation

$$g^{s_{ij}} = \prod_{z=0}^{t-1} B_{i,z}^{j^z} = g^{\sum_{z=0}^{t-1} a_{i,z} j^z} \tag{2.1}$$

4. Each party i sums all its received shares s_{ji} to obtain

$$x_i = \sum_{j=1}^{l} s_{ji} = \sum_{j=1}^{l} p_j(i)$$

[9] Note that when working on the exponent we are modulo q which is the order of the generator g and the output values such as g^a are modulo p which is the field we are working on.

which is its share of the private key x. The verification key of the party is defined as $h_i = g^{x_i}$. This verification key can be used to prove correct usage of the private share (see Section 2.6.3). Other parties than i can compute this key using the broadcasted values $B_{i,z}$ as

$$\prod_{j=1}^{l}\prod_{z=0}^{t-1} B_{i,z}^{j^z} = g^{\sum_{j=1}^{l}\sum_{z=0}^{t-1} a_{i,j}j^z} = g^{x_i} = h_i$$

After the key generation takes place the secret key is actually the $p(0)$ value of the polynomial

$$p(x) = \sum_{z=0}^{t}\sum_{i=1}^{l} a_{i,z}x^z$$

which is the sum of the individually generated polynomials and each party holds a share x_i which is equal to $p(i)$. Encryption is performed as with plain El-Gamal using the public key h. For the decryption the reconstruction of the polynomial $p(x)$ is required and this will be discussed in a following section.

This protocol's performance is outside the scope of this study thus we will not focus into it. For the interested reader a more detailed discussion is in [30]. Similar protocols can be found in [28] and [12].

2.4.2 El-Gamal threshold decryption

In order to decrypt an encrypted value with the public key h the secret value from the polynomial $p(x)$ has to be used but in a way that it doesn't get revealed to any of the parties.We give a description of the method as well as some complexities because this is an important part of the conditional gate, which will be discussed later.

Description

To decrypt a message (a, b), by a set of players A, where each player has a private share x_i the following steps are used. Note that A must have at least t players, which is the minimum number needed for deciphering.

1. Each party i computes $d_i = a^{x_i}$ and broadcasts d_i and a proof that $d_i = a^{x_i}$ (for the proof see Section 2.6.3);
2. Each party verifies the received proofs and calculates the secret message as
$$m = \frac{b}{\prod_{i\in A} d_i^{\lambda_{A,i}}},\ \lambda_{A,i} = \prod_{j\in A\setminus\{i\}} \frac{j}{j-i} \tag{2.2}$$
which is based on Lagrange's interpolation algorithm, to reconstruct a polynomial passing through t points.

We have to note here that the decryption protocol above has to be performed in a private channel between the parties that share the private key, or the output of the decryption will become known to any eavesdroppers.

Computational complexity

Each party must compute a single exponentiation ($d_i = a^{x_i}$), calculate Equation (2.2), generate one proof and verify $t - 1$ proofs. The proof is described in Section 2.6.3 on page 19 and each one needs[10] two exponentiations to be computed and four to be verified. In both cases the output of a hash function is required.

To calculate the Equation (2.2) each party has to perform $t \cdot (t-1) + t = t^2$ multiplications, t exponentiations and one division. Here we will focus on the complexity of the exponentiations and ignore the complexities of multiplications and other operations because their complexity is at least one order of magnitude lower than exponentiations. Thus the overall bit complexity with respect to exponentiations[11] is

$$1 + 2 + 4(t-1) + t = \tag{2.3}$$
$$= (5t - 1) \tag{2.4}$$

Memory space requirements

For these computations the storage of the d_i values is the only requirement. In a case where there are no cheating parties the required space is

$$t \log_2 p \text{ bits} \tag{2.5}$$

Randomness complexity

It is interesting to see the randomness complexity per party of the algorithm. For the decryption step, no random numbers are required. However the proof requires one random number u of bit size $\log_2 q$ to be generated. So the requirements for this algorithm, in bits, are

$$\log_2 q \tag{2.6}$$

Round complexity

Eventhough the number of messages exchanged in this phase is t the round complexity is

$$1 = O(1) \tag{2.7}$$

That is because none of the messages depend on each other and according to our definition they compose a single round.

[10] Here we violate the order by doing a forward reference, but we believe it serves keeping the description simpler.

[11] Modular exponentiation complexity is of order $O(m^3)$.

Communication complexity

Thus we can see that, if no dishonest parties exist, for n parties and for threshold t we need to broadcast t messages to perform this protocol. Each message contains a number from the group d_i and a non-interactive proof consisting of two values from the subgroup. Thus the communication complexity is

$$t(\log_2 p + 2\log_2 q) \tag{2.8}$$

The communication complexity for this algorithm remains the same if members in the set A increase or decrease, given that they are more than t. It only depends on the number t and the field.

2.5 Conditional gate

The conditional gate [31] is an interactive protocol that computes the product of two homomorphic encrypted values, as long as one of these values is from a binary[12] domain. The computations occur between parties that share the private key of the encrypted values. We focus on this protocol by giving a detailed description and complexities since the conditional gate is the most important building block of the *Approximate matcher* protocol.

2.5.1 Preliminaries

In this protocol the notion of a Pedersen commitment of a number is used. That is given a randomly selected $s \in \mathbb{S} \setminus \{1\}$, a commitment to a number $m \in \mathbb{Z}_q$ is value $b = g^m s^r$, with $r \in_R \mathbb{Z}_q$. That commitment is revealed by opening the values of m and r. We will symbolize the commitment of a value m as $\ll m \gg$. For more information on the commitment and in-depth reasoning for its usage in the protocol see [31].

2.5.2 Description

We give a rough idea of the protocol for input values $[\![x]\!]$ and $[\![y]\!]$ where $x \in \{0, 1\}$ and $y \in \mathbb{Z}_q$. Initially convert x to $\{-1, 1\}$ range, blind both by letting parties multiply it with a random integer from the $\{-1, 1\}$ range and then decrypt the blinded $[\![x]\!]$ using threshold decryption. That way a multiplication of $[\![x]\!]$ and $[\![y]\!]$ can be calculated as $x \otimes [\![y]\!]$. The detailed protocol for n parties is shown in Figure 2.1.

Remember that we have threshold decryption which means that at least t parties are required to decryption. With t being the threshold in El-Gamal decryption.

[12] With the term binary we explicitly mean a two-valued domain and not only the $\{0, 1\}$ domain.

Input: $[\![x]\!], [\![y]\!], x \in \{0,1\}, y \in \mathbb{Z}_q$

Steps:

1. All parties convert the binary input value to $\{-1,1\}$ range

$$\begin{aligned} [\![x']\!] &= (2 \otimes [\![x]\!]) \oplus (-1) \\ [\![x_{-1}]\!] &= [\![x']\!] \\ [\![y_{-1}]\!] &= [\![y]\!] \end{aligned}$$

2. For each party starting with party i=0, blind $[\![x]\!]$ and $[\![y]\!]$
   (a) generate $s_i \in_R \{-1,1\}$
   (b) set $\ll s_i \gg = g^{s_i} h^r$
   (c) $[\![x_i]\!] = [\![s_i \otimes [\![x_{i-1}]\!]]\!]_{\mathrm{R}}$
   (d) $[\![y_i]\!] = [\![s_i \otimes [\![y_{i-1}]\!]]\!]_{\mathrm{R}}$
   (e) generate proof P_i to assure that $[\![x_i]\!]$ and $[\![y_i]\!]$ are correctly generated; Also called Double private multiplier proof; For the proof see Section 2.6.2;
   (f) broadcast P_i, $\ll s_i \gg$, $[\![x_i]\!]$, $[\![y_i]\!]$ to all parties;
3. All parties verify the proof P_i and save $\ll s_i \gg, 0 \leq i \leq n$
4. El-Gamal threshold decryption of $[\![x_n]\!]$
5. Check if x_n is in the $\{-1,1\}$ range
   If not ask for proofs that $s_i \in \{-1,1\}$ and restart the protocol (without the cheating party). The proof is described in Section 2.6.1.
6. All parties undo the initial conversion of x and y

$$\begin{aligned} [\![x'y]\!] &= [\![y_n]\!] \otimes x_n \\ [\![xy]\!] &= ([\![x'y]\!] \oplus [\![y]\!]) \otimes (2)^{-1} \end{aligned}$$

Output: $[\![xy]\!]$

Figure 2.1: The Conditional Gate.

2.5.3 Computational complexity

Here we will try to give the computational complexity per party. For the Double private multiplier proof P_i to be generated 10 exponentiations are required whilst to verify it 15 are required. For the protocol itself it is tricky to calculate the needed exponentiations. Initially to convert x to the range $\{-1,1\}$ a squaring is required, but we will not count this as an exponentiation. Similarly when raising to the power of -1 or 1 we will also not count it. Thus the exponentiations left are the one from Pedersen commitment and 4 exponentiations for the random re-encryption operations.

Overall the computational complexity for exponentiations, including the El-

Gamal decryption step, is

$$\underbrace{10+(n-1)(15)}_{\text{private multiplier proof}} + \underbrace{5}_{\text{conditional gate}} + \underbrace{(5t-1)}_{\text{El-Gamal decryption}} = \\ = (5t+15n-1)$$

When $n = t$ we have the minimum numer of parties that can perform this protocol. For this number of parties the complexity becomes

$$(20n-1) \tag{2.9}$$

2.5.4 Memory space requirements

Each party must store $[\![x']\!], [\![y]\!], \ll s_i \gg$, and temporarily in order to verify the proof, the values $P_i (= 5\log_2 p), [\![x_i]\!], [\![y_i]\!]$ are needed. The size of a homomorphic encryption is $2\log_2 p$ bits, and a Pedersen commitment occupies $\log_2 p$ bits. By including the El-Gamal decryption, from Equation (2.5), we get an upper bound to the total space requirements

$$\begin{aligned}(\text{Conditional gate}) + (\text{El-Gamal decryption}) &= \\ = (n-1)(14\log_2 p) + (t\log_2 p) &= \\ = (14n + t - 14)\log_2 p &= \\ = O(n\log p)&\end{aligned}$$

where n is the number of parties executing the protocol and t the threshold for El-Gamal decryption. For the minimum number of parties possible the complexity is

$$(15n-14)\log_2 p \tag{2.10}$$

2.5.5 Randomness complexity

In the conditional gate protocol the random numbers needed are in

- generation of $s_i \in \{-1, 1\}$, 1 bit;
- Pedersen commitment for $\ll s_i \gg$, $\log_2 q$ bits;
- re-encryption of $[\![x_i]\!]$ and $[\![y_i]\!]$, $\log_2 q$ bits each;
- proof P_i requires $4\log_2 q$ bit numbers;
- El-Gamal decryption needs 1 number of $\log_2 q$ bits (see Equation (2.6)).

Thus in total the requirements, of random bits per party, for the conditional gate are

$$(8\log_2 q) + 1 \tag{2.11}$$

2.5.6 Round complexity

The number of rounds in this protocol also depends on the number of rounds in the El-Gamal decryption. In the two party case the number of rounds is constant since only two messages are exchanged for the conditional gate and two for the El-Gamal decryption. That is complexity of $O(1)$. In the n-party case the number of rounds is n plus the single round of the El-Gamal decryption. Thus the complexity is

$$n + 1 = O(n) \tag{2.12}$$

How a run of the protocol looks like, when $n = t$, is shown in Figure 2.2. The two party case is shown in Figure 2.3.

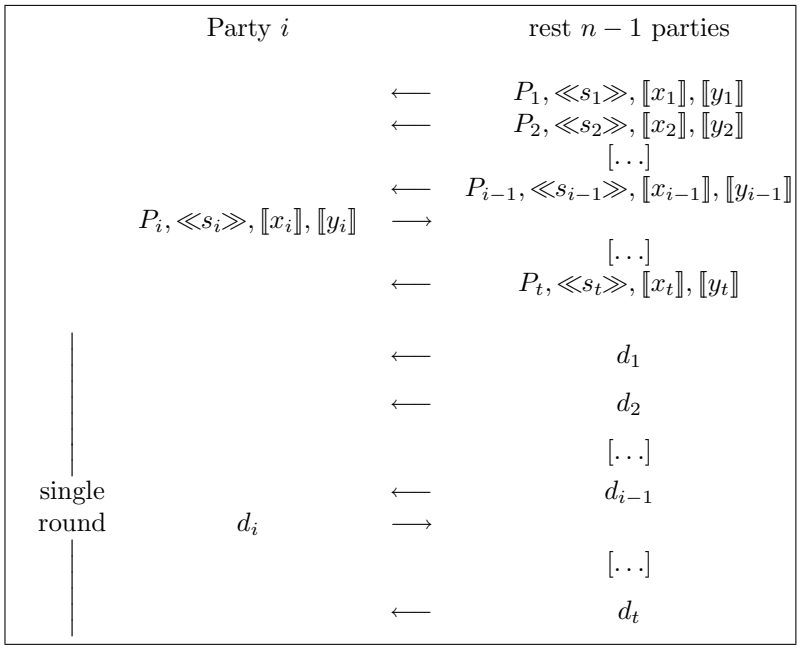

Figure 2.2: The message sequence of the conditional gate.

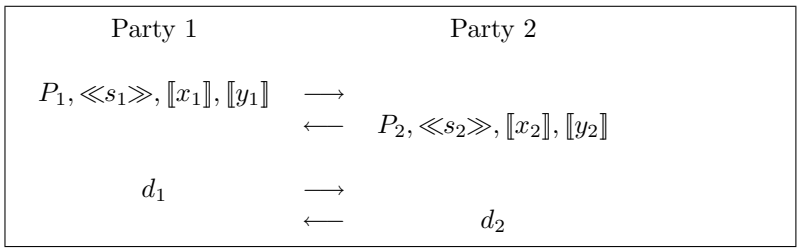

Figure 2.3: The message sequence of the conditional gate for two parties.

2.5.7 Communication complexity

In this section we will try to examine the total complexity of the exchanged data of the conditional gate for n parties and threshold t. In the protocol above we can see that in a run of the protocol where everybody is honest, each party has to broadcast the proof P_i, $\ll s_i \gg, [\![x_i]\!], [\![y_i]\!]$ and perform an El-Gamal threshold decryption.

The proof outputs 5 values from the subgroup we are working on (see Section 2.6.2), thus in total we need to transfer 5 values from the subgroup $\mathbb{Z}_q$ and 5 values from the group $\mathbb{Z}_p^*$. Thus, by also considering Equation (2.8) the complexity in bits would be

$$\begin{aligned}(\text{Conditional gate compl.}) + (\text{El-Gamal decryption compl.}) = \\ = 5n(\log_2 p + \log_2 q) + t(\log_2 p + 2\log_2 q) = \\ = (5n + t)\log_2 p + (5n + 2t)\log_2 q = \\ = O(n \log p)\end{aligned}$$

When performed by the minimum possible number of parties possible, that is $n = t$, the complexity becomes

$$(6\log_2 p + 7\log_2 q)n \tag{2.13}$$

As an example we will test the case where the prime p is a 1024 bits number, q is 160 bits and two parties execute a conditional gate. Then both parties have to transmit at least 14528 bytes.

2.6 Sigma proofs

In this section we list the proofs that are used throughout the discussed protocols. Those proofs are used to ensure the protocols' resistance to malicious participants that violate the protocol rules and send specially crafted values in order to gain advantage over the other participants, for example by learning the content of an encrypted value.

A *Sigma Proof* or Σ-Proof [30, Zero-Knowledge Proofs] is a class of protocols between parties, called the prover and the verifier. In these protocols the prover convinces the verifier on the validity of a statement or on the knowledge of something. Here our proofs are based on the so-called sigma protocols with the Fiat-Shamir[9] heuristic applied. This allows for non-interactive proofs of knowledge.

For a non-interactive *Sigma Proof* to be evaluated the usage of a cryptographic hash function is required, that acts a random oracle[13] $\mathcal{H}$. The input to the hash function are an arbitrary number of values from the $\mathbb{Z}_p^*$ group. When these values are interpreted as integers care must be taken to avoid collisions that arise from the concatenation of them[14].

2.6.1 Conditional gate: Value in range

This proof is used in the Conditional Gate, at a case of a failure, to verify that the values sent by a participant are indeed within a range. More specifically for the Pedersen commitment $\ll s_i \gg = g^{s_i} h^r$, we need a Σ-Proof that shows that $s_i \in \{-1, 1\}$ and r is known. The proof that we are going to use is an OR composition for the value of s_i. The non-interactive version is shown in Table 2.4.

2.6.2 Conditional gate: Double Private multiplier

In the conditional gate, when the values of x_i and y_i are calculated as $[\![x_i]\!] = [\![s_i \otimes [\![x_{i-1}]\!]]\!]_{\mathrm{R}}$ and $[\![y_i]\!] = [\![s_i \otimes [\![y_{i-1}]\!]]\!]_{\mathrm{R}}$, the party that calculated the value has to send a proof that his result has been correctly generated. That is, given

$$\begin{aligned}
\ll s_i \gg &= g^{s_i} h^r \\
[\![x_{i-1}]\!] &= (c, d) \\
[\![x_i]\!] &= (g^z, h^z) \cdot (c^{s_i}, d^{s_i}) = \\
&= (g^z c^{s_i}, h^z d^{s_i}) \\
[\![y_{i-1}]\!] &= (e, f) \\
[\![y_i]\!] &= (g^w, h^w) \cdot (e^{s_i}, f^{s_i}) = \\
&= (g^w e^{s_i}, h^w f^{s_i})
\end{aligned}$$

[13] This is an ideal hash function. For a formal definition see [30].

[14] An example would be $\mathcal{H}(5, 5) = \mathcal{H}(55)$. The solution we follow is to expand the numbers to the maximum size allowed in their field (by prepending zeros). The example above would then be $\mathcal{H}(05, 05) \neq \mathcal{H}(55)$ in the field $\mathbb{Z}_{59}$. This method is similar to the one used in [27].

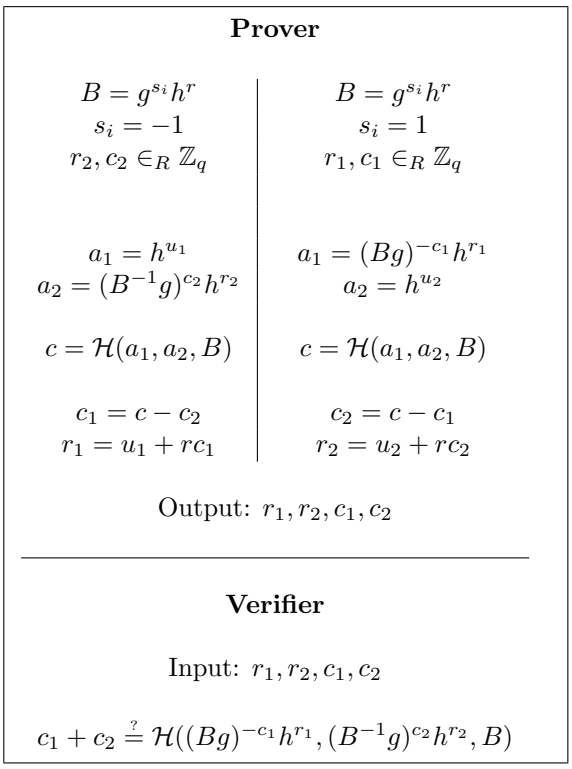

Figure 2.4: Non-interactive sigma proof for the conditional gate. It ensures that the Pedersen commitment was properly generated.

and that $w, r, z \in_R \mathbb{S}$ one has to prove knowledge of r, s_i, z, w satisfying

$$\begin{aligned} A &= g^{s_i} h^r \\ B &= g^z c^{s_i} \\ C &= h^z d^{s_i} \\ D &= g^w e^{s_i} \\ E &= h^w f^{s_i} \end{aligned}$$

This consists of AND and EQ compositions for the values above. The non-interactive version of the sigma protocol for this proof is shown in Figure 2.5.

2.6.3 El-Gamal threshold decryption

In Table 2.6 we list a non-interactive sigma proof used in the El-Gamal threshold decryption (see Section 2.4.2 on page 11). With this proof a party proves that the value d_i he produced was calculated using his private share. It is an EQ composition for the values $d_i = a^{x_i}$ and $h_i = g^{x_i}$.

Prover

$$A = g^{s_i} h^r, B = g^z c^{s_i},$$
$$C = h^z d^{s_i}, D = g^w e^{s_i}, E = h^w f^{s_i}$$
$$u_1, u_2, u_3, u_4 \in_R \mathbb{Z}_q$$

$$a_1 = g^{u_1} h^{u_2}$$
$$a_2 = g^{u_3} c^{u_1}$$
$$a_3 = h^{u_3} d^{u_1}$$
$$a_4 = g^{u_4} e^{u_1}$$
$$a_5 = h^{u_4} f^{u_1}$$

$$c = \mathcal{H}(a_1, a_2, a_3, a_4, a_5, A, B, C, D, E)$$

$$r_1 = u_1 + cs_i$$
$$r_2 = u_2 + cr$$
$$r_3 = u_3 + cz$$
$$r_4 = u_4 + cw$$

Output: (r_1, r_2, r_3, r_4, c)

Verifier

Input: (r_1, r_2, r_3, r_4, c)

$$c \stackrel{?}{=} \mathcal{H}(g^{r_1} h^{r_2} A^{-c}, g^{r_3} c^{r_1} B^{-c}, h^{r_3} d^{r_1} C^{-c}, g^{r_4} e^{r_1} D^{-c},$$
$$h^{r_4} f^{r_1} E^{-c}, A, B, C, D, E)$$

Figure 2.5: Non-interactive sigma proof for double private multiplier.

2.6.4 Private XOR

In the case where a party wants to compute the exclusive or of an encrypted value $[\![y]\!] = (a, b)$ with a known value x, we list a non-interactive sigma proof, to prove that it has been correctly calculated and x is known.

The way to compute the XOR is

$$\begin{aligned} [\![x \texttt{ xor } y]\!] &= [\![[\![y]\!] \oplus x \ominus 2(x \otimes [\![y]\!])]\!]_{\mathrm{R}} \\ &= [\![((1 - 2x) \otimes [\![y]\!]) \oplus x]\!]_{\mathrm{R}} \end{aligned}$$

Thus given the encrypted value $(a, b) = [\![y]\!]$ we have to calculate a value $(c, d) = [\![x \texttt{ xor } y]\!]$ such as

$$\begin{aligned} (c, d) &= (a^{1-2x}, b^{1-2x} g^x) \cdot (g^r, h^r), \ r \in_R \mathbb{Z}_q \\ &= (a^{1-2x} g^r, b^{1-2x} g^x h^r) \end{aligned}$$

The proof for the correctness of this result and the knowledge of x is shown in Figure 2.7.

Prover

$$d_i = a^{x_i}, h_i = g^{x_i}$$
$$u \in_R \mathbb{Z}_q$$

$$a_1 = a^u$$
$$a_2 = g^u$$

$$c = \mathcal{H}(a_1, a_2, d_i, h_i)$$

$$r = u + cx_i$$

Output: (r, c)

Verifier

Input: (r, c)

$$c \stackrel{?}{=} \mathcal{H}(a^r d_i^{-c}, g^r h_i^{-c}, d_i, h_i)$$

Figure 2.6: Non-interactive sigma proof for the El-Gamal threshold decryption.

Prover

$$A = a^{1-2x} g^r, B = b^{1-2x} g^x h^r$$
$$u_1, u_2 \in_R \mathbb{Z}_q$$

$$a_1 = a^{1-2u_1} g^{u_2}$$
$$a_2 = b(b^{-2}g)^{u_1} h^{u_2}$$

$$c = \mathcal{H}(a_1, a_2, A, B)$$

$$r_1 = u_1 + cx$$
$$r_2 = u_2 + cr$$

Output: (r_1, r_2, c)

Verifier

Input: (r_1, r_2, c)

$$c \stackrel{?}{=} \mathcal{H}(a^{1+c-2r_1} g^{r_2} A^{-c}, b^{1+c-2r_1} g^{r_1} h^{r_2} B^{-c}, A, B)$$

Figure 2.7: Non-interactive sigma proof for private XOR.

Chapter 3

Approximate Matcher

3.1 Approximate Matcher protocol

The *Approximate matcher* is a *Profile matching* protocol that checks whether the Hamming distance of two encrypted bitstrings is within a given threshold T. The protocol does not leak any other information on the bitstrings.

We give the protocol in brief for n parties. We assume that the parties already posses threshold El-Gamal key shares.

1. Use exclusive or to the given bitstrings to get the different bits in them;
2. add the individual bits of the output of the XOR step;
3. compare the output of the addition step with the threshold. If it is less or equal to return true. Otherwise false.

An example of how those steps work, when the values are not encrypted is shown in Figure 3.1. For the real protocol, those steps have to be executed using encrypted values and thus the algorithms discussed in the previous chapter will be used.

In the following sections we give a detailed description of the protocol as well as the complexities that arise from a "by the book" implementation. In the next chapters we demonstrate that there is much room for improvement.

We also describe briefly some variants of the *Approximate matcher* protocol. Those apply for the cases where the bitstrings are known to some parties, or the threshold is also encrypted and combinations.

3.2 Description

The description of the algorithms give here is based on [32]. According to the authors the algorithms used in that paper were selected to minimize computational complexity versus round complexity. In our study we focus on this

Input: the bit strings 01011, 11111 and threshold is 1.

1. 01011 XOR 11111 = 10100
2. 1+0+1+0+0 = 2
3. $2 \leq 1$

Output: false

Figure 3.1: The *Approximate matcher* protocol on non-encrypted bit strings.

version of the protocol. For the following algorithms remember that the values are encrypted with homomorphic El-Gamal, and moreover we assume that the minimum required number of parties are participating in the protocol, i.e, the are equal to the El-Gamal threshold[1].

Input: Two m-bit encrypted binary strings, threshold T

$$[\![S_1]\!] = ([\![x_0]\!], [\![x_1]\!], \ldots, [\![x_{m-1}]\!])$$
$$[\![S_2]\!] = ([\![y_0]\!], [\![y_1]\!], \ldots, [\![y_{m-1}]\!])$$

Output: If HammingDistance$(S_1, S_2) \leq T$ then true. Otherwise false.

XOR step

In this step the goal is to compute the `exclusive or` of the input bitstrings. That is compute $H_i = (x_i \text{ xor } y_i)$, $0 \leq i \leq m-1$. The following algorithm from [31] is used to calculate the exclusive or of the encrypted values:

$$\begin{array}{rcl} \texttt{For all } 0 \leq i < m-1 & & \\ [\![v_i]\!] & = & [\![x_i]\!] \otimes_b [\![y_i]\!] \\ [\![H_i]\!] & = & [\![x_i]\!] \oplus [\![y_i]\!] \ominus (2 \otimes [\![v_i]\!]) \\ & = & [\![x_i \text{ xor } y_i]\!] \end{array}$$

The output is the encrypted bitstring:

$$([\![H_0]\!], [\![H_1]\!], \ldots, [\![H_{m-1}]\!])$$

As we can see one conditional gate operation is needed for each bit to be XORed.

Addition step

Here we want to evaluate the sum of the individual bits of the output of the XOR step. On the first sight this might look an easy step since addition is an

[1] We will make the reasons for this assumptution more concrete in Section 4.1.1 on page 33.

available operation in our cryptosystem, but the tricky part is the fact that we need[2] the output as an encrypted bitstring.

Thus we will evaluate the sum using the secure adder from [32]. That is we will add the individual bits of $[\![H_i]\!]$ in a way that they lead to a number in its encrypted binary form such as $([\![S_0]\!], [\![S_1]\!], \ldots, [\![S_s]\!])$.

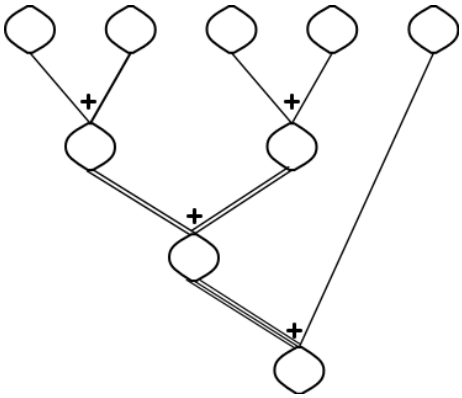

Figure 3.2: A parallelizable addition algorithm. The circles denote a value which can consist of any number of bits and the lines denote the number of bits to be added.

To perform the addition we will use an addition tree such as the one in Figure 3.2. We name the values to be added $[\![x_i]\!]$ and $[\![y_i]\!]$, $0 \leq i < k$, with k being the size in bits of the values to be added, initially that size would be one, but as we move on deeper levels in the tree the size of the bitstrings increases.

We can get an encrypted binary representation of their sum $[\![z_i]\!]$ with the following algorithm

$$
\begin{array}{lcl}
[\![c_0]\!] & = & [\![x_0]\!] \otimes_b [\![y_0]\!] \\
[\![z_0]\!] & = & [\![x_0]\!] \oplus [\![y_0]\!] \ominus (2 \otimes [\![c_0]\!])
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{For all } 1 \leq i < k & & \\
[\![w_i]\!] & = & [\![y_i]\!] \otimes_b [\![c_{i-1}]\!] \\
[\![c_i]\!] & = & [\![x_i]\!] \otimes_b ([\![y_i]\!] \oplus [\![c_{i-1}]\!] \ominus (2 \otimes [\![w_i]\!])) \oplus [\![w_i]\!] \\
[\![z_i]\!] & = & [\![x_i]\!] \oplus [\![y_i]\!] \oplus [\![c_{i-1}]\!] \ominus (2 \otimes [\![c_i]\!])
\end{array}
$$

Output: $([\![c_{k-1}]\!], [\![z_{k-1}]\!], [\![z_{k-2}]\!], \ldots, [\![z_0]\!])$

Apply this algorithm for each pair of bits of $[\![H_i]\!]$ starting with $[\![H_0]\!]$ and $[\![H_1]\!]$, then add their sums and so on. So we get a bitstring $[\![S_i]\!]$, $0 \leq i \leq s$ that has size[3] of $s = \log_2(m+1)$ bits.

[2] Because it will be used by the comparison algorithm which requires two bitstrings.

[3] Because the maximum value of the sum is m, and every number x such that $0 \leq x \leq m$ can be written using $\log_2(m+1)$ bits. Since we are dealing with encrypted values the result cannot be expressed with less bits than the maximum. That is because the result is not known, so it is not possible to remove the leading zeros.

This algorithm only applies when adding equal bitstrings. When non equal bitstrings are to be added, such as the case in the last step of Figure 3.2, the values of the smaller bitstring have to be eliminated at some point in the algorithm[4].

Comparison step

In this step we need to compare the output of the addition step with the threshold.

Input: $[\![S_0]\!], [\![S_1]\!], \ldots, [\![S_s]\!]$ and $T_0, T_1, \ldots, T_s$, $s = \log_2(m+1)$

$$[\![t_0]\!] = [\![S_0]\!] \otimes (1 \ominus T_0)$$

`For all` $1 \leq i \leq s$

$$[\![v_i]\!] = T_i \otimes [\![t_{i-1}]\!]$$
$$[\![t_i]\!] = [\![t_{i-1}]\!] \ominus [\![v_i]\!] \ominus [\![S_i]\!] \otimes_b ([\![t_{i-1}]\!] \oplus T_i \ominus (2 \otimes [\![v_i]\!]) \ominus 1)$$

The output is

$$[\![\omega]\!] = 1 \oplus ((-1) \otimes [\![t_s]\!])$$

Final step

After the comparison step the result is available, although as an encrypted value $[\![\omega]\!]$. For the parties to be able to read the output, an El-Gamal threshold decryption of $[\![\omega]\!]$ is performed. Since the output is binary they can verify the result by testing equality against powers of the subgroup generator.

3.3 Complexities

In the following paragraphs we try to summarize the different complexities of this protocol given a "by the book" implementation of it. Those complexities will be used as a basis to calculate the improvement of the optimizations discussed in the next chapters.

3.3.1 Computational complexity

In this section we will try to give the complexity of the *Approximate matcher* by counting the required number of conditional gates.

[4]It might seem tempting to insert some encryption of zeros to make the bit strings equal but that is not feasible since each party will calculate different encryptions of the same values, which is undesirable.

XOR step In the XOR phase a single conditional gate is required per operation, thus in total we need as many gates as the number of bits m of the input.

Addition step For the secure adder, a number of $(m-1)$ secure additions are required for the digits of an m-bit number to be added. For simplicity we will restrict our calculations for the case where m is a power of two $m = 2^{\mu}$. In that case on each level of the addition (see Figure 3.3) we have the same number of bits to be added. The additions done in the first level require 1 bit operations, on the second level 2 bits and so on. The total number of levels[5] is $\log_2 m$.

For 1 bit addition the adder requires a single conditional gate to be evaluated, 3 for 2 bits and in general $1 + 2(m-1)$ for m bit numbers.

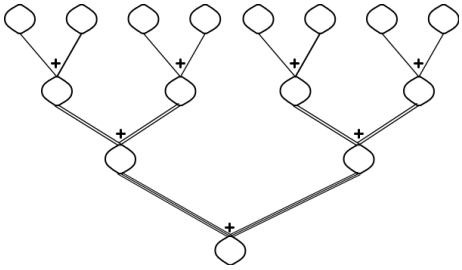

Figure 3.3: An addition tree of a number of bits that is a power of two.

Thus the total number of conditional gates required for the secure adder is

$$\underbrace{\frac{m}{2} \cdot 1}_{\text{level 1}} + \underbrace{\frac{m}{2^2} \cdot 3}_{\text{level 2}} + \ldots + \underbrace{\frac{m}{2^{\mu}} \cdot (2\mu - 1)}_{\text{level } \mu} =$$

$$= m \cdot 2^{-\mu}(3 \cdot 2^{\mu} - 2\mu - 3) =$$

$$= 3m - 2\mu - 3 = 3m - 2\log_2 m - 3$$

Since the output of the adder is in the range $[0, m]$ it should consist of maximum $\log_2(m+1)$ binary digits.

Comparison step In the comparison phase one conditional gate per bit is required. Thus it would be

$$\log_2(m+1) \text{ gates}$$

Conclusion For m bit numbers we get the number of required conditional gates, being

$$4m + \log_2(\frac{m+1}{m^2}) - 3 \text{ gates.} \tag{3.1}$$

[5] That is because on each step of the adder the numbers to be added are half of the previous step. Thus the adder finishes when $\frac{n}{2^k} = 1 \rightarrow k = \log_2 m$. And that k would be the number of levels.

Based on the complexity for a single conditional gate in Equation (2.9), the computational complexity with respect to exponentiations for n parties is

$$(4m + \log_2(\frac{m+1}{m^2}) - 3)(20n - 1)$$

3.3.2 Memory space requirements

In a straightforward implementation the maximum memory needed for the *Approximate matcher* is the memory needed by a single conditional gate. Thus the same values as in Section 2.5.4 apply.

3.3.3 Randomness complexity

The randomness complexity in the protocol is mainly due to the conditional gates calculations. Given that the *Approximate matcher* needs at most $4m + \log_2(\frac{m+1}{m^2}) - 3$ gates for m bit input and the calculations in Section 2.5.5, an upper bound to the random bits required is

$$\begin{aligned}&\text{(number of gates)} \cdot \text{(Randomness per gate)} \\ &= (4m + \log_2(\frac{m+1}{m^2}) - 3)((8\log_2 q) + 1) = \\ &= O(m \log q)\end{aligned}$$

As an example for an 160 bit subgroup and 32 bit input, to the *Approximate matcher*, a lower limit on the number of random bytes required to be retrieved for a *Approximate matcher* execution is 16900.

3.3.4 Round complexity

The number of rounds in the protocol heavily depends on the number of conditional gates to be evaluated. In the conditional gate the number of rounds required is tight to the number of players interacting. Typically we can have a round complexity that is the number of players plus one or $n + 1$. And based on the number of gates calculated the round complexity is

$$\begin{aligned}&\text{(number of gates)} \cdot \text{(round complexity per gate)} \\ &(4m + \log_2(\frac{m+1}{m^2}) - 3)(n + 1) = O(nm)\end{aligned}$$

3.3.5 Communication complexity

Given the number of the conditional gates to be evaluated and the facts from Section 2.5.7 we can find out the communication complexity of this protocol.

Clearly the number of bits to be exchanged is

$$\begin{aligned}(\text{Max number of gates}) \cdot (\text{complexity of each gate}) = \\ = (4m + \log_2(\frac{m+1}{m^2}) - 3)(n(6\log_2 p + 7\log_2 q)) = \\ = O(mn \log p)\end{aligned}$$

3.4 Variants of the Approximate Matcher protocol

The *Approximate matcher* protocol may apply, with minor modifications, to other cases where the threshold is encrypted or the bit strings are known, and combinations of these cases. To classify all the cases we assume the following possibilities for each value

- private: At least one party knows this value;
- protected: No single party knows this value; It is given encrypted.

With these cases in mind and by looking combinations of these in the bit strings and threshold, we can see that the *Approximate matcher* could even be optimized for some combinations.

We will try to briefly cover those cases in the following paragraphs.

3.4.1 One of the bitstrings is private

In this case there is at least one party that knows the private values of x_i[6]. Then the XOR step of the protocol can be optimized by letting this party calculate the XOR output by itself and provide a proof of correctness (see Figure 2.7 on page 21) to the rest of the parties. To calculate the output the algorithm below is used:

$$\begin{array}{lll} & \texttt{For all } 0 \leq i < m-1 & \\ [\![H_i]\!] & = & x_i \oplus [\![y_i]\!] \otimes (1 - 2x_i) \\ & = & [\![x_i \text{ xor } y_i]\!] \end{array}$$

The output is the encrypted bitstring:

$$([\![H_0]\!], [\![H_1]\!], \ldots, [\![H_{m-1}]\!])$$

We call this the Private XOR step. Since it avoids any use of conditional gates, this protocol is much more efficient and independent of the network's properties.

The addition step is the same as with the *Approximate matcher* since again the output of the XOR step is again an encrypted bitstring. The comparison step depends on the type of the threshold.

[6]The selection of x_i is arbitrary, that case also applies for y_i.

This case also applies when both bitstrings are private, as long as they are not known by the same party.

3.4.2 The threshold is protected

No matter what the previous steps looked like, when the threshold is protected only the comparison step needs to be adjusted. That is because in all cases we get an encrypted bitstring as output from the addition step. To do the comparison of the previous step output with the encrypted threshold $[\![T]\!]$ the following variation of the comparison algorithm is used:

Input: $[\![S_0]\!], [\![S_1]\!], \ldots, [\![S_s]\!]$ and $[\![T_0]\!], [\![T_1]\!], \ldots, [\![T_s]\!]$, $s = \log_2(m+1)$

$$[\![t_0]\!] = [\![S_0]\!] \otimes_b (1 \ominus [\![T_0]\!])$$

`For all` $1 \leq i \leq s$

$$[\![v_i]\!] = [\![T_i]\!] \otimes_b [\![t_{i-1}]\!]$$
$$[\![t_i]\!] = [\![t_{i-1}]\!] \ominus [\![v_i]\!] \ominus [\![S_i]\!] \otimes_b ([\![t_{i-1}]\!] \oplus [\![T_i]\!] \ominus (2 \otimes [\![v_i]\!]) \ominus 1)$$

The output is

$$[\![\omega]\!] = 1 \oplus ((-1) \otimes [\![t_s]\!])$$

That algorithm has an extra conditional gate evaluation per bit than the algorithm with plain T.

3.4.3 Semi-honest model

Although this study focuses on the case where malicious parties are participating in the protocol, this requirement is not always necessary. For example when the software used is verifiably tamper proof[7] we can trust the parties to calculate the messages correctly but we can't trust them not to infer something from the communication transcript and the data they obtain. In other words we consider only passive attackers and this is called the semi-honest case model. A formal definition of the model can be found in [23].

The *Approximate matcher* protocol can be more efficient in this model. That is because the Sigma Proofs are not needed any longer and thus for a conditional gate to be calculated all the operations for the proofs in Section 2.6.2 and Section 2.6.3 could be discarded. Given that the complexity of the conditional gate in exponentiations is (from Equation (2.9)) $20n - 1$ without the proofs will drop to

$$n + 6$$

[7] We will not discuss how to do this or whether this is possible at all, but assume it could be done.

which is a serious improvement. By extending this to the *Approximate matcher* complexity we have a complexity in exponentiations of (based on Equation (3.1))

$$(4m + \log_2(\frac{m+1}{m^2}) - 3)(n+6) \tag{3.2}$$

We can see that in this model the *Approximate matcher* in the two party case has $\frac{20 \cdot n - 1}{n+6} = \frac{39}{8} \approx 5$ times less exponentiations than the malicious model. Moreover the number of exponentiations of the private bitstring variant is further decreased because of the lack of proofs for the private XOR phase.

Chapter 4

Generic optimizations

In this chapter we will describe possible optimizations for the involved protocols. Our main focus will be the conditional gate, because it is the algorithm used most in the *Approximate matcher* and it has the highest complexity.

The optimizations found in this sector also apply to other protocols based on the conditional gate. By generalizing we could also argue that these optimizations could also apply to multi-party computations protocols that involve a high round complexity operation.

4.1 Single conditional gate

4.1.1 Reducing the number of parties

A quite obvious but important optimization of the protocol, without an impact on the security could be the following. Instead of performing the step 2 of the algorithm all the n parties, a threshold t could be added, so that only t parties interact on this algorithm. This speeds up the calculations and allows the protocol to be executed with a lower complexity if $t < n$. The interacting parties though, need to be able to perform an El-Gamal decryption, thus this number t should be at least the same number as the threshold in El-Gamal decryption[1].

In that case the communication complexity becomes

$$14t \log_2 p$$

and the computational complexity drops to

$$(20t - 10)$$

[1] By using n parties to perform the conditional gate, instead of t does not add to security (t is the El-Gamal threshold). That is because if t parties can conspire, they can gain advantage in the conditional gate calculation by being able to recover the input value x, even if the parties performing the calculation are $n > t$.

Thus both complexities now depend on t which is less than n. This justified our calculations for the complexities of the protocols for the case of the minimum possible participating parties, when $n = t$.

Of course with this optimization robustness[2] of the protocol is not lost. If a party cheats, it is detected, eliminated and replaced by another idle party (we assume that there are non active parties since $t < n$). This comes at the cost of an additional run of the algorithm.

4.1.2 Multiple exponentiations

In several cases such as the sigma proof generation and verification the multiplication of two elements raised in some exponent is required. That could be something like $g^x h^y$ or $a^x b^y c^z$ and so on. These are called multiple exponentiations and the time to compute them can be reduced by using algorithms such as the ones described in [34]. In the context of this project we will not use nor discuss further the effect of this optimization. Some reasoning behind this decision is given in the last paragraph of Section 4.2.2.

4.2 Multiple conditional gates

In the case of the evaluation of multiple conditional gates, for n parties, several optimizations can be applied. The goal of these optimizations is to minimize the total execution time of the gates to be computed. The interactive nature of the conditional gate, introduces idle time while a party is waiting for the results of the others to be received. We show that by efficient scheduling of the operations and applying some precomputations, improvement of the overall complexity can be achieved.

4.2.1 Parallel initiation

When having many independent conditional gates to be evaluated there is no reason to initiate all of them by the same party. Moreover by doing it, the other parties will have to remain idle waiting for the output of his calculations. A better approach is to divide the gates among all parties, thus all of them could start computing immediately. This has the advantage that no party is left idle and thus execution time is minimized[3].

For G gates to be evaluated and n parties this method scales well since it parallelizes execution and may reduce the running time by the number of available parties for computation (when $G > n$). Let's check that in detail. Normally, without any optimizations, the time needed to execute the G gates in a serialized way would be nG time slots, and a time slot is the time each party needs

[2]This is the resilience of the protocol in respect with dishonest parties. For a detailed discussion see [31].

[3]This reduces the time required for computation of a gate when the computational power of the parties is similar and the gates are equally distributed. When a party is very slow comparing to the others he becomes a bottleneck, no matter how the gates are initially divided.

to evaluate his part of a conditional gate, which we assume it's the same for everyone[4].

When this optimization is applied the running time becomes $\mathcal{T} \approx \frac{nG}{n} = G$, assuming that the conditional gates to be evaluated can be equally divided to all parties. The effect in the two party case for two conditional gates can be seen in Figure 4.1. Remember that in this case 4 messages are required to be exchanged per conditional gate.

It is important to note that since the running time of this method is actually independent of the number of parties performing it, by increasing the number of parties from n to $(n + k)$ while maintaining the number of gates, there is no increase in the execution time of the optimized case, whereas in the sequential execution the time is increased by (kG) slots.

If the gates cannot be divided to all parties we use the following trick. G can be written as $G = kn + \lambda$, with $\lambda < n$. The first part kn can be equally divided to all parties thus takes nk time slots. λ is less than n thus it needs n time slots to be calculated. So to sum up for a random $G = kn + \lambda$ we have execution time of

$$\mathcal{T} = (k + 1)n = \lceil \frac{G}{n} \rceil n \tag{4.1}$$

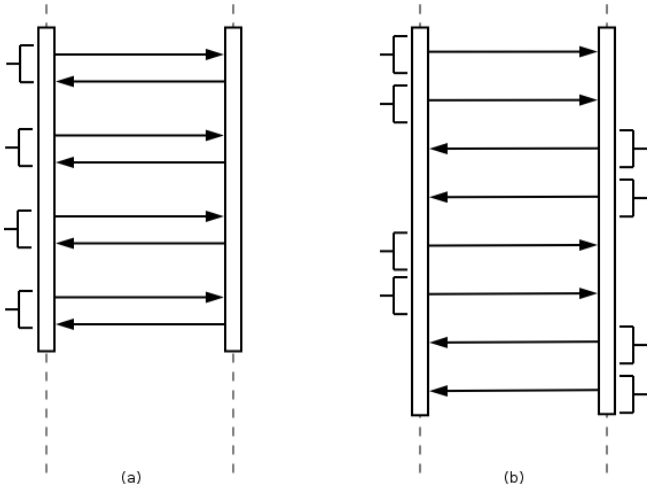

Figure 4.1: The effect of parallel initiation (a) in reducing time slots required compared to sequential execution (b). The brackets represent a single time slot.

4.2.2 Precomputations

Random re-encryption and the sigma proofs require several exponentiations that could be precomputed. For example for a random re-encryption several numbers of the form (g^r, h^r), where r is random in $\mathbb{Z}_q$ are used. Since g and

[4]Here we are consistent with our definition of running time, but we introduce parallelism, thus we count time slots instead of operations to get a perception of the running time.

h are known, they could be computed during the CPU's idle time by a worker thread. Similarly, the proofs require numbers of the form g^{u_1}, h^{u_2}, where u_1, u_2 are random in $\mathbb{Z}_q$. Thus, if precomputed, these random numbers can be later on requested, with no cost whatsoever. Moreover given that most operating systems produce random numbers using a Pseudo Random Number Generator, the effect of running the PRNG during the protocol execution is also reduced.

Let's see then how many numbers are required by this generator to produce. Each gate requires

- 2 pairs (g^r, h^r) for the random re-encryptions;
- 1 number $g^{s_i} h^r$ for a Pedersen commitment;
- 1 number g^{u_1} for the El-Gamal decryption;
- 2 pairs (g^r, h^r) and a number h^{u_2} for the conditional gate proof P_i (see Section 2.6.2).

To allow easy reference to these random number pairs we name and summarize their format in Table 4.1.

name	format	constrains
re-encryption pair	(g^r, h^r)	$r \in \mathbb{Z}_q$
Pedersen commitment number	$g^s h^r$	$s \in \{-1, 1\}$, $r \in \mathbb{Z}_q$
El-Gamal decryption number	g^r	$r \in \mathbb{Z}_q$
conditional gate proof number	h^r	$r \in \mathbb{Z}_q$

Table 4.1: The random numbers used in precomputations.

Thus per gate we need 4 re-encryption pairs, 1 Pedersen commitment number, 1 El-Gamal decryption number and 1 conditional gate proof number.

Of course this generator does not need to calculate everything in advance. It could work in parallel with the protocol execution and generate the numbers using the CPU's idle time. Note also that in most of the cases the generator need not only to return the base raised to the random exponent result, but also the random exponent as well[5].

When this optimization is applied the computational complexity of the conditional gate becomes

$$(20n - 10)$$

because now we have 9 exponentiations less per conditional gate compared to the original complexity of Equation (2.9).

Combining with multiple exponentiations In cases were a random number is multiplied with an other one, such as $g^r h^x$, where g, h, x are known and

[5]For example in the Sigma proofs both u_1 and g^{u_1} are required.

r is random, it is questionable which optimization between precomputations or multiple exponentiations is more efficient to calculate the output.

With precomputations, g^r is available on demand with no cost, thus a single exponentiation for h^x and a multiplication would be required, whilst if we use multiple exponentiations, a random number request and a double exponentiation are required. Thus in this particular case, using the precomputations method seem more efficient, although a definite answer to this problem can only be given by studying the specific complexities of the algorithms used for random generation, exponentiation and multiplication. This was outside the scope of this project.

4.2.3 Combining outputs

While evaluating several independent conditional gates, it might be wise to reduce the number of rounds needed for conditional gates as much as possible. This could be done by every party by combining the intermediate outputs of all the conditional gates in one packet and broadcasting it.

Although it might seem that this could be a potential waste of resources since no parties can begin calculations before the first "big" message is transmitted. However this is a very useful optimization for any protocol, where the number of conditional gates that need to be evaluated is a number much larger than the number of parties. That is because we can combine this optimization with parallel initiation and thus no parties will remain idle.

This optimization depends not only on the number of gates that each party has to evaluate, but mainly on the number of gates that are available to be evaluated each point in time. This distinction might not be clear, but there are cases where some gates depend on previously calculated data, hence they cannot be initiated before the currently evaluated gates are fully executed.

Round complexity

If implemented naively the round complexity of m conditional gates to be evaluated by n parties will be (using Equation (2.12))

$$\begin{aligned}(\text{number of gates}) \cdot (\text{complexity of each gate}) = \\ m(n+1) = O(nm)\end{aligned}$$

Whereas the round complexity of this optimization is

$$n + 1 = O(n)$$

Compared to the naive implementation the effect of this optimization is quite impressive. That is because the round complexity for k gates is the same as the round complexity of a single conditional gate.

On the other hand if this optimization is combined with parallel initiation then the round complexity increases since each party initiates in parallel a different

set of the gates, thus we have complexity of

$$n(n+1) = O(n^2)$$

But in both cases of this optimization the round complexity depends only in the number of participants, and is independent of the number of gates that are evaluated.

Memory space requirements

On the other hand, the memory space required for this optimization increases. The amount of the increase is directly proportional to the number of gates that are being calculated. Thus for m gates the memory required is

$$\begin{aligned}(\text{number of gates}) \cdot (\text{memory space per gate}) = \\ = m(15n - 14)\log_2 p = O(n \log p)\end{aligned}$$

Chapter 5

Approximate Matcher optimizations

In the next sections we discuss ways of improving the execution time of the *Approximate matcher* protocol. Since the major component of this protocol is the conditional gate, the optimizations discussed in the previous chapter directly apply. In that case we discuss their effect on this protocol and calculate the improvement in complexities.

5.1 Parallel initiation

The *Approximate matcher* protocol allows for a level of parallelism of operations between parties, thus allowing optimizations such as the Parallel initiation described in Section 4.2.1. We will study this optimization's effects on each phase of the *Approximate matcher* protocol.

XOR step In the XOR phase several independent conditional gates have to be evaluated making them suitable for parallel initiation. In the case of an m-bit string, m gates need to be calculated. By using the result from Section 4.2.1 we get an execution time for the XOR phase of

$$\lceil \frac{m}{n} \rceil n$$

where T is the amount of time needed by a single party to perform the conditional gate.

Addition step The secure adder, if implemented properly allows for a level of parallelization. Instead of summing the first two bits, adding to their sum the third, and so on (see Figure 5.1), a better algorithm that is suited for parallel initiation is needed. This is achieved, by applying summation to distinguished

pairs of bits, and then adding their sum (see Figures 3.2 and 3.3). In that case the time needed for the adder would be

$$\sum_{i=1}^{\mu}(\lceil\frac{m(2i-1)}{n2^i}\rceil n)$$

The formula above is derived by summing the conditional gates per level in the addition tree[1] and applying Equation (4.1).

Comparison step The third part of the algorithm, the secure comparator cannot be parallelized. That is because every step of it depends on previous outputs, thus forbidding parallelization. Thus, given the input is $\log_2 m + 1$ bits and we need one conditional gate per bit, we have running time of

$$n \log_2 (m+1)$$

Conclusion In total the running time would be

$$\begin{aligned}
\mathcal{T} = \underbrace{\lceil\frac{m}{n}\rceil n}_{\text{XOR}} + \underbrace{\sum_{i=1}^{\mu}(\lceil\frac{m(2i-1)}{n2^i}\rceil n)}_{\text{addition}} + \underbrace{n \log_2 (m+1)}_{\text{comparison}} = \\
= n(\lceil\frac{m}{n}\rceil + \log_2 (m+1) + \sum_{i=1}^{\mu}\lceil\frac{m(2i-1)}{n2^i}\rceil) \approx \\
\approx (m + n \log_2 (m+1) + \sum_{i=1}^{\mu}\frac{m}{2^i}(2i-1)) = \\
= (m + n \log_2(m+1) + (-3 + 3m - 2\log_2 m)) = \\
= (4m + n \log_2(m+1) - 2\log_2(m) - 3) = \\
= O(m + n \log(m+1))
\end{aligned}$$

We can see that the term which depends on the number of parties, $n \log_2(m+1)$, is due to the comparison step which is sequential and the other terms that are independent of the number of parties are due to XOR and addition steps. Thus we see that as parties increase the comparison step becomes the bottleneck while as the bitstrings increase the bottlenecks are the XOR and addition steps.

But let's now see at which at which level the above can improve the efficiency of the *Approximate matcher*. We try the fraction

$$\frac{\text{(running time with parallel initiation)}}{\text{(running time without parallel initiation)}}$$

[1] See the discussion in Section 3.3.1 for how the gates per level are calculated.

which will give the improvement in execution time. This is

$$\frac{(4m + n\log_2(m+1) - 2\log_2 m - 3)}{(4m + \log_2(m+1) - 2\log_2 m - 3)n} =$$
$$= \frac{(4m + \log_2(m+1) - 2\log_2 m - 3) + (n-1)\log_2(m+1)}{n(4m + \log_2(m+1) - 2\log_2(m) - 3)} =$$
$$= \frac{1}{n} + \frac{(n-1)\log_2(m+1)}{n(4m - 2\log_2 m + \log_2(m+1) - 3)}$$

and by taking the limit of m to infinity we have the improvement level for large bit strings. That would be

$$\lim_{m\to\infty}(\frac{1}{n} + \frac{(n-1)\log_2(m+1)}{n(4m - 2\log_2 m + \log_2(m+1) - 3)}) = \frac{1}{n} \tag{5.1}$$

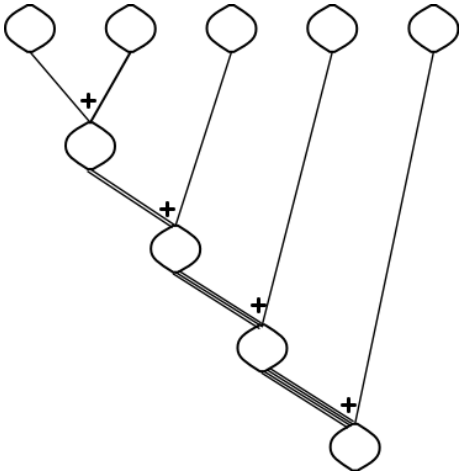

Figure 5.1: A sequential addition algorithm.

Thus we see that this scheduling optimization can be up to n times faster, for n parties, without any change in the computational complexity.

5.2 Precomputations

Since the operation used most during the *Approximate matcher* protocol execution is the conditional gate (see Section 4.2.2) we expect precomputations to increase the execution speed of the *Approximate matcher*. The increase factor depends directly on the number of gates that need to be calculated (see Section 3.3.1), although the way precomputations are applied should not be disregarded. When precomputations occur long time before, the complexity of each conditional gate drops to the level seen in Section 4.2.2. Otherwise, when they occur during the protocol execution by utilizing idle time, their effect heavily depends on the availability of idle time.

random number type	amount needed
re-encryption pair	$\frac{m}{2} \cdot 4 = 2m$
Pedersen commitment number	$\frac{m}{2}$
El-Gamal decryption number	$\frac{m}{2}$
conditional gate proof number	$\frac{m}{2}$

Table 5.1: The amount of numbers that need to be delivered by the random number generator for each iteration of the XOR step and the first level of the addition step.

Two party case

Here we will elaborate more in the two party case where the random generator starts simultaneously and runs in parallel with the protocol. In that case we will focus on the random number requirements between I/O time. The I/O time acts as idle time for the parallel random generator since almost no CPU time is used at that point.

For m-bit input to the *Approximate matcher* protocol we expect each party to calculate $\frac{m}{2}$ gates for each iteration[2] of the XOR step, when parallel initiation of the gates is used. Thus given the needs per gate (shown in Section 4.2.2) the amount of numbers that the random generator should compute in between these steps is shown in Table 5.1.

In the addition step the maximum number of parallel executed conditional gates is on the first level, where we have $\frac{m}{2}$ gates at a time. Thus the table for the needs of XOR step still applies. In the comparison step we have only a single gate to be calculated at a time.

So for the *Approximate matcher* protocol an ideal random generator should be able to deliver the amount of numbers shown in Table 5.1 at once when requested. That means that this amount of numbers should be generated during I/O and idle time.

5.3 Combining outputs

Round complexity

As described in Section 4.2.3, it is possible to reduce the number of rounds of the executed conditional gates by combining their outputs and transmitting them in a single message. This of course requires a level of parallelism in the calculated gates. We try to calculate the effect of this optimization on the *Approximate matcher* protocol.

XOR step The XOR step can make heavy use of this optimization. The fact that all the conditional gates are independent allows for their intermediate

[2]By iteration we mean the point where both parties have to communicate to calculate the conditional gates.

results to be combined. Thus the number of rounds for the XOR step drops to the number of rounds of a single conditional gate, which is $n+1$ for n parties. That is independent of the size of the input.

Addition step The number of gates that are executed in parallel for m-bit input, when m is 2^μ, depends on the level of the addition tree. It has been calculated in the Computational complexity section (Section 3.3.1) that we have $\log_2 m$ levels of the addition tree and on each level k there are $\frac{m}{2^k} \cdot (2k-1)$ conditional gates to compute. At each time only $\frac{m}{2^k}$ can be executed simultaneously. That means that the number of rounds for the addition step would be

$$\begin{aligned}
&\text{(rounds per gate)(number of gates)} = \\
&= (n+1)(\underbrace{1}_{\text{level 1}} + \underbrace{3}_{\text{level 2}} + \ldots + \underbrace{(2\mu-1)}_{\text{level } \mu}) = \\
&= (n+1) \cdot (\mu^2) = \\
&= (n+1)(\log_2^2 m) \text{ rounds}
\end{aligned}$$

Comparison step During the comparison step all required conditional gates are dependent thus the use of this optimization does not improve anything. The number of rounds is again

$$(n+1) \cdot (\log_2(m+1))$$

Conclusion Thus the total number of rounds, for n parties and m-bit strings, when using this optimization would be

$$\begin{aligned}
&\underbrace{n+1}_{\text{XOR}} + \underbrace{(n+1)(\log_2^2 m)}_{\text{addition}} + \underbrace{(n+1) \cdot (\log_2(m+1))}_{\text{comparison}} = \\
&= (n+1)(\log_2^2 m + \log_2(m+1)) = \\
&= O(n \log^2 m)
\end{aligned}$$

This is quite an improvement of the original version which had round complexity of $O(nm)$.

Memory space requirements

The memory needed for the *Approximate matcher* with the above optimization depends basically on the number of gates to be calculated simultaneously. The XOR phase is the phase with the most independent conditional gates to be calculated thus this would be the most memory consuming. If we assume k-bit strings, n parties and ω the number of gates that each party can calculate in parallel, then each party has to calculate at once at most

$$[\frac{k}{n\omega}] \text{ gates}$$

Thus

$$\begin{aligned}(\text{Max simult. number of gates}) \cdot (\text{Memory needed per gate}) = \\ = [\frac{k}{n\omega}](15t - 14)\log_2 p \text{ bits}\end{aligned}$$

This value is divided by the number of parties when parallel initiation is applied and is increased by the memory cache for precomputations.

5.4 Combining optimizations

In several cases combining optimizations might be negative for some factors. For example the parallel initiation method when combined with the precomputations method might not improve performance as expected. That would be because precomputations depend on the idle time of the protocol, and with parallel initiation the idle time is dramatically reduced.

The same applies on the mixing of the parallel initiation with the round reducibility. As we saw in the previous chapter (see Section 4.2.3) we expect the number of rounds to increase at maximum by a factor of the number of players.

Chapter 6

Architecture of the software

For the purposes of this project we developed an implementation of the algorithms and the protocols discussed in the previous chapters. Here we discuss the requirements we set for this implementation and the high level architecture of it.

6.1 Requirements

The main purpose of this implementation is to act as a tool for evaluation of the performance of the underlying algorithms. Those algorithms are to be implemented only in the 2-party setup, however that should be in way that it can be easily extended for the n-party case. All the requirements that were set for this implementation are listed in a formal detailed format below.

1. Implement El-Gamal threshold key generation for 2 parties.
2. Implement El-Gamal key encryption.
3. Implement El-Gamal threshold key decryption for 2 parties.
4. Implement the precomputations optimization to be used in multiple conditional gates evaluation.
5. Implement the parallel initiation optimization to be used in multiple conditional gates evaluation.
6. Implement the Conditional gate for 2 parties.
7. Implement the *Approximate matcher* for 2 parties that both of them holds an equal length bitstring.
8. The threshold should be public or given in encrypted form.
9. The communication protocol between the parties is TCP/IP in a peer to peer connection.
10. The communication protocol should be defined in an easy to extend protocol, such as ASN.1 or XML.

11. No external threats are to be considered.
12. Be able to handle a second application of profile matching, and demonstrate this by implementing the variants of the *Approximate matcher*.
13. Be designed in a way that it is easy to reuse components.
14. Run on the `Win32` platform.

In addition to these requirements some additional functionality was added, during implementation time, to fulfill the needs for the testing phase. Those were

- precise timing of the *Approximate matcher* protocol;
- calculation of the application's idle time;
- usage statistics of the random cache;
- unit testing and benchmarks.

6.2 Threats

The threat that our implementation has to cope with, is Cheating parties that participate in the protocol.

No external threats are to be considered, such as attackers that cause a denial of service attack, attackers that remove, add or replay messages from the stream. That is because the protocol is assumed to run over an already secure layer, such as TLS. Although this implementation will not include this feature, it is assumed to be provided in a real world deployment.

As a future extension possibility we note that the structure of the *Approximate matcher* protocol allows for integration with a security layer. That is because of the initial step which is key generation, the peers are provided with private and public key shares, that they can use later on for secure communications. Thus the implementation should not forbid a future extension to support a secure communications layer.

6.3 Use cases

What is our program expected to offer to a user? A list of use cases is given in Figure 6.1. As we can see the user is expected to interact with the program in order to provide the required data for the *Approximate matcher* protocol. In addition he is expected to interact to enable functionality such as start the key exchange, or start comparing the binary strings (profiles) with the peer. The interface should try to avoid confusions and prevent illegal combinations of actions.

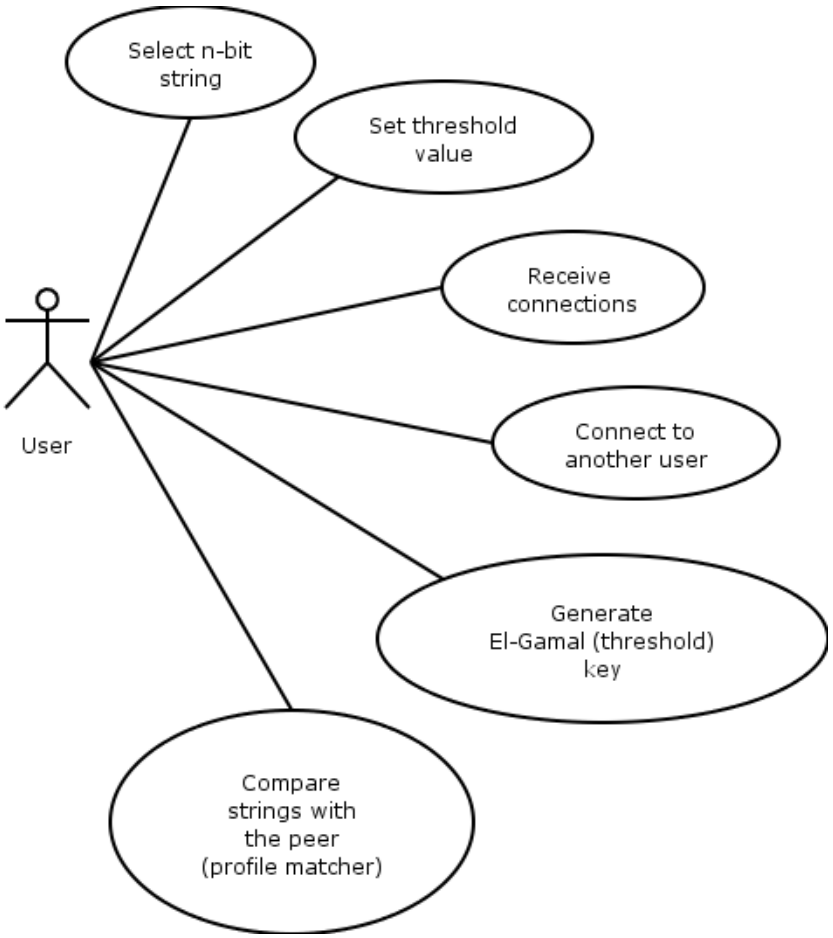

Figure 6.1: The use cases.

6.4 Architectural views

Although this is a small implementation of a protocol we decided to write down an architectural description of it. That is mainly because parts of this system are expected to be reused and modified to accomodate for different algorithms, such as replacing the El-Gamal cryptosystem, and different protocols, i.e., using the conditional gate for a different protocol than *Approximate matcher*. For this to be achieved a written description of the application will help future implementors get an overview of the application without digging into source code.

In this section we will make use of the ideas in [16] to present different views of the design of the software. A brief description of the views that we are going to use is:

- The *conceptual* architectural view, which models the application as a collection of decomposable and interconnected conceptual components;

- The *module* architectural view, which captures the concurrency and synchronization aspects of the design.

We skip the *Execution* view which is obvious in our design. The use cases of the software will also be discussed to show the correspondence between the views and make the design decisions clear.

In the following subsections we will discuss the architectural design of the *Approximate matcher* application in parallel with the rationale that lead to this design decisions. More thoughts on the rationale and decisions will be discussed in Section 7.1.

6.4.1 Conceptual Architecture view

In this view we see the application as an entity composed by different components that are assigned different tasks. We will not focus into details but rather give a brief overview of the architecture.

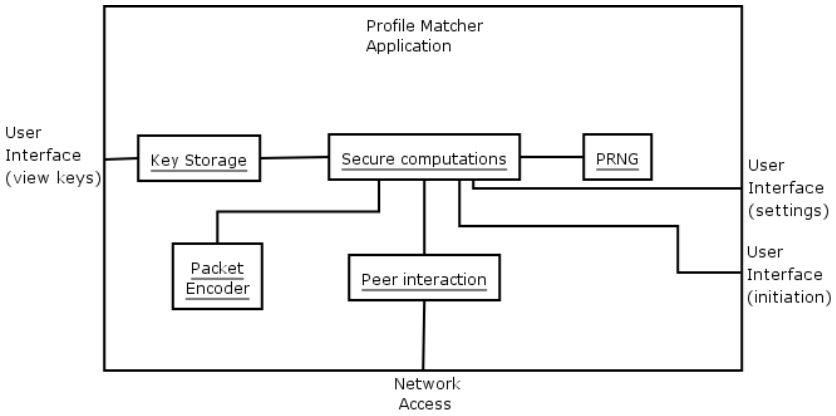

Figure 6.2: A high-level configuration of the *Approximate matcher* application. The boxes represent components and the lines interaction between them.

Initially we define the following components (see Figure 6.2) of the application:

- Secure Computations component;
- PRNG component;
- Peer interaction component;
- Key Storage;
- Packet Encoder component.

These are also the key aspects of the application. We need some user input to set the data for the *Approximate matcher*, set the address of the peer and view the peers that a key exchange with them has already occurred.

A refined version of the application is shown in Figure 6.3. In that figure it should be clear that concepts are described and not actual classes or instantiated

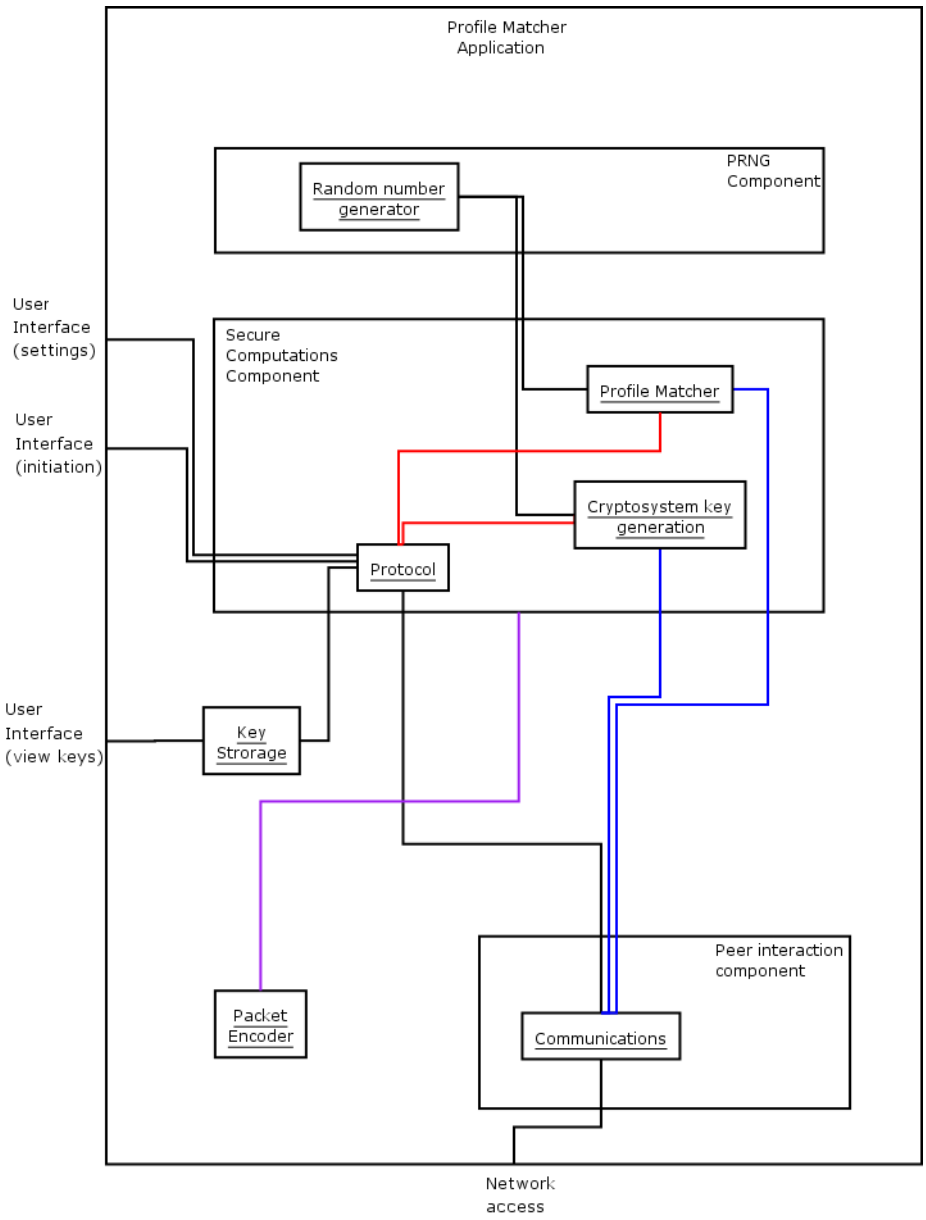

Figure 6.3: An initial high-level configuration of the *Approximate matcher* Application. The boxes represent components of the system and the lines correspond to interaction points. Colored lines have been used to make interactions distinct and serve no other purpose.

objects. Each concept may not directly correspond to a class. The mapping will be described in the Module Architectural view section.

The Secure Computations component

The Secure Computations component is the main element of the application. It contains the components that implement the *Approximate matcher* protocol. The The component named Protocol is used to gather the required data set by the user interface, perform an initial handshaking with the peer and check whether a key exists using the Key Storage component. Based on the initial negotiation it starts the *Approximate matcher* protocol or initiates a key exchange.

The Profile matcher component takes control once the Protocol has setup the parameters for the connection (such as key exchange and user input). It's purpose is to make the internal calculations for the *Approximate matcher* protocol. It is connected to the Conditional Gates which is responsible for making the computations with the peers, thus needs to use the Communications component.

The Cryptosystem key generation is used by the Protocol to generate a shared key with the peer if this doesn't already exist. It makes use of the Communications component to interact with the peer.

The PRNG component

The PRNG component is responsible to generate random numbers that are required by other components. It is intended to run in parallel with the other components in order to be able to do precomputations of the random numbers. This will allow faster delivery of the requested numbers.

The Communications component

Finally there is the communications component. It handles the actual transmission and reception over the network, as well as the packet encoding by using the Packet encoder component. The Key storage component will store and retrieve private and public keys corresponding to the cryptosystem.

6.4.2 Module Architecture view

The purpose of this section is to define modules and organize them into layers. Layers provide independence between parts of the system, so that a change in the layer doesn't affect the whole system. Modules can interact with other modules within the same layer. If interaction is required between layers then the layer should provide the interface.

An initial mapping of the components into the layers is given in Figure 6.4 and a mapping of the components into subsystems and modules[1] follows in Table 6.1. In the following sections we will expand the components into modules and subsystems.

[1]The subsystems are modules in practice, but we want to make clean that they are independently executed, and may even run on different CPUs.

Component	Module	
Random Number Generator	PRNG	Subsystem
Key Storage	KeyStorage	Module
Profile Matcher (protocol)	ProfileMatcher	Subsystem
Protocol	Protocol	Module
Cryptosystem Key Generation	CryptoSystemKeyGeneration	Module
Profile Matcher (algorithms)	ConditionalGateCalculator	Module
Packet Encoder	PacketEncoder	Module
Communications	Communications	Module

Table 6.1: Mappings of components to modules.

The main idea is to separate functionality into independent layers so usage can be restricted to public interfaces. We also focus on the algorithmic aspect of the design so we will keep out the description of the interaction with the user interface. The detailed diagram with the layers is shown in Figure 6.5 and the interdependencies between them can be seen in Figure 6.4.

The interfaces of the *Approximate matcher* application are described and explained in [24].

Protocol layer

To accommodate the *Approximate matcher* protocol we create the Protocol. It contains the `Profile Matcher` subsystem and the Protocol Module. The `Protocol` module is responsible for the initial setup of parameters with the peer, the key generation, if needed, and the initiation of the *Approximate matcher*. It relies on the `ProfileMatcher` module, the `KeyStorage` and the `Communications` modules using their layer interfaces.

The `ProfileMatcher` subsystem consists of the `ProfileMatcher`, `Adder`, `XOR` and `Comparator` modules that implement the actual algorithms. They all depend on the Algorithms Layer for the Conditional gates calculations.

Algorithms layer

The Algorithms layer holds the crypto functionality including the Conditional Gates and the cryptographic protocols. Those are split into the `CryptoSystem KeyGeneration`, `CryptoSystemComputations` and the `ConditionalGatesCalculator` modules. All of these modules depend on the functionality provided by the PRNG layer and both the `CryptoSystemKeyGeneration` and `ConditionalGatesCalculator` depend on the interface of the BackEnd layer to access the communication channel.

PRNG layer

This layer consists only of the `PRNG` module that provides access to random numbers. It is to be executed in a separate thread than the main application

in order to be able to perform precomputations needed by the underlying cryptographic protocols. The name PRNG was chosen because that functionality need not to be known by the other modules that make use of it, thus they can just consider it as a random number provider.

Key Storage layer

This layer with the `KeyStorage` provides the functionality to access past keys exchanged with a peer, and store new ones.

Communications layer

This layer provides access to the system level functionality of Networking to the upper layers.

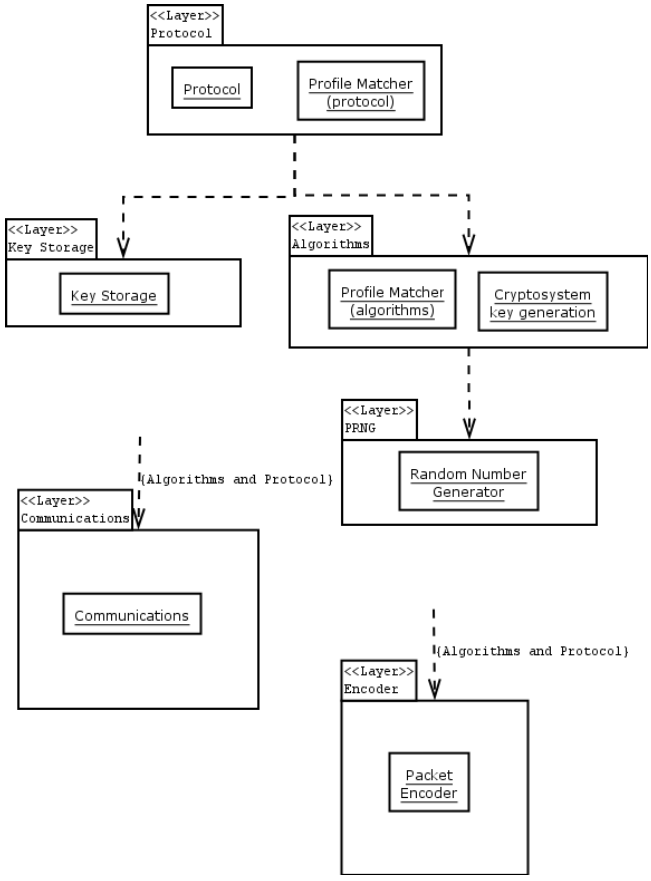

Figure 6.4: An initial creation of layers based on the components of the conceptual view.

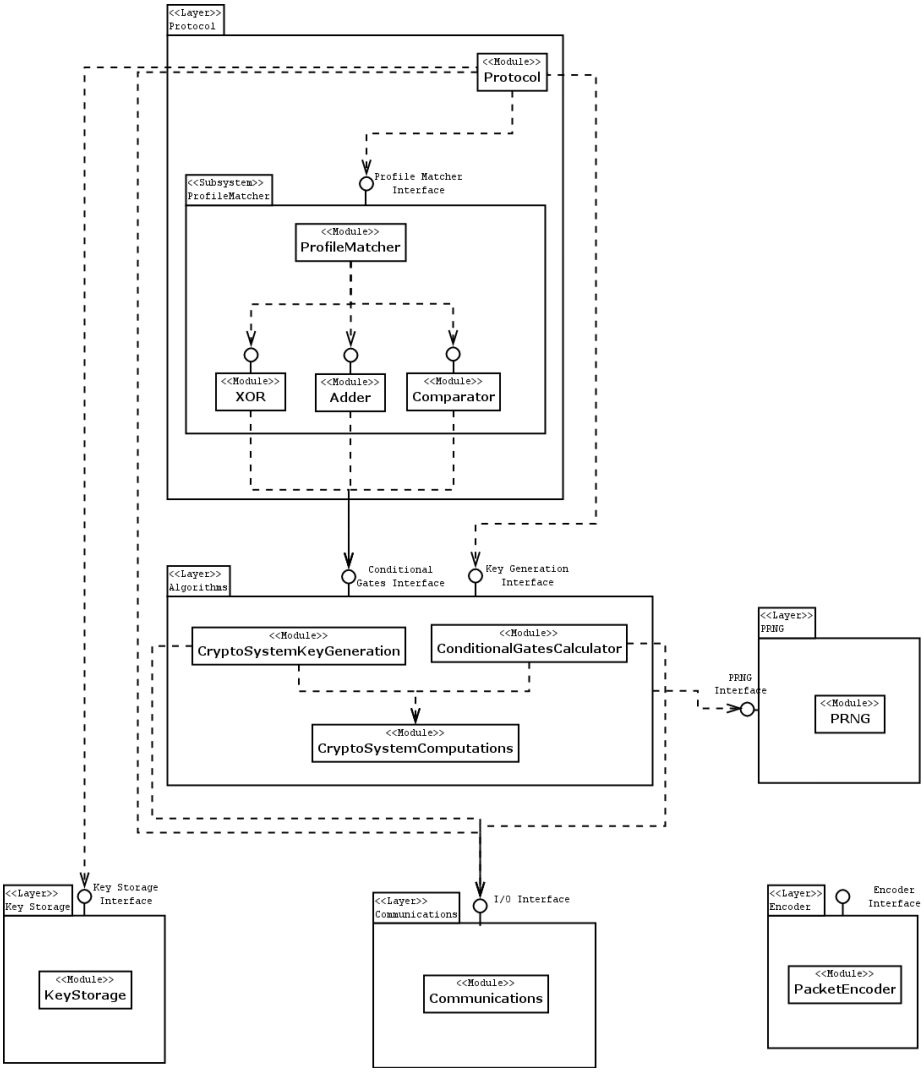

Figure 6.5: The Layers with the modules and subsystems.

Chapter 7

Implementation issues

In the following sections of this chapter, we give an overview of the most important design decisions and the rationale behind them. In addition in Section 7.2 we describe the communication protocols involved.

7.1 Modules and subsystems

7.1.1 Random Number Generator

This protocol like most of the cryptographic protocols assume a random source capable to provide a quite large amount of random numbers. However this is not trivial in modern operating systems. Different systems provide different interfaces with different semantics to provide random bytes. For example the `Win32` API provides a system call `CryptGenRandom` to generate random bytes, that depends on a cryptographic provider context (for no obvious reason). However nothing about the details of this generator are explained in the function's description [5] except for

> *This function is often used to generate random initialization vectors and salt values.*

which does not offer a warm feeling. Other operating systems such as Linux and FreeBSD provide special files in the operating system such as `/dev/random` and `/dev/urandom` to offer random bytes. Even though in both of them these files are linked with an entropy gatherer within the kernel, that times hardware interrupts, network usage etc., the interface is still non uniform. For example Linux's `/dev/random`[14] blocks when it thinks that the entropy within the kernel is not enough, whilst FreeBSD's `/dev/random`[26] does not have this limitation. There are several discussions on which behavior is better (see links from [3]) and both sides seem to be passionate enough[4].

The bottom line is that random bytes are not easy to obtain, and even if somebody obtains them there is no assurance on the randomness. One can only be assured on the way they are produced. Relying on the entropy counters,

such as the ones in the Linux kernel, does not provide assurance too[14]. The FreeBSD's approach and the approach in [3] seem to be the most promising since they utilize algorithms such as [17] and [29, Fortuna] that are designed to cope with low entropy pools and even with input to the PRNG that comes from malicious attackers.

Our implementation is expected to function on the Windows platform thus we will not use anything else except the `Win32` API for the PRNG and ignore the fact that we know nothing about it. This decision should be reconsidered for a real world implementation. On the `GNU/Linux` system we make use of the `/dev/urandom` device.

Parallel generation

To implement the optimizations based on precomputations a way to parallelize the random number generator was required. For this we designed an interface to access random numbers in the specific forms described in Section 4.2.2. That way the precomputations component was only visible within the random generator back end. The callers had no idea whether it was used or not.

We used a low priority worker thread to compute random numbers in the required forms while the protocol was executing. Thus we were effectively using the idle time of the CPU and the time of the I/O for the precomputations. The average size of the cache for the precomputed numbers was made configurable, although different types of numbers had different size of cache based on their usage ratio[1]. When the generator was asked for a number but didn't have any in the cache, it was generating one on the fly in the high priority thread.

7.1.2 Profile Matcher (algorithms)

Side-channel attacks

Although not part of this study, we believe it is important to give some hints about the possibility of side-channel attacks in the involved protocols. That is because even if the protocols used are proved not to leak any other information that the output, this is done using models that do not consider other attacks except the ones possible in the model. In the following paragraphs we list briefly some first observations on the possibility of side channel attacks, and most specifically timing attacks, which are important due to the interactive nature of this protocol[2].

As it can be seen in Section 2.3.2 because of the El-Gamal encryption all the operations between encrypted data are independent of the plaintext value. This prevents side channel attacks that observe the timing of operations.

However operations that involve unencrypted values such as the addition and multiplication with elements in $\mathbb{Z}_q$ take up time that is affected by the input's

[1] which based on the calculations in Section 5.2, verified by measurements.

[2] One can easily detect when the calculations were finished by checking the time the next reply was sent.

value. Thus this may be a point to be investigated for a real world implementation.

The conditional gate operation (Section 2.5) in the blinding step, where a multiplication with a value from the $\{1, -1\}$ range is required, care must be taken not to provide any information of this value. That is because a multiplication with 1 is a no-op, whilst a multiplication with -1 requires an inversion. A way might be to perform a multiplication with -1, if this wasn't the selected number, and discard the output.

Moreover while testing the output of a decryption of a homomorphic encrypted value, care must be taken not to leak the result by spending more time in some results than others.

7.1.3 Profile Matcher (protocol)

Conditional Gate Worker

The *Approximate matcher* protocol involves subprotocols that depend on several calculations on El-Gamal encrypted values. All of the calculations except for the Conditional Gate can be easily performed without any interaction between the calculating parties. The Conditional Gate on the other hand needs (for two parties) a two round message exchange involving 1 exchange per round. Instead of executing each gate separately we implement the optimizations discussed in Section 4.2. That is the *Parallel initiation* and the *Combining outputs*. That way, since the *Approximate matcher* involves a large number of Conditional Gates, we keep the message exchange to the minimum possible.

The component that handles the calculations for conditional gates is called `ConditionalGateWorker` and is intended to be used as depicted in Figure 7.1. It is initially filled in with a number of gates and as soon as no more gates are expected it starts executing them as shown in Figure 7.2.

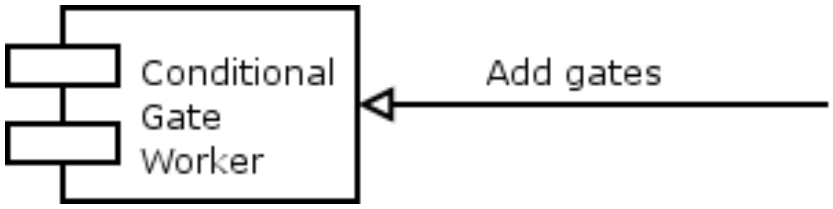

Figure 7.1: Conditional Gate worker: Adding gates. Each component of the *Approximate matcher* can add gates to the worker.

XOR phase In the XOR phase all the conditional gates that need to be calculated do not depend on previous steps outputs of the algorithm (see Section 3.1). Thus all the gates are fed directly to a `ConditionalGateWorker`.

Addition phase The Addition phase is more complex, because we have a multi-level algorithm where the Conditional Gates involved depend on the out-

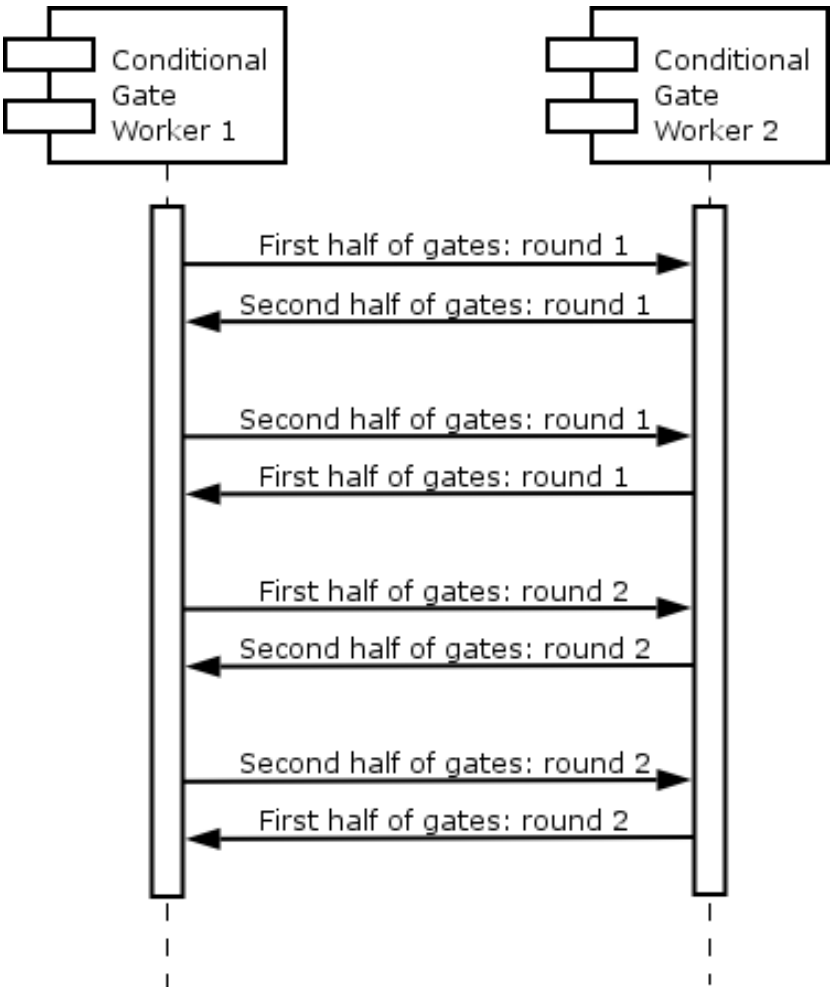

Figure 7.2: Conditional Gate worker: Execution. The Workers correspond to two different parties, and each one begins by calculating a different subset of the available gates. This is the implementation of the parallel initiation optimization.

puts of previous levels. However as we saw in Figure 3.2 a level of parallelism in the addition algorithm can be achieved. Our design is depicted in Figure 7.3. It depends on executing each addition on a different thread. These threads use a single `ConditionalGateWorker` to compute the Conditional Gates. That way the Conditional Gate messages of each addition thread are combined with the other threads'. In addition if the number of gates computed is 2 or more, parallel initiation is used.

Timing

The timing requirements for the protocol were the following:

- precise timing of the *Approximate matcher* protocol;

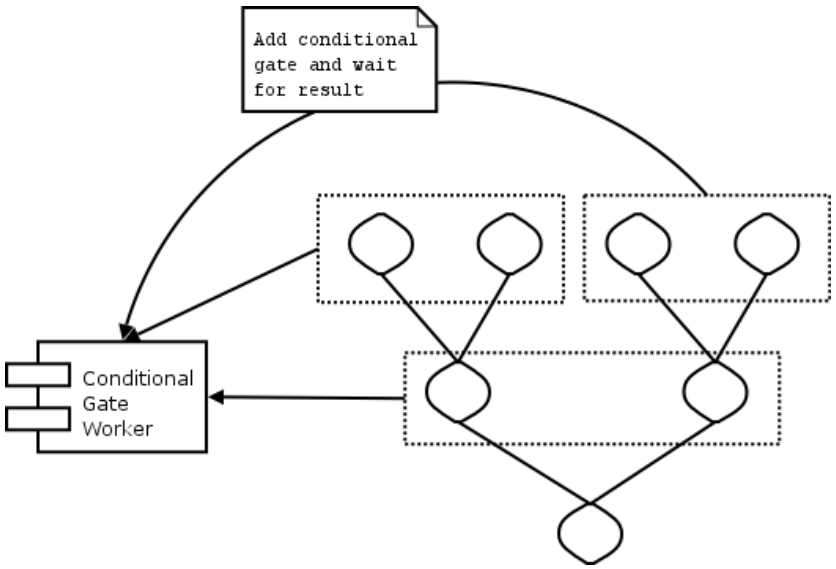

Figure 7.3: *Approximate matcher*: Adder. Each box represents a different execution thread. The boxes on the same level are executed in parallel.

- calculation of the idle time during protocol execution.

To do precise timing in the `Win32` platform we used the high-resolution time function `timeGetTime`, which allows for up to 10 milliseconds accuracy. In GNU/Linux system we used the `clock_gettime` API function that is available in all POSIX compliant systems. This has accuracy that depends on the CPU frequency but for our purposes we restricted timing to millisecond precision.

Calculating the idle time of the protocol was done as described below. When no precomputations thread was available the idle time was estimated by counting the time the application spent waiting input through the network. When a precomputations thread was present the idle time was again estimated by counting the network waiting time if the precomputations thread was idle, i.e. no calculations were done.

7.1.4 Cryptosystem Key Generation

Big number operations

For the cryptosystem to be implemented, as we described it, a way to perform operations with big numbers is required. In our case big is defined to be numbers of size 1024-6894 bits. Since most CPUs are limited to 32 or 64-bits operations the native data types cannot be used. Fortunately there are several libraries (or packages) that we can use. Those are

- `GMP`: The GNU Multiple Precision Arithmetic Library[13], which is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. According to their web site the main target applications for GMP are cryptography applications and research, Internet security applications, algebra systems, computational algebra research, etc.
- `libtommath`: LibTomMath[6] provides highly optimized and portable routines for a vast majority of integer based number theoretic applications.
- `libgcrypt`: This is a general purpose cryptographic library based on code from GnuPG. It includes a hardened version of the `GMP` library.
- `OpenSSL`: This is a general purpose cryptographic toolkit.
- and more ...

There are quite numerous possibilities, so which one to select? If we think about performance then `GMP` outperforms any of the other options, but this is a library that was never intended to be used in real world security applications. A careful study of its source code reveals code vulnerable to denial of service. A typical example is that GMP will call `exit()` if it cannot allocate space for a number, leading to easily exploited denial of service attacks.

`Libgcrypt` did modify the `GMP` code in a way to be used in security applications and thus it gets the benefits of the speed issue. A drawback is the fact that it is based on old code of `GMP`. `Libtommath` was also specifically designed for security applications and is a well-written library with a clean interface. It is highly portable since it was written in C with no assembler optimizations , but this of course comes at a performance cost. The `OpenSSL` library contains big number operations as part of the toolkit but it also does not achieve the performance of `GMP`.

For our application the `GMP` library was selected, mainly because the author was familiar with `libgcrypt` which has almost the same API and porting `GMP` to `Win32` platform was easy.

El-Gamal key generation

The El-Gamal cryptosystem requires the generation of a multiplicative group and a prime order subgroup. We use an algorithm, similar to the one described in [22] optimized for our needs, to generate a prime number of the form $p = 2qw+1$, where q is also prime. The reasons behind the choice of this special form will be

Generate $p = 2qw + 1$, where p and q have size $l(p)$ and $l(q)$ respectively.

1. Generate a prime q of size $l(q)$
2. Generate a random odd number w of size $l(p) - l(q)$ and test if $p = 2qw + 1$ is prime. If not repeat this step.

Having generated the prime p we find a generator of a subgroup of order q.

1. Generate a random number r in the group
2. Calculate $c = r^{\frac{p-1}{q}} = r^{2w}$
3. If $c = 1$ return to the first step. Otherwise the generator is c.

Figure 7.4: Prime order group and subgroup generation

given below. Then we find a generator of order q using the algorithm described in [33, Finding a generator for $\mathbb{Z}_p^*$].

Our algorithm for group generation is shown in detail in Figure 7.4. The prime number generation is done by generating random odd numbers and testing them for primality using the Miller-Rabin algorithm (see [25]).

The reason for the choice of a prime of the form $p = 2qw + 1$ is to make known attacks against the discrete logarithm problem (see [25, The discrete logarithm problem]) difficult to mount. That is because the multiplicative group generated by the prime p has order $p - 1 = 2qw$ that includes q as a factor. This means that p will resist the attacks described in [38] and [25], if the size of q is big enough. How "big enough" is, is discussed in the next paragraphs.

Security levels

While implementing the threshold El-Gamal cryptosystem we faced the problem of determining the length of generated parameters, such as the group's size and the order of the subgroup, as well as the cryptographic primitives to be used such as the hash function. We wanted our implementation to use algorithms with similar security properties, that is, have no weak link in the implementation. For this reason we used the definition of security levels found in [8].

The security level is defined as the amount of work required to break an algorithm measured in steps. For example to brute force an 128 bits AES key 2^{128} steps are required. The amount of CPU instructions that each step requires is not important for the algorithms we use. The difference between a million instructions and a single one does not practically affect the computational effort needed in the magnitude of numbers we are working on.

In [21] estimates for the required parameter sizes for various cryptosystems with respect to a cost are given[3]. Furthermore [20] does a comparison of the security

[3]The authors of this article implicitly suggest that security level should be selected by

levels for public key algorithms with the levels offered by the AES symmetric cipher. We summarize their results from [20] for today's attacks in Table 7.1.

Security level (in $\log_2$(steps))	El-Gamal Prime p size (in bits)	Order of generator g (in bits)	Hash function output length (in bits)
72	1024	160	160
128	2644	256	256
192	6897	384	384
256	13840	512	512

Table 7.1: The security levels implemented. The first level doesn't correspond to values from the referenced papers but rather commonly used values in today's protocols.

7.1.5 Packet Encoder

There are several ways to encode a packet to store or to be transmitted over the network. Standard bodies usually define ways to do it. The W3 Consortium defined XML for that purpose, ITU believes ASN.1 [36] should be used and so on. However if we check random protocols from IETF we can see that in practice there is no standard way. Very few protocols are defined in XML, such as Jabber, even fewer in ASN.1, i.e., LDAP, and the majority they just use a custom made protocol, say SMTP, HTTP and TLS. On the other hand outside IETF several security protocols, especially the sponsored by RSA Security PKCS standards are based on ASN.1.

A question that might arise, is whether there is a need for a standardized way of encoding messages and structures. We don't think there is a definite answer to this question since even today where ASN.1 and XML are mature, new protocols avoid using them. Maybe there isn't. However there are some advantages from using a standardized way. These are

- It is easier to describe the protocol to somebody familiar with the standard. For example in ASN.1 you can simply say encode this integer in the `packet` structure below using the DER encoding rules, and everybody familiar with the standard will understand what to do.

```
packet ::= SEQUENCE {
   x   INTEGER,
   y   INTEGER
}
```

- The encoding is automated using available tools. For example for the DER encoding rules of ASN.1 one can find tools such as the one described in [10], or for XML the `libxml`, which allow for automated encoding and

letting the user specify a cost, for example, the year until when he'd like his data to be secure. We consider it a very good approach to specify the sizes of the security parameters, especially given the current common approaches that let the user decide which algorithms and key sizes to use.

decoding with few source code lines. For custom made protocols there are no such tools (we couldn't find a tool to encode TLS packets).

For our implementation we will use the ASN.1 language to specify the packet structures and the DER encoding rules to perform the encoding. That design decision was arbitrary although influenced by the fact that several security protocols already use it, such as the PKIX and PKCS standards.

7.1.6 Communications

During the conditional gate study we assumed, for the communication complexity calculations, that there is a way for individual peers to communicate using a broadcast method. With the traditional peer to peer communication models this is not possible without having a central server, or having individual hosts forward packets. In both cases this is clearly inefficient since the communication complexity increases due to retransmissions. A way to overcome this is to use the broadcast feature of IP version 4, which seems to offer the required functionality. However this prevents parties from remote networks from communicating since broadcasted packets cannot be transmitted outside a local network area and moreover broadcast is not supported by the new IPv6.

The most suited solution seems to be the multicast[35, Multicasting] feature of the IP protocol. Multicasting allows a targeted broadcast between parties over the network. Each party must volunteer and join the multicast network to be able to receive the transmitted packets. Thus our initial assumption is fullfilled. However multicast links are unreliable by design[4] and require protocols to announce sessions such as [15]. Reliable extensions for multicast links, such as [1] and [7], have been proposed, although complete implementations of these protocols are not yet available.

For this reason and moreover because we focus on two-party computations our implementation uses a peer to peer TCP/IP connection, which is sufficient for our purposes. The networking API we are going to use is the Berkeley Sockets API which is available in all POSIX compliant operating systems including `Win32`.

7.1.7 User interface

For the purposes of the user interface of the application we considered using a graphical interface. The choices we had were

- `Microsoft Foundation Classes`: The "native" way of creating windows in the `Win32` platform;
- `GTK+ and GTK-`: A user interface that is widely used in GNU/Linux applications and is the basis of the GNOME window environment, mostly used by C and C++ applications;

[4] They depend on UDP message delivery.

- `QT`: The basis of the KDE window environment, based on an extended version of C++;
- `WxWidgets`: A cross-platform framework for graphical user interface applications, based on a subset of C++.

All of the above were portable at least on the `Win32` and GNU/Linux platforms[5]. Our choice was the `WxWidgets` not only because it is available for our language of choice (C++), but mainly because it provided a portable Graphical User Interface and a compatibility layer for each platform that involved Threads, Semaphores and other operating system depended functionality.

7.2 Communication protocol overview

In this section we will give a way to perform the *Profile matching* for two parties over a reliable communication line. The next sections will show how a typical message exchange between two participants looks like. The exact contents of the messages are shown in Appendix C as well as the contents of packets intented for storage purposes.

In the following sections we assume a client-server model, where the client is the initiator of the connection and the server is the one accepting it. That choice was made for descriptive purposes only and serves no other purpose since the protocol is symmetric. The messages we use are either of specific type that are named and defined accordingly, as well signal messages that contain few bits of information.

7.2.1 Initial handshake

Here in the initial handshake the client advertises itself to the server. The participants give a hint to each other so they can check whether they already share an El-Gamal key pair. Additionally a security level is negotiated and the server provides the client with all the parameters needed for the *Profile matching* protocol, such as the threshold, enabled optimizations etc.

Client		Server
ClientHelloPDU	$\longrightarrow$	
	$\longleftarrow$	ServerHelloPDU
IDPDU	$\longrightarrow$	
	$\longleftarrow$	IDPDU

The `IDPDU` message includes the numerical ID to be used in the subsequent key exchange and the protocol. It matches any previously established IDs if connecting to a known user[6].

[5] Even the MFC is available in linux via `libwine`.

[6] This might not make much sense in the two party case since each party can store the other party's ID, but for n parties a way to map the participants to protocol IDs is required.

If a party doesn't share a key with the peer they notify the peer with a signal of type `BEGIN_KX`, otherwise with a signal of type `PROCEED_PROTO`. In the case both signals from the peers are of type `PROCEED_PROTO` they proceed to the main protocol. Otherwise the move on with a fresh key generation. An exchange of messages, where the Client side has a key stored, but the Server side doesn't (may have been deleted for some reason) would be

Client		Server
SignalPDU(PROCEED_PROTO)	$\longrightarrow$	
	$\longleftarrow$	SignalPDU(BEGIN_KX)

7.2.2 El-Gamal key generation

The message exchange for the El-Gamal key generation follows the general case algorithm from Section 2.4.1. The mapping to messages is straightforward, except for the `ElGamalGroupParameters` which is a message sent by the client that contains El-Gamal group parameters that correspond to the selected security level. Those parameters are generated by the client.

Client		Server
ElGamalGroupParameters	$\longrightarrow$	
ThresholdElGamalGenerationCommitment	$\longrightarrow$	
	$\longleftarrow$	Threshold...Commitment
ThresholdElGamalGenerationShare	$\longrightarrow$	
	$\longleftarrow$	Threshold...Share
SignalPDU(DONE)	$\longrightarrow$	
	$\longleftarrow$	SignalPDU(DONE)

7.2.3 Conditional Gate

The following message exchange is to allow calculations of conditional gates. This is directly based on the Conditional Gate worker (see Section 7.1.3). The number of exchanged messages is 4 and this is the minimum number of messages needed to evaluate a conditional gate work for two parties (see Section 2.5). In those exchanged messages the output of many conditional gates are combined together distinguished by an ID number[7]. This combination of many conditional gates into four messages is the implementation of the Round reducing optimization (see Section 4.2.3).

[7] That number is calculated based on the position of the protocol the conditional gate was run, thus is the same for both peers.

Client		Server
ConditionalGatesBunchPDU1	$\longrightarrow$	
	$\longleftarrow$	ConditionalGatesBunchPDU1
ConditionalGatesBunchPDU2	$\longrightarrow$	
	$\longleftarrow$	ConditionalGatesBunchPDU2

7.2.4 Profile Matcher

Here a run of the *Profile matching* protocol is displayed. The protocol is split into steps, the XOR, the addition, the comparison and the final decryption step. On each step several conditional gate messages are exchanged. Each step is terminated using a signal of type DONE. The final step is terminated with a signal of type BYE. After this message is exchanged each party should be able to calculate the output of the profile matching algorithm.

Client		Server
BigStringPDU	$\longrightarrow$	
	$\longleftarrow$	BitStringPDU
	Conditional gates for XOR step	
	$\vdots$	
SignalPDU(DONE)	$\longrightarrow$	
	$\longleftarrow$	SignalPDU(DONE)
	Conditional gates for Addition step	
	$\vdots$	
SignalPDU(DONE)	$\longrightarrow$	
	$\longleftarrow$	SignalPDU(DONE)
	Conditional gates for Comparison step	
	$\vdots$	
SignalPDU(DONE)	$\longrightarrow$	
	$\longleftarrow$	SignalPDU(DONE)
ThresholdElGamalDecryptionShare	$\longrightarrow$	
	$\longleftarrow$	Threshold. . . Share
SignalPDU(BYE)	$\longrightarrow$	
	$\longleftarrow$	SignalPDU(BYE)

Chapter 8

Results

After having discussed the protocols and their implementation, it is interesting to compare and see the effect of the optimizations discussed in previous chapters. This is the purpose of this chapter. In the first section, we give a brief description of the test tool that was developed. The measurements taken using that tool and some discussion are given in Section 8.2. The final part, Section 8.3, summarizes the advantages and disadvantages of this specific protocol, as well as for the selected set of algorithms.

8.1 Deliverables

The delivered application is a test tool implemented by following the architecture and design discussed in the previous chapters. It is a 10000-lines application that fulfills the requirements and can handle the use cases set in Section 6.3. Except for the *Approximate matcher* protocol, it implements the two variants of it (see Section 3.4) and the following optimizations:

- Parallel initiation;
- Precomputations;
- Combining outputs.

The user interface was designed in a way to allow enabling and disabling various optimizations, plus giving controls to internals of the implementation such as the cache size and the priority of the precomputations thread. Due to the selection of the interface back-end the application could be compiled and executed on the GNU/Linux as well as the `Win32` platforms, although extensive testing was only performed on the `Win32` platform.

A screenshot of the `Win32` user interface is shown in Figure 8.1 and a screenshot from the interface in GNU/Linux is in Figure 8.2.

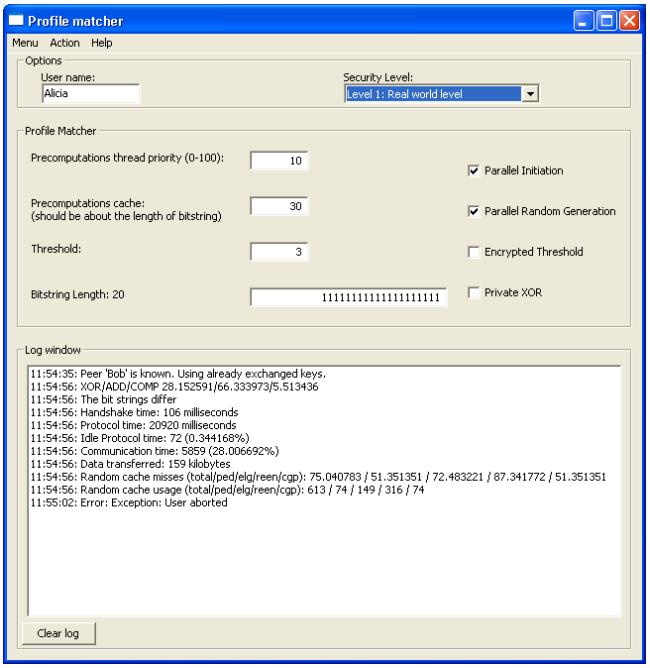

Figure 8.1: The application's user interface. The log messages correspond to a successful protocol run.

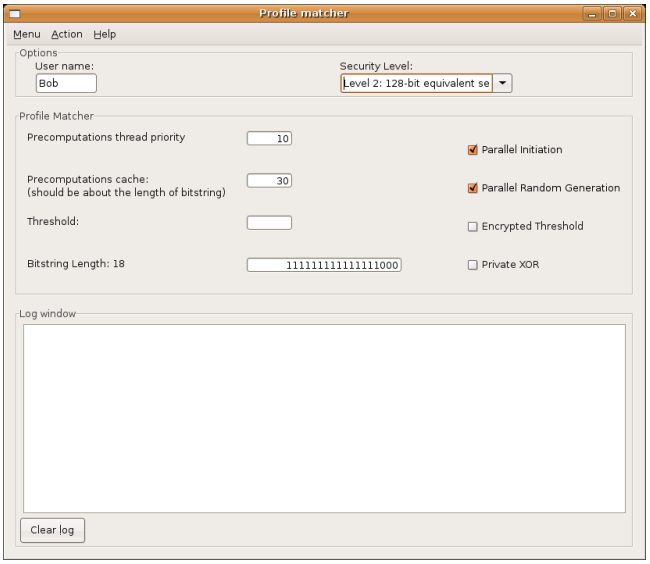

Figure 8.2: The application's user interface on GNU/Linux using the GTK widgets.

8.2 Results

In the following paragraphs we discuss the performance of the optimizations compared to an unoptimized, naive implementation, the performance of the protocol across different kinds of networks and its scalability over various complexities. At the end we discuss about the accuracy of the values measured with the implementation.

In all measurements, unless stated otherwise the security parameter is fixed to a 1024 bit El-Gamal group, with exponent size of 160 bits. This is the commonly used today security level (see Section 7.1.4).

8.2.1 Performance of the protocol

To give an overview of the protocol's running time we summarize in Table 8.1 the running time for various bitstring sizes when all optimizations are enabled.

bitstring size in bits	total time
16	18 sec
32	29 sec
64	48 sec
128	85 sec
256	2,5 min
512	5 min
1024	9,5 min

Table 8.1: In this table we give an overview for the amount of time spent in the protocol when all optimizations are enabled.

More details about the running time and improvements are given in the next sections.

8.2.2 Performance of optimizations

In this section we are going to give a comparison of the performance of a naive implementation of the protocol, without any optimizations applied, versus the optimized versions. When feasible we also give the expected, from the models developed in the previous chapters, values for comparison.

For the measurements, the test program was run on two similar computers, one being an Intel Pentium 4, 2.00GHz and the other of the same type in 2.40GHz[1]. Both systems were connected in the same 100 MBit ethernet switch which was used for the communication.

It should be noted that the combining outputs optimization (see Section 5.3) is always applied, due to the design of our implementation.

[1] They had a speed difference in modular exponentiations of about 15%, and for this reason the measurements were done on the slowest of the two.

To give the comparison between optimizations we run the protocol with the same input, the same threshold and the same priority and cache for the random number generator. The tests were performed using varying bit string lengths. The raw output values of these tests are listed in Table 8.2 on page 76. On several occasions the performance across different security levels was tested.

In the tests, the following values were measured:

- Total time of execution;
- Idle time;
- Communication time;[2]
- Amount of data transferred;
- Messages exchanged;
- Amount of random data requested;
- Number of modular exponentiations;
- Time spent in modular exponentiations.

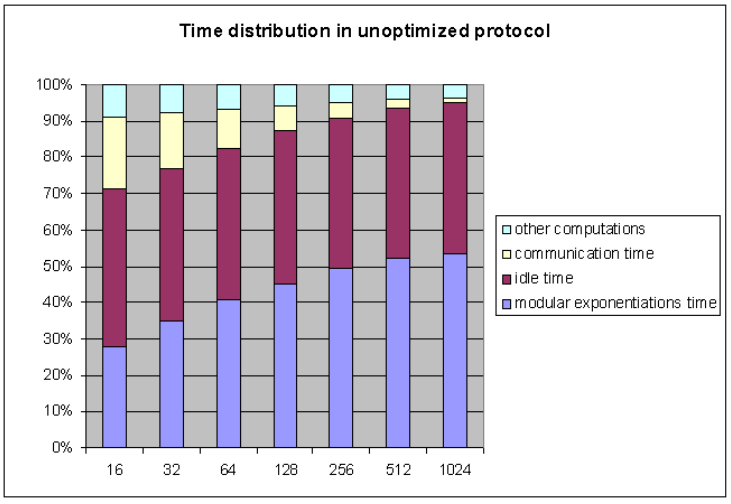

Figure 8.3: The distribution of time spent in the protocol when no optimizations are applied. The data are shown for various bit string sizes, and fixed El-Gamal group parameters.

Optimizations

We can see from Table 8.2 and Figure 8.3, that the idle time[3] of the protocol accounts for more than 40% of the total execution time when no optimizations

[2] This number proved to be of little significance, since it heavily depends on the OS' internal caches. We observed several times that the amount of time needed for the `send()` system call to return from sending large amounts of data, could be less than sending fewer data.

[3] See Section 7.1.3 for how idle time is defined and calculated.

are applied. This percentage is fixed and doesn't depend on the size of the bit strings. Thus we see that there is plenty of space for improvement.

In the next paragraphs we discuss the effect of each optimization that was implemented.

Parallel initiation Parallel initiation as expected reduces the idle time to over 90% for the bit strings of size larger than 256 bits. That leads to an equal in length reduce in total execution time, which is more than 40% less than the unoptimized case. We summarize the decrease in time in the Table 8.3.

In small bitstrings, where the communication time occupies a non-negligible amount of time, the decrease of the idle time can be as low as 50%. This is because of the fact that part of one party's communication time is the other party's idle time. However as bitstring sizes increase and the communication time becomes insignificant, our implementation gets closer to the calculated in Equation (5.1), limit in time complexity, which is a reduce of 50% for the 2-party case.

This optimization's effects are visible when comparing Figure 8.3 with figure Figure 8.4. There the reduction in idle time is clearly visible.

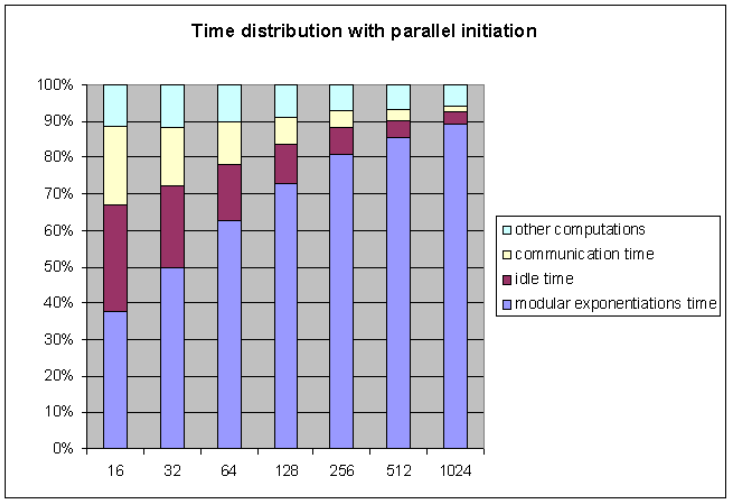

Figure 8.4: The distribution of time spent in the protocol when parallel initiation is enabled. The data are shown for various bit string sizes, and fixed El-Gamal group parameters.

Precomputations As was discussed in Section 7.1.1, in our implementation the precomputations optimization was implemented in way that it starts on a lower priority thread, at the time of the protocol initiation. That effectively means that we had a parallel random generator utilizing the CPU's idle time. The effect of this generator can be seen in Figure 8.5. The idle time is reduced, and the effect becomes more visible when the bitstring sizes increase.

The cache size used by default by the random generator, was the same as the size of the bitstring. For rationale see Section 5.2. This size proved to be effective, at the cost of a linear increase in the memory usage of the generator, depending on the size of the bitstrings.

As we can see from Table 8.4 although precomputations manage to reduce idle time the level of optimization is not of the same order as parallel initiation. This is of course expected since the precomputations improve only a small fraction of the total computational load, thus the bottleneck of waiting the peer to finish the computations still remains, although reduced.

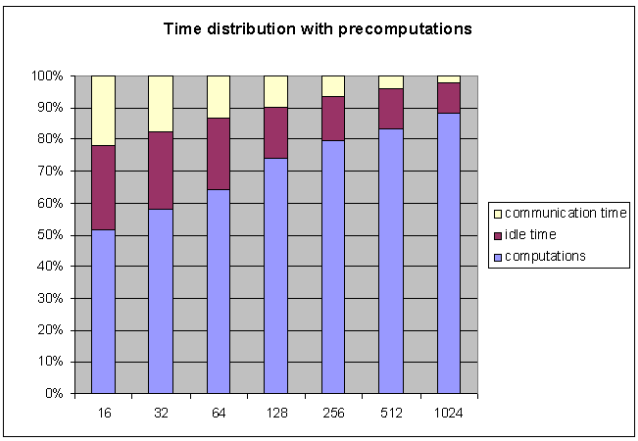

Figure 8.5: The distribution of time spent in the protocol when precomputations are enabled. The data are shown for various bit string sizes, and fixed El-Gamal group parameters.

Combining outputs It is interesting to see the effect of this optimization. From the *Approximate matcher*'s round complexity discussion (see Section 3.3.4) we would expect the number of messages exchanged[4] to be linear to the number of bits of the bit strings. However because this optimization is used we can see in Figure 8.6 that our expectation for logarithmic growth (see Section 5.3) is correct. In Table 8.5 we list the actual messages exchanged in the protocol compared to the expected message complexity.

The expected message complexity was calculated based on the round complexity formula in Section 3.3.4, by replacing the factor $n + 1$ that depends on the number of players with the number $2 + 2 = 4$, because we only have 2 parties and El-Gamal decryption uses two messages even though they are in a single round. Thus we calculated the message complexity as

$$4(\log_2^2 m + \log_2(m + 1))$$

We can see that our calculations match the number of messages exchanged, except for a difference of 16 messages. Those are messages used by the communication protocol, for signaling and termination of the protocol. We can also see

[4]In the two party case the number of messages exchange and the round complexity are analogous.

the difference in the number of messages when parallel initiation is enabled[5]. The scaling of the number of messages is visible in Figure 8.6.

A comparison with the expected number of messages without this optimization is given in Figure 8.7. That expected number was calculated as above based on the results of Section 3.3.4.

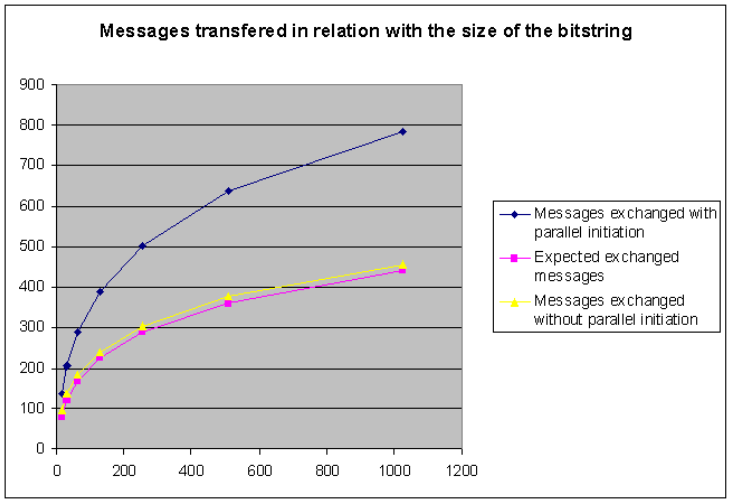

Figure 8.6: The number of messages exchanged in the protocol. We can see the increase in the number of messages due to parallel initiation.

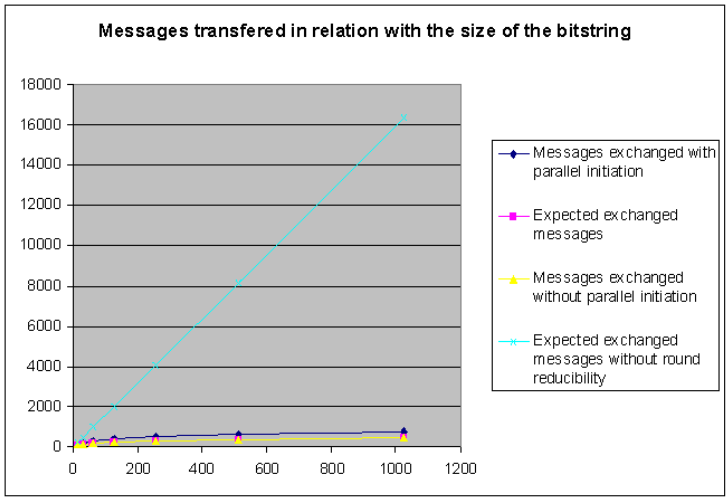

Figure 8.7: The number of messages exchanged in the protocol compared with the expected number of messages without the combining outputs optimization.

[5]When parallel initiation is used the amount of messages exchanged increases by a factor that is less than 2 (the number of players). See also the discussion about combining optimizations in Section 5.4.

Combining optimizations As we already saw with *Combining outputs* and *Parallel initiation*, combining optimizations will not give the sum of the individual improvements. This is visible in our case where the combination of parallel initiation and precomputations optimizations, on average reduces the total time of execution at about 4-5% more than parallel initiation alone. The level of reduce in total time is shown in Table 8.6.

The main reason for this is that the amount of idle time was reduced due to parallel initiations, thus less time was available for precomputations. More reasoning is given in Section 5.4. The total and idle time improvement for bitstrings of size 1024 when those optimizations are applied is shown in Figure 8.9. The time distribution is shown in Figure 8.8.

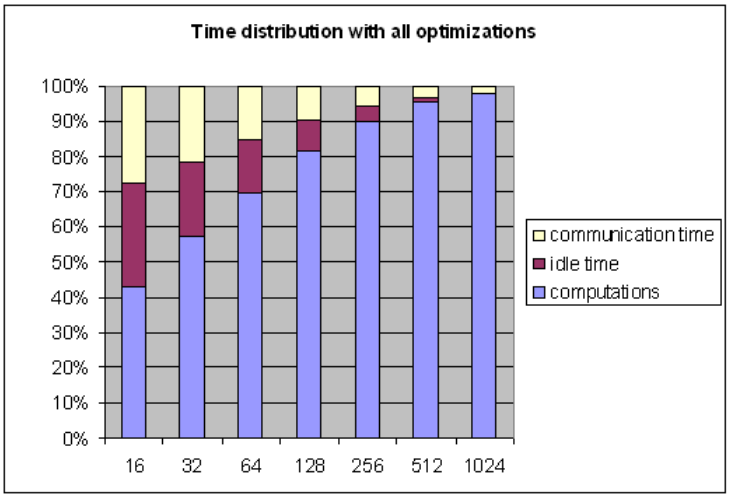

Figure 8.8: The distribution of time spent in the protocol when both parallel initiation and precomputations are enabled. The data are shown for various bit string sizes, and fixed El-Gamal group parameters.

Time distribution Another interesting figure to see is the effect of optimizations in the time distribution across the three phases of the protocol (XOR, Addition and Comparison). For bitstrings of size 256, the distribution is shown in Table 8.7 and visualized at Figure 8.10.

We can see in that figure that, as expected, the XOR and Addition steps have the largest contribution to total time, whilst the comparison step has a very small contribution. That is because the input to the XOR step is only a fraction of the input to the XOR and the addition steps.

In addition, it is visible that parallel initiation changes the balance and the calculations for the comparison step have increased their percentage in the total time. That is something expected since the comparison phase has only sequential calculations of conditional gates, while both XOR and addition steps can be parallelized. Thus the latter steps take advantage of the parallel initiation whilst the comparison step is still having the same execution time, and so increased its percentage.

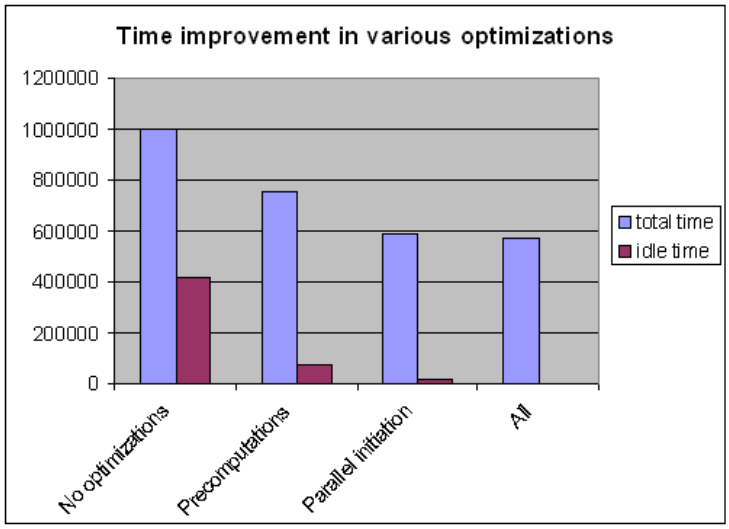

Figure 8.9: The total and idle time spent in the protocol when both parallel initiation and precomputations are enabled. The data are shown for bit strings of size 1024, and fixed El-Gamal group parameters.

The precomputations optimization affects the same way all three steps thus, as expected, no dramatic change in the time ratio occurred[6].

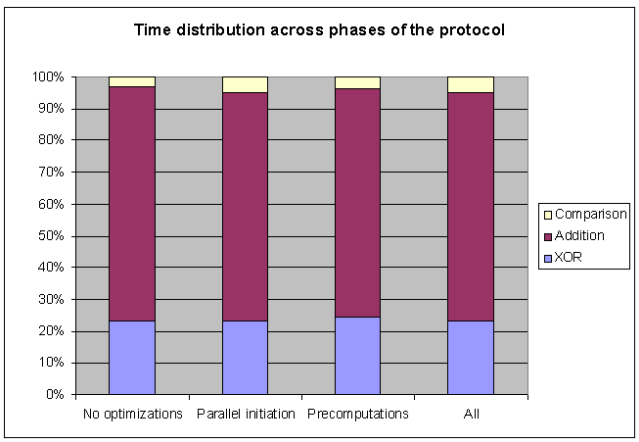

Figure 8.10: The time distribution between different phases of the protocol for different optimizations.

[6]Except maybe for the XOR step, which is the first to start, thus less time is available for precomputations.

bit string size	total time	idle time	communication time	data transferred	messages	random data	modular exponentiations	time in exponentiations
No optimizations								
16	26565	11528	5310	123	96	13	2795	7423
32	45674	19246	7028	258	136	24	5868	15896
64	79572	33142	8380	530	184	46	12061	32509
128	145556	61395	9855	1076	240	90	24494	65872
256	268476	111352	11559	2173	304	179	49407	132749
512	516220	214156	12741	4365	376	358	99280	268909
1024	1003863	419488	14512	8753	456	714	199073	535229
Parallel initiation								
16	19827	5798	4245	123	136	13	2795	7494
32	31191	6940	5049	258	204	24	5868	15593
64	51712	7972	5989	530	288	46	12061	32459
128	91039	9749	6763	1076	388	90	24494	66477
256	166078	12003	7664	2173	504	179	49407	134374
512	309422	13925	9583	4365	636	358	99280	264821
1024	588125	18163	10813	8753	784	714	199073	525220
Precomputations								
16	24172	8235	6719	123	96	13	2877	
32	39812	11704	8580	258	136	24	6129	
64	67265	17419	10485	530	184	46	12322	
128	116703	20939	12452	1076	240	90	24757	
256	211750	30946	14590	2173	304	179	49669	
512	395172	51964	16171	4365	376	358	99544	
1024	758078	72268	18048	8753	456	714	211214	
All optimizations								
16	18375	5421	5061	123	136	13	2877	
32	29313	6266	6248	258	204	24	6142	
64	48781	7151	7548	530	288	46	12719	
128	85203	7295	8389	1077	388	90	25290	
256	155531	6579	9047	2173	504	179	52369	
512	294328	3157	10026	4367	636	358	105315	
1024	571266	0	11127	8754	784	714	204890	

Table 8.2: The performance of the *Approximate matcher* for various bit string sizes. The time unit is milliseconds and the data unit is kilobytes (1kb=1000 bytes). The time spent in exponentiations was not measured when precomputations were enabled, because of practical problems arising from the multi-threaded architecture.

Bit string size	Idle time percentage	Decrease in idle time	Decrease in total time
16	43%	49%	25%
32	42%	64%	32%
64	42%	75%	35%
128	41%	84%	37%
256	41%	89%	38%
512	41%	93%	40%
1024	42%	96%	41%

Table 8.3: The decrease in total and idle time of the protocol when parallel initiation is used.

Bit string size	Idle time percentage	Decrease in idle time	Decrease in total time
16	43%	29%	9%
32	42%	39%	13%
64	42%	47%	15%
128	41%	66%	20%
256	41%	72%	21%
512	41%	76%	23%
1024	42%	83%	24%

Table 8.4: The decrease in total and idle time of the protocol when precomputations are used.

Bit string size	Messages exchanged with parallel initiation	Messages exchanged without parallel initiation	Expected message complexity (without parallel initiation)
16	136	96	80
32	204	136	120
64	288	184	168
128	388	240	224
256	504	304	288
512	636	376	360
1024	784	456	440

Table 8.5: The number of messages exchanged in the protocol.

Bit string size	Idle time percentage	Decrease in idle time	Decrease in total time
16	43%	52%	30%
32	42%	67%	36%
64	42%	78%	37%
128	41%	88%	41%
256	41%	94%	42%
512	41%	99%	43%
1024	42%	100%	43%

Table 8.6: The decrease in total time of the protocol when all optimizations are applied. What is impressive is the fact that with the precomputations optimization helped reduce the total time by a factor that is more than the idle time.

	XOR	Addition	Comparison
No optimizations	23%	74%	3%
Parallel initiation	23%	72%	5%
Precomputations	24,5%	72%	3,5%
Both	23%	72%	5%

Table 8.7: The time distribution across different stages of the protocol.

8.2.3 Performance over various networks

In the previous section we saw how various optimizations affected communication and round complexity. But examining the round and communication complexities alone, does not give an accurate image of the expected communication time. Our test network happened to have low latency (high speed) and high bandwidth[7] (capacity). But how would a protocol like the *Approximate matcher* perform on different networks?

In Table 8.8 we can see how latency and bandwidth can affect the protocol on some types of networking infrastructures. The values in this table were calculated based on the values of Table 8.2. The calculations for the latency overhead were done by adding the latency per messages for all the messages exchanged, and for the capacity overhead by dividing the total number of data exchanged with the capacity.

Although we will not elaborate, we can clearly see that the main overhead is caused by communication complexity, both in high (ethernet) and low (modem) bandwidth links. On the other hand links like a satellite link, which have extremely high latency are affected by the round complexity severely. But even in that case we can see that the linear growth of the amount of data to be transferred, compared to the logarithmic growth of the round complexity, reduces very fast the contribution of latency in the total communication overhead time.

8.2.4 Performance of the variants

In the description of the *Approximate matcher* we discussed about possible variants where the input bitstrings are private or the threshold is encrypted. In these paragraphs we only summarize the performance of the variants in the malicious model without discussing any details. Some indicative values can be seen in Table 8.9 on the next page.

In that table we see that when the Private XOR variant is used the protocol is about 15% faster. This could be further improved when there is private input to both parties. In that case they can use parallel initiation to speed up the XOR phase. As expected when the threshold is encrypted the protocol time is increased due to the extra conditional gates needed during the comparison phase.

[7] A good introduction to these terms can be found in [2].

bit string size	time overhead due to capacity	time overhead due to latency	contribution of latency
28800 modem, with 3.6kb/s bandwidth and latency of 100ms			
16	34167	13600	28%
32	71667	20400	22%
64	147222	38800	20%
128	299167	50400	14%
256	603611	63600	10%
512	1213056	63600	4%
1024	2431667	78400	3%
10Mbit ethernet, with 1250kb/s bandwidth and latency of 0.3ms			
16	98	40.8	29%
32	206	61.2	23%
64	424	86.4	27%
128	861	116.4	22%
256	1738	151.2	8%
512	3494	190.8	5%
1024	7003	235.2	3%
1Mbit satellite, with 125kb/s bandwidth and latency of 500ms			
16	984	68000	99%
32	2064	102000	98%
64	4240	144000	97%
128	8616	194000	95%
256	17384	252000	93%
512	34936	318000	90%
1024	70032	392000	84%

Table 8.8: The expected performance of the *Approximate matcher* in different types of network infrastructure. The calculations here are based on the data from Table 8.2. The values of latency and bandwidth of different infrastructures might not be perfectly accurate.

bit string size	*Approximate matcher*	Private XOR *Approximate matcher*	Encrypted threshold *Approximate matcher*
16	18 sec	16,5 sec	22,5 sec
32	29 sec	25,5 sec	34,5 sec
64	48 sec	42 sec	55 sec
128	85 sec	71 sec	92,5 sec
256	2,5 min	2 min	2,7 min

Table 8.9: The performance of the *Approximate matcher* and variants for several bitstrings.

8.2.5 Scalability

An interesting question is how does the protocol scale with the size of the input. We will mostly focus on the growth in relation to the bit string size, although the growth due to security parameter increase will also be discussed. The factors we will examine are

- Computational complexity;
- Randomness complexity;
- Communication complexity;
- Round complexity.

Computational complexity

We can get an idea of the growth of the computational complexity by looking the values in Table 8.2 and the figures 8.11 and 8.12, which show the total time growth and the amount of modular exponentiations in relation with the size of the bit string. As expected (see Section 5.1) the scaling is linear $O(m)$.

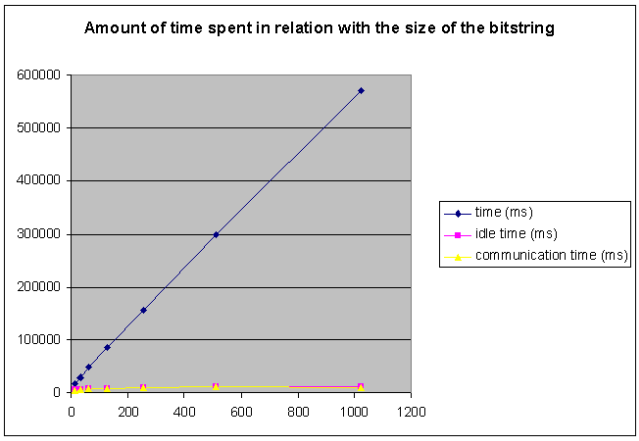

Figure 8.11: The total time in relation with the size of the bit strings. The time unit is milliseconds.

Both of these figures rely on El-Gamal parameters of 1024 bits base and 160 bits exponent. The scalability of the protocol as a function of the security parameter (see Section 7.1.4 on page 61) can be seen from the values in Table 8.10 and the graph in Figure 8.13. The growth factor is of the order of the exponentiation complexity which is typically $O(k^3)$ for k-bit input.

Randomness complexity

In Table 8.2 and Figure 8.14 we can see how the amount of random bytes request scales with the size of the input. They are linearly dependent as already indicated by the calculations in Section 3.3.1.

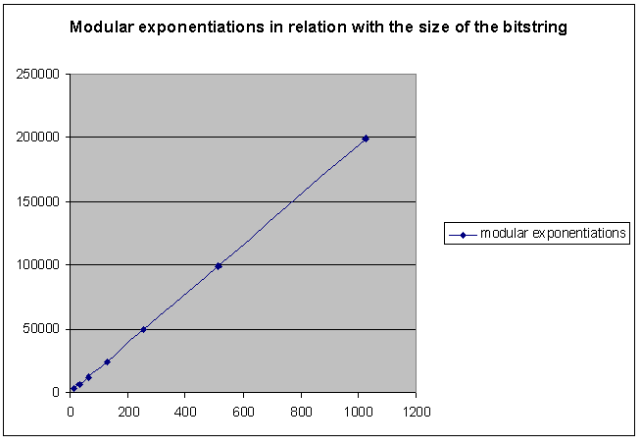

Figure 8.12: The number of modular exponentiations performed by the *Approximate matcher* in relation with the size of the bit strings.

security level	total time	idle time	communication time	data transferred	messages	random data	modular exponentiations	time in exponentiations
72	166078	12003	7664	2173	504	179	49407	134374
128	1325656	38253	4659	4995	504	276	49407	1245688
192	9515187	241076	6409	12159	504	410	49407	9073098

Table 8.10: The performance of the *Approximate matcher* with parallel initiation for 256 bit strings and varying security levels. The time unit is milliseconds and the data unit is kilobytes (1kb=1000 bytes).

Communication complexity

Another important factor of the protocol is how the amount of data exchanged scales with the sizes of the bitstrings to be compared. For a fixed security parameter, the measurements are shown in Table 8.2 and visualized in Figure 8.15.

As expected from Section 3.3.5 the scaling is linear with the size m of the bitstrings. However we can see a deviation with the expected values, which is expected due to the overhead of the communication protocol.

Round complexity

As already discussed above the round complexity is equivalent to the message complexity, thus Figure 8.6 gives an accurate image of the logarithmic growth of the round complexity with the size of the bit strings. That complexity is not affected by any increase in the size of security parameter.

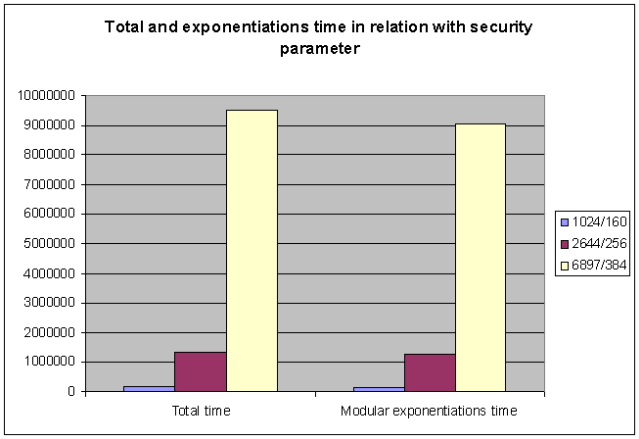

Figure 8.13: Total time and exponentiation time in relation with the El-Gamal parameter's size. In the parameters the first number indicates the base size and the second the size of the exponent. The time unit is milliseconds.

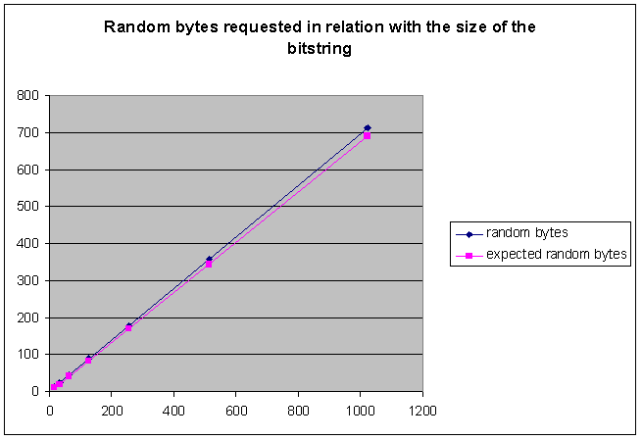

Figure 8.14: The amount of random data required in relation with the size of the bit strings. The data unit is one kilobyte and the El-Gamal parameters used are of 1024 bits size.

8.2.6 About the measurements

Is our implementation optimal? A question that might arise, is whether the previous results are accurate. That is whether our implementation actually spends time on the calculations, and the measured times are accurate and don't depend on the time spent for the user interface update etc. We will try to measure effectiveness by checking ratio of the amount of time spent in exponentiations (the most expensive operation in the protocol) to the total time of the protocol minus the idle and communication times. That would be

$$\frac{\text{(exponentiations time)}}{\text{(total time)} - \text{(idle time)} - \text{(communication time)}}$$

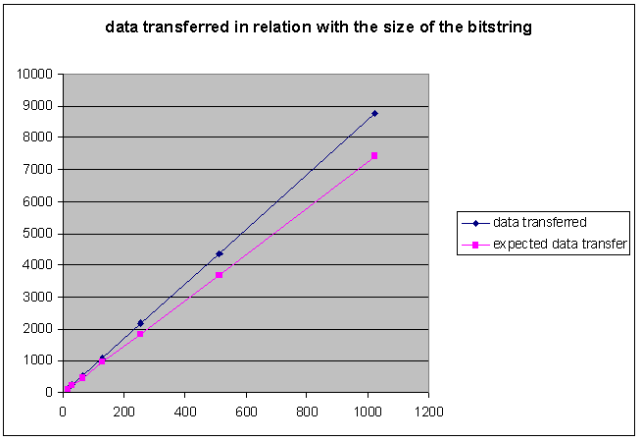

Figure 8.15: The amount of data exchanged in relation with the size of the bit strings. The data unit is one kilobyte, and the El-Gamal parameters used are of 1024 bits size.

This ratio gives the percentage of processing time, spent in exponentiations when not idle. It would give an accurate indicator on the efficiency of the implementation. The ratios are shown in Table 8.11 which uses data from the parallel initiations measurements in Table 8.2.

Bit string size	Total time	Idle time	Communication time	Exponentiation time	Ratio
16	19827	5798	4245	7494	0.77
32	31191	6940	5049	15593	0.81
64	51712	7972	5989	32459	0.86
128	91039	9749	6763	66477	0.89
256	166078	12003	7664	134374	0.92
512	309422	13925	9583	264821	0.93
1024	588125	18163	10813	525220	0.94

Table 8.11: The ratio of spent in exponentiations over total time for various bit string sizes.

Thus we can see that for bitstring sizes of more than 128 the ratio is very close to being optimal. For the small bitstrings the values shown are not that indicative since our optimality "ratio" only considers exponentiations and ignores multiplications and other cryptographic operations such as hashing, that could account for significant time when the total time is low.

The ratio increases even further with the increase of the security parameters as we can see in Table 8.12.

Thus with those measurements, we can be confident with our results especially when the size of bitstrings is over 128. An other side effect of these measurements is that they also justify our decision in the analysis phase to focus on the number of exponentiations to measure the computational complexity.

security level	Total time	Idle time	Communication time	Exponentiation time	Ratio
72	166078	12003	7664	134374	0.92
128	1325656	38253	4659	1245688	0.97
256	9515187	241076	6409	9073098	0.98

Table 8.12: The ratio of spent in exponentiations over total time for different El-Gamal group sizes and bit string size of 256 bits.

8.3 Conclusions

On the *Approximate matcher* protocol

Having seen this implementation of a *Approximate matcher* protocol and its performance, we could summarize its strong and weak points. We start with the strong points.

- Practical although non real-time[8];
  The time needed to perform the protocol increases with the size of the input to be compared, and ranges from tens of seconds to several minutes.
- Growth of computations, data transferred and required random numbers needed is linear with the input;
  This growth ensures that the protocol scales well and the amount resources needed to perform it, will not magnify unreasonably.
- Round complexity growth is logarithmic with the size of input;
  The amount of rounds needed for the computations will increase with less pace than the above complexities.
- No idle time;
  We have shown that the protocol can be implemented in a way that no idle time occurs during the protocol execution.
- All complexities can be calculated in advance;
  An other important fact that could be listed as an advantage, is the fact that all complexities can be calculated in advance, and thus one can have a good precision of resources needed before the calculations are performed.

But weak points are not of low importance. Those are the following

- The currently used cryptographic algorithms depend heavily on exponentiations;
  Although nothing is really wrong about exponentiations, they are an expensive operation and typical algorithms to perform them have a complexity of $O(k^3)$ with the size of their input, which is proportional to the security parameter selected.
- Not efficient enough for real-time applications;
  Although the total execution time is practical and at the order of seconds and minutes, currently it is not enough for real time applications, unless powerfull or dedicated CPUs are used.

Other important issues that need not to be ignored

- The amount of random bytes required is high;
  Although randomness complexity grows in a linear way, the total amount of random bytes needed can be quite high. This fact should influence the decision of selecting a cryptographic PRNG, and use constructions that

[8] It should be noted that when the malicious model requirement is relaxed to a semi-honest model (see Section 3.4.3) a 5 times improvement factor in execution time is expected, which might make the protocol efficient for some real-time applications.

use algorithms of lower magnitude than exponentiations, so that they don't affect seriously the total execution time.

- The underlying networking infrastructure is important;
  We saw in previous sections that network latency and bandwidth are important to the protocol's performance. We noted that as the input strings increase bandwidth becomes the deciding factor, whilst for small sizes the latency is more crucial. Thus in both cases care must be taken when selecting the infrastructure, with respect to the expected input values.
- Side channel attacks;
  Because of the way the protocols used operate it is important for attackers and participating parties to not be able to distinguish the values used by the protocol, either the input values or the randomly generated ones. Some first observations are summarized in Section 7.1.2.

On the optimizations

We have seen in detail that the effect of the Precomputations and Parallel initiation optimizations has a huge impact in the total running time of the protocol. However it was shown in the previous section that the combination of these two offered a small improvement over Parallel initiation alone. Thus implementations that want to avoid the complexity of parallel threads could avoid using precomputations at a low performance cost.

The above is valid with the assumption that the Combining outputs optimization is always used. Otherwise as it was shown before the rounds will increase dramatically for large bit string sizes, effectively making all of our previous measurements and results irrelevant.

With the latter optimization enabled the number of messages exchanged increases logarithmic with the size of the input, which is much lower than the linear increase of other factors such as computational and communication complexity. Thus effectively reduces the cost of interaction compared to the other complexities for long bitstrings.

Chapter 9

Summary

In this case study of the *Approximate matcher* protocol we have shown that while the primitives used, such as the conditional gate, are of quite high round complexity, the combination of these to form a higher level protocol, can keep the round complexity at acceptable growth levels with the input size. We also show that the disadvantages of high round complexity can be reversed and used efficiently for improving the protocol's performance. That was the idle time, caused by the need to wait for the other parties to reply and we also used that time to perform precomputations of values that will be used at later stage of the protocol.

On the other hand communication complexity proved to be not an unimportant factor for the performance of the protocol. Because as the input bitstrings increase in size it grows faster than the number of rounds (linear compared to logarithmic), it becomes the main factor of the communication delay in common network links.

With scheduling improvements, such as parallel initiation, and the precomputations during idle time, the computational complexity of the protocol became the main, if not the only, factor of the time spent in the protocol. But is then our main goal of minimizing the running time fulfilled? The above fact suggests for yes. The optimizations that are now expected to improve the running time are optimizations that affect the protocols' computational complexity with respect to exponentiations. Thus we believe that this is the direction for future optimizations of this protocol.

Appendix A

Acronyms

ASN.1 Abstract Syntax Notation 1 is a standard and flexible notation that describes data structures for representing, encoding, transmitting, and decoding data. It provides a set of formal rules for describing the structure of objects that are independent of machine-specific encoding techniques and is a precise, formal notation that removes ambiguities.

IETF The Internet Engineering Task Force is an open, all-volunteer standards organization, with no formal membership or membership requirements. It develops and promotes Internet standards.

PDU Protocol data unit. This is a common term used to describe communication protocols.

POSIX This is an acronym of the Portable Operating System Interface. It is the collective name of a family of related standards specified by the IEEE to define the application programming interface for software compatible with variants of the Unix OS. They are formally designated as IEEE 1003 and the international standard name is ISO/IEC 9945.

PRNG Pseudo Random Number Generator.

TCP/IP The Transmission Control Protocol and Internet Protocol. They form the most popular reliable Internet communication protocol.

TLS The Transport Layer Security protocol is a protocol standardized by IETF to provide a secure transport layer to application protocols. Its designed was based on the Netscape's Secure Sockets Layer version 3.

UDP User Datagram Protocol is one of the core protocols of the Internet protocol suite. Unlike TCP it does not provide reliability nor ordering guarantee.

UML The Unified Modeling Language is an object modeling and specification language used in software engineering. UML includes a standardized graphical notation that may be used to create an abstract model of a system: the UML model.

VSS VSS stands for Verifiable Secret Sharing and is a secret sharing scheme which is required to withstand active attacks by the participating parties.

XML The Extensible Markup Language (XML) is a W3C-recommended general-purpose markup language for creating special-purpose markup languages, capable of describing many different kinds of data. It is a simplified subset of Standard Generalized Markup Language (SGML).

Appendix B

El-Gamal threshold key generation

Special case $t = l$

When the number of participating parties is the same as the threshold of the cryptosystem the key generation procedure can be simplified (see [30]). Here we give a brief description.

Instead of basing the key generation on the Feldman's VSS scheme we base it on a simpler scheme that involves no polynomials.

- Each party P_i acting as a dealer generates the random values

$$\begin{aligned} x_i &\in_R \mathbb{Z}_q \\ s_{ij},\ 1 \leq j \leq l,\ j \neq i &\in \mathbb{Z}_p \\ s_{ii} = x_i - \sum_{i \neq j,\ 1 \leq i \leq l}^{l} s_{ij} &\mod q \end{aligned}$$

- He sends privately to party j the value s_{ij} and broadcasts the commitments

$$\begin{aligned} B_0 &= g^{x_i} \\ B_j &= g^{s_{ij}},\ 1 \leq j \leq l \end{aligned}$$

- Each party P_j receiving the s_{ij} and the commitments verifies the validity as

$$g^{s_{ij}} \prod_{1 \leq z \leq l,\ z \neq i} B_z = B_0$$

The secret is then the sum of the individually generated secrets

$$x = \sum_{i=1}^{l} \sum_{j=1}^{l} s_{ij} = \sum_{i=1}^{l} x_i$$

Decryption of a value $(a, b) = (g^r, mh^r)$ can be done by having each party i calculate and broadcast the value

$$r_i = a^{-\sum_{j=1}^{l} s_{ji}}$$

and then multiplying those values together. Thus

$$m = b \prod_{i=1}^{l} r_i = ba^{-\sum_{i=1}^{l} \sum_{j=1}^{l} s_{ij}} = ba^{-x}$$

This method has similar properties as the polynomial method (see Section 2.4.1) and is more efficient in the key generation part. However we make no use of this method in our implementation, in order to allow easy extension to a real threshold system, thus we discuss this algorithm no further.

Appendix C

Data formats

Here we give a description of the transferred and stored data formats used by the implementation. In order to encode data the ASN.1 DER encoding format is used. Many of the sequences used here were inspired by the X.509 [37] definitions of public keys and certificates.

Packet contents and format

Below we give the format of the generic packet used by the communication protocol. All other packets are embedded in this one in the data field.

```
MessageType ::= ENUMERATED {
   HANDSHAKE   (0),
   PROTOCOL    (1)
}

Packet ::= SEQUENCE {
   version INTEGER(0),
   type MessageType,
   data OCTET STRING
}
```

Initial Handshake

The messages in the initial handshake are important to establish a relation with a peer. Identification information is included, initially as strings, such as `clientName` and `serverName`, and after that the protocol specific identification. This is not included in the initial `Hello` PDUs since the relation between the two parties is not known at that point.

```
ClientHelloPDU ::= SEQUENCE {
    version INTEGER(0),
```

```
    clientName PrintableString
}

Threshold ::= CHOICE {
    plain [0] INTEGER,
    encrypted [1] EncryptedBitString
}

ServerHelloPDU ::= SEQUENCE {
    version INTEGER(0),
    serverName PrintableString,
    threshold Threshold,
    parallelInitiation BOOLEAN, -- for conditional gates
    bitStringLength INTEGER
}

IDPDU ::= ElGamalID
```

The `MessageType` for these messages is `HANDSHAKE`.

Sigma proofs

Since a sigma proof always contains a set of values c and a sequence of r integer values, it will be represented as

```
AlgorithmIdentifier  ::=  SEQUENCE  {
    algorithm               OBJECT IDENTIFIER,
    parameters              ANY DEFINED BY algorithm OPTIONAL
}

SigmaProof ::= SEQUENCE {
    algorithm   AlgorithmIdentifier, -- the hash algorithm
    c           SEQUENCE OF INTEGER,
    r           SEQUENCE OF INTEGER
}
```

El-Gamal key generation

As described in Section 2.4.1, the following messages are required for an El-Gamal key generation. For the `ThresholdElGamalGenerationShare` we combine the broadcasted values `B` with the value `s` which is privately sent to a party. That is because our implementation focuses on the two-party case, thus splitting these messages seems redundant. However for the n-party case these messages should be separated.

```
ElGamalID ::= INTEGER

ElGamalGroupParameters ::= SEQUENCE {
```

```
    g   INTEGER, -- generator of a (multiplicative) subgroup of order q
    q   INTEGER, -- the order of the generator
    n   INTEGER  -- the order of the group
}

ThresholdElGamalGenerationCommitment ::= INTEGER

ThresholdElGamalGenerationShare ::= SEQUENCE {
    s   INTEGER,
    B   SEQUENCE SIZE(1..MAX) OF INTEGER
}
```

The `MessageType` for these messages is `PROTOCOL`. The format to store El-Gamal public and private keys is:

```
ThresholdElGamalPrivateKeyShare ::= SEQUENCE {
    groupName  PrintableString,
    group      ElGamalGroupParameters,
    memberId   ElGamalID,
    x          INTEGER,
    h          INTEGER, -- h=g^x
    public_h   INTEGER
}

ThresholdElGamalPartyPublicKey ::= SEQUENCE {
    memberId   ElGamalID,
    h          INTEGER
}

ThresholdElGamalPublicKey ::= SEQUENCE {
    groupName  PrintableString,
    group      ElGamalGroupParameters,
    h          INTEGER,
    hi         SEQUENCE SIZE(1..MAX) OF ThresholdElGamalPartyPublicKey
}
```

El-Gamal threshold decryption

This message contains the value $d = a^{x_i}$ and a proof that $d = a^{x_i}, h_i = g^{x_i}$.

```
ThresholdElGamalDecryptionShare ::= SEQUENCE {
    d   INTEGER,
    p   SigmaProof
}
```

The `MessageType` for these messages is `PROTOCOL`.

Conditional Gate

During the conditional gate execution the $P_i, \ll s_i \gg, [\![x_i]\!]$ and $[\![y_i]\!]$ values are transmitted. These are contained in the following data units.

```
ElGamalEncryptedValue ::= SEQUENCE {
    a   INTEGER,
    b   INTEGER
}

PedersenCommitmentValue ::= INTEGER

ConditionalGateData ::= SEQUENCE {
    p   SigmaProof,
    c   PedersenCommitmentValue,
    e1  ElGamalEncryptedValue,
    e2  ElGamalEncryptedValue
}

ConditionalGatePDU1 ::= SEQUENCE {
    id  INTEGER, -- gate id
    m   ConditionalGateData
}

ConditionalGatePDU2 ::= SEQUENCE {
    id  INTEGER, -- gate id
    m   ThresholdElGamalDecryptionShare
}

ConditionalGatesBunchPDU1 ::= SEQUENCE SIZE(1..MAX) OF ConditionalGatePDU1

ConditionalGatesBunchPDU2 ::= SEQUENCE SIZE(1..MAX) OF ConditionalGatePDU2
```

The `MessageType` for these messages is `PROTOCOL`.

Profile Matcher

```
EncryptedBitString ::= SEQUENCE SIZE(1..MAX) OF ElGamalEncryptedValue

BitStringPDU ::= EncryptedBitString
```

Signalling

For purposes of signal exchanges between the peers the following PDU is used

```
SignalPDU ::= ENUMERATED {
    DONE     (0),
    BYE      (1),
```

```
    BEGIN_KX (2)
    PROCEED_PROTO (3),
    ABORT (4)
}
```

Appendix D

El-Gamal parameters and keys

For the measurements and the testing of the *Profile matching* protocol we generated keys for two parties, named Eve and Bob. Those keys are listed in the paragraphs below.

Eve's private key

The private key of Eve is listed below and is encoded as a `ThresholdElGamalPrivateKeyShare` sequence.

```
    0:d=0  hl=4 l= 585 cons: SEQUENCE
    4:d=1  hl=2 l=   7 prim:  PRINTABLESTRING   :Eve-Bob
   13:d=1  hl=4 l= 285 cons:  SEQUENCE
   17:d=2  hl=3 l= 128 prim:   INTEGER           :0716B4A1FCE396687 \
    4757F3F5CE4A2EDDB3971DC0E5A9BCEB97F1067FC5691D3BBFB60D2A5EEBDE7 \
    BE50B4F098E440CB324D1967E0EEAE794FE75366367C81E108C819968FA6D7A \
    562A565D1C9AB84E2C3B9C88EED3A07E56576C2F70CFBD33A7FC0684C773C6E \
    115A49EC04EA223B0B356025E9C6DA0D423C91D2E652F60D9B
  148:d=2  hl=2 l=  21 prim:   INTEGER           :B262B867327CB3046 \
    089389E59E4CA1F1376EBD7
  171:d=2  hl=3 l= 128 prim:   INTEGER           :19EA7D21DCB84ED77 \
    DC1E0443261CAC1C34CE1B436D1B24B8C4F1151B4A5D2866D5E89CD357E3066 \
    AD09EFD46EBB156BD25393E8831DABB2DCAED77363A8B19881F75839BA96284 \
    44D12E6E3D4F4F404008809760AC9B6311305BAA97B67804E66B10CBCA2D88A \
    1380D7F3C8B5533CB3DE7C7107D36EE731CB0DB68E7396665F
  302:d=1  hl=2 l=   1 prim:  INTEGER           :02
  305:d=1  hl=2 l=  20 prim:  INTEGER           :16040D1CE8717DD9B7 \
    5B4FE2F3AF295B6C0CCF3A
  327:d=1  hl=3 l= 128 prim:  INTEGER           :069168DAD107444394 \
    821E09FF67AB3FEE83F5BD77215C3CA912F70AF386BAF44C03BBDDC7214AECA \
    64862D8354CC1BAEB8525DC4EF2A24F59D3CE3D12F724D5226AA26094112EB9 \
```

```
   A22D1490BEC5ABDEA3DB90546A1B7524790FE47E9F01007D452172AE9BD77F3 \
   2675767243CC125A6CC6562D0703D7F966CC3975D2E83F580
 458:d=1  hl=3 l= 128 prim:  INTEGER           :024AEA451554359FB8 \
   928A53C133630A6C0D82262E5AAF887227CC1F2667DBEC3F40FF024FB01B7EA \
   DDB4E066261F5248A30AD367A2498F2F8327D10E8B5792AD0D5C8341A4BE744 \
   B1C189DBA118CBFCF60E8BFB3EC5161F4B83112E20D2F6D41F86A240A498CA0 \
   2BB6B8C36CB6119E369A1CB72A21DC22CF0DB0C4C3C1D4264
```

Bob's private key

```
   0:d=0  hl=4 l= 585 cons: SEQUENCE
   4:d=1  hl=2 l=   7 prim:  PRINTABLESTRING   :Bob-Eve
  13:d=1  hl=4 l= 285 cons:  SEQUENCE
  17:d=2  hl=3 l= 128 prim:   INTEGER           :0716B4A1FCE396687 \
    4757F3F5CE4A2EDDB3971DC0E5A9BCEB97F1067FC5691D3BBFB60D2A5EEBDE \
    7BE50B4F098E440CB324D1967E0EEAE794FE75366367C81E108C819968FA6D \
    7A562A565D1C9AB84E2C3B9C88EED3A07E56576C2F70CFBD33A7FC0684C773 \
    C6E115A49EC04EA223B0B356025E9C6DA0D423C91D2E652F60D9B
 148:d=2  hl=2 l=  21 prim:   INTEGER           :B262B867327CB3046 \
   089389E59E4CA1F1376EBD7
 171:d=2  hl=3 l= 128 prim:   INTEGER           :19EA7D21DCB84ED77 \
   DC1E0443261CAC1C34CE1B436D1B24B8C4F1151B4A5D2866D5E89CD357E3066 \
   AD09EFD46EBB156BD25393E8831DABB2DCAED77363A8B19881F75839BA96284 \
   44D12E6E3D4F4F404008809760AC9B6311305BAA97B67804E66B10CBCA2D88A \
   1380D7F3C8B5533CB3DE7C7107D36EE731CB0DB68E7396665F
 302:d=1  hl=2 l=   1 prim:  INTEGER           :01
 305:d=1  hl=2 l=  20 prim:  INTEGER           :0D582CF06F3AD5B980 \
   C6696DAE76767DE9B7EC3F
 327:d=1  hl=3 l= 128 prim:  INTEGER           :042C73D09F1720E080 \
   F2350FACF96D84BD5BE93CF8999892D9503A8BE63B3B124DD0176BA27669F4D \
   A0C1ABCDAA7FB9E0C02BA48F4C4197E7EACFD1102F2AAEC0AE99074F34A74E1 \
   1E96D41028BED4CE6AD8842B3AD019FF83C4531002939DFD06C70F81612A1A5 \
   6141CF2DCCBC17E1AC5A6E69D7F116C8F95665D4A68D4125A
 458:d=1  hl=3 l= 128 prim:  INTEGER           :024AEA451554359FB8 \
   928A53C133630A6C0D82262E5AAF887227CC1F2667DBEC3F40FF024FB01B7EA \
   DDB4E066261F5248A30AD367A2498F2F8327D10E8B5792AD0D5C8341A4BE744 \
   B1C189DBA118CBFCF60E8BFB3EC5161F4B83112E20D2F6D41F86A240A498CA0 \
   2BB6B8C36CB6119E369A1CB72A21DC22CF0DB0C4C3C1D4264
```

The public key

This is the public key formatted as `ThresholdElGamalPublicKey`.

```
   0:d=0  hl=4 l= 707 cons: SEQUENCE
   4:d=1  hl=2 l=   7 prim:  PRINTABLESTRING   :Bob-Eve
  13:d=1  hl=4 l= 285 cons:  SEQUENCE
  17:d=2  hl=3 l= 128 prim:   INTEGER           :0716B4A1FCE396687 \
```

```
   4757F3F5CE4A2EDDB3971DC0E5A9BCEB97F1067FC5691D3BBFB60D2A5EEBDE \
   7BE50B4F098E440CB324D1967E0EEAE794FE75366367C81E108C819968FA6D \
   7A562A565D1C9AB84E2C3B9C88EED3A07E56576C2F70CFBD33A7FC0684C773 \
   C6E115A49EC04EA223B0B356025E9C6DA0D423C91D2E652F60D9B
148:d=2  hl=2 l=  21 prim:   INTEGER           :B262B867327CB3046 \
   089389E59E4CA1F1376EBD7
171:d=2  hl=3 l= 128 prim:   INTEGER           :19EA7D21DCB84ED77 \
  DC1E0443261CAC1C34CE1B436D1B24B8C4F1151B4A5D2866D5E89CD357E3066 \
  AD09EFD46EBB156BD25393E8831DABB2DCAED77363A8B19881F75839BA96284 \
  44D12E6E3D4F4F404008809760AC9B6311305BAA97B67804E66B10CBCA2D88A \
  1380D7F3C8B5533CB3DE7C7107D36EE731CB0DB68E7396665F
302:d=1  hl=3 l= 128 prim:  INTEGER           :024AEA451554359FB8 \
  928A53C133630A6C0D82262E5AAF887227CC1F2667DBEC3F40FF024FB01B7EA \
  DDB4E066261F5248A30AD367A2498F2F8327D10E8B5792AD0D5C8341A4BE744 \
  B1C189DBA118CBFCF60E8BFB3EC5161F4B83112E20D2F6D41F86A240A498CA0 \
  2BB6B8C36CB6119E369A1CB72A21DC22CF0DB0C4C3C1D4264
433:d=1  hl=4 l= 274 cons:  SEQUENCE
437:d=2  hl=3 l= 134 cons:   SEQUENCE
440:d=3  hl=2 l=   1 prim:    INTEGER           :01
443:d=3  hl=3 l= 128 prim:    INTEGER           :042C73D09F1720E0 \
  80F2350FACF96D84BD5BE93CF8999892D9503A8BE63B3B124DD0176BA27669F \
  4DA0C1ABCDAA7FB9E0C02BA48F4C4197E7EACFD1102F2AAEC0AE99074F34A74 \
  E11E96D41028BED4CE6AD8842B3AD019FF83C4531002939DFD06C70F81612A1 \
  A56141CF2DCCBC17E1AC5A6E69D7F116C8F95665D4A68D4125A
574:d=2  hl=3 l= 134 cons:   SEQUENCE
577:d=3  hl=2 l=   1 prim:    INTEGER           :02
580:d=3  hl=3 l= 128 prim:    INTEGER           :069168DAD1074443 \
  94821E09FF67AB3FEE83F5BD77215C3CA912F70AF386BAF44C03BBDDC7214AE \
  CA64862D8354CC1BAEB8525DC4EF2A24F59D3CE3D12F724D5226AA26094112E \
  B9A22D1490BEC5ABDEA3DB90546A1B7524790FE47E9F01007D452172AE9BD77 \
  F32675767243CC125A6CC6562D0703D7F966CC3975D2E83F580
```

Bibliography

[1] B. Adamson, C. Bormann, M. Handley, and J. Macker. RFC3940: Negative-acknowledgment (NACK)-oriented reliable multicast (NORM) protocol. Request for comments, 2004. IETF Network working group.

[2] S. Cheshire. It's the latency, stupid, 1996. http://rescomp.stanford.edu/~cheshire/rants/Latency.html.

[3] J. L. Cooke. Explanation of and improvements on /dev/random. http://jlcooke.ca/random/.

[4] J. L. Cooke. "[proposal/patch] fortuna prng in /dev/random", 2004. Posting to the Linux Kernel Mailing List. http://marc.theaimsgroup.com/?l=linux-kernel.

[5] CryptGenRandom function description, 2006. Microsoft Developer's Network. http://msdn.microsoft.com.

[6] T. St. Denis. *LibTomMath User Manual*. T. St. Denis, 2005. Available from http://cvs.sourceforge.net/viewcvs.py/tcl/libtommath/bn.pdf?rev=1.1.1.5.

[7] T. Speakman et. al. RFC3208: PGM reliable transport protocol specification. Request for comments, 2004. IETF Network working group.

[8] N. Ferguson and B. Schneier. *Practical Cryptography*. Willey, 2003.

[9] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. *Lecture notes in Computer Science*, 263, 1987. In Proc. of the 12th CRYPTO Conference.

[10] F. Fiorina and S. Josefsson. *Abstract Syntax Notation One (ASN.1) library for the GNU system*. Free Software Foundation, 2006. Available from http://josefsson.org/gnutls/manual/libtasn1/libtasn1.pdf.

[11] R. Gennaro, Y. Ishai, E. Kushilevitz, and T. Rabin. The round complexity of verifiable secret sharing and secure multicast. In *Proceedings of the Annual ACM Symposium on Theory of Computing, STOC'01*, 2001.

[12] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Lecture notes in Computer Science*, 1592, 1999.

[13] T. Granlund. *The GNU Multiple Precision Arithmetic Library*. Free Software Foundation, 2004. Available from http://www.swox.com/gmp/gmp-man-4.1.4.pdf.

[14] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator, 2006. Available from http://eprint.iacr.org/2006/086.

[15] M. Handley, C. Perkins, and E. Whelan. RFC2974: Session announcement protocol. Request for comments, 2000. IETF Network working group.

[16] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Wesley, 2001.

[17] J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator, 1999. Available from http://www.schneier.com/paper-yarrow.ps.gz.

[18] D. E. Knuth and A. C. Yao. The complexity of nonuniform random number generation. In J. F. Traub, editor, *Algorithms and complexity: New directions and recent results*, 1976.

[19] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge university press, 1987.

[20] A. K. Lenstra. Unbelievable security matching AES security using public key systems.

[21] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4), 2001.

[22] C. H. Lim and P. J. Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. *Lecture notes in Computer Science*, 1294, 1997.

[23] Y. Lindell and B. Pinkas. Privacy preserving data mining. *Journal of Cryptology*, 15(3), 2002.

[24] N. Mavrogiannopoulos. Profile matcher API reference manual, 2006. Internal document.

[25] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997.

[26] M. R. V. Murray. An implementation of the yarrow PRNG for FreeBSD. In *Proceedings of the BSDCon 2002*, 2002. Available from http://www.usenix.org/publications/library/proceedings/bsdcon02/full_papers/murray/murray.ps.

[27] P1363 Working Group of the Microprocessor Standards Committee. *Draft Standard for Specifications for Password based Public Key Cryptographic Techniques*. IEEE, 2006.

[28] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. *Lecture notes in Computer Science*, 576, 1992. Advances in Cryptology - CRYPTO '91.

[29] B. Schneier and N. Ferguson. *Practical Cryptography*. Wiley, 2003.

[30] B. Schoenmakers. *Cryptographic protocols*. Technische Universiteit Eindhoven, 2004.

[31] B. Schoenmakers and P. Tuyls. Practical two-party computation based on the conditional gate. *Lecture notes in Computer Science*, 3329, 2004.

[32] B. Schoenmakers and P. Tuyls. Private profile matching. In *Proceedings of the second Philips Symposium on Intelligent Algorithms, SOIA'04*, 2004.

[33] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2005.

[34] M. Stam. *Speeding up Subgroup Cryptosystems*. PhD thesis, Technische Universiteit Eindhoven, 2003.

[35] W. R. Stevens. *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*. Prentice Hall, 1998.

[36] International Telecommunication Union. *Recommendation X.680-683: Information Technology - Abstract Syntax Notation One (ASN.1)*. International Telecommunication Union, 1994.

[37] International Telecommunication Union. *Recommendation X.509: Information Technology - Open Systems Interconnection - The Directory: Authentication Framework*. International Telecommunication Union, 1997.

[38] P. C. van Oorschot and M. J. Wiener. On diffie-hellman key agreement with short exponents. *Lecture notes in Computer Science*, 1070, 1996.

[39] J. Villavicenc. "no entropy and no output at /dev/random (quick question)", 2004. Posting to the Linux Kernel Mailing List. http://marc.theaimsgroup.com/?l=linux-kernel.

[40] A. C. Yao. Protocols for secure computations. In *IEEE Foundations of computer science*, 1982.