Distributed RSA Key Generation
An Implementation for the Two-Party Scenario

Diploma Thesis
by
Georg Lippold
August 19th, 2006

Prof. Dr. Michael Hortmann
Prof. Dr. Hans-Jörg Kreowski
Abstract

Standard digital certificates bear only one signature. This is a single point of failure, since this signature is usually generated by one person on one computer. If the secret key in the signature generator is compromised, the complete chain of trust for all certificate holders is broken. A very common asymmetric cipher used for certificates is RSA, as described in Kalinski and Staddon [KS98].

To avoid a compromise of the signature generator, secret keys used in sensitive environments like trust centers are usually distributed between different entities. This is most easily done by a trusted dealer generates key shares. But this is only a shift of the problem: If the dealer is compromised, so is the trust in the key.

Boneh and Franklin [BF97] developed distributed RSA key generation for $k$ parties using the protocol described in Ben-Or et al. [BOGW88]. The resulting shared RSA key is $\left\lfloor \frac{k-1}{2} \right\rfloor$ private. Later, Gilboa [Gil99] introduced two party distributed RSA key generation that is $k - 1$ private. In 2003, Straub [Str03] partially improved Gilboa’s protocol by generating multi-prime RSA keys for two parties.

In this thesis, we implement Gilboa’s protocol, which uses parts of the original protocol developed by Boneh and Franklin.

For this, an implementation of the Benaloh cryptosystem [Ben94] and the Goldwasser-Micali cryptosystem [GM] had to be done, since there are (to my knowledge) no public implementations available. Furthermore, Oblivious Transfers as described in Even et al. [EGL85] were implemented. They are used to compute the private exponent of the RSA key from the generated shares of the RSA modulus.
# Contents

1 Introduction 7

2 Mathematical Preliminaries and Tools 9
   2.1 Notations ........................................................... 9
   2.2 Fast Exponentiation ............................................... 10
   2.3 Euclidean Algorithm ............................................. 10
   2.4 Extended Euclidean Algorithm ................................. 11
   2.5 Chinese Remainder Theorem .................................. 12
   2.6 Fermat Test .......................................................... 12
   2.7 Legendre Symbol .................................................. 14
   2.8 Jacobi Symbol ..................................................... 14
   2.9 Twisted Group of $\mathbb{Z}_N$ .................................. 15

3 Cryptosystems 17
   3.1 RSA Cryptosystem .................................................. 19
   3.2 Goldwasser-Micali Cryptosystem ............................... 20
      3.2.1 General Description ...................................... 20
      3.2.2 System Setup ................................................. 21
   3.3 Benaloh Cryptosystem ............................................ 22
      3.3.1 General Description ...................................... 22
      3.3.2 System Setup ................................................. 23
   3.4 Naccache-Stern Cryptosystem ................................. 25
      3.4.1 General Description ...................................... 25
      3.4.2 System Setup ................................................. 25

4 Constructing a Shared RSA Key 29
   4.1 Preliminary Remarks ............................................. 29
      4.1.1 Implementation Details .................................. 30
   4.2 Gilboa’s Protocol .................................................. 32
      4.2.1 Description .................................................. 32
      4.2.2 Program Design ............................................. 33
Chapter 1

Introduction

The motivation for the development of techniques for distributed RSA key generation is clear: In sensible areas concerned with computer security, there is a need to impose a multi-eye principle. The RSA cryptosystem was developed by Ron Rivest, Adi Shamir, and Len Adleman and bears their initials as an identifier. It is a widely adopted standard (see Kalinski and Staddon [KS98]) that is used by many programs. Usually, RSA keys for sensitive environments are built by a trusted dealer who generates a standard RSA key and then splits it into multiple parts.

The problem with a trusted dealer is that there is again a single point of failure: If the dealer is compromised, so is the key. So there arises a need to securely compute a shared RSA key such that neither party ever has the full key or is able to reconstruct the full key from the communication protocol.

Starting with Boneh and Franklin [BF97] in 1997, there was a development of techniques to jointly generate shared RSA secret keys. Boneh and Franklin developed a scheme that allowed \( k \geq 3 \) parties to jointly generate a key for that any coalition of at most \( \left\lfloor \frac{k-1}{2} \right\rfloor \) parties cannot recover the complete key. They used the BGW protocol (see Ben-Or et al. [BOGW88]) to compute the sharing. For the Boneh-Franklin protocol an open source implementation is available from the ITTC website (see [Bon99]).

There exist a few papers about two-party RSA key generation but until today no open source software for the two party scenario is available. To verify the claims made by the authors an open implementation is both necessary and useful.

In the course of this thesis such an implementation is developed. In chapter 2 on page 9 the mathematical tools needed are proven and an example implementation for the algorithms is given where applicable.

Chapter 3 on page 17 deals with the cryptosystems used in the process of generating a shared RSA key. I explain the underlying principle for each
cryptosystem, show how the system is set up for two parties and give a sample implementation – except for the RSA cryptosystem which has several open implementations.

Implementation-specific details and the complete process of the generation of a shared RSA key are explained in chapter 4 on page 29. First, a shared RSA modulus $N$ is constructed. A specially designed bi-primality test verifies that the modulus is not composed of more than two primes. Moduli that consist of more than two primes are detected with probability at least $\frac{1}{2}$ in each run of the test. The technique of Oblivious Transfers is explained which is in turn used to compute the private exponent $d$ of the RSA key. The chapter finishes with the computation of the private exponent.

The experimentally verified results are presented in chapter 5 on page 49. Here, a performance analysis and a statistical verification of the process is given. The outcome is compared to the claims of Gilboa in [Gil99] and evaluated for compatibility with off-the-shelf software.

A short outlook on what could be implemented in further steps of the software is given in chapter 6 on page 53. A classification into the scientific context concludes the thesis.

At the end of every entry in the bibliography is a back reference of the pages on which the entry is cited. Furthermore, the bibliography contains references that are only available online and are not printed. Thus, they may be subject to change with time and the status in which I viewed them may not be recoverable (although the Way Back Machine on http://www.archive.org may be able to). To prevent this, and to facilitate the recapitulation of the current work, all literature used is also on the accompanying CD in the folder DiplomaThesis/literature.
Chapter 2

Mathematical Preliminaries and Tools

In the course of this thesis, there are some mathematical recipes used that may or may not be known to the reader. I give a short introduction into the most commonly used techniques throughout this thesis. Not all functions described here are implemented in the standard Java API, with which the accompanying application is programmed. Therefore I give example implementations where applicable.

2.1 Notations

There are some symbols used in the calculations, a short overview of their meaning is given in table 2.1.

<table>
<thead>
<tr>
<th>condition:equation</th>
<th>equation holds for condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \prod_{i}^{j} )</td>
<td>Product from index ( i ) to index ( j )</td>
</tr>
<tr>
<td>( \sum_{i}^{j} )</td>
<td>Sum from index ( i ) to index ( j )</td>
</tr>
<tr>
<td>( x \leftarrow N )</td>
<td>( x ) uniformly at random from ( N )</td>
</tr>
<tr>
<td>\forall \</td>
<td>For all</td>
</tr>
<tr>
<td>( \in )</td>
<td>Element of</td>
</tr>
<tr>
<td>( \mathbb{Z}_p )</td>
<td>The field of ( x ) mod ( p ) with ( p ) prime for all ( x \in \mathbb{N} )</td>
</tr>
</tbody>
</table>

Table 2.1: Mathematical Symbols
2.2 Fast Exponentiation

Exponentiation is widely used throughout the thesis. The reader may think that exponentiation is a rather expensive operation because it involves many multiplications. But we can use a shortcut by defining

\[
\text{Power}(x, n) = \begin{cases} 
  x & \text{if } n = 1 \\
  \text{Power}(x^2, n/2) & \text{if } n \mod 2 = 0 \\
  x \cdot \text{Power}(x^2, n - 1/2) & \text{if } n \mod 2 = 1.
\end{cases}
\]

For more algorithms on exponentiation, please see Menezes et al. page 613 – 618 [MvOV96].

2.3 Euclidean Algorithm

The Euclidean Algorithm (see also Menezes et al. page 66 [MvOV96] and Wikipedia [Wik06b]) determines the greatest common divisor of two numbers \(a\) and \(b\) without the need to factor them. In this thesis it is used as \(\gcd(a, b)\).

Without loss of generality, if \(a > b\) we can write

\[r_0 = q_1 r_1 + r_2\]

with \(r_0 = a\) and \(r_1 = b\). \(q_1\) is the quotient of the division of \(r_0\) by \(r_1\) and \(r_2\) is the remainder. Since \(r_2 = r_0 - q_1 r_1\), any divisor of \(r_0\) and \(r_1\) is also a divisor of \(r_2\):

If there is a common divisor \(d\) of \(r_0\) and \(r_1\), then we have \(r_0 = dx\) and \(r_1 = dy\), thus \(r_2 = dx - q_1 dy = d(x - q_1 y)\). Since \(r_2, r_0\) and \(r_1\) have the same divisor, we can continue the search with the smaller two of them, \(r_1\) and \(r_2\).

Eventually, it will be found that \(r_{n-1} = q_n \cdot r_n + 0\), now \(r_n\) is the greatest common divisor of \(r_0 = a\) and \(r_1 = b\).

The algorithm is a very nice recursion:

```java
int gcd(a, b) { 
    if(b == 0) { 
        return a; 
    } 
    return gcd(b, a mod b); 
}
```

The Euclidean Algorithm is implemented in the standard Java class Big-Integer. The method is named \(\gcd()\).
2.4 Extended Euclidean Algorithm

The Extended Euclidean Algorithm is basically the Euclidean Algorithm with bookkeeping. It determines \( ax + by = \gcd(a, b) \).

\[
\begin{align*}
ax + by & = \gcd(a, b) \\
\mod a & \Rightarrow \quad 0 + by = \gcd(a, b) \quad (\mod a) \\
\mod b & \Rightarrow \quad ax + 0 = \gcd(a, b) \quad (\mod b).
\end{align*}
\]

Thus, if \( \gcd(a, b) = 1 \) (\( a \) and \( b \) are coprime), then \( y \) is the multiplicative inverse of \( b \) in \( \mathbb{Z}_a \) and \( x \) is the multiplicative inverse of \( a \) in \( \mathbb{Z}_b \).

As we have seen in section 2.3 on the facing page, the Euclidean Algorithm is

\[
r_{k+1} = r_{k-1} - q_k r_k. \quad (2.1)
\]

Let us take two additional terms of the same form, namely

\[
s_{k+1} = s_{k-1} - q_k s_k \\
t_{k+1} = t_{k-1} - q_k t_k.
\]

With \( s_0 = 0, s_1 = 1 \) and \( t_0 = 1, t_1 = 0 \) we can write

\[
r_k = s_k r_0 + t_k r_1. \quad (2.2)
\]

The following proof shows this by induction over \( k \):

\[
\begin{align*}
k = 0 : & \quad r_0 = 1 \cdot r_0 + 0 \cdot r_1 \\
k = 1 : & \quad r_1 = 0 \cdot r_0 + 1 \cdot r_1. \quad (2.3)
\end{align*}
\]

Now observe \( r_{k+1} \) for \( k \geq 1 \):

\[
r_{k+1} \overset{2.1}{=} r_{k-1} - q_k r_k \overset{2.2 \text{ & } 2.3}{=} s_{k-1} r_0 + t_{k-1} r_1 - q_k (s_k r_0 + t_k r_1) \\
= (s_{k-1} - q_k s_k) r_0 + (t_{k-1} - q_k t_k) r_1 = s_{k+1} r_0 + t_{k+1} r_1.
\]

Since \( r_0 = a \) and \( r_1 = b \) and \( r_k = \gcd(a, b) \) we now have an algorithm that finds \( x \) and \( y \) as stated above.
2.5 Chinese Remainder Theorem

The Chinese Remainder Theorem (see also Menezes et al. page 68–69 and 611–613 [MvOV96] and Wikipedia [Wik06a]) was developed by the mathematician Sun Tsu Suan-Ching in the 4th century A.D. It is applied to re-compose an integer \( x \) from its residues modulo pairwise coprime integers \( p_1, p_2, \ldots, p_n \). Pairwise coprime means that for two integers \( p_i, p_j \), \( p_i \neq p_j : \gcd(p_i, p_j) = 1 \).

Thus, we want to find an \( x \) for that

\[
x \mod p_i = a_i. \tag{2.4}
\]

With \( N = \prod_{i=1}^{n} p_i \) and \( \gcd(p_i, \frac{N}{p_i}) = 1 \) by construction, we can use the Extended Euclidean Algorithm (see section 2.4 on the page before) to solve

\[
x \cdot p_i + y \cdot \frac{N}{p_i} = 1. \tag{2.5}
\]

If we take 2.5 modulo \( p_i \) we get

\[
y \cdot \frac{N}{p_i} \mod p_i = 1
\]

and modulo \( p_j \)

\[
\forall p_j \neq p_i : \quad y \cdot \frac{N}{p_i} \mod p_j = 0.
\]

Thus, to reconstruct an \( x \) that fulfills equation 2.4 we have with \( e_i = y \cdot \frac{N}{p_i} \):

\[
x = \sum_{i=1}^{n} a_i e_i \pmod{N}.
\]

Other solutions are \( x + q \cdot N \) for \( q = 1, 2, \ldots \)

The Java code for the Chinese Remainder Theorem is in listing A.1 on page 61.

2.6 Fermat Test

The Fermat Test is also known as Fermat’s Little Theorem (see also Weisstein [Wei04]). It is based on the observation that for any natural number \( a \) and any prime \( p \) the following condition holds true:
2.6. **FERMAT TEST**

\[
a^p \mod p = a \mod p
\]

\[
\Rightarrow (a^p - a) \mod p = 0
\]

This can be proved by induction over \(a\). The anchor is \((1^p - 1) \mod p = 0\) for \(a = 1\). Now consider

\[
(a + 1)^p - (a + 1).
\]

Using the binomial theorem, \((a + 1)^p\) expands to

\[
(a + 1)^p = a^p + \binom{p}{1} a^{p-1} + \binom{p}{2} a^{p-2} \ldots + \binom{p}{p-1} a + 1
\]

\[
\Rightarrow (a + 1)^p - a^p - 1 = \binom{p}{1} a^{p-1} + \binom{p}{2} a^{p-2} \ldots + \binom{p}{p-1} a.
\]

If \(p\) is prime, \(p\) divides

\[
\binom{p}{1} a^{p-1} + \binom{p}{2} a^{p-2} \ldots + \binom{p}{p-1} a.
\]

Thus, \(p\) also divides

\[
(a + 1)^p - a^p - 1
\]

and therefore \(p\) also divides

\[
(a + 1)^p - a^p - 1 + (a^p - a)
\]

since \((a^p - a)\) is divisible by \(p\) as proved in the anchor. Now we have:

\[
(a + 1)^p - a^p - 1 + (a^p - a) = (a + 1)^p - (a + 1)
\]

and thus the induction holds for any \(a\).

If \(p\) does not divide \(a\), we can divide \((a^p - a) \mod p = 0\) by \(a \mod p\) and get

\[
a^{p-1} - 1 \mod p = 0.
\]

This provides us with a primality test: If \(a^{p-1} \mod p = 1 \mod p\), then we have a strong indication that \(p\) may be prime. If this test is done for different \(a\)'s it is an even stronger indication that \(p\) may be prime. It is not a complete primality test, since there are Carmichael Numbers for which
this test is always true. These numbers are of the form \( N = p_1 \cdot p_2 \cdot p_3 = (6k + 1)(12k + 1)(18k + 1) = 1296k^3 + 396k^2 + 36k + 1 \), if \( p_1 = 6k + 1 \) and \( p_2 = 12k + 1 \) and \( p_3 = 18k + 1 \) are prime. Thus, \( N - 1 \) is a multiple of \( 36k \) which is also the least common multiple of \( p_1 - 1, p_2 - 1 \) and \( p_3 - 1 \). Since \( p_1, p_2 \) and \( p_3 \) are prime, the equation \( a^{p_x - 1} \mod p_x = 1 \mod p_x \) holds for each of them. Hence, since \( N \) is the product of the three primes, \( a^{N - 1} \mod N = 1 \). For further details about Carmichael Numbers, see also Weisstein [Wei05].

2.7 Legendre Symbol

With the Fermat Test (see 2.6 on page 12), we can now test any \( a \) in \( \mathbb{Z}_p \) with \( p \) prime, as to whether it fulfills the condition that \( r^2 = a \). If such an \( r \) exists, it is said that \( a \) has “a quadratic residue modulo \( p \)”.

We define the Legendre Symbol as \( \left( \frac{a}{p} \right) \) for any integer \( a \) and any prime \( p \). The Legendre Symbol has the following properties:

\[
\left( \frac{a}{p} \right) = \begin{cases} 
0 & \text{if } p \text{ divides } a \\
1 & \text{if } a \text{ is a quadratic residue modulo } p \\
-1 & \text{if } a \text{ is not a quadratic residue modulo } p.
\end{cases}
\]

The Fermat Test showed that for any prime \( p \) and any integer \( a 
\]
\[ a^{p-1} \mod p = 1 \mod p. \]

For \( a = r^2 \), we have
\[ a^{\frac{p-1}{2}} \mod p = (r^2)^{\frac{p-1}{2}} \mod p = r^{p-1} \mod p = 1 \mod p. \]

Thus we can easily test for quadratic residues modulo \( p \) by computing \( a^{\frac{p-1}{2}} \).

The Java source code for determining the Legendre Symbol is in listing A.2 on page 61.

2.8 Jacobi Symbol

The Jacobi Symbol is a generalization of the Legendre Symbol. For any positive odd integer \( q \), the Jacobi Symbol is defined as

\[
J(a, q) = \left( \frac{a}{q} \right) = \left( \frac{a}{p_1} \right)^{b_1} \cdot \left( \frac{a}{p_2} \right)^{b_2} \cdot \ldots \cdot \left( \frac{a}{p_n} \right)^{b_m}
\]
2.9. TWISTED GROUP OF $\mathbb{Z}_N$

where $q = p_1^{b_1} \cdot p_2^{b_2} \cdots \cdot p_n^{b_n}$ is the prime factorization of $q$ and $\left( \frac{a}{p_x} \right)$ is the Legendre Symbol for prime $p_x$.

There are some shortcuts for the computation of $\left( \frac{a}{q} \right)$ (taken from page 4 in Fröschler [Fl05]):

- If $a = 0$ return 0.
- If $a = 1$ return 1.
- If $a > q$ return $J(a \bmod q, q)$.
- If $a \bmod 4 = 0$ return $J(a \bmod 4, q)$.
- If $a \bmod 2 = 0$ and ($q \bmod 8 = 1$ or $q \bmod 8 = 7$) return $J(a/2, q)$.
- If $a \bmod 2 = 0$ and ($q \bmod 8 = 3$ or $q \bmod 8 = 5$) return $-J(a/2, q)$.
- If $a \bmod 4 = 1$ or $q \bmod 4 = 1$ return $J(q \bmod a, a)$.
- If $a \bmod 4 = 3$ or $q \bmod 4 = 3$ return $-J(q \bmod a, a)$.

The Java source code to compute the Jacobi Symbol is in listing A.3 on page 62.

2.9 Twisted Group of $\mathbb{Z}_N$

In the ring $\mathbb{Z}_N[\frac{x}{x^2+1}]$ with $N \equiv 3 \bmod 4$, all elements are of the form $ax + b$ where $a$ and $b$ are Elements of $\mathbb{Z}_N$. If we multiply two elements, we get a term that contains $x^2$. Since $x^2 + 1$ shall always be zero, we set $x^2 = -1$.

This lets us calculate with the elements of $\mathbb{Z}_N[\frac{x}{x^2+1}]$ analogously to the complex numbers if we simply eliminate $x$ from $ax + b$ by using

\[(a, b) + (c, d) = ((a + c), (b + d))\]
\[(a, b) \cdot (c, d) = ((ac - bd), (ad + bc))\]

and reducing modulo $\mathbb{Z}_N$.

$\mathbb{Z}_N$ itself is then a sub ring of $\mathbb{Z}_N[\frac{x}{x^2+1}]$. This can be seen by setting $(e, 0)$ for any $e \in \mathbb{Z}_N$. All these elements behave in analogously to the native elements of $\mathbb{Z}_N$.

The “Twisted Group” (the name is adopted from Straub [Str03, page 2]) of $\mathbb{Z}_N$ now is the Group $\mathbb{T}_N = \frac{\mathbb{Z}_N[x]}{x^2+1}/\mathbb{Z}_N$. In $\mathbb{Z}_N[x]/(x^2 + 1)$, we have $(a, b)(a, -b) = (a^2 + b^2, 0)$. Thus, in $\mathbb{T}_N$, the multiplicative inverse of $(a, b)$ is $(a, -b)$ and all elements $(a, 0)$ with $a \in \mathbb{Z}_N$ are remapped to $(1, 0)$. 

Chapter 3

Cryptosystems

It is common knowledge that there are two kinds of encryption: The first are the symmetric ciphers that encrypt a given plaintext into a ciphertext using a key that is known to both parties. The same key is used for decryption by applying it to the ciphertext.

Then, there are asymmetric ciphers that consist of a public key and private key. Both keys are constructed such that they do mathematically inverse operations. In this thesis, only asymmetric ciphers are used.

For asymmetric ciphers, one can further differentiate between deterministic and probabilistic encryption.

Deterministic asymmetric encryption encrypts the same plaintext always to the same ciphertext.

Probabilistic encryption, also known as homomorphic encryption, has characteristics that are very useful for generating shared secrets:

- There are multiple encodings for the same message using the same key pair and algorithm. Thus, it is much harder to do chosen plaintext attacks.

- It is possible to apply arithmetical operations like addition, subtraction and multiplication to encrypted messages without the need to decipher them.

- Upon encryption, the user gains a certificate with which he can prove that he encrypted the message. This certificate is not recoverable upon decryption.

A drawback is that homomorphic encryption has a very high expansion rate when converting plaintext into ciphertext.
All operations in homomorphic encryption schemes are modulo $N = p \cdot q$ and usually there is a maximum plaintext message size that can be encrypted in a single step. Throughout this thesis the message limit is called $r$ and only messages smaller than $r$ can be encrypted. Thus, addition and multiplication works only for small messages that can be encrypted in one step. Operations can be applied to larger messages by splitting the message into smaller parts so that the operation is transparently carried on. Then the operation is applied to each part and the result is reconstructed from the modified parts. In this thesis, the Chinese Remainder Theorem is used to this end.

Addition of encrypted messages is achieved by multiplying the messages:

$$c((m_1 + m_2) \mod r) = c(m_1) \cdot c(m_2) \mod N.$$  

Multiplication is done via exponentiation, here the exponent has to be in plaintext

$$c((m_1 \cdot m_2) \mod r) = c(m_1)^{m_2} \mod N.$$  

In the course of this thesis, multiple encryptions schemes are used to exchange messages between the two parties. Here is a short overview:

The first asymmetric cipher I would like to mention is the RSA cipher. It is widely used during the Oblivious Transfers (see 4.3 on page 40) and the generation of a shared RSA key pair is also the goal of this thesis.

In order to use the protocol described by Gilboa in [Gil99], the Goldwasser-Micali cryptosystem as first published by in [GM] as well as the Benaloh cryptosystem that was published in [Ben94] had to be implemented. Additionally, the Naccache-Stern cryptosystem as explained in [NS98] was implemented for later optimizations.

In this chapter the similarities and differences between these systems will be examined and a short explanation on the cryptographic mechanisms used therein will be given. In appendix A.2 on page 64 implementation details for the systems are available. For a complete implementation please see the accompanying source distribution.

In the folder

DistribRSA/src/de/uni_bremen/informatik/lippold/cryptosystems

you will find the sources for the mentioned classes.
3.1 RSA Cryptosystem

The RSA cryptosystem is named after its developers Ron Rivest, Adi Shamir, and Len Adleman. It is based on the hardness of factoring the product of two large primes $N = p \cdot q$ with $p \neq q$. The RSA cryptosystem is probably the most widely used deterministic asymmetric cipher today.

In a group $G$ with order $|G|$, for every element $a \in G$ there is a order $k = \text{ord}(a)$ so that $a^k = 1$. It is obvious that $k$ divides $|G|$ since $|G|$ must be the least common multiple of all $k_i$. Thus we can write $\text{ord}(G) = \text{ord}(a) \cdot q$.

We now have

$$a^{\text{ord}(G)} = a^{\text{ord}(a) \cdot q} = (a^{\text{ord}(a)})^{q} = 1^q = 1.$$ 

We are now interested in the order of $\mathbb{Z}_N$, $|\mathbb{Z}_N|$ (see also Wikipedia [Wik06d] and Wikipedia [Wik06c]). Since $N = p \cdot q$, we must remove all multiples of $p$ and $q$ from $\mathbb{Z}_N$ to get the order of $\mathbb{Z}_N$. For $p$ we have the elements

$$0 \cdot p, p, 2p, \ldots, (q - 1)p$$ 

which are $q$ elements, and for $q$ we get likewise $p$ elements that have to be removed. Thus the order of $\mathbb{Z}_N$ is $\sum_{p=q}^{N} -p - q + 1 = (p - 1)(q - 1)$ since $0$ was counted twice. And thus in $\mathbb{Z}_N$ we have $\forall x \in \mathbb{Z}_N : x^{(p-1)(q-1)} = 1$ and therefore $x^{(p-1)(q-1)+1} = x^{(p-1)(q-1)} \cdot x = x$.

This leads us to a cryptosystem:

- Let $\varphi(N) = (p - 1)(q - 1)$. Choose an $e$ that fulfills $\gcd(e, \varphi(N)) = 1$ (a widely used but potentially unsafe $e$ is $2^{16} + 1$, for details please see Fouque et al. [FKJM+06]). Compute $d$ so that $de = 1 \mod \varphi(N)$. From the Extended Euclidean Algorithm (see section 2.4 on page 11) we know that we can find $d$ and $b$ such that

$$b\varphi(N) + de = \gcd(\varphi(N), e) = 1.$$ 

Taking this modulo $\varphi(N)$, we get $0 + de \equiv 1 \pmod{\varphi(N)}$ and thus $de - 1 \equiv \varphi(N)$.

- Now we can encrypt an $x \in \mathbb{Z}_N$: Let $y = x^e$ be the encryption. To decrypt, we simply compute $y^d = (x^e)^d = x^{ed} = x^{(de-1)+1} = x^{b\varphi(N)+1} = (x^{b\varphi(N)})x = 1 \cdot x = x$. 


Thus, we have an elegant cryptosystem: Encryption is done by computing \( y = x^e \) and decryption by \( y^d = x \). If one publishes \( e \) and \( N \) and keeps \( d \) secret, everybody can encrypt but only the person knowing \( d \) (which could be reconstructed from \( p \) and \( q \)) can decrypt. Thus, we have established an asymmetric cipher system.

### 3.2 Goldwasser-Micali Cryptosystem

#### 3.2.1 General Description

A description of the Goldwasser-Micali cryptosystem is given in \([GM]\). The security is based on two assumptions:

- It is not feasible to factor the product of two large primes \( N = p \cdot q \).
  
  This problem is known as integer factoring.

- If the Jacobi symbol for a given integer \( x < N \) is \( J(x, N) = 1 \), it is not decidable whether \( x \) is of the form \( x = a \cdot a \) so that \( J(x, p) = J(x, q) = 1 \Rightarrow J(x, N) = 1 \) or of the form \( J(x, p) = -1 \) and \( J(x, q) = -1 \), so \( J(x, N) = J(x, p) \cdot J(x, q) = 1 \).
  
  This problem is known as quadratic residue problem.

In \( \mathbb{Z}_p \), there are \( \frac{p-1}{2} \) quadratic residues, excluding zero. They stem from the numbers \( 1^2, 2^2, \ldots, (\frac{p-1}{2})^2 \). Evidently, there are also \( \frac{p-1}{2} \) quadratic non-residues. In \( \mathbb{Z}_N \) with \( N = p \cdot q \), \( p \) prime, \( q \) prime and \( p \neq q \), there are \( N - (p-1) \cdot (q-1) \) divisors of zero. The elements with \( J(x, N) = 1 \) stem from the \( \frac{p-1}{2} \cdot \frac{q-1}{2} = \frac{(p-1)(q-1)}{4} \) quadratic residues in \( p \) and \( q \) with \( J(x, p) = J(x, q) = 1 \) and the \( \frac{p-1}{2} \cdot \frac{q-1}{2} = \frac{(p-1)(q-1)}{4} \) quadratic non-residues in \( p \) and \( q \) with \( J(x, p) = -1 \) and \( J(x, q) = -1 \).

Thus we have \( \frac{(p-1)(q-1)}{2} \) elements with \( J(x, N) = 1 \). Analogously, we get \( \frac{p-1}{2} \cdot \frac{q-1}{2} \) elements with \( J(p, x) = 1 \) and \( J(q, x) = -1 \) and again \( \frac{(p-1)(q-1)}{4} \) elements with \( J(q, x) = 1 \) and \( J(p, x) = -1 \). Thus, we also have \( \frac{(p-1)(q-1)}{2} \) elements \( x \) with \( J(x, N) = -1 \).

Therefore, given that \( J(x, N) = 1 \), one cannot decide whether \( x \) is a quadratic residue or not if one cannot factor \( N \).

Given the factorization of \( N \), one can simply test if an integer \( x \) with Jacobi symbol 1 is a quadratic residue: If \( J(x, p) = 1 \) and \( J(x, q) = 1 \), then \( x \) is a quadratic residue.

Now, if there were a means to compute the root of a given square in \( N \), the factorization problem would be solved:
3.2. GOLDWASSER-MICALI CRYPTOSYSTEM

<table>
<thead>
<tr>
<th>Goldwasser-Micali cryptosystem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Private key parameters</strong></td>
</tr>
<tr>
<td>$p$ prime</td>
</tr>
<tr>
<td>$q$ prime</td>
</tr>
<tr>
<td><strong>Public key parameters</strong></td>
</tr>
<tr>
<td>$N = p \cdot q$</td>
</tr>
<tr>
<td>{ $y \in N \mid J(y, N) = 1$ $\wedge$ $J(y, p) = J(y, q) = -1$ }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plaintext</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m \in {0, 1}$ (1 bit)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ciphertext</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c(m) \in \mathbb{Z}_N$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Random element</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \leftarrow \mathbb{Z}_N$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \leftarrow \mathbb{Z}_N : u^2 \cdot y^m \mod N$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J(c(m), p) \wedge J(c(m), q) \equiv -1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expansion rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \rightarrow N$</td>
</tr>
</tbody>
</table>

Table 3.1: Goldwasser-Micali parameters

Any element $x \in \mathbb{Z}_N$ is composed of the two remainders modulo $p$ and $q$. Thus, $x$ can be written as $x = x_p \cdot x_q = x_{\mathbb{Z}_N}$. For any square $x^2 \in \mathbb{Z}_N$, there obviously exist the roots $-x_p \cdot -x_q, -x_p \cdot x_q, x_p \cdot -x_q$ and $x_p \cdot x_q$ (because squared all these integers lead to $x^2 \cdot r^2$). If we could compute the root of a given $x^2$, the algorithm would either give all four possible roots, or with probability $\frac{1}{2}$ a root that is not $x$ and not $-x$, let us call it $r$.

In $\mathbb{Z}_N$, we have $0 = x^2 - r^2 = (x+r)(x-r)$. Thus, $N$ divides $(x+r)(x-r)$. Now we can easily factor $N$, since $gcd(x + r, N)$ and likewise $gcd(x - r, N)$ gives a factor of $N$.

Thus, solving the quadratic residue problem is as hard as factoring $N$.

A summary of the Goldwasser-Micali system is in table 3.1.

3.2.2 System Setup

**Key generation**

For a cryptosystem with strength $k$ bit, Alice chooses two primes $p$ and $q$, each of size $\frac{k}{2}$ bit. She sets $N = p \cdot q$. She then finds an integer $y$ for which $J(y, N) = 1$ but $J(y, p) = J(y, q) = -1$. and then publishes $y$ and $N$ but keeps $p$ and $q$ secret. The Java code for the key generation is in listing A.4 on page 64.
CHAPTER 3. CRYPTOSYSTEMS

Encryption

Bob, who wants to send a message \( m \) which is represented as a bit string of length \( n \), encrypts each bit \( m[i] \) in the following way and sends it to Alice:

Bob chooses a random integer \( u \in \mathbb{Z}_N \). He then computes \( u^2 \cdot y^{m[i]} \mod N \). Clearly, if \( m[i] = 0 \), then \( J(c(m), N) = J(u^2, N) = 1 \). On the other hand, if \( m[i] = 1 \), then \( J(c(m), N) = J(u^2 \cdot y, N) = J(u^2, N) \cdot J(y, N) = 1 \). Thus, all messages that Bob sends to Alice have the same Jacobi symbol, and the content cannot be determined without knowing the factorization of \( N \). The Java code for Bob is in listing A.5 on page 64.

Decryption

Alice now decrypts the message:

If \( c(m) \) is a quadratic residue modulo \( p \) and also a quadratic residue modulo \( q \), then Bob’s bit was 0, else it was 1. The Java code for Alice is in listing A.6 on page 65.

3.3 Benaloh Cryptosystem

3.3.1 General Description

Benaloh explains his cryptosystem in [Ben94]. The security of this cryptosystem is based on two problems which are cryptographically hard:

1. The “higher residuosity” or “discrete logarithm” problem: “Given \( z, r \) and \( n \) of unknown factorization, there is no known polynomial time algorithm to determine whether or not there exists an \( x \) such that \( z = x^r \mod n \).” (from Benaloh in [Ben94])

The inverse operation, the exponentiation, is feasible as shown in section 2.2 on page 10.

The same problem is also used in the Naccache-Stern cipher (see section 3.4 on page 25) and the popular Diffie-Hellman [Res99] cryptosystem.

2. The problem of factoring the product of two large primes.

The system consists of a modulus \( N = p \cdot q \) with \( p, q \) prime and \( p \neq q \), a public base for exponentiation \( y \), and the upper limit for exponents \( r \) which is an odd integer chosen before system setup. In this system, the message is stored in the exponent. Then, \( p, q \) and \( y \) are generated as follows:
3.3. BENALOH CRYPTOSYSTEM

- $r$ divides $p - 1$ and $\gcd(r, \frac{p-1}{r}) = 1$
- $\gcd(r, q - 1) = 1$
- $N = p \cdot q$, thus $r$ divides $\varphi(N) = (p-1)(q-1)$ exactly once.
- $\gcd(y, N) = 1$ and $y^{\frac{(p-1)(q-1)}{r}} \mod N \neq 1$

A message $m < r$ is encrypted as $c(m) = y^m \cdot u^r \mod N$, where $u$ is a random element in $\mathbb{Z}_N$. It is easy to compute the plaintext $m$, given $\varphi(N) = (p-1)(q-1)$:

\[
\forall x \in \mathbb{Z}_N : x^{(p-1)(q-1)+1} = x \text{ and } x \in \mathbb{Z}_N : x^{(p-1)(q-1)} = 1 \text{ as we have seen in section 3.1 on page 19. Now we can compute } m \text{ from } c(m) \text{ (all computations modulo } N) :
\]

\[
c(m)^{\frac{(p-1)(q-1)}{r}} = (y^m \cdot u^r)^{\frac{(p-1)(q-1)}{r}} = y^{\frac{m(p-1)(q-1)}{r}} \cdot u^{\frac{(p-1)(q-1)}{r}} = y^{\frac{m(p-1)(q-1)}{r}}.
\]

Since we chose $y$ in such a way that $y^{\frac{(p-1)(q-1)}{r}} \neq 1$, there exists a canonical value $y^{\frac{(p-1)(q-1)}{r}}$ for every possible message $m < r$. If we pre-compute this value for all possible messages $\{0, 1, \ldots, r-1\}$, we can then simply look it up in a table and retrieve the original message.

A short overview of the Benaloh system is in table 3.2 on the following page.

### 3.3.2 System Setup

#### Key Generation

Before generating her primes $p$ and $q$, Alice has to decide two parameters influencing the key generation:

- An odd integer $r > 2$ that determines the maximum message size for her Benaloh system and
- the bit length of $N$.

Then, she generates $p$.

Since $p$ has to fulfill the requirement that $r$ divides $p - 1$, and $\frac{p-1}{r}$ and $p$ shall be coprime, it is wise to choose a prime $p'$ that has $N/2 - r - x$ bit as a starting point, where $x$ is a chosen value. Naccache and Stern [NS98] recommend that $x$ has 24 bits. This is useful since it takes some time to generate large primes with sufficient certainty.
Benaloh cryptosystem

### Private key parameters

- \( p \) prime, \( r \mid (p - 1) \land \gcd(r, \frac{p-1}{r}) = 1 \)
- \( q \) prime, \( \gcd(r, q - 1) = 1 \)

### Public key parameters

- \( N = p \cdot q \)
- \( y \in N \mid gcd(y, N) = 1 \land y^{(p-1)(q-1) \mod r} \neq 1 \)
- \( r \) The maximum message size

<table>
<thead>
<tr>
<th>Plaintext</th>
<th>Ciphertext</th>
<th>Random element</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m \in {0, 1, \ldots, r - 1} )</td>
<td>( c(m) \in \mathbb{Z}_N )</td>
<td>( u \leftarrow \mathbb{Z}_N )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encryption</th>
<th>Decryption</th>
<th>Expansion rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \leftarrow \mathbb{Z}_N : u^r \cdot y^m \mod N )</td>
<td>( c(m)^{(p-1)(q-1) \mod r} \mod N ) and table lookup</td>
<td>( r \rightarrow N )</td>
</tr>
</tbody>
</table>

### Expansion rate

Table 3.2: Benaloh parameters

Once \( p' \) is determined, Alice can generate a prime \( p \) with the desired properties by calculating \( p = 2 \cdot r \cdot p' \cdot x + 1 \) with \( x \) being a 24 bit prime. She then tests whether \( p \) is prime and if not, she just chooses a new \( x \) (which is much smaller than \( p' \)) until she finds a suitable prime that fulfills all criteria. This search is very fast compared to the brute force search.

Since \( q \) and \( y \) are generated according to the specification, there is not much to optimize if \( r \) is prime.

The Java code for the key generation is in listing A.7 on page 65.

### Encryption

Encryption works like in the Goldwasser-Michali system with the only difference being the message size.

The code for Benaloh encryption is in listing A.5 on page 64.

### Decryption

For decryption, Alice computes \( c(m)^{(p-1)(q-1) \mod r} \mod N \) as described above. She then deciphers the message by looking up the correct value for \( m \) in the precomputed table.
3.4. NACCACHE-STERN CRYPTOSYSTEM

The Java code for decryption is in listing A.8 on page 67.

3.4 Naccache-Stern Cryptosystem

3.4.1 General Description

The Naccache-Stern cryptosystem is an optimization of the Benaloh cryptosystem (see section 3.3 on page 22). It is described by Naccache and Stern in [NS98] and is based on the same cryptographic assumptions which lie at the base of the Benaloh system.

For system setup a list \( P \) of small pairwise coprime odd integers (i.e. \( \forall p_i, p_j \in P, p_i \neq p_j : \gcd(p_i, p_j) = 1 \)) is generated. This list has usually (but not necessarily) an even number of entries \( k \) and is split into two halves, \( u \) and \( v \), where \( u = \prod_{i=1}^{k/2} p_i \) and \( v = \prod_{i=k/2+1}^{k} p_i \). Two “large” primes \( a \) and \( b \) are generated in such a way that \( p = 2au + 1 \) and \( q = 2bv + 1 \) are prime and have the desired bit length to compose \( N \).

For \( N = pq \) a base \( y \) is generated, such that \( \forall p_i \in P : y^{(p-1)(q-1)/p_i} \neq 1 \) (analogously to the Benaloh cryptosystem in section 3.3 on page 22). Naturally, encryption is likewise performed by encrypting a message \( m < \prod p_i \) as \( c(m) = y^m \mod N \). Optionally, the encrypted value can be obfuscated by setting \( c(m) = y^m \cdot u^{\prod p_i} \mod N \) with \( u \in \mathbb{Z}_N \) uniformly at random. However, this is not necessary here since the message space is much larger compared to the Benaloh cryptosystem (usually > 160 bit), so it cannot be searched.

Decryption works analogously to the Benaloh system, with the difference that here many Benaloh systems are used at once: For each \( p_i \in P \), the value of \( m_{p_i} = c(m)^{\frac{(p_i-1)(q_i-1)}{p_i}} \) is computed and the cleartext value is looked up in the table of \( p_i \). By means of the Chinese Remainder Theorem (see 2.5 on page 12) the original value of \( m \) can be restored: Clearly, \( m \) is \( \text{CRT}(m_i, p_i) \). Thus, the message \( m \) is decrypted.

A quick reference sheet for the Naccache-Stern cryptosystem is in table 3.3 on the next page

3.4.2 System Setup

Key Generation

Naccache-Stern key generation can be optimized by using threads, since \( p \) and \( q \) can be generated independently from each other. After \( p \) and \( q \) are generated, \( y \) can be generated modularly for every prime \( p_i \) by finding
### Naccache-Stern cryptosystem

<table>
<thead>
<tr>
<th>Private key parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( u, v )</td>
<td>list of coprime integers</td>
</tr>
<tr>
<td>( a, b )</td>
<td>“large” primes</td>
</tr>
<tr>
<td>( p )</td>
<td>prime, ( p = 2au + 1 )</td>
</tr>
<tr>
<td>( q )</td>
<td>prime, ( q = 2bv + 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Public key parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>( N = p \cdot q )</td>
</tr>
<tr>
<td>( y )</td>
<td>( { y \in N \mid \text{ord}(y, N) \geq 4 \land \forall p_i \in u, v : y^{(p-1)(q-1)} \mod N \neq 1 } )</td>
</tr>
<tr>
<td>( \sigma = u \cdot v )</td>
<td>The maximum message size</td>
</tr>
</tbody>
</table>

| Plaintext | \( m \in \{0, 1, \ldots, \sigma - 1 \} \) |
| Ciphertext | \( c(m) \in \mathbb{Z}_N \) |
| Random element | \( u \leftarrow \mathbb{Z}_N \) |

| Encryption | \( u \leftarrow \mathbb{Z}_N : u^\sigma \cdot y^m \mod N \) |
| Decryption | \( c(m) \frac{(p-1)(q-1)}{n} \mod N \), table lookup and CRT |
| Expansion rate | \( \sigma \rightarrow N \ (\approx 1:4) \) |

Table 3.3: Naccache-Stern parameters
3.4. NACCACHE-STERN CRYPTOSYSTEM

\[ y_i : y_i \prod_{p_i}^{p-1} (q-1) \neq 1. \] Now, the product \( \prod_{i=1}^{k} y_i \mod N = y \) has the desired property with very high probability. In my implementation, the key generation process is fully threaded and makes use of all available processors. During key generation, finding \( p \) and \( q \) is optimized by using 24 bit tuning primes, as suggested in Naccache and Stern [NS98].

The Java code for the Naccache-Stern key generation is in listing A.9 on page 67.

Encryption

Encryption is similar to the encryption process in Benaloh, but here the random value \( u \) is not compulsory since the message space has already \( \approx 160 \) bit.

The encryption is in listing A.10 on page 68.

Decryption

Since decryption can be done independently for every single prime, it can be threaded. This is done in my implementation: For each small prime a DecryptBySmallPrime thread is started which submits the decryption value to the originating thread. Since I can start multiple threads at once, the time needed for decryption decreases linearly by the amount of available processors.

The decryption process is in listing A.11 on page 69.
Chapter 4

Constructing a Shared RSA Key

4.1 Preliminary Remarks

The two party RSA key generation is implemented as a classical client-server system where the server either has a list containing authorized clients or asks its administrator for the information with every connecting client. The protocol is designed in such a way that both parties are assumed to be honestly following the protocol but curiously interpreting every message sent. They cannot learn the secrets of the other party by analyzing the messages. In this setting, mutual authentication is useful because the generated key cannot be used without the other party. Thus keys are usually generated only with reliable parties. To this end, mutually authenticated SSL connections are used for communication, where all messages are signed and encrypted automatically. Furthermore, the certificates used to establish the SSL connection are used to identify the parties. The encryption of the SSL connection is not necessary for the security of the protocol.

There are mainly two reasons why the software is not implemented in a peer to peer setting:

1. Peer to peer computing is useful when a large group of people wants to do something together. Since this implementation is tailored to the two party scenario, it is more efficient if one person is the client and the other is the server.

2. When using true peer to peer technology with automatic peer discovery (without fixed rendezvous servers), one cannot guarantee that both parties actually find each other. Peer to peer networks tend to form
clusters of computers that are strongly interconnected and have very few if any connections to other clusters. Thus, the two parties wanting to generate a shared RSA key may not be able to find each other in the peer to peer network.

Moreover, as peer to peer networks usually add some kind of abstraction layer over the actual network, it may also not be easy to reconstruct IP addresses from P2P addresses for direct communication.

By using a classical client-server solution, these two problems are overcome.

Before using the protocols explained below, the parties have to agree on the bit length $k$ of the generated RSA key, as well as on the protocol used, and have to set up the needed cryptosystems.

\subsection{4.1.1 Implementation Details}

\subsubsection{Certificates for Authentication}

When the application starts, the user is asked whether he wants to create a new certificate and private key on the fly or import both from his PKCS\#12 keystore (as defined in RSA Laboratories \cite{RSA93a}). With this certificate and private key the application has authentication credentials that are cryptographically strong and are later used to authentify against the other party. Certificates from other parties can also be stored in the keystore after successful authentication. Thus, if two parties exchange their certificates once, the identity of the other party needs no confirmation in successive runs of the application.

\subsubsection{Persistent Settings}

In my implementation, it is possible to store all settings in a configuration file that can be supplied to the application on startup. If a valid configuration file is presented, the application tries to generate a shared RSA key without user interaction. Thus, scenarios for larger organizations where multiple shared RSA keys are needed or keys must be generated automatically are possible.

\subsubsection{Size of the Generated RSA Key}

The size of the generated RSA key is the smaller of the preferences of client and server, because the homomorphic encryption systems for generating the shared key are set up before the generation. If the larger size was chosen, the homomorphic systems would also need to be stronger. In the process of
generating a shared RSA key the cryptosystems used are always as hard or harder than the generated key. Therefore, the encryption added by the SSL protocol is irrelevant.

**Agreement on a Random Element**

During the protocol the parties have to agree on a random element of a ring. These agreements are necessary in two crucial steps of the protocol: the element used for the primality test and the element used as public exponent of the shared RSA key.

Thus the security of the resulting RSA key highly depends on the inability of any party to influence the agreement. Neither Boneh and Franklin [BF97] nor Gilboa [Gil99] publish a protocol to reach a secure agreement, so I will give a short example of such a protocol:

- Each party chooses a random element of the ring. It then computes the hash value of the element with a cryptographically strong hash algorithm and publishes the hash value. After both parties have published the hash value, their elements are fixed and every party is sure that its element is truly random.

- The parties publish their elements and add them. The resulting element is the element agreed upon. Multiplication should not be used in the ring because then one party can choose a divisor of zero to get “special” results.

- If a prime number is needed, one party can search for the first prime after the element agreed upon and publish it. If both parties are convinced that the number is prime and indeed the next possible element, then the process of finding a prime number can be significantly reduced in large rings. For elements that have other special properties, the same principle can be used.

**Protocol used**

Currently, the choice of the protocol depends on the client; the server adopts as necessary. A configuration option that lists the allowed protocols for the server is not yet implemented but would be a nice addition. The current implementation contains three steps:

1. Preparation of a mutually authenticated channel. To this end, a SSL connection with verified certificates is created between both parties.
2. Generation of a shared modulus over the authenticated channel. The current implementation allows for multiple protocols to be used. Actually, only Gilboa’s Protocol as described in Gilboa [Gil99] is fully implemented. Work on the protocol designed by Straub [Str03] has begun but is not finished yet.

3. Generation of a private and a public exponent matching the shared modulus. The protocol used here is Gilboa’s adoption to the two party scenario of the protocol developed by Boneh and Franklin in [BF97].

4.2 Gilboa’s Protocol

4.2.1 Description

Gilboa’s protocol is inspired by the original work of Boneh and Franklin [BF97] but tailored to the two party scenario by using either Oblivious Transfers, Oblivious Polynomial Evaluation or Benaloh cryptosystems. In this thesis, only Benaloh cryptosystems are used to generate $N$, since in section 8 of Gilboa [Gil99] it is stated that this is the fastest way to generate $N$. After the generation of $N$ finished, Oblivious Transfers are used to construct the private RSA exponent $d$ for each party.

As Gilboa [Gil99] explains in section 3 of his paper, the protocol is split into four steps (although Gilboa lists five):

1. The server and the client agree on the bit length $k$ of the modulus $N$. The parties make sure that both $p$ and $q$ are equal to $3 \mod 4$ by setting the server’s shares of $p$ and $q$, $p_s$ and $q_s$, equal to $3 \mod 4$ and the clients shares $p_c$ and $q_c$ equal to $0 \mod 4$. All other information about $p_i$ and $q_i$ is kept secret.

2. For each party, a list of the first $l$ primes $p$ is constructed so that the bit length of $\prod_{i=0}^{l} p_i > k$. For each of these primes a Benaloh cryptosystem is set up and the sharing of $(p_s + p_c)(q_s + q_c) \mod p_i$ is computed (see section 4.2.3 on page 34). By using the Chinese Remainder Theorem (see section 2.5 on page 12), $N$ is reconstructed from the shares. If $(p_s + p_c)(q_s + q_c) \mod p_i = 0$ is detected, each party chooses new shares and the process is restarted.

3. After an $N$ has been found that is relatively prime to all generating primes and has at least the desired bit length $k$, two bi-primality tests are executed: One adopted Fermat test (see section 2.6 on page 12 and
section 4.2.4 on page 37) and a Fermat test in the twisted group that catches the few elements that may have passed the first test.

4. After the $N$ has been found and shared by both parties, they compute their private decryption exponent $d$. To this end, Oblivious Transfers are used. At the end of the computation, the parties have a shared RSA key. Since this computation can be used in both Gilboa’s protocol and Straub’s protocol, it is implemented in its own package that is executed after the shared modulus is found.

### 4.2.2 Program Design

<table>
<thead>
<tr>
<th>State</th>
<th>Command</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0) Initial State</td>
<td>Preferred Shared Key Size</td>
<td>(1)</td>
</tr>
<tr>
<td>(1) Exchange Key Size</td>
<td>Start Benaloh Setup</td>
<td>(2)</td>
</tr>
<tr>
<td>(2) Benaloh Setup</td>
<td>Benaloh Setup</td>
<td>(2)</td>
</tr>
<tr>
<td>(2) Benaloh Setup</td>
<td>Benaloh Setup Finished</td>
<td>(3)</td>
</tr>
<tr>
<td>(2) Benaloh Setup</td>
<td>Benaloh Setup Failed</td>
<td>(X)</td>
</tr>
<tr>
<td>(3) Benaloh Setup Finished</td>
<td>Start Candidate Generation</td>
<td>(4)</td>
</tr>
<tr>
<td>(4) Candidate Generation</td>
<td>Conversion Mod Prime</td>
<td>(4)</td>
</tr>
<tr>
<td>(4) Candidate Generation</td>
<td>N Mod Prime $Is 0$</td>
<td>(3)</td>
</tr>
<tr>
<td>(4) Candidate Generation</td>
<td>Generation Finished</td>
<td>(5)</td>
</tr>
<tr>
<td>(5) Candidate Found</td>
<td>N Failed</td>
<td>(3)</td>
</tr>
<tr>
<td>(5) Candidate Found</td>
<td>Start Verification</td>
<td>(6)</td>
</tr>
<tr>
<td>(6) Candidate Verification</td>
<td>Conversion Mod Prime</td>
<td>(6)</td>
</tr>
<tr>
<td>(6) Candidate Verification</td>
<td>Verification Failed</td>
<td>(3)</td>
</tr>
<tr>
<td>(6) Candidate Verification</td>
<td>Verification Finished</td>
<td>(7)</td>
</tr>
<tr>
<td>(7) Candidate Verified</td>
<td>New N</td>
<td>(8)</td>
</tr>
<tr>
<td>(8) New N</td>
<td>N Failed</td>
<td>(3)</td>
</tr>
<tr>
<td>(8) New N</td>
<td>Bi-primality Test</td>
<td>(9)</td>
</tr>
<tr>
<td>(9) Bi-primality Test</td>
<td>Witness</td>
<td>(9)</td>
</tr>
<tr>
<td>(9) Bi-primality Test</td>
<td>N Verified</td>
<td>(X)</td>
</tr>
<tr>
<td>(X) Finish</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: States used in Gilboa’s protocol

The program is designed as a finite state machine in which for every state only a few commands are possible. State transitions are executed by defined commands and for every state there is a very limited set of following states. During state transitions, there are always sanity checks on the environment;
thus corruption can be detected early and execution can be aborted if necessary.

For the Gilboa Protocol, all states with the allowed state transitions are shown in table 4.1 on the page before. The corresponding state machine is displayed in diagram 4.2 on the facing page.

Furthermore, for every set of commands that can be executed in a state, a strict order is implemented. With these precautions, both parties are very strictly bound to the protocol.

At first, the client sends the public keys of his Benaloh cryptosystems to the server. They are generated using the first $l$ primes $p_i$ so that the bit length of $\prod_{i=1}^{l} p_i > k$. The server verifies this condition by checking that at least $l$ Benaloh public keys were sent and that the product of the upper limits for the message size $r_i$ is larger than the desired bit length of the shared key. Only if both criteria are met, the reconstruction of $N$ is possible with the Chinese Remainder Theorem.

4.2.3 Generation of N

The server generates two integers $p_s$ and $q_s$ that are both equal to 3 mod 4. The client generates two integer $p_c$ and $q_c$ that are equal to 0 mod 4.

Together, they want to compute $N = (p_s + p_c)(q_s + q_c)$. We know from the Chinese Remainder Theorem that we can recompose an integer from its remainders modulo coprime integers (see section 2.5 on page 12). In a Benaloh cryptosystem (see section 3.3 on page 22), all operations are modulo the message border $r$. Since the Benaloh encryption scheme is homomorphic, we can perform addition and multiplication on encrypted messages without decrypting them. If we use multiple Benaloh systems with pairwise coprime message borders, we can do mathematical operations on very large messages in small steps and recompose the operation on the large message by the Chinese Remainder Theorem.

With a close look at $N = (p_s + p_c)(q_s + q_c) = p_sp_s + p_sp_c + p_cq_s + p_cq_c$ we see that if the client encrypts $p_c$ to $c(p_c)$ and $q_c$ to $c(q_c)$ and sends both to the server, then the server can compute

$$c(N') = c(p_c)^{p_s} \cdot c(q_c)^{p_s} \cdot c(p_sq_s)$$

without decrypting $c(p_c)$ and $c(q_c)$. Upon decryption, $c(N')$ equals $p_sp_s + p_sp_c + p_sq_s$. Now the client can add $p_sq_c$ to it and gets $N$. Since Benaloh systems limit messages below the message border $r$, the client constructs one Benaloh system for every $p_i$ and reconstructs $N$ using the Chinese Remainder Theorem. So effectively the server computes for every $p_i$
Table 4.2: Finite state machine for Gilboa’s protocol
and the client computes $N_i = (p_i q_i) \mod p_i$. With the Chinese Remainder Theorem $N$ can be restored from the $N_i$ and $p_i$.

**Privacy of the Parties**

The privacy of the client is based on the hardness of the Benaloh cryptosystems it uses. If on average half of the cryptosystems are broken then its shares are transparent to the server since they can be recovered by Chinese Remaindering (recall that the bit length of $p$ and $q$ is half the bit length of $N$). But since all Benaloh cryptosystems used in the generation of $N$ have the same bit length as $N$, each is probably as hard to break as factoring $N$ is. So this is not a real threat.

The privacy of the server is based on the fact that every message sent in a Benaloh system is randomized by $u$. Since it is essential for the client that the Benaloh systems used have the same magnitude as $N$, $u$ is also a pseudo random element in $\mathbb{Z}_N$. The decryption cannot recover $u$ from the encrypted message and thus the client gains only the information that $c(N_i' \mod p_i)$ is a random element in the partition $(N - p_i q_i) \mod p_i$ of $\mathbb{Z}_N$. It cannot gain any new information from it.

**Optimization**

Gilboa proposes in section 5.2 of [Gil99] that both parties compute $N$ by picking random elements in $p_i$ and compute $N_i$ separately for every element as described above. If it turns out that $N \mod p_i = 0$, then both parties pick new shares for $p_i$. At the end, $N$ can be reconstructed from the $N_i$ by means of the Chinese Remainder Theorem. This optimization is currently implemented.

Another optimization that has not yet been implemented would be to use Naccache-Stern cryptosystems to compute $N$. If the primes used during the transfers are split into groups such that every group forms one $\sigma$ of a Naccache-Stern cryptosystem, the number of transfers would be significantly reduced.

Since $\sigma$ consists of many primes, it is possible to check during decryption for every single prime if $N \mod p_i = 0$. For the few cases where this will happen, Benaloh cryptosystems can be used to recompute until $N \mod p_i \neq 0$. This would decrease the communication needed approximately by a factor.
of 10 since the expansion rate of Naccache-Stern encoded messages is much better than the expansion rate of messages encrypted with Benaloh systems.

The Java source code for the generation of N is shown in section A.12 on page 72 for the client and in section A.13 on page 75 for the server.

4.2.4 Primality Test

The primality test is taken from section 4 of Boneh and Franklin [BF97] as Gilboa recommends. For m parties, the test is m − 1 private and thus is also suitable for the two party scenario.

Boneh and Franklin look at all possible settings for an RSA N that is composed of two primes p and q which are equal to 3 modulo 4.

During the test, the parties detect an N that is not a product of two distinct primes with probability at least 1/2 for every complete run. Experience shows that the test is even stronger: When a prime passes a single test, it usually passes repeated tests as well.

The test has the following steps:

1. The parties agree on a g with \( J(g, N) = 1 \). We already know that there are two possibilities for the quadratic residues of such a g in \( N = p \cdot q \) (see section 3.2 on page 20), namely 1 and −1. This g is used for the test.

2. The client computes

\[
 v_c = g^{\frac{p - q}{4}}
\]

and the server computes

\[
 v_s = g^{\frac{N - p - q + 1}{4}}.
\]

They combine to gain \( v = v_c \cdot v_s = g^{\frac{N - p - q + 1}{4}} \). They check whether \( v = \pm 1 \). If not, N is not the product of two distinct primes.

3. Additionally, the parties generates a random element from the twisted group of N, \( h \in T_N \) (see also section 2.9 on page 15).

For both primes p and q, \( x^2 + 1 \) is irreducible in \( \mathbb{Z}_N[x] \) since they are equal to 3 modulo 4 (for a proof, see Rogers [Rog05], page 1). Thus, \( \mathbb{Z}_N[x]/(x^2 + 1) \) is a quadratic extension of \( \mathbb{Z}_N \) with \( N = p \cdot q \). The multiplicative group \( T_p = \frac{\mathbb{Z}_p[x]}{x^2 + 1} / \mathbb{Z}_p \) has order \( p + 1 \); similarly, \( T_q \) has order \( q + 1 \). Thus, \( T_N \) has order \((p + 1)(q + 1)\) and for all \( h \in T_N : h^{(p+1)(q+1)} = 1 = (1, 0)\).
CHAPTER 4. CONSTRUCTING A SHARED RSA KEY

For the chosen element, the client computes \( u_c = h^{p + q} \) and the server computes \( u_s = h^{N + p + q} \). They compute \( u = u_c \cdot u_s = h^{N + p + q} \) and verify that \( u = 1 = (1, 0) \). If not, \( N \) is not the product of two distinct primes, as will be shown below.

4. This test is repeated until the security parameters for primes of both parties are met (usually 25 to 30 times).

Boneh and Franklin prove that the standard Fermat test already catches almost all \( N \) that are not the product of two primes, if \( g \) is chosen at random in \( N \). By using a \( g \in G \) for that \( J(g, N) = 1 \), they make sure that \( J(g, p) = J(g, q) \). Recalling the definition of the Jacobi Symbol (see section 2.8 on page 14), and the fact that both \( q^{-1} \) and \( p^{-1} \) are uneven (because \( p = q = 3 \mod 4 \)), we can write

\[
g^{N - p - q + 1} = g^{(p-1)(q-1)} = \left( g^{p-1} \right)^{\frac{q-1}{2}} = J(g, p) \quad J(g, p)
g^{N - p - q + 1} = g^{(q-1)(p-1)} = \left( g^{q-1} \right)^{\frac{p-1}{2}} = J(g, q) \quad J(g, q).
\]

Now consider an \( N \) that is not the product of two distinct primes with probability of at least \( \frac{1}{2} \). We have to show that \( |H| \leq \frac{1}{2}|G| \). Since \( G \) is closed under multiplication it suffices to find a single element of \( G \) that is not in \( H \). We have to consider four cases:

- \( N \) consists of 3 or more factors, thus \( s \geq 3 \). Let \( r_2 \) and \( r_3 \) be distinct prime factors of \( N \) with the same exponent modulo 2. With three different primes making up \( N \), such a pair must exist. Let \( r_1 \) be a distinct prime factor different from \( r_2 \) and \( r_3 \).

Define \( g \in G \) (with \( a \) being a quadratic non-residue modulo \( r_3 \)): 
4.2. GILBOA’S PROTOCOL

\[ g \equiv 1 \mod r_1 \]
\[ g \equiv -1 \mod r_2 \]
\[ g \equiv \begin{cases} 1 \mod r_3 & \text{if } J(r_2, N) = 1 \\ a \mod r_3 & \text{if } J(r_2, N) = -1 \end{cases} \]
\[ g \equiv 1 \mod r_i. \]

By definition, \( J(g, N) = J(g, r_1)^{d_1} \cdot J(g, r_2)^{d_2} \cdot J(g, r_3)^{d_3} \ldots \). Since \( a \) is a quadratic non-residue modulo \( r_3 \), \( J(g, r_3) = -1 \) if \( J(g, r_2) = -1 \) and since the exponents modulo 2 are equal, \( J(g, N) = 1 \). If \( J(g, r_2) = 1 \), then \( J(g, r_3) = 1 \) and \( J(g, N) = 1 \), too. A suitable \( g \) can be reconstructed from the moduli by the Chinese Remainder Theorem. Thus, \( g \in G \).

Since \( e \) is odd, we have

\[
\begin{align*}
g^e \mod r_1 &= g \mod r_1 = 1 \\
g^e \mod r_2 &= g \mod r_2 = -1.
\end{align*}
\]

To meet both conditions at once, \( g \) cannot be \( \pm 1 \). Thus we have proven the existence of a \( g \in G \) that is not in \( H \) for this case.

- Consider the condition that \( \gcd(p, q) > 1 \). Then a prime \( r \geq 3 \) exists which divides both \( p \) and \( q \) (it can’t be 2 because \( N \) is odd). We have \( N = r^{d_1} \cdot p' \cdot r^{d_2} \cdot q' \) and thus \( \varphi(N) = r^k(r-1)(p'-1)(q'-1) \) with \( k \geq 1 \). Thus we have an element \( g \) of order \( r \) in \( \mathbb{Z}_N \) and therefore \( g^r = 1 \). Since \( r \) is odd, we have \( J(g, N) = J(g, N)^r = J(g^r, N) = J(1, N) = 1 \).

Therefore \( g \) is an element of \( G \). We know that \( r \) divides both \( p \) and \( q \) and thus does not divide \( N - p - q + 1 = 4e \). Thus, \( g^4 \neq \pm 1 \) and also \( g^e \neq \pm 1 \), and therefore \( g \notin H \).

- An \( N \) that is not caught by the upper two cases must be of the form \( p \cdot q = r_1^{d_1}r_2^{d_2} \) with \( \gcd(r_1, r_2) = 1 \) and at least one of \( d_1 \) or \( d_2 \) greater than 1.

Let us set \( d_1 > 1 \). We now have as before \( N = r_1^{d_1-1}(r_1 - 1)r_2^{d_2-1}(r_2 - 1) \). Thus, again we have an element \( g \) with order \( r_1 \) and \( J(g, N) = 1 \) as above. If \( q \mod r_1^{d_1-1} \neq 1 \), then again \( N - p - q + 1 \) is not divisible by \( r_1^{d_1-1} \). As above, \( g \notin H \) since \( g^e \neq \pm 1 \).
• The last case is for \( N = r_1^{d_1} \cdot r_2^{d_2} \) where \( d_1 > 1 \) and \( q \mod r_1^{d_1-1} = 1 \). An \( N \) of this form has a probability to pass the first three tests. Since \( p \equiv q \equiv 3 \mod 4 \) we know that \( r_1 \equiv r_2 \equiv 3 \mod 4 \). For this case, the test in the twisted group will fail with probability at least \( \frac{1}{2} \):

Define the group \( H' = \{ h \in \mathbb{T}_N \mid h^{(p+1)(q+1)} = 1 \} \) which is a subgroup of \( \mathbb{T}_N \). Again, we have to show that \( |H'| \leq \frac{1}{2} |\mathbb{T}_N| \). Again, proving the existence of a single element of \( \mathbb{T}_N \) that is not in \( H' \) is sufficient because \( \mathbb{T}_N \) is closed under multiplication.

Since \( p = r_1^{d_1} \) we know that the group \( \mathbb{T}_p \) has order \( r_1^{d_1-1}(r_1+1) \). Thus an element \( g \) of order \( r_1 \) exists in \( \mathbb{T}_p \) and similarly an element \( g' \) exists in \( \mathbb{T}_N \) with order \( r_1 \). Since we know that \( q \mod r_1 \equiv 1 \), it is clear that \( r_1 \) does not divide \( q+1 \). Thus \( r_1 \) cannot divide \( N + p + q + 1 \) and \( g^{(p+1)(q+1)} \neq 1 \). Again we have found an element that is in \( \mathbb{T}_N \) but not in \( H' \) and can therefore be sure that we detect a composite \( N \) at least with probability \( \frac{1}{2} \).

The Java source code for the client’s prime test is in appendix A.14 on page 77. The corresponding code for the server is in appendix A.15 on page 81.

### 4.3 Oblivious Transfers

Oblivious Transfers enable party A to query party B for exactly one secret out of many secrets that party B holds. Party B sends that secret, but does not know which of the secrets was requested. Additionally, party A cannot learn the other secrets. In the protocol for computing the private RSA keys from the shared modulus \( N \) this protocol is always used for two secrets. I will now describe “one out of two” or \( \binom{2}{1} \) Oblivious Transfers as described originally in Even et al. [EGL85] and in short in Wikipedia [Wik06e]:

• Party B generates an RSA key pair and sends the public exponent \( e \) and the modulus \( N \) to party A. For its secrets \( m_0 \) and \( m_1 \) in \( \mathbb{Z}_N \), it prepares two random messages in \( \mathbb{Z}_N \), \( x_0 \) and \( x_1 \). It then sends \( x_0 \) and \( x_1 \) to party A.

• Party A also generates a random message \( k \) in \( \mathbb{Z}_N \). It encrypts \( k \) with B’s public key to \( c(k) \). Depending on the secret \( A \) wants to know, it sends \( k_x \), which is either \( c(k) + x_0 \) or \( c(k) + x_1 \) to \( B \).
4.3. OBLIVIOUS TRANSFERS

- Party B decrypts two messages: \( k_0 = k_x - x_0 \) and \( k_1 = k_x - x_1 \), from which only one reveals the correct \( k \) (but B doesn’t know which). The decryption of the other message leads to a random element in \( \mathbb{Z}_N \) that party A cannot guess.

B now sends \( k_0 + m_0 \) and \( k_1 + m_1 \) to party A

- Party A chooses the message corresponding to its choice of \( x_0 \) or \( x_1 \) and subtracts \( k \) from it. It gains the correct message from B but cannot recover B’s second secret.

It should be noted that the secrets \( m_0 \) and \( m_1 \) can be of arbitrary size because they are transferred additively. If they are larger than \( N \), the high-order bits of the secrets are exposed.

4.3.1 Converting Shared Secrets

Using Oblivious Transfers in a ring \( R \), we can transform a multiplicatively shared secret \( a \cdot b \) into an additively shared secret \( x+y \) so that \( a \cdot b = s = x + y \). The same process can also be used to transform an additively shared secret into a multiplicatively shared secret. The protocol presented here is based on the protocol described in Gilboa [Gil99, section 4.1 and section 6].

Note that every element in the ring \( R \) can be encoded using \( \rho = \log_2 |R| \) bits. Thus, if mathematical operations can be carried out bitwise by using Oblivious Transfers, there are exactly \( \rho \) transfers needed for the computation. In the process described below, party A always has the secrets \( a \) and \( x \) and party B always has the secrets \( b \) and \( y \).

Party B chooses uniformly at random and independently \( \rho \) elements of \( R \). Let \( s_0, s_1, \ldots, s_{\rho-1} \) denote the randomly chosen elements.

Party B additionally has a private element \( b \),

- either by choosing a random element \( r \) in \( R \) if an additively shared secret shall be converted into a multiplicatively shared secret,

- or by setting it to the correct value \( b \) if a multiplicatively shared secret shall be converted into an additively shared secret.

Party B prepares \( \rho \) pairs of elements \( m_i \) with

\[
m_i = (m_i^0, m_i^1) = (s_i, 2^i b \cdot s_i).
\]

To convert the shared secret from one type to the other, party A uses its secret denoted by \( a \). Let \( a_{\rho-1}, \ldots, a_1, a_0 \) denote the bits of party A’s secret.
Now, the parties execute $\rho \left( \frac{2}{1} \right)$ Oblivious Transfers. In the $i$\textsuperscript{th} invocation, party A chooses from the tuple $(m_i^0, m_i^1)$ the $m_i^x$ that matches the bit in $a_i$. Thus, if $a_i = 0$, it chooses $m_i^0$ and if $a_i = 1$ it chooses $m_i^1$.

Party A computes $x = \sum_{i=0}^{\rho-1} m_i^{a_i}$ and party B computes $y = -\sum_{i=0}^{\rho-1} m_i^0 = -\sum_{i=0}^{\rho-1} s_i$.

The parties share $x + y = a \cdot b$. For a multiplicatively shared secret that shall be converted into an additively shared secret we are done.

But an additively shared secret that shall be converted into a multiplicatively shared secret needs more attention: Currently party A holds $x$ and $a$ and party B holds $y$, $r$ and $b$.

Party B sends $y + r b$ (since $r$ is a random element in $R$, the secrets $y$ and $b$ are blinded) to party A. Party A computes $x + y + r b = r \cdot a + r \cdot b = r \cdot (a + b)$ and party B computes $r^{-1}$ in $R$; the conversion is finished.

Finally we have to prove that $x + y = a \cdot b$ over the ring $R$. The binary representation of $a$ is given by $a_{\rho-1}, \ldots, a_1, a_0$. Thus $a = \sum_{i=0}^{\rho-1} a_i \cdot 2^i$ and as proven in [Gil99, section 4.1] we find that

\[
x + y = \sum_{i=0}^{\rho-1} a_i \cdot 2^i - \sum_{i=0}^{\rho-1} s_i \\
= \sum_{i=0}^{\rho-1} (a_i \cdot 2^i \cdot b + s_i) - \sum_{i=0}^{\rho-1} s_i \\
= \sum_{i=0}^{\rho-1} (a_i \cdot 2^i \cdot b) + \sum_{i=0}^{\rho-1} s_i - \sum_{i=0}^{\rho-1} s_i \\
= b \sum_{i=0}^{\rho-1} (a_i \cdot 2^i) \\
= a \cdot b.
\]

The code for Party B can be found in listing A.16 on page 85, the code for Party A is analogously in listing A.17 on page 88.
4.4 Computing the Public and the Private Exponent

The client and the server share \( N = (p_c + p_s)(q_c + q_s) \) in an additive manner and thus also share \( \varphi(N) = (p - 1)(q - 1) = N - p - q + 1 = N - p_c - p_s - q_c - q_s + 1 \) additively. Since in our setting the server holds the shares that are equivalent to 3 mod 4, it is somewhat natural to split \( \varphi(N) = \varphi_s + \varphi_c = (N - p_s - q_s + 1) + (-p_c - q_c) \). To compute an RSA key, the parties have to agree on a public exponent \( e \) and compute the matching private exponent \( d \) so that \( de \mod \varphi(N) = 1 \).

As also described in section 3.1 on page 19, the recent development in RSA side channel attacks has shown that RSA public exponents \( e \) such as 3 or \( 2^{16} + 1 \) are not very secure and should be avoided. For details please see Fouque et al. [FKJM06]. Similar problems arise if the private exponent is too small (see Wiener [Wie89] and Hinek [Hin06] for details). To avoid this, both parties agree on a random public exponent in \( \mathbb{Z}_N \) that is prime and thus yields a very high probability that both the private and the public exponent will not be susceptible to attacks.

Usually RSA private keys are constructed by inverting \( e \) modulo \( \varphi(N) \) by use of the Chinese Remainder Theorem which is described in section 2.5 on page 12. Unfortunately, \( \varphi(N) \) must be known to all parties to do this and thus the secret shares of the client and the server would be exposed.

Boneh and Franklin [BF97] describe how to generate a shared RSA private key without the need to make reductions modulo \( \varphi(N) \):

- First compute \( \zeta = \varphi(N)^{-1} \mod e \).
- Set \( T = -\zeta \cdot \varphi(N) + 1 \). Modulo \( e \) we have

\[
T \mod e = -\zeta (\varphi(N) \mod e) + 1
= -\left( \frac{\varphi(N)}{\varphi(N) \mod e} \right) + 1
= -1 + 1 = 0
\]

and modulo \( \varphi(N) \)

\[
T \mod \varphi(N) = -\zeta (\varphi(N) \mod \varphi(N)) + 1 = 1.
\]
Set \( d = \frac{T}{e} \). Thus, \( de = 1 \mod \varphi(N) \).

Four steps are needed to apply this computation to the two-party scenario. The process is described in section 6 of Gilboa [Gil99]:

1. Transform the additive sharing of \( \varphi(N) = \varphi_c + \varphi_s \) into a multiplicative sharing \( \varphi_{cm} \cdot \varphi_{sm} \) modulo \( e \). We use the protocol described in section 4.3.1 on page 41 with the ring \( \mathbb{Z}_e \).

The server incorporates party A in the protocol and his secret element \( a \) is \( \varphi_s \) modulo \( e \). The client takes party B and his \( b \) is a random element \( r \) of \( \mathbb{Z}_e \).

As we want to convert an additively shared secret into a multiplicatively shared secret, at the end of the conversion the client sends \( y + r \cdot \varphi_c \) modulo \( e \) to the server as described in the protocol. The server holds \( \varphi_{sm} = r \cdot (\varphi_s + \varphi_c) \) and the client holds \( \varphi_{cm} = r^{-1} \). Hence, both parties hold multiplicative shares of \( \varphi(N) = \varphi_{sm} \cdot \varphi_{cm} \) (modulo \( e \)).

2. The parties invert their shares modulo \( e \). The server holds

\[
\zeta_s = \frac{1}{\varphi_{sm}} = \frac{1}{r((\varphi_s + \varphi_c) \mod e)}
\]

and the client holds

\[
\zeta_c = \frac{1}{\varphi_{cm}} = \frac{1}{r^{-1}} = r.
\]

Thus, they hold multiplicative shares of \( \varphi(N)^{-1} \) modulo \( e \):

\[
\zeta_s \cdot \zeta_c = \frac{1}{r((\varphi_s + \varphi_c) \mod e)} \cdot r = \frac{1 \mod e}{(\varphi_s + \varphi_c) \mod e} = \varphi(N)^{-1} \mod e.
\]

Additionally the server negates his share so that the parties now share \( -\zeta = -\zeta_s \cdot \zeta_c = -\varphi(N)^{-1} \mod e \).
3. The shares of $-\varphi(N)^{-1}$ modulo $e$ are re-converted into additive sharings $\psi_c$ and $\psi_s$ using the algorithm given in section 4.3.1 on page 41. The ring used is again $Z_e$.

The server’s initial secret $a$ is $-\zeta_s$ and the client’s initial secret $b$ is $\zeta_c$. At the end of the protocol the parties share $x+y = -\zeta_s \cdot \zeta_c$. This result can be used, but to follow Gilboa’s protocol exactly, the sharing is not finished when the server holds $x$ and the client holds $y$.

Instead, the client additionally generates a $\psi_c$ uniformly at random from $Z_e$. It then sends $y - \psi_c$ to the server so that the server has $\psi_s = x + y - \psi_c = -\zeta_s \zeta_c - \psi_c$. This additional step can be skipped but does not have to, since it does not significantly slow down the protocol either.

4. The parties now compute $T - 1 = -\zeta \cdot \varphi(N)$ from Boneh and Franklin’s protocol. The additive sharing of $\zeta$ is $\psi_s + \psi_c$ as computed in the previous step and the additive sharing of $\varphi(N) = \varphi_s + \varphi_c = (N - p_s - q_s + 1) + (-p_c - q_c)$ is also known to both parties. Thus

$$T - 1 = -\zeta \cdot \varphi(N)$$
$$= (\varphi_s + \varphi_c)(\psi_s + \psi_c)$$
$$= \varphi_s \psi_s + \varphi_s \psi_c + \varphi_c \psi_s + \varphi_c \psi_c.$$ 

The elements needed to compute $T - 1$ are shared by the server and the client. The current sharing of $T - 1$ is both additive and multiplicative. To convert it into a sharing that is only additive, both sides need to convert the multiplicative sharings of $\varphi_s \psi_c$ to $x_1 + y_1$ and $\varphi_c \psi_s$ to $x_2 + y_2$.

The server always holds $x_i$ and the client always holds $y_i$. The sharings are converted using Oblivious Transfers in a ring $R$ with $R > 4e \cdot N$. In the current implementation, the ring used is $Z_k$ with $k$ being the first prime number that is larger than $4eN$. The ring has to be larger than $4eN$ because each element of $T - 1$ is composed of $\frac{\varphi_i \psi_i}{\in Z_N} < N \cdot e$.

Since we have 4 shared elements, we can safely say that any ring $Z_k$ with $k > 4eN$ is sufficient for the computation of $T - 1$.

After the conversion, both parties hold elements of $Z_k$. However, the addition of the elements may lead to an element that is larger than $k$: The parties cannot reveal their elements, thus reductions modulo $k$ can only be made on a “per party” basis and not on the sum of the elements.
Following this argumentation, the computed value is either \( T - 1 \) or \( T - 1 + k \). The latter case is not acceptable, because \( T + k \mod \varphi(N) \neq 1 \).

To avoid this problem and complete the computation of \( T - 1 \), the client chooses uniformly at random an element \( y \) of \( \mathbb{Z}_{k/2} \) and sends \( y + y_1 + y_2 + \varphi_c \psi_c \) to the server. The server now holds \( \varphi_s \psi_s + x_1 + x_2 + (y + y_1 + y_2 + \varphi_c \psi_c) = (T - 1) + y \). This value modulo \( k \) is smaller than \( k \). The client’s secret is \(-y\) which is in \( \mathbb{Z}_{k/2} \). Added, the parties have an element that is surely in \( \mathbb{Z}_k \) and thus the correct \( T - 1 \). By choosing \( y \) from \( \mathbb{Z}_{k/2} \) the client’s security is not significantly reduced since it looses only a single bit of security. Since this bit could also be guessed in an attempt to recover its secret, the loss does not signify.

Finally, the server sets

\[
d_s = \frac{\varphi_s \psi_s + x_1 + x_2 + (y + y_1 + y_2 + \varphi_c \psi_c)}{e} = \frac{(T - 1) + y}{e}
\]

and the client sets

\[
d_c = -\frac{y + 1}{e}.
\]

Finally we get

\[
d_s + d_c = \frac{(T - 1) + y - y + 1}{e} = \frac{T}{e}
\Rightarrow e \cdot (d_s + d_c) \mod \varphi(N) = 1.
\]

Both parties hold additive shares of the secret exponent, thus they can decrypt a encrypted message \( y = c(x) = x^e \) by multiplication of the decrypted values:
4.4. COMPUTING THE PUBLIC AND THE PRIVATE EXPONENT

\[ y^d = (x^e)^{d_s} \cdot (x^e)^{d_c} = (x^e)^{d_s} + d_c = (x^e)^d = x^{(ed-1)+1} = x^{\phi(N)+1} = x^{\phi(N)} \cdot x = x. \]
CHAPTER 4. CONSTRUCTING A SHARED RSA KEY
Chapter 5

Results

In order to judge the fitness of the software for practical use, a performance analysis is required. Mostly, the analysis focuses on aspects such as computational effort versus key size – the key size is directly related to the security of the key – as well as possible optimizations to the implementation. Finally, the question is addressed, whether it is likely that existing software can be used with the keys generated here.

5.1 Performance

When the implementation was finished, I noticed that one client only used \( \approx 30\% \) of the the CPU when generating a 1024 bit key. This increased the running time by a factor three, which was unacceptable. After a lengthy analysis I found out that the TCP protocol used for communication seems to wait until either enough data for sending a packet is available or a timeout for constructing packets has passed. Since the information sent is usually less than the data needed for a packet, the communication was impaired because the TCP stack slowed the data exchange.

I found a workaround by setting the TCP options TCP_NODELAY to \texttt{true} and setting TCP_TRAFFIC_CLASS to IPTOS_LOWDELAY and IPTOS_THROUGHPUT. This led to a much better processor usage of \( \approx 80\% \).

5.1.1 Statistics

The search for an \( N \) that is a product of two primes is nondeterministic, since both parties choose their parts randomly and \( N \) is verified only against the first few primes during generation. On average, the computation of a 130 bit RSA key takes 20 seconds on a AMD Athlon XP 2500+. In that case,
the average communication between both parties sums to 500 kB where the client generates roughly 350 kB and the server generates about 150 kB of traffic. A large part of the bandwidth is used by the protocol to generate the private exponent $d$; here the client uses 216 kB of his traffic and the server needs 54 kB.

In section 8 of [Gil99], Gilboa maintains that a 1024 bit key has a communication complexity of 42 MB on average, when two primes are used per Benaloh key as opposed to one in the current implementation.

I received the following data for twenty runs of the protocol to generate 1024 bit keys with unpaired Benaloh keys, without SSL overhead, via loopback network on a 8x Xeon 3.2 GHz, with 10 clients and one server executing simultaneously:

<table>
<thead>
<tr>
<th>Run</th>
<th>Time</th>
<th>Traffic generated</th>
<th>Tries</th>
</tr>
</thead>
<tbody>
<tr>
<td>#20</td>
<td>0:41:59</td>
<td>7045535</td>
<td>1911246</td>
</tr>
<tr>
<td>#17</td>
<td>0:44:17</td>
<td>10423907</td>
<td>3604966</td>
</tr>
<tr>
<td>#01</td>
<td>0:47:54</td>
<td>12140554</td>
<td>5210535</td>
</tr>
<tr>
<td>#03</td>
<td>0:54:54</td>
<td>17580750</td>
<td>7545371</td>
</tr>
<tr>
<td>#12</td>
<td>0:58:11</td>
<td>17357593</td>
<td>8035923</td>
</tr>
<tr>
<td>#02</td>
<td>1:11:07</td>
<td>23148323</td>
<td>9934857</td>
</tr>
<tr>
<td>#11</td>
<td>1:34:20</td>
<td>40390020</td>
<td>18651728</td>
</tr>
<tr>
<td>#13</td>
<td>2:17:25</td>
<td>68191228</td>
<td>32605370</td>
</tr>
<tr>
<td>#07</td>
<td>2:17:27</td>
<td>72772452</td>
<td>34911252</td>
</tr>
<tr>
<td>#18</td>
<td>2:26:19</td>
<td>76017226</td>
<td>36545303</td>
</tr>
<tr>
<td>#04</td>
<td>2:24:43</td>
<td>79007433</td>
<td>38038436</td>
</tr>
<tr>
<td>#08</td>
<td>2:44:12</td>
<td>86896689</td>
<td>42000429</td>
</tr>
<tr>
<td>#15</td>
<td>2:41:25</td>
<td>88956596</td>
<td>43032935</td>
</tr>
<tr>
<td>#05</td>
<td>3:26:31</td>
<td>96007829</td>
<td>46575500</td>
</tr>
<tr>
<td>#16</td>
<td>2:49:27</td>
<td>99812655</td>
<td>48482706</td>
</tr>
<tr>
<td>#09</td>
<td>3:55:07</td>
<td>109519716</td>
<td>53355402</td>
</tr>
<tr>
<td>#10</td>
<td>5:35:27</td>
<td>203188471</td>
<td>100390472</td>
</tr>
<tr>
<td>#14</td>
<td>4:52:26</td>
<td>252834571</td>
<td>125315065</td>
</tr>
<tr>
<td>#06</td>
<td>9:54:02</td>
<td>417250385</td>
<td>207853153</td>
</tr>
<tr>
<td>#19</td>
<td>7:03:15</td>
<td>428536888</td>
<td>213518269</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>2:57:36</strong></td>
<td><strong>110206443</strong></td>
<td><strong>53875945</strong></td>
</tr>
</tbody>
</table>

On average, it takes 2846 rounds of the protocol to generate a 1024 bit key and the whole process generates 164 MB of network traffic. This is four times
as much as Gilboa claimed in section 8 of [Gil99]. From Gilboa’s results with paired keys I would expect only twice as much traffic in my implementation. Since Gilboa’s implementation is not transparent, it is not quite clear where the differences between his and my implementation lie. However, the most likely candidate is the size of the transferred data set. Here, I send uncompressed Java objects which, while they might result in some overhead, will certainly not be responsible for the large difference in traffic.

Interestingly, if I consider only the ten fastest runs of the algorithm, the average traffic suddenly diminishes to ≈ 50MB on average, which is much closer to Gilboa’s result.

5.2 Usability with Existing RSA Software

The Java API supports only RSA keys with 512 or more bits. Thus, smaller keys cannot be saved to a PKCS#8 structure (as defined by the RSA Laboratories in [RSA93b]) for later use. Instead, all relevant data to reconstruct the key is logged and the user has to reconstruct the key on his own.

Keys with 512 bits or more are written to a standard PKCS#8 structure. The public key can be successfully reconstructed from the structure but is also saved in a X.509 structure for later use.

Standard RSA software such as OpenSSL (http://www.openssl.org/) or the Java BouncyCastle library (http://www.bouncycastle.org/) use a very fast algorithm for RSA decryption. The algorithm basically decrypts modulo $p$ and modulo $q$ and uses the Chinese Remainder Theorem to recompose the correct value.

The private RSA keys generated with this software cannot include $p$ or $q$ because neither party is supposed to know the values. Thus, standard libraries and standard software will not work with the generated private keys. The generated public key does not have this problem and can be used normally.
Chapter 6

Summary

The focus of the summary is twofold: On one hand, venues for future work are explored; on the other hand, the thesis is placed in relation to its scientific context.

6.1 Outlook

With the current software, distributed RSA keys for two parties can be successfully generated so that neither party ever possessed the complete key and is also not able to reconstruct the complete key from the communication protocol. But software is never perfect, so there still exist a few improvements that should be included in future releases:

- Gilboa’s protocol should be using Naccache-Stern cryptosystems instead of Benaloh cryptosystems as described in section 4.2.3 on page 36. I think that this will speed up the computation of $N$ by a factor of more than 2.

- Other protocols for generating shared RSA keys should be integrated as well. I would like to mention Straub’s protocol as described in [Str03] which will probably give a dramatic decrease in key generation time. The current algorithm has to find two primes simultaneously. This implies a running time in the order of $k^2$ where $k$ is the bit size of the generated keys. By generating keys with only one shared prime, the running time will be in the order of $k$. A disadvantage is that the resulting RSA key will be composed of three primes, which will further reduce its usability with off the shelf software.

- It would be nice to have protocol whitelists for the server so that protocols deemed insecure can be excluded.
CHAPTER 6. SUMMARY

• An integration of the multi party protocol developed by Boneh and Franklin in [BF97] should be achieved. This would open the software to a broader range of users since then it would also allow more than two parties to generate a shared RSA key. The primality test could be re-used; other parts of the software would have to be adapted.

• I would like to see an integration of the software into a software for managing Certification Authorities, e.g. OpenCA (http://www.openca.org). This would result in an even broader range of potential users.

Although there are improvements that can be made, the software is in a state that is usable and reliable and therefore releasable as an open source project.

6.2 Conclusion

In the course of this thesis, a software for generating shared RSA keys in the two party scenario has been developed. Apart from the work of Boneh and Franklin [BF97], Gilboa [Gil99] and Straub [Str03] there exists to my knowledge no further literature on the subject of generating shared RSA keys.

Moreover, the algorithms used in the process are not included in standard cryptographic libraries so that a verification and extension of the work of the three authors initially requires a lot of preparation.

The cryptographic algorithms implemented in the course of this thesis were submitted to the BouncyCastle library (http://www.bouncycastle.org). This may facilitate further development on the subject. Additionally, the software accompanying the thesis has a design that allows for easy integration of other protocols tailored to generate two-party RSA keys and will soon be made publicly available as an open source project.

The performance of the actual software is about half as good as proposed by Gilboa but can be boosted by using the Naccache-Stern cipher instead of Benaloh’s. The Naccache-Stern cipher is fully implemented, only the integration into Gilboa’s protocol is missing.

This thesis is one of the few, if any, that provide a complete coverage of the process of generating shared RSA keys for two parties. By explaining the needed mathematical background, then introducing homomorphic encryption and finally putting together and explaining the elements used in the protocol designed by Gilboa and Boneh and Franklin, it enables the reader to quickly grasp the theoretical background. Combined with the complete source code
for the application as a reference to the reader, it is probably one of the most complete guides on the subject.
Bibliography


57
[Fl05] Jana Fröschler. Probabilistische Algorithmen für PRIMES. 


Appendix A

Appendix: Source Code

A.1 Mathematical Tools

Listing A.1: Chinese Remainder Theorem in PrimeUtils.java

```
/**
 * Computes the integer x that is expressed through the given primes and the
 * congruences with the chinese remainder theorem (CRT).
 * @param congruences the congruences c_i
 * @param primes the primes p_i
 * @return an integer x for that x % p_i == c_i
 */
public static BigInteger chineseRemainder(final Vector congruences, final Vector primes) {
    BigInteger all = ONE;
    for (int i = 0; i < primes.size(); i++) {
        all = all.multiply((BigInteger) primes.elementAt(i));
    }
    BigInteger a; BigInteger b; BigInteger b;
    BigInteger tmp;
    for (int i = 0; i < primes.size(); i++) {
        a = (BigInteger) primes.elementAt(i);
        b = all.divide(a);
        b_ = b.modInverse(a);
        tmp = tmp.multiply((BigInteger) congruences.elementAt(i));
        retval = retval.add(tmp).mod(all);
    }
    return retval;
}
```

Listing A.2: Legendre Symbol in PrimeUtils.java

```
/**
 * Computes the Legendre symbol of a and p.
 * 
```
public static int legendreSymbol(final BigInteger a, final BigInteger p) {
    BigInteger tmp = a;
    tmp = tmp.mod(p);
    final BigInteger sym = tmp.modPow((p.subtract(ONE)).divide(TWO), p);
    if (sym.equals(ZERO)) {
        return 0;
    }
    if (sym.equals(ONE)) {
        return 1;
    }
    // mod in Java always returns positive integers
    if (sym.equals(p.subtract(ONE)) || (sym.mod(p)).equals(p.subtract(ONE))) {
        return -1;
    }
    // shouldn’t get here
    throw new IllegalArgumentException("probably the second argument is not prime");
}

Listing A.3: Jacobi Symbol in PrimeUtils.java

public static int jacobiSymbol(final BigInteger a, final BigInteger p) {
    if (a.equals(ZERO)) {
        return 0;
    }
    if (a.equals(ONE)) {
        return 1;
    }
    if (a.compareTo(p) > 0) {
        return jacobiSymbol(a.mod(p), p);
    }
    if (a.mod(FOUR).equals(ZERO)) {
        return jacobiSymbol(a.divide(FOUR), p);
    }
    if (a.mod(TWO).equals(ZERO)
        && (p.mod(EIGHT).equals(ONE) || p.mod(EIGHT).equals(SEVEN))) {
        return jacobiSymbol(a.divide(TWO), p);
    }
    if (a.mod(TWO).equals(ZERO)
        && (p.mod(EIGHT).equals(THREE) || p.mod(EIGHT).equals(FIVE))) {
        return -1 * jacobiSymbol(a.divide(TWO), p);
    }
    if (a.mod(FOUR).equals(ONE) || p.mod(FOUR).equals(ONE)) {
        return 1;
    }
    return -1;
}
\textbf{A.1. MATHEMATICAL TOOLS}

\begin{verbatim}
95 return jacobiSymbol(p.mod(a), a);

if (a.mod(FOUR).equals(THREE) || p.mod(FOUR).equals(THREE)) {
    return (-1) * jacobiSymbol(p.mod(a), a);
}

// should never get here
throw new IllegalArgumentException("probably the second argument is even");
\end{verbatim}
### A.2 Cryptosystems

Listing A.4: Goldwasser-Micali key generation in BenalohKeyPairGenerator.java

```java
/**
   * Generates a new Goldwasser–Micali key pair using the given key generation parameters.
   * @return An AsymmetricCipherKeyPair containing a Goldwasser–Micali key pair.
   */
private AsymmetricCipherKeyPair generateGMKeyPair() {
    // bp are the key generation parameters supplied by the user
    BigInteger p = new BigInteger(bp.getStrength() / 2, bp.getPrimeCertainty(), bp.getRandom());
    BigInteger q = new BigInteger(bp.getStrength() / 2, bp.getPrimeCertainty(), bp.getRandom());
    BigInteger n = p.multiply(q);
    do {
        BigInteger y = PrimeUtils.getRandom(n, bp.getRandom());
    } while (PrimeUtils.jacobiSymbol(y, p) != -1 && PrimeUtils.jacobiSymbol(y, q) != -1);
    return new AsymmetricCipherKeyPair(new BenalohKeyParameters(y, r, n),
                                       new BenalohPrivateKeyParameters(y, r, n, p, q));
}
```

Listing A.5: Benaloh & Goldwasser-Micali encryption in BenalohEngine.java

```java
/**
   * Encrypts a BigInteger representing the plaintext message with the public key.
   * @param plain The plaintext message
   * @return The encrypted plaintext message as BigInteger.toByteArray()
   */
private byte[] encrypt(final BigInteger plain) {
    BigInteger encrypted = bkp.getY().modPow(plain, bkp.getModulus());
    final byte[] tmp = encrypted.toByteArray();
    System.arraycopy(tmp, 0, output, output.length - tmp.length, tmp.length);
    if (debug) {
        System.out.println("Encrypted value is: " + new BigInteger(1, output));
    }
    Arrays.fill(output, Byte.parseByte("0"));
    return output;
}
```
A.2. CRYPTOSYSTEMS

Listing A.6: Goldwasser-Micali decryption in BenalohEngine.java

```java
/**
 * Decrypts a BigInteger representing an encrypted bit with the
 * Goldwasser-Micali algorithm
 * @param encrypted.The encrypted message
 * @return The decrypted message in a BigInteger.toByteArray()
 */
private byte[] decryptGM(final BigInteger encrypted) {
    final BenalohPrivateKeyParameters privKey = (BenalohPrivateKeyParameters) bkp;
    final int pRes = PrimeUtils.jacobiSymbol(encrypted, privKey.getPQ()[0]);
    final int qRes = PrimeUtils.jacobiSymbol(encrypted, privKey.getPQ()[1]);

    if (debug) {
        System.out.println("pRes is "+ pRes);
        System.out.println("qRes is "+ qRes);
    }

    if ((pRes == 1) && (qRes == 1)) {
        return BigInteger.ZERO.toByteArray();
    } else {
        return BigInteger.ONE.toByteArray();
    }
}
```

Listing A.7: Benaloh key generation in BenalohKeyPairGenerator.java

```java
/**
 * Generates a Benaloh key pair using the given key generation parameters.
 * @return An AsymmetricCipherKeyPair containing a Benaloh key pair.
 */
private AsymmetricCipherKeyPair generateBenalohKeyPair() {
    // Generate p
    final BigInteger startP = new BigInteger(bp.getStrength() / 2
    - r.bitLength() - 24, bp.getPrimeCertainty(), bp
    .getRandom());
    while (true) {
        p = startP.multiply(BigInteger.valueOf(2));
        p = p.multiply(new BigInteger(24, bp
        .getPrimeCertainty(), bp.getRandom()));
        p = p.multiply(r);
        p = p.add(BigInteger.ONE);
        final BigInteger pMinus1 = p.subtract(BigInteger.ONE);

        // r shall divide p-1
        if (!pMinus1.mod(r).equals(BigInteger.ZERO)) {
            if (debug) {
                System.out.println("r does not divide p-1");
            }
            continue;
        }

        // (p-1) / r and r shall be relatively prime
        if (!r.gcd(pMinus1.divide(r)).equals(BigInteger.ONE)) {
            if (debug) {
                System.out.println("p-1 /r and r are not relatively prime");
            }
            break;
        }
    }
    return new AsymmetricCipherKeyPair(p, q);
}
```
APPENDIX A. APPENDIX: SOURCE CODE

```java
continue;

if (!p.isProbablePrime(bp.getPrimeCertainty())) {
    if (debug) {
        System.out.println("not prime: " + p);
    }
    continue;
}
if (debug) {
    System.out.println("New Benaloh p * + p * with " + p.bitLength() + " bit.");
} break;

// Generate q
while (true) {
    q = new BigInteger(bp.getStrength() / 2, bp.getPrimeCertainty(), bp.getRandom());
    if (debug) {
        System.out.println("New Benaloh q * + q");
    }
    // r and (q-1) shall be relatively prime.
    if (!r.gcd(q.subtract(BigInteger.ONE)).equals(BigInteger.ONE)) {
        continue;
    }
    break;
}
if (debug) {
    System.out.println("New Benaloh q " + q + " with " + q.bitLength() + " bit.");
}
n = p.multiply(q);
if (debug) {
    System.out.println("New Benaloh n " + n + " with " + n.bitLength() + " bit.");
}
while (true) {
    y = PrimeUtils.getRandom(n, bp.getRandom());
    if (debug) {
        System.out.println("New Benaloh y * + y");
    }
    if (!y.gcd(n).equals(BigInteger.ONE)) {
        continue;
    }
    if (y.modPow(((p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE)).divide(r), n))
        .equals(BigInteger.ONE)) {
        continue;
    }
    break;
}
return new AsymmetricCipherKeyPair(new BenalohKeyParameters(y, r, n), new BenalohPrivateKeyParameters(y, r, n, p, q));
```
### A.2. CRYPTOSYSTEMS

#### Listing A.8: Benaloh decryption in BenalohEngine.java

```java
/**
 * Decrypts a BigInteger that is a Benaloh encrypted message.
 * @param encrypted The encrypted message
 * @return A BigInteger.toByteArray() representing the decrypted message.
 */
private byte[] decryptBenaloh(final BigInteger encrypted) {
    final BenalohPrivateKeyParameters privKey = (BenalohPrivateKeyParameters) bkp;

    // set up lookup table if necessary
    if (privKey.getLookupList().size() == 0) {
        privKey.setupLookupList();
    }

    BigInteger[] pq = privKey.getPQ();
    BigInteger pSub1 = pq[0].subtract(BigInteger.ONE);
    BigInteger qSub1 = pq[1].subtract(BigInteger.ONE);
    BigInteger exp = pSub1.multiply(qSub1).divide(bkp.getMessageBorder());
    BigInteger res = PrimeUtils.chineseModPow(encrypted, exp, pq);

    final byte[] output = privKey.getMessageBorder().toByteArray();
    Arrays.fill(output, Byte.parseByte("0"));
    final BigInteger decrypted = (BigInteger) privKey.getLookupList().get(res);
    final byte[] tmp = decrypted.toByteArray();
    System.arraycopy(tmp, 0, output, output.length - tmp.length, tmp.length);
    if (debug) {
        System.out.println("Decrypted value is " + decrypted);
    }
    return output;
}
```

#### Listing A.9: Naccache Stern key generation in NaccacheSternKeyPairGenerator.java

```java
/**
 * Generates a new NaccacheSternKeyPair using the number of processors given
 * in the constructor and the parameters supplied in the init() method.
 * @see org.bouncycastle.crypto.AsymmetricCipherKeyPairGenerator#generateKeyPair()
 */
public AsymmetricCipherKeyPair generateKeyPair() {
    // Permute the prime list individually
    smallPrimes = permuteList(smallPrimes, rand);

    // compute u and v
    u = BigInteger.ONE;
    v = BigInteger.ONE;

    for (int i = 0; i < smallPrimes.size() / 2; i++) {
        u = u.multiply((BigInteger) smallPrimes.get(i));
    }
    for (int i = smallPrimes.size() / 2; i < smallPrimes.size(); i++) {
        v = v.multiply((BigInteger) smallPrimes.get(i));
    }

    // the upper bound for messages, sigma
```
sigma = u.multiply(v);

// generate a and b (threaded)
generateAB();

if (debug) {
    System.out.println("generating p and q");
}

// generate p and q (threaded)
generatePQ();

// n and \phi(n)
n = p.multiply(q);
phi_n = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));

if (debug) {
    System.out.println("generating y");
}

// compute y, NCS'98 calls it g (threaded)
computeY();

if (debug) {
    System.out.println();
    System.out.println("found new NaccacheStern cipher variables:");
    System.out.println("smallPrimes: " + smallPrimes);
    System.out.println("sigma:..." + sigma + " (" + sigma.bitLength() + " bits)");
    System.out.println("a:.......");
    System.out.println("b:.......");
    System.out.println("p':.......");
    System.out.println("q':.......");
    System.out.println("n:.......");
    System.out.println("phi(n):..." + phi_n);
    System.out.println("y:..........." + y);
    System.out.println();
}

return new AsymmetricCipherKeyPair(new NaccacheSternKeyParameters falsen, y, n, sigma), new NaccacheSternPrivateKeyParameters(y, n, sigma, smallPrimes, p, q, debug, processorCount));

Listing A.10: Naccache Stern encryption in NaccacheSternEngine.java

/**
 * Encrypts a BigInteger aka Plaintext with the public key. Uses
 * probabilistic encryption if a certificate is set.
 *
 * @param plain
 * The BigInteger to encrypt
 *
 * @return The byte[] representation of the encrypted BigInteger (i.e.
 * encrypted.toByteArray())
 *
 */
public byte[] encrypt(final BigInteger plain) {
    final byte[] output = key.getModulus().toByteArray();
    Arrays.fill(output, Byte.parseByte("0"));
    BigInteger encrypted = key.getY().modPow(plain, key.getModulus());
    return output;
// Certificate is the uniformly at random generated obfuscating element
// from $Z_n$
if (certificate != null) {
    encrypted = certificate.modPow(key.getSigma(), key.getModulus())
        .multiply(encrypted).mod(key.getModulus());
}

final byte[] tmp = encrypted.toByteArray();
System.arraycopy(tmp, 0, output, output.length - tmp.length, tmp.length);
if (debug) {
    System.out.println("Encrypted value is: " + new BigInteger(1, output));
}
return output;

/**
 * Decrypts a BigInteger that represents the encrypted message.
 * @param input The message
 * @return A byte array that contains the decrypted message as BigInteger.toByteArray().
 */
private synchronized byte[] decrypt(final BigInteger input) {
    final NaccacheSternPrivateKeyParameters priv = (NaccacheSternPrivateKeyParameters) key;
    final Vector primes = priv.getSmallPrimes();
    plain = new Vector(primes.size());
    threads = new Vector();

    // Get Chinese Remainders of CipherText
    final Object tmp = new Object();
    Thread t;
    for (int i = 0; i < primes.size(); i++) {
        // insert objects into plain, so I can use Vector.setElementAt()
        plain.add(tmp);
        smallPrime = (BigInteger) primes.get(i);
        t = new DecryptBySmallPrime(smallPrime, input);
        threads.add(t);
    }

    final Vector runningThreads = new Vector();

    synchronized (waitFor) {
        for (int i = 0; (i < threads.size()) && (i < processorCount); i++) {
            t = (Thread) threads.get(i);
            runningThreads.add(t);
            t.start();
        }

        while (threads.size() > 0) {
            try {
                waitFor.wait();
            } catch (final InterruptedException e) {
            }
            for (int i = 0; i < threads.size(); i++) {
                t = (Thread) threads.get(i);
                t =...
if (!runningThreads.contains(t)) {
    runningThreads.add(t);
    t.start();
    break;
}
}
}
for (int i = 0; i < runningThreads.size(); i++) {
    t = (Thread) runningThreads.get(i);
    try {
        t.join();
    } catch (final InterruptedException e) {
    }
}

// compute message from remainders
final BigInteger ret_val = PrimeUtils.chineseRemainder(plain, primes);
return ret_val.toByteArray();

/**
 * Callback function for decryption threads. They submit the decrypted value
 * modulo their prime.
 * @param t The decryption thread submitting.
 */
private void submitDecryptionValue(final DecryptBySmallPrime t) {
    final NaccacheSternPrivateKeyParameters priv = (NaccacheSternPrivateKeyParameters) key;
    final Vector smallPrimes = priv.getSmallPrimes();
    plain.setElementAt(BigInteger.valueOf(t.lookedup), smallPrimes.indexOf(BigInteger.valueOf(t.smallPrime)));
    synchronized (threads) {
        threads.remove(t);
    }
    synchronized (waitFor) {
        waitFor.notifyAll();
    }
}

/**
 * DecryptBySmallPrime computes the plaintext modulo a small prime t from
 * the ciphertext. With the Chinese Remainder Theorem, the complete
 * plaintext can be restored.
 * @author lippoll
 */
class DecryptBySmallPrime extends Thread {
    private final BigInteger smallPrime;
    private final BigInteger input;
    private final NaccacheSternPrivateKeyParameters privKey;
    private int lookedup;
    private boolean debug;

    DecryptBySmallPrime(final BigInteger smallPrime, final BigInteger input) {
        super();
public void run() {
    BigInteger[] pq = privKey.getPQ();
    BigInteger exponent = pq[0].subtract(BigInteger.ONE);
    exponent = exponent.multiply(pq[1].subtract(BigInteger.ONE));
    exponent = exponent.divide(smallPrime);
    Vector al = (Vector) privKey.getLookupTable().get(smallPrime);
    lookedup = al.indexOf(PrimeUtils.chineseModPow(input, exponent, pq));
    if (debug) {
        System.out.println("decryption for prime " + smallPrime + " finished.");
    }
    submitDecryptionValue(this);
}
A.3 Protocols

Listing A.12: Client generation of N in GilboaClient.java

```java
/**
 * Computes a sharing for a given prime
 * @param prime the prime to use for the sharing
 * @return true if the sharing is successful, false if the protocol needs to
 * be restarted.
 * @throws Exception if the other party talks about a different prime
 */
private boolean sendSharingModPrime(final BigInteger prime) throws Exception {
    int position;
    // first retrieve the correct position of our prime lists
    if (super.actualState.equals(states[4])) {
        position = genPrimes.indexOf(prime);
    } else {
        position = verPrimes.indexOf(prime) + genPrimes.size();
    }
    // get the correct pre-computed Benaloh systems
    final BenalohEngine pubEngine = ((BenalohSystem) benalohSystems.get(prime)).getPubEngine();
    final BenalohEngine privEngine = ((BenalohSystem) benalohSystems.get(prime)).getPrivEngine();

    // set N mod p_i to 0, thus N is divided by p_i
    int nModPi = 0;
    BigInteger[] pq = generateNewPrimes(prime);
    pModPrime = pq[0];
    qModPrime = pq[1];
    // set them in the lists that are later used to reconstruct N
    // via CRT
    pComposites.set(position, pModPrime);
    qComposites.set(position, qModPrime);
} else {
    // p and q are set, calculate values
    pModPrime = p.mod(prime);
    qModPrime = q.mod(prime);
}

// encrypt p mod p_i and q mod p_i
final byte[] xl = pubEngine.processBlock(pModPrime.toByteArray(), 0, pModPrime.toByteArray().length);
final byte[] x2 = pubEngine.processBlock(qModPrime.toByteArray(), 0, qModPrime.toByteArray().length);
```

A.3. PROTOCOLS

```java
// write them to the server
out.write(GilboaProtocol.CONVERSION_MODPRIME);
out.writeObject(prime);
out.writeObject(z1);
out.writeObject(z2);
out.flush();

n = BigInteger.ZERO;
// make sure that both parties are at the same step of the protocol
if (in.read() == GilboaProtocol.COMPUTED_N)
    final BigInteger remPrime = (BigInteger) in.readObject();
// check that both parties operate on the same prime
if (prime.equals(remPrime)) {
    // read in the servers computed result
    final byte[] z_data = new byte[in.readInt()];
in.readFully(z_data);
    // and decrypt it
    n = new BigInteger(1, privEngine.processBlock(z_data, 0, z_data.length));
    // compute the final N mod p + 1
    n = n.add((pModPrime.multiply(qModPrime).mod(prime)).mod(prime));
    // test whether N != 0
    if (n.equals(BigInteger.ZERO)) {
        // N is zero mod prime
        log.debug(REMOTE_IP + " Found that N mod " + prime + " is 0. \n");
        if (super.actualState.equals(states[4])) {
            // generate new shares mod prime, we still can
            // adjust
            out.writeObject(GilboaProtocol.N_MODPRIME_IS_0);
            out.flush();
        } else {
            // Testing rsaN failed.
            // Restart generation.
            log.info(REMOTE_IP + " Restarting generation. " + "N is divisible by " + prime);
            out.writeObject(GilboaProtocol.N_FAILED);
        }
    } else {
        // successfully generated N mod prime
        nModPi = n.intValue();
        // set it in the list from which N is recomposed
        super.actualState = states[3];
        // and restart generation of N
        out.writeObject(GilboaProtocol.START_CANDIDATE_GENERATION);
        out.flush();
        // inform caller about failure
        return false;
    }
```

// Successfully generated N mod prime
nModPi = n.intValue();
// set it in the list from which N is recomposed
nModPrimes.set(position, n);
// inform other party about success
out.write(GilboaProtocol.COMPUTED_N);
out.flush();
}

} else {
    throw new Exception(PROTO_VIOL
         + "Not talking about the same prime.");
}

// finally set n to the actual value so that the process restarts if
// it is 0
nModPi = n.intValue();

// if N != 0 mod prime, return true
return true;
Listing A.13: Server generation of N in GilboaServer.java

```java
/**
 * Computes N' = p_c * q_s + q_c * p_s + p_s * q_s mod <t>prime</t>. In the
 * Gilboa paper these are named z1, z2 and z3, N' is named z.
 * @param prime
 * @param The prime for that N' shall be computed
 * @throws Exception
 * @if the protocol is violated.
 */
private void computeN(final BigInteger prime) throws Exception {
    // get the other parties public benaloh system
    final BenalohSystem bs = (BenalohSystem) benalohSystems.get(prime);
    final BenalohEngine pubEngine = bs.getPubEngine();

    // read in clients p
    byte[] p_c;
    if (in.read() == GilboaProtocol.REMOTE_Z1) {
        final int size = in.readInt();
        p_c = new byte[size];
        in.readFully(p_c);
    } else {
        throw new Exception(PROTO_VIOL + "REMOTE_Z1");
    }

    // read in clients q
    byte[] q_c;
    if (in.read() == GilboaProtocol.REMOTE_Z2) {
        final int size = in.readInt();
        q_c = new byte[size];
        in.readFully(q_c);
    } else {
        throw new Exception(PROTO_VIOL + "REMOTE_Z2");
    }

    // set own sharings of p and q mod prime
    BigInteger pModPrime;
    BigInteger qModPrime;
    if (super.actualState.equals(states[4])) {
        // while generating p and q
        final int pos = genPrimes.indexOf(prime);
        Biginteger[] pq = generateNewPrimes(prime);
        // set p and q in the lists used to reconstruct them via CRT
        pComposites.set(pos, pq[0]);
        qComposites.set(pos, pq[1]);
        pModPrime = pq[0];
        qModPrime = pq[1];
    } else {
        // p and q are fixed, set correct value
        pModPrime = p.mod(prime);
        qModPrime = q.mod(prime);
    }

    // exponentiate p_c with pModPrime, this is equivalent to a plaintext
    // multiplication
    p_c = pubEngine.multiplyCryptedBlock(p_c, qModPrime);
    // exponentiate q_c with pModPrime, this is equivalent to a plaintext
    // multiplication
    q_c = pubEngine.multiplyCryptedBlock(q_c, pModPrime);

    // Compute p * q mod prime
    final byte[] pqModPrime = pModPrime.multiply(qModPrime).mod(prime).toByteArray();

    // encrypt it
```

A.3. PROTOCOLS
byte[] n_ = pubEngine.processBlock( pqModPrime, 0, pqModPrime.length );

// multiply all encrypted blocks successively, this is equivalent to
// addition in plaintext
n_ = pubEngine.addCryptedBlocks( n_, q_c );
n_ = pubEngine.addCryptedBlocks( n_, p_c );

// inform the client about the newly computed N'
out.writeObject( GilboaProtocol.COMPUTED_N );
out.writeObject( prime );
out.writeInt( n_.length );
out.write( n_ );
out.flush();

// check result of complete computation
final int command = in.read();
if (command == GilboaProtocol.COMPUTED_N ) {
    // everything went well, p * q mod prime != 0
    if (super.actualState.equals( states[4] )) {
        computedZs.set( genPrimes.indexOf( prime ), prime );
    }
    return;
} else if ( (command == GilboaProtocol.N_MOD_PRIME_IS_0) &&
    super.actualState.equals( states[4] ) ) {
    log.info(REMOTE_IP + " Got N_MOD_PRIME_IS_0 for prime " + prime);
} else if (command == GilboaProtocol.N_FAILED) {
    log.info(REMOTE_IP + " Restarting generation, rsaN failed for " + prime);
    // reset state machine to correct state
    super.actualState = states[3];
} else {
    // got protocol violation, exit
    final String msg = PROTO_VIOL + " Wrong command: " + command;
    log.debug(REMOTE_IP + " actual state is " + super.actualState);
    log.fatal(REMOTE_IP + msg);
    throw new Exception(msg);
}
Listing A.14: Client BiPrime test of N in GilboaClient.java

```java
/**
 * Tests if the rsaN is composed of exactly two primes. First, the
 * candidates of both parties are fixed and then the test is performed by
 * using the candidate that is the sum of both candidates.
 * @return True if the test is successful.
 * @throws Exception If the protocol is violated or the parties disagree about the
 * composite.
 */
private boolean sendWitness() throws Exception {
    // Inform the server that a BiPrime test starts
    out.write(GilboaProtocol.BIPRIME_TEST);
    BigInteger witness;
    // fix our half of the candidate
    witness = PrimeUtils.getRandom(rsaN, rand);
    SHA1Digest sha1 = new SHA1Digest();
    byte[] wData = witness.toByteArray();
    byte[] digest = new byte[sha1.getDigestSize()];
    sha1.doFinal(digest, 0);
    // and inform server about the candidate but do not reveal it
    out.write(GilboaProtocol.SHA1_DIGEST);
    out.write(digest.length);
    out.write(digest);
    out.flush();
    // read the servers candidate
    if (in.read() == GilboaProtocol.SHA1_DIGEST) {
        byte[] remoteDigest = new byte[in.read()];
        in.readFully(remoteDigest);
        // reveal our candidate
        out.writeObject(witness);
        out.flush();
        // read the remote candidate and verify it
        BigInteger remCand = (BigInteger) in.readObject();
        wData = remCand.toByteArray();
        sha1.reset();
        sha1.update(wData, 0, wData.length);
        sha1.doFinal(digest, 0);
        if (((new BigInteger(1, remoteDigest)).equals(new BigInteger(1, digest)))) {
            // compose the final candidate of both sharings
            witness = witness.add(remCand);
            while (PrimeUtils.jacobiSymbol(witness, rsaN) != 1) {
                witness = witness.add(BigInteger.ONE);
            }
        } else {
            String msg = REMOTE_IP + PROTO_VIOL + " SHA-1 digest not correct";
            log.fatal(msg);
            throw new Exception(msg);
        }
    } else {
        String msg = REMOTE_IP + PROTO_VIOL + "expected SHA1_DIGEST";
        log.fatal(msg);
        throw new Exception(msg);
    }

    // Do the simple test that catches almost all
    final BigInteger myExp = ((p.add(q)).negate()).divide(four);
    BigInteger jWitness = witness.modPow(myExp, rsaN);
    jWitness = jWitness.mod(rsaN);
    // log.debug(REMOTE_IP + "Sending Witness" + witness);
    out.write(GilboaProtocol.WITNESS);
```
out.writeObject(witness);
out.writeObject(jWitness);
out.flush();
BigInteger remWitness;
if ((in.read() == GilboaProtocol.WITNESS) {
  remWitness = (BigInteger) in.readObject();
} else {
  final String msg = PROTO_VIOL + "expecting WITNESS";
  log.fatal(REMOTE_IP + msg);
  throw new Exception(msg);
}
jWitness = remWitness.multiply(jWitness);
int retval = in.read();
if ((jWitness.equals(BigInteger.ONE) || jWitness.equals(rsaN)
  && (retval == GilboaProtocol.N_VERIFIED)) {
  // If it succeeded, do a final test in the twisted group

  // First fix our share of the twisted candidate
  BigInteger[] twWitness = PrimeUtils.getRandomTwistedElement(rsaN, rand);
  sha1.reset();
  wData = twWitness[0].toByteArray();
  sha1.update(wData, 0, wData.length);
  wData = twWitness[1].toByteArray();
  sha1.update(wData, 0, wData.length);
  sha1.doFinal(digest, 0);
  // and inform the server about it
  out.write(GilboaProtocol.SHA1_DIGEST);
  out.write(digest.length);
  out.flush();
  // read the server's hash value
  if ((in.read() == GilboaProtocol.SHA1_DIGEST) {
    byte[] remoteDigest = new byte[in.read()];
    in.readFully(remoteDigest);
    // and reveal our element
    out.writeObject(twWitness[0]);
    out.writeObject(twWitness[1]);
    out.flush();
    // read the server's element
    BigInteger[] remCand = new BigInteger[2];
    remCand[0] = (BigInteger) in.readObject();
    remCand[1] = (BigInteger) in.readObject();
    // and verify it
    wData = remCand[0].toByteArray();
    sha1.reset();
    wData = remCand[1].toByteArray();
    sha1.update(wData, 0, wData.length);
    sha1.doFinal(digest, 0);
    if (!((new BigInteger(1, remoteDigest)).equals(new BigInteger(1, digest)))) {
      // If all is correct, fix the final element
      twWitness = PrimeUtils.twistedAdd(twWitness, remCand);
A.3. PROTOCOLS

79

} else {
    String msg = REMOTE_IP + PROTO_VIOL + " SHA-1 digest " + "of transferred object is not correct";
    log.fatal(msg);
    throw new Exception(msg);
}
}
} else {
    String msg = REMOTE_IP + PROTO_VIOL + "expected SHA1_DIGEST";
    log.fatal(msg);
    throw new Exception(msg);
}

// Do the clients part of the test in the twisted group
BigInteger[] jTwWitness = PrimeUtils.twistedModPow(twWitness, p.add(q), rsaN);
// and inform the server about it
out.writeObject(GilboaProtocol.TW_WITNESS);
out.writeObject(twWitness[0]);
out.writeObject(twWitness[1]);
out.writeObject(jTwWitness[0]);
out.writeObject(jTwWitness[1]);
out.flush();

// Read the servers results
final BigInteger[] remTwWitness = new BigInteger[2];
if (in.read() == GilboaProtocol.TW_WITNESS) {
    remTwWitness[0] = (BigInteger) in.readObject();
    remTwWitness[1] = (BigInteger) in.readObject();
}

// and combine the results to get the final result
jTwWitness = PrimeUtils.twistedModMult(remTwWitness, jTwWitness, rsaN);

 retval = in.read();
if (jTwWitness[1].equals(BigInteger.ZERO)) {
    if (retval == GilboaProtocol.N_VERIFIED) {
        // if both parties are sure that rsaN is prime for this candidate, return true to the calling function
        log.info(REMOTE_IP + " For witness " + witness + ", the rsaN passes the test");
        out.writeObject(GilboaProtocol.N_VERIFIED);
        out.flush();
        return true;
    } else {
        // Throw exception on disagreement and exit
        final String msg = "Other side thinks that * + rsaN is not prime, but I do";
        log.fatal(REMOTE_IP + msg);
        throw new Exception(msg);
    }
} else {
    if (retval == GilboaProtocol.N_FAILED) {
        // Else return false if both parties agree that rsaN is not composed of exactly two primes.
        log.info(REMOTE_IP + "8 failed BiPrimality test");
        return false;
    } else {
        // Throw exception on disagreement and exit
        final String msg = "Other side thinks that * + rsaN is prime, but I don't";
        log.fatal(REMOTE_IP + msg);
        throw new Exception(msg);
    }
}
} else {
    // Return false if the first test failed already
    log.info("RSA N failed first BiPrimality test. Restarting.");
    return false;
}
A.3. PROTOCOLS

Listing A.15: Server BiPrime test of N in GilboaServer.java

```java
/**
 * Does a biPrime test for the RSA N. Notifies the client of the result and
 * resets the server if an invalid rsaN is detected.
 * @throws Exception
 *     If the protocol is not followed correctly.
 */
private void doBiPrimeTest() throws Exception {
    BigInteger witness;

    // First, fix the shares of both parties without revealing them
    if (in.read() == GilboaProtocol.SHA1_DIGEST) {
        // Read in the other side's sha-1 digest
        byte[] remoteDigest = new byte[in.read()];
        in.readFully(remoteDigest);

        // Compute our own witness part
        witness = PrimeUtils.getRandom(rsaN, rand);

        // and inform the other party about it without revealing it
        byte[] wData = witness.toByteArray();
        SHA1Digest shal = new SHA1Digest();
        shal.update(wData, 0, wData.length);
        byte[] digest = new byte[shal.getDigestSize()];
        shal.doFinal(digest, 0);
        out.write(GilboaProtocol.SHA1_DIGEST);
        out.write(digest.length);
        out.write(digest);
        out.flush();

        // Read the other parties candidate and verify it
        BigInteger remCand = (BigInteger)in.readObject();
        wData = remCand.toByteArray();
        shal.reset();
        shal.update(wData, 0, wData.length);
        shal.doFinal(digest, 0);
        if ((new BigInteger(1, remoteDigest)).equals(new BigInteger(1, digest))) {
            // if it is correct, send our own candidate
            out.writeObject(witness);
            out.flush();

            // and add both candidates to get the final candidate
            witness = witness.add(remCand);

            // find the next needed value
            while (PrimeUtils.jacobiSymbol(witness, rsaN) != 1) {
                witness = witness.add(BigInteger.ONE);
            }
        } else {
            String msg = REMOTE_IP + PROTO_VIOL + "SHA-1 digest " + "of transferred object is not correct";
            log.fatal(msg);
            throw new Exception(msg);
        }
    } else {
        String msg = REMOTE_IP + PROTO_VIOL + "expected SHA1_DIGEST";
        log.fatal(msg);
        throw new Exception(msg);
    }

    // Then do a simple test. If this test fails already, we do not have to
```
APPENDIX A. APPENDIX: SOURCE CODE

```java
// test in the twisted group
if (in.read() == GilboaProtocol.WITNESS) {

    // verify that the other party found the same witness
    BigInteger compWitness = (BigInteger) in.readObject();
    if (compWitness.equals(witness)) {

        // read the other parties result
        final BigInteger remResult = (BigInteger) in.readObject();
        // compute our own result
        BigInteger jWitness = witness.modPow((rsaN.subtract(p).subtract(q).add(BigInteger.ONE)).divide(BigInteger.valueOf(4)), rsaN);
        jWitness = jWitness.mod(rsaN);
        // and inform the client about our part
        out.writeObject(GilboaProtocol.WITNESS);
        out.writeObject(jWitness);
        out.flush();

        // compute the final result
        jWitness = jWitness.multiply(remResult);
        jWitness = jWitness.mod(rsaN);
        if (jWitness.equals(BigInteger.ONE) || jWitness.equals(rsaN.subtract(BigInteger.ONE))) {
            // if successful inform the client
            out.writeObject(GilboaProtocol.N_VERIFIED);
            out.flush();
        } else {
            // if unsuccessful inform client
            log.info(REMOTE_IP + " Found witness that rsaN is not a product of two primes.");
            out.writeObject(GilboaProtocol.N_FAILED);
            out.flush();

            // and reset state machine so that a new N must be generated
            super.actualState = states[3];
            primeTries.clear();
            return;
        }
    }
    else {
        String msg = REMOTE_IP + PROTO_VIOLET "Not using the same witness";
        log.fatal(msg);
        throw new Exception(msg);
    }
}

// The previous test was successful. Now do a test in the twisted group.
BigInteger twWitness[] = PrimeUtils.getRandomTwistedElement(rsaN, rand);
if (in.read() == GilboaProtocol.SHA1DIGEST) {
    // read the sha-1 digest of the client’s candidate byte[] remoteDigest = new byte[in.read()];
    in.readFully(remoteDigest);
    // compute our own digest
    byte[] wData = twWitness[0].toByteArray();
    SHA1Digest shal = new SHA1Digest();
    shal.update(wData, 0, wData.length);
    wData = twWitness[1].toByteArray();
    shal.update(wData, 0, wData.length);
    byte[] digest = new byte[shal.getDigestSize()];
    shal.doFinal(digest, 0);

    // and inform the client about it
    out.writeObject(GilboaProtocol.SHA1DIGEST);
```

A.3. PROTOCOLS

out.write(digest.length);
out.write(digest);
out.flush();

// read in the client's candidate
BigInteger remWitness[] = new BigInteger[2];
remWitness[0] = (BigInteger) in.readObject();
remWitness[1] = (BigInteger) in.readObject();

// and verify its digest
sha1.reset();
wData = remWitness[0].toByteArray();
sha1.update(wData, 0, wData.length);
wData = remWitness[1].toByteArray();
sha1.update(wData, 0, wData.length);
sha1.doFinal(digest, 0);

if ((new BigInteger(1, remoteDigest)).equals(new BigInteger(1, digest))) {
  // if digest is correct, reveal our candidate
  out.writeObject(twWitness[0]);
  out.writeObject(twWitness[1]);
  out.flush();
  // and compute final composed candidate
  twWitness = PrimeUtils.twistedAdd(remWitness, twWitness);
} else {
  // else abort computation
  String msg = REMOTE_IP + PROTO_VIOL + " SHA-1 digest "
            + " of transferred object is not correct";
  log.fatal(msg);
  throw new Exception(msg);
}

else {
  String msg = REMOTE_IP + PROTO_VIOL + " expected SHA1_DIGEST";
  log.fatal(msg);
  throw new Exception(msg);
}

// read the client's composed candidate
final BigInteger remTwWitnessResult[] = new BigInteger[2];
if (in.read() == GilboaProtocol.TW_WITNESS) {
  final BigInteger remWitness[] = new BigInteger[2];
  remWitness[0] = (BigInteger) in.readObject();
  remWitness[1] = (BigInteger) in.readObject();
  if (!remWitness[0].equals(twWitness[0])
            || !remWitness[1].equals(twWitness[1])) {
    // if they are not equal, the other side is cheating, exit
    throw new Exception("witnesses not equal");
  }
  // read the client's part of the test
  remTwWitnessResult[0] = (BigInteger) in.readObject();
  remTwWitnessResult[1] = (BigInteger) in.readObject();
} else {
  final String msg = PROTO_VIOL + " expecting TW_WITNESS";
  log.fatal(REMOTE_IP + msg);
  throw new Exception(msg);
}

// compute our part of the test
BigInteger[] jTwWitness = PrimeUtils.twistedModPow(twWitness, rsaN.add(p).add(q).add(new BigInteger.ONE), rsaN);
// and inform the client about it
out.writeObject(GilboaProtocol.TW_WITNESS);
out.writeObject(jTwWitness[0]);
out.writeObject(jTwWitness[1]);
out.flush();
// compute the final outcome
jTwWitness = PrimeUtils.twistedModMult(jTwWitness, remTwWitnessResult, rsaN);

if (jTwWitness[1].equals(BigInteger.ZERO)) {
    // if successful, inform the client
    log.info(REMOTE_IP + " For witness " + witness + " , the rsaN passes the test ");
    out.write(GilboaProtocol.N_VERIFIED);
    out.flush();
    // hopefully it thinks the same ;)
    if (in.read() == GilboaProtocol.N_VERIFIED) {
        log.info(REMOTE_IP + " RSA verified for " + witness);
        // Record the successful test in our statistics, so we can
        // verify later that we did enough tests
        primeTries.add(witness);
    } else {
        // if parties disagree, exit
        final String msg = " Other side thinks that " + rsaN + " is not prime, but I do ";
        log.fatal(REMOTE_IP + msg);
        throw new Exception(msg);
    }
} else {
    // if unsuccessful, also inform the client
    log.info(REMOTE_IP + " Found witness " + " that rsaN is not a product of two primes. ");
    out.write(GilboaProtocol.N_FAILED);
    out.flush();
    // and reset state machine
    super.actualState = states[3];
    primeTries.clear();
}

/* Prepares the RSA key pair needed to perform an oblivious transfer and
 * sends it to the other party
 * @throws Exception
 * If oblivious transfers could not be prepared correctly.
 */
private void setupOblTransfers(final BigInteger ring) throws Exception{
    log.debug("Setting up oblivious transfers " + "for desired exponent");
    log.info("Generating RSA key pair in ring " + ring + " for oblivious transfers");
    // Do not give pubExp too few 1-bits to avoid SideChannel Attacks
    BigInteger pubExp;
    do {
        pubExp = new BigInteger(ring.bitLength(), rand);
        if (pubExp.isProbablePrime(certainty)) {
            break;
        }
    } while (true);
    // Construct the RSA key that has size of the ring
    final RSAPublicKeyParameters param = new RSAPublicKeyParameters(pubExp, rand, ring.bitLength(), certainty);
    final RSAPublicKeyPairGenerator generator = new RSAPublicKeyPairGenerator();
    generator.init(param);
    final AsymmetricCipherKeyPair pair = generator.generateKeyPair();
    final RSAPublicKeyParameters oblPubKey = (RSAPublicKeyParameters) pair.getPublic();
    key = (RSAPrivateCrtKeyParameters) pair.getPrivate();
    // Write the data needed for the oblivious transfer to the server
    out.writeObject(KeyProtocol.OBL_RING);
    out.writeObject(ring);
    out.writeObject(KeyProtocol.OBL_EXP);
    out.writeObject(oblPubKey.getModulus());
    // verify that all went well
    if ((in.read() == KeyProtocol.OBLIVIOUS_SETUP) {
        return;
    } else {
        String msg = REMOTE_IP + PROTO_VIOL + " expecting OBLIVIOUS_SETUP":
        log.fatal(msg);
        throw new Exception(msg);
    }
}

/**
 * Does i 2−l oblivious transfers to enable the other side to learn the
 * needed bits of my value and returns my share of the computation.
 * @param myValue
 * @throws Exception
 * If something goes wrong
 */
private BigInteger startObliviousTransfer(final BigInteger ring,
                                           final BigInteger value) throws Exception{
    log.debug("starting Oblivious Transfer");
// make sure that value is an element of the ring
final BigInteger myValue = value.mod(ring);

// initialize y, which will hold the sum of our m0
BigInteger y = BigInteger.ZERO;

// start the transfer
out.write(KeyProtocol.OBL_START);
out.flush();

// verify other side is ready
if (in.read() != KeyProtocol.OBL_START) {
    final String msg = REMOTE_IP + PROTO_VIOL + " Expecting OBL_START";
    log.fatal(msg);
    throw new Exception(msg);
}

// do ring.bitLength() oblivious transfers to convert an element
for (int i = 0; i < ring.bitLength(); i++) {
    // inform server that an obliv transfer is coming
    out.write(KeyProtocol.OBL_TRANSFER);

    // prepare two random messages x0 and x1
    final BigInteger x0 = PrimeUtils.getRandom(ring, rand);
    final BigInteger x1 = PrimeUtils.getRandom(ring, rand);

    // and write them to the server
    out.write(KeyProtocol.OBL_X0);
    out.writeObject(x0);
    out.write(KeyProtocol.OBL_X1);
    out.writeObject(x1);
    out.flush();

    // server chooses one and adds his encrypted k. We read it
    if (in.read() == KeyProtocol.OBL_KX) {
        final BigInteger kx = (BigInteger) in.readObject();

        // and compute the two possible results
        final BigInteger k0 = rsaDecrypt(kx.subtract(x0));
        final BigInteger k1 = rsaDecrypt(kx.subtract(x1));

        // prepare our messages m0 (random) and m1 = 2^i * m0
        final BigInteger m0 = PrimeUtils.getRandom(ring, rand);
        BigInteger m1 = x.multiply(myValue);
        m1 = m1.mod(ring);
        y = y.add(m0).mod(ring);

        // add them to the two possible decrypts and send each to the server
        out.write(KeyProtocol.OBL_M0K0);
        out.writeObject(k0.add(m0));
        out.writeObject(KeyProtocol.OBL_M1K1);
        out.writeObject(k1.add(m1));
        out.flush();

        // server chooses correct one, one transfer is finished
    } else {
        final String msg = REMOTE_IP + PROTO_VIOL + " Expecting OBL_KX";
        log.fatal(msg);
        throw new Exception(msg);
    }

    // finally, negate our sum
    y = y.negate();
// and make sure that it is a ring element (the sum may be larger than the ring)
y = y.mod(ring);
log.debug("Oblivious Transfer finished");
return y;
}
Listing A.17: Oblivious Transfer for Party A in ComputeKeyServer.java

```java
//
// Does a Oblivious Transfer for one bit of <tt>value</tt>. In each
// transfer the bit is increased by one, so that in the end all bits have
// been transferred.
//
// @throws Exception
// If either the protocol is violated or the connection is
// closed.
//
private void obliviousTransfer(final BigInteger ring, final BigInteger value)
    throws Exception {
    // make sure that value is in \( \mathbb{Z}_N \)
    final BigInteger myValue = value.mod(ring);
    actualBit++;
    log.debug("Now executing bit " + actualBit + "/" + ring.bitLength());
    // construct private element
    BigInteger k = PrimeUtils.getRandom(ring, rand);
    // and make sure that it can be encrypted
    k = k.mod(oblModulus);
    // and encrypt it
    BigInteger kx = k.modPow(oblExp, oblModulus);
    // read the other parties prepared messages
    BigInteger m0, m1;
    if (in.read() == KeyProtocol.OBL_X0) {
        m0 = (BigInteger) in.readObject();
    } else {
        final String msg = PROTO_VIOL + REMOTE_IP + " Expecting OBL_X0";
        log.fatal(msg);
        throw new Exception(msg);
    }
    if (in.read() == KeyProtocol.OBL_X1) {
        m1 = (BigInteger) in.readObject();
    } else {
        final String msg = PROTO_VIOL + REMOTE_IP + " Expecting OBL_X1";
        log.fatal(msg);
        throw new Exception(msg);
    }
    // and add the correct message to my encrypted value
    if (myValue.testBit(actualBit)) {
        kx = kx.add(m1);
    } else {
        kx = kx.add(m0);
    }
    // write the sum to the other party
    out.writeObject(KeyProtocol.OBL_KX);
    out.writeObject(kx);
    out.flush();
    // and read the answers
    BigInteger m0k0, m1k1;
    if (in.read() == KeyProtocol.OBL_M0K0) {
        m0k0 = (BigInteger) in.readObject();
    } else {
        final String msg = PROTO_VIOL + REMOTE_IP + " Expecting OBL_M0K0";
        log.fatal(msg);
        throw new Exception(msg);
    }
    if (in.read() == KeyProtocol.OBL_M1K1) {
        m1k1 = (BigInteger) in.readObject();
    } else {
        final String msg = PROTO_VIOL + REMOTE_IP + " Expecting OBL_M1K1";
        log.fatal(msg);
        throw new Exception(msg);
    }
    //
```
A.3. PROTOCOLS

```java
if (in.read() == KeyProtocol.OBL_M1K1) {
    m1k1 = (BigInteger) in.readObject();
} else {
    final String msg = PROTO_VIO + REMOTE_IP + " Expecting OBL_M1K1";
    log.fatal(msg);
    throw new Exception(msg);
}

// choose the correct answer
BigInteger m;
if (myValue.testBit(actualBit)) {
    m = m1k1.subtract(k);
} else {
    m = m0k0.subtract(k);
}

// and record it for later addition with other elements
oblMessages.set(actualBit, m);
```