Secure Computing and Distributed Solutions to the Secret Santa Problem

Johannes Verelst
jverelst@cs.uu.nl

Version 0.5, final draft
# Table Of Contents

1 Introduction .................................................. 4  
  1.1 Organisation of this Thesis ......................... 4  
  1.2 Cryptography Introduction .......................... 4  
  1.3 Security ............................................... 6  
    1.3.1 Security Notions ................................. 7  
    1.3.2 Adversaries .................................. 8  

2 Problem Description ........................................... 10

3 Previous Work .................................................. 11  
  3.1 Trusted Third Party .................................. 11  
  3.2 Tel's Algorithm ...................................... 11  

4 Secure Computations .......................................... 13  
  4.1 Existing Protocols .................................... 13  
    4.1.1 Garbled Circuits ................................ 13  
    4.1.2 Verifiable Secret Sharing ....................... 14  
    4.1.3 Joint Homomorphic Encryption ................. 15  
  4.2 Application to Secret Santa ......................... 16  
  4.3 Solution ............................................... 16  
    4.3.1 Generation of Random Bits ..................... 18  
    4.3.2 Complexity ..................................... 18  
    4.3.3 Implementation ................................ 18  

5 Mix-Networks .................................................. 21  
  5.1 Existing Protocols .................................... 21  
    5.1.1 Encryption Chains ............................... 21  
    5.1.2 Re-encryption .................................. 21  
  5.2 Application to Secret Santa ......................... 22  
  5.3 Solution ............................................... 22  
    5.3.1 Security ......................................... 25  
    5.3.2 Complexity ..................................... 26  
    5.3.3 Implementation ................................ 26
# Implementation

6 Implementation  

## 6.1 Libraries  

6.1 Libraries ........................................................................... 27  

## 6.2 Environment  

6.2 Environment .................................................................... 27  

### 6.2.1 Choice of programming language  

6.2.1 Choice of programming language ....................................... 27  

### 6.2.2 Choice of server environment  

6.2.2 Choice of server environment ........................................... 29  

## 6.3 Design  

6.3 Design ............................................................................. 29  

### 6.3.1 Initialisation  

6.3.1 Initialisation .................................................................... 30  

### 6.3.2 Registration  

6.3.2 Registration ................................................................... 30  

### 6.3.3 Submitting a Public Key  

6.3.3 Submitting a Public Key ................................................... 31  

### 6.3.4 Forming a Group  

6.3.4 Forming a Group ................................................................ 31  

### 6.3.5 Key Download  

6.3.5 Key Download .................................................................. 33  

### 6.3.6 Mixing  

6.3.6 Mixing ........................................................................... 33  

### 6.3.7 Proof Verification  

6.3.7 Proof Verification ................................................................ 34  

### 6.3.8 Plaintext Equality Test  

6.3.8 Plaintext Equality Test ..................................................... 34  

### 6.3.9 Joint Decryption  

6.3.9 Joint Decryption ............................................................... 35  

## 6.4 Evaluation  

6.4 Evaluation ........................................................................ 35  

# Bibliography

A Bibliography ........................................................................ 37  

# Implementation Details

B Implementation Details .......................................................... 41  

## B.1 Libraries  

B.1 Libraries ........................................................................... 41  

### B.1.1 JCE  

B.1.1 JCE ............................................................................. 41  

### B.1.2 Cryptix  

B.1.2 Cryptix ........................................................................ 41  

### B.1.3 Flexiprovider  

B.1.3 Flexiprovider ................................................................... 42  

### B.1.4 Bouncy Castle  

B.1.4 Bouncy Castle ................................................................... 42  

## B.2 Software Design  

B.2 Software Design ................................................................ 42  

### B.2.1 Classes  

B.2.1 Classes ........................................................................ 42  

## B.3 Database Design  

B.3 Database Design ................................................................. 44  

### B.3.1 User  

B.3.1 User ............................................................................ 44  

### B.3.2 Usergroup  

B.3.2 Usergroup ...................................................................... 45  

### B.3.3 Membership  

B.3.3 Membership ................................................................... 45  

### B.3.4 Mixdata  

B.3.4 Mixdata .......................................................................... 45  

### B.3.5 Plaintext Equality Test  

B.3.5 Plaintext Equality Test .................................................... 46  

### B.3.6 Joint Decryption  

B.3.6 Joint Decryption .............................................................. 46
1. Introduction

Suppose we have a group of friends who wish to give each other presents with Christmas. They create ballots with their name on it, fold it, and put them in a hat. After shuffling them, every participant takes out a ballot with a name on it; he just picked the name of the person he has to buy a present for. In case somebody picked his own name, all ballots are folded again and re-shuffled. This problem is known as Secret Santa. In the Netherlands, this method is frequently used at the celebration known as Sinterklaas.

While this real-life example seems trivial, various complications arise when the friends are not located in the same room, for instance if some of them live abroad. In this case we use a computer to create ballots and a permutation, we could let it e-mail these ballots to the participants. Unfortunately, we need to trust one of the friends to create a valid permutation and take his word for not looking at the ballots.

In this paper, we will research the possibilities for securely creating a permutation with no fixed points (nobody will receive a ballot with his own name), where nobody learns more than the name of the person he has to buy a present for. We give away that we did indeed find solutions using different cryptographic approaches.

1.1 Organisation of this Thesis

In this thesis, we will investigate a protocol for the Secret Santa problem. To do this, we need to find a protocol that allows us to create a derangement of ballots. To the best of the author’s knowledge, there are two ways to achieve this. The first is by using Secure Computation (chapter 4), the second is by using a network of Mix-Servers (chapter 5). In this preface we will give some cryptographic definitions for those readers that are unfamiliar with this subject. This thesis ends with a discussion about the implementation of the mixing solution.

1.2 Cryptography Introduction

ElGamal Encryption In this thesis we frequently use the ElGamal asymmetric encryption algorithm. ElGamal is a so-called probabilistic encryption system; given a key and a plaintext, there are many possible ciphertexts.

The basis of ElGamal is a group $\mathbb{Z}_p$, where $q$ and $p$ are primes so that $p = kq + 1$, given an integer $k$. Given a known generator $g$ of group $\mathbb{Z}_p^*$ and a private key $x \in \mathbb{Z}_{p-1}$, the public key $y = g^x$ in group $\mathbb{Z}_p^*$. For encryption of a message $m$, we first choose a random $k \in \mathbb{Z}_{p-1}$. Our ciphertext is the
1.2. Cryptography Introduction

tuple \( c = (g^k, m \cdot b^k) \). Decryption \( m' \) of a tuple \((u, v)\) consists of calculating \( m' = v \cdot u^{-a} \).

One of the interesting properties of ElGamal is that it is *multiplicative homomorph*, meaning that the multiplication of two ciphertexts gives a ciphertext that is an encryption of the multiplication of the plaintexts: \( E(a) \cdot E(b) = E(a \cdot b) \). The special case of multiplication with \( E(1) \) gives us a new ciphertext that decrypts to the same plaintext, but with different ‘exterior’. This property of a cryptosystem is called *self-reducibility*.

**Secure Computing** The notion of *Secure Computations* was first described by Yao[Yao82]. He describes it as the problem where \( m \) people wish to compute a function \( f \) over private inputs, where everybody is to learn the output but nobody should be able to learn anything of the other \( m-1 \) private inputs. Nowadays, the phrase *Multi-Party Computation* seems to be more popular. Another frequently used phrase is *Secure Function Evaluation*.

**Secure Circuit Evaluation** *Secure Function Evaluation* means the evaluation of secure functions: functions that are garbled. The general idea is that Alice has a private function, and Bob has private input. Bob wishes to compute Alice’s function on his private data, but doesn’t want to disclose this data to Alice. Alice on the other hand does not want Bob to know the internals of her function.

This idea can be generalised to *Secure Circuit Evaluation*: a group of participants have a boolean circuit which they want to evaluate on their private input. Since virtually any function can be written as a boolean circuit (remember that this is what computers do: evaluation of boolean operations), Secure Circuit Evaluation gives us a very powerful mechanism to do secure computations.

**Verifiable Secret Sharing** Another solution to the problem of Secure Computing is the sharing of all secrets between all participants. Every participant receives shares of the private inputs of all other participants, and he participates in a protocol where the participants jointly evaluate a publicly known boolean circuit. While some of the steps in the circuit can be done locally by participants, others may require quite some communication. After circuit-evaluation, every participant possesses a share of the result, which must be combined to reveal the result to all.

With *Verifiable Secret Sharing*, all participants can verify that they received a valid share of the secret, and also that all computations done by others were correct.

**Homomorphic Encryption** Another approach to publishing private information, is to publish it in encrypted form. If we want to do computations on encrypted values, we need an encryption algorithm with so-called *homomorphic* properties. This means that two (or more) ciphertexts can be combined, where the result is the encryption of some arithmetic function of the corresponding
plaintexts. Both the RSA and the ElGamal encryption algorithms are multiplicative homomorphic, which means that ciphertexts can be combined to obtain an encryption of the multiplication of the plaintexts. In the case of these two algorithms the combination is the multiplication of the ciphertexts.

**Group Cryptography** In conventional asymmetric cryptography, there is one public key to encrypt values, and one private key to decrypt them. Some applications however require that keys are distributed between parties, where only a subset of $m$ out of $n$ parties can decrypt values. This is known as Group Cryptography. In Group Cryptography there are two cases; the first is where a secret can only be decrypted when all the parties work together ($m = n$); this is known as a Veto Protocol. The alternative is a Threshold Protocol, where any subset of $m \leq n$ parties can decrypt secrets. A very well known threshold protocol is that of Shamir[Sha79]

**Oblivious Transfer** Some protocols require a recipient Bob to receive information from Alice without Alice knowing which information it was exactly. This procedure is called Oblivious Transfer, or OT. Most famous is one-out-of-two OT, where Bob receives exactly one of two of Alice’s secret bits, without Alice knowing which one Bob received.

**Secure Computation with Homomorphic Encryption** Instead of giving all participants shares of all secrets, it is also possible to encrypt all secrets using group cryptography. If the encryption algorithm has homomorphic properties, these secrets can be combined in a way to allow for secure computations. After all computations, the group members cooperate to decrypt the result.

## 1.3 Security

When analysing cryptographic protocols, the most important issue is that of the security of a protocol. But what exactly does the word security mean? In 1992, Micali and Rogaway noted [MR92] that, while secure protocols had been around for almost a decade, there still didn’t exist a clear definition of this phrase. The work of Micali and Rogaway is centred around the notion of an Ideal Process, where a protocol is compared to execution in an ideal world, where a Trusted Third Party does all computations. A protocol is considered secure if it is as secure as the ideal process.

Even in a perfectly secure evaluation, there may still be some cheating. Although an adversary may not be able to interfere with the computation, he can still send incorrect messages to the Trusted Third Party in order to obstruct the computation. This is however a problem that cannot be solved in the Ideal Process setting, therefore unsolvable by any protocol. Only when knowledge about a participant’s secret is known, others can verify whether or not the participant is lying about his input.
1.3. Security

1.3.1 Security Notions

In this section we discuss all the definitions regarding security properties of a protocol. Using this strict division for security proofs is opposed by Micali and Rogaway [MR92], but we feel it is necessary to explain which notions need to be taken into account when designing a secure protocol.

Security Model

In cryptography, two security models are known. First there is the notion of Information-Theoretic security. This means that for all combinations of a plaintext and a ciphertext, there is a key that could have been used to generate the ciphertext. This means the ciphertext does not disclose any information about the plaintext. To an adversary, all plaintexts are possible given a ciphertext. The One-Time Pad system has this property: it consists of applying a XOR to the plaintext with a random key of the same length as the plaintext.

While Information-Theoretic security sounds great, the fact that keys need to be as long as the plaintext poses a serious problem for practical applications. Therefore, the notion of Cryptographic security is used: cracking the key might be theoretically possible, but doing so would be infeasible considering the amount of work involved. Some cryptographic algorithms rely on structure to obtain this property (consider for example the symmetric protocols DES and AES/Rijndael), others rely on certain mathematical assumptions (like the assumption that factoring certain large composite numbers with large prime divisors is hard; the basis of RSA).

When dealing with an adversary that has unbounded computing resources, only protocols that are secure in the Information-Theoretic sense are secure. Protocols that have Cryptographic security are only secure against adversaries with polynomial-bounded computing resources, and only if the relevant computational assumption holds.

When applied to secure protocols, the two phrases tend to have a slightly different meaning. When spoken of the Cryptographic Model, it is assumed that all players have access to private communication between two participants, but they are unable to read it since a cryptographic protocol has been used. The phrase Information-Theoretic Model (also called Secure Channels) means that communication lines between any pair of players are guaranteed secure against an adversary with unbounded computing powers.

Privacy

One very important notion in secure protocols is that private information has to be kept private. This means that if a user adds his private input into the protocol, this information will not be disclosed to anybody.

There is a caveat here, because the output of a protocol may say something about the input. Take for example a protocol where two millionaires wish to calculate the sum of their wealth without disclosing their wealth to each-other. When they succeed in calculating the sum, they can just subtract their privately known wealth from it, and recover the private input of the other participant.
When applying a protocol to a specific problem, one should take into account that some information about private inputs can be revealed by the output of the algorithm.

**Independence of Inputs**

It must not be possible for an adversary to send input that compromises the input of honest participants. The difference with the previous notion is that the protocol needs to be resilient against faulty input. Not only should the protocol ensure privacy when all participants are honest, this should still be guaranteed when an adversary sends faulty input.

**Correctness**

When output is sent to users, it should be correct. This means that if some participant cheats during joint computations, this must be detected. An adversary may not be able to disrupt a protocol more than by lying about his initial input.

**Fairness**

When corrupted parties can only receive their output when honest parties receive their output, a protocol is called *fair*; a violator will not learn the output before an honest player. Galil, Haber and Yung defined this property as *synchronicity*, and showed in [GHY88] how to extend any multi-party protocol to be synchronous.

### 1.3.2 Adversaries

In cryptographic publications, a lot of effort is spent on proving a protocol secure. But secure against what? To be able to create a proof, we need to know the capabilities of the *Adversary*. We assume that this adversary *corrupts* a number of $t$ out of $n$ parties. This means that the adversary has complete control over these participants.

While this definition seems overly strong (in real life, you usually deal with curious people that do not collaborate but try to cheat alone), it has the advantage of being very general. When looking at the Internet, we realise that these kinds of adversaries do exist in real life. Think of hackers who have compromised several machines, or people who are snooping on network traffic.

**Active vs. Passive**

The weakest kind of adversary is called *passive*, he does see all data flowing to and from his $t$ corrupted parties, and he also has knowledge of all secret information of his corrupted parties. These corrupted parties do however not deviate from the protocol. An *active* adversary can also modify data, he can bring bogus information into the protocol.

The passive adversary can be compared to somebody who is listening for all network traffic, but who cannot inject any. An active adversary is doing a *man
in the middle-attack, where he changes all information flowing to and from the compromised party.

**Static vs. Dynamic**

Furthermore, there is the distinction between *static* (non-adaptive) and *dynamic* (adaptive) adversaries. For a non-adaptive adversary, the list of corrupted parties remains static over time. An adaptive adversary may choose to corrupt new parties during the execution of the protocol.
2. Problem Description

The concrete problem we try to solve is the following: how to create a distribution $\pi$ of ballots for $n$ people where the following requirements are satisfied:

- The distribution is a random permutation, with the extra property that there are no fixed points ($\pi(i) \neq i$). A permutation with this property is called a *derangement*.

- No participant $i$ may learn any other information than $\pi(i)$, he may not be able to influence the permutation, or gain knowledge that others do not possess (besides $\pi(i)$)

- No participant may be able to obstruct the generation of the permutation without being caught.

When comparing this list of requirements to the real-world scenario, we can see some resemblance:

- When users shuffle the ballots with the names, they create a random permutation.

- Users may only open the ballot they receive.

- Users may not add/remove ballots during the shuffle.

In the real world scenario we can trust on one very useful property of white paper: two individual folded ballots have the same exterior and are therefore indistinguishable. This property is lost when translating the protocol to a digital world, so this will be our main problem to fix.

Furthermore we have investigated whether or not we can improve on the real world protocol. In a real-world scenario we trust that people come forward with fixed points. We show that we can improve upon this using cryptographic protocols.
3. Previous Work

In this chapter we describe several protocols that generate a valid permutation, but that do not satisfy all of our requirements.

3.1 Trusted Third Party

The easiest way of dealing with the Secret Santa problem is to use a Trusted Third Party to generate the permutation. Third parties that do exactly this already exist for this very problem, one in form of a script on a website[SL]. Unfortunately, we need to trust the webmaster of given site: if one of the participants would collude with the webmaster then he would know the distribution, which is exactly what we are trying to prevent.

If nobody colludes with the Trusted Third Party, then this scheme is perfect; nobody sees any information besides the ballot he receives.

3.2 Tel’s Algorithm

In his book [Tel02], Tel proposes an algorithm for Secret Santa using RSA encryption. This algorithm consists of the following steps:

1. A dealer is chosen, he creates all ballots.
2. The dealer uses his public RSA key and encrypts all ballots which are extended with a random value, similar to the PKCS #1 standard.
3. The dealer generates a permutation of the ballots and publishes it.
4. The dealer sends the ballots to the first participant, who takes out ballot and passes the remaining ballots to the following participant. At the end of the round, the last ballot is given to the dealer.
5. Everybody starts a protocol with the dealer to blindly open the ballot he received.
6. If anybody sees his name on the ballot, he publishes it to all other participants, so they can all verify that the encrypted version was published during step 3. The entire protocol will be rerun from step 2.

While this protocol does generate a valid distribution of the ballots, several problems are present which are also mentioned by the author[Tel02]:

- We need to trust the dealer to generate valid ballots. It is not possible for the dealer to show that ballots are valid since that would allow participants to recognise individual ballots.
• A participant is not enforced to report the fact that the ballot he received contains his name. The possibility of creating a valid permutation is completely based on the sincerity of the participants.

• A participant could let the dealer blindly decrypt another ballot than the one he received, to gain more information of the permutation. Of course he wouldn’t be able to open up his own ballot anymore, assuming that he isn’t colluding with the dealer.

Nevertheless, this is a very interesting example protocol, since it shows that many problems have to be overcome when we are moving from blank ballots in the real world to digital information.
4. Secure Computations

In this chapter, we give a summary of secure protocols that have been developed over the last two decades. We will not go into the specific details of every protocol, but we will give a functional overview instead.

Too many protocols have been developed to be discussed here, we focus on those who were either important in the development of secure protocols, or those who will be useful for our specific problem.

Currently, there are three major areas in which research on secure protocols has been done. These three (Garbled Circuits, Verifiable Secret Sharing and Multi-party Computations with Homomorphic Encryption) will be discussed in the following section.

4.1 Existing Protocols

4.1.1 Garbled Circuits

Yao, 1982

In 1982, Yao was the first to describe the notion of Secure Computations [Yao82]. He described the problem of two millionaires, trying to establish which one is richer without disclosing their wealth. Yao described three possible solutions to this problem, followed by a generic solution using a garbled circuit. Micali and Rogaway noted[MR92] that Yao’s definition of security was rather weak, the protocol could only defeat adversaries with very limited capabilities.

Goldreich, Micali and Wigderson, 1987

Goldreich, Micali and Wigderson [GMW87] were the first to describe a general approach for secure computation. They base their idea on Combined Oblivious Transfer, where every gate evaluation in a boolean circuit requires an OT-session between the participants.

To extend this idea to a system with multiple parties, they introduce the notion of Secret Sharing, where all bits of the individual secrets are shared between participants.

The revolutionary result of the paper was that they provided a proof that every game can be played, when more than half of the participants are honest. They note that many real life situations can be described as games.

One big drawback of the protocol is that an active adversary may at most corrupt $\frac{n}{2}$ parties, otherwise all private inputs are compromised. Furthermore, the huge amount of communication required makes this protocol impractical.
In his PhD thesis [Rog91], Rogaway presents more efficient approach to the idea formed by Goldreich et al. He reuses the idea of garbled circuits, but his protocol needs less communication for gate evaluations.

The latest work that continues on the idea of garbled circuits is that of Cachin, Camenisch, Kilian and Müller [CCKM00].

### 4.1.2 Verifiable Secret Sharing

With Verifiable Secret Sharing, as explained in section 1.2, the secret of one participant is shared between all participants in the protocol. The **Verifiable** part means that the honesty of all parties can be verified during the execution of the protocol. Participants perform calculations with their shares of the secrets, and obtain a share of the result.

Where Goldreich et al. constructed a garbled circuit to evaluate function on private data, Rabin and Ben-Or [RBO89] decided to use a publicly known circuit, but garble the inputs. They use Verifiable Secret Sharing to distribute secrets of all parties. During computations, participants use their shares of the secrets to create a share of the result.

The result of their work is a protocol where addition of two shares and multiplication of a share by a constant are both possible without communication: any participant can do them locally. For multiplication of two secrets however, a distributed protocol is needed, where participants gain a share of the multiplication after a substantial amount of communication rounds.

For sharing secrets, they choose Shamir’s protocol [Sha79], where a polynomial is created with \( \frac{n}{2} \) zero-points. This means that any coalition of \( \frac{n}{2} + 1 \) participants can read the secret, meaning that this protocol is only applicable with groups of at least \( \frac{n}{2} \) honest players.

The protocol of Rabin and Ben-Or was one of the first that could defeat an active adversary, as long as he corrupted less than half of the participants.

The latest contribution to the field of secure computing with Verifiable Secret Sharing is that of Cramer, Damgård and Maurer [CDM00]. They show that from any **Linear Secret Sharing Scheme** a VSS scheme can be built, which is secure for adversary structures that are more general than the known threshold adversaries of \( n/2 \) for the passive and \( n/3 \) for the active case.
4.1.3 Joint Homomorphic Encryption

Franklin and Haber, 1996

Franklin and Haber show [FH96] how to create a protocol for secure computation based on Joint Encryption. They generate a composite number $N$ of which the two prime factors are unknown, and encrypt bits as a 4-tuple using ElGamal-like encryption. Decryption is only possible if all parties cooperate, encryption can be done by all. Since ElGamal is probabilistic, there are many different ways of encrypting a single bit.

They show that evaluating the NOT on a bit can be done locally, an AND gate requires communication between all parties. This means that the complexity of evaluating a circuit is linear in the number of participants $n$ and the number of AND gates $D$ in the circuit: $O(nD)$.

The protocol of Franklin and Haber suffers from two major problems. First of all, it is only secure against passive adversaries. Secondly, they require the RSA modulus $N$ to be generated without knowledge of its prime factors. Franklin and Haber suggest that this should be done by a central server, which means that a trusted third party is still needed. Nowadays, practical solutions exist [DK01] for a group to jointly generate an RSA modulus.

Sander, Young and Yung, 1999

In 1999, Sander, Young and Yung [SYY99] created an algorithm that resembles that of Franklin and Haber, since they reuse the idea of using a tuple of encrypted values to represent a single bit. They define both the OR and the NOT in a way that allows for local computations, meaning that a boolean circuit can entirely be calculated locally. Unfortunately, the length of the tuple is doubled in every step of the circuit, meaning that a bit in level $l$ of the circuit has a length of $2^l$.

The algorithm is proven using induction; if we have a bit in level $l$, we can create a negation of this bit in level $l + 1$ with double length. Likewise, if we have two bits in level $l$, we can create an OR of those two in level $l + 1$, also with length double of the concatenation of the bits.

It is obvious that, while this protocol allows local operations, it is only practical for circuits with limited depth, so called $NC^1$ circuits. If the depth of the circuit is logarithmic, then this protocol has polynomial complexity.

Cramer, Damgård and Nielsen, 2001

A big drawback of the protocol of Franklin and Haber [FH96] is that it only works when the adversary is passive. Cramer et al. describe a method [CDN01] that bears great similarities with this protocol, but is resilient against an active adversary that corrupts at most $n/2$ parties.

The protocol is very generic, it does not rely on a specific encryption algorithm. Any suitable algorithm that allows for Threshold Homomorphic Encryption can be used; they show the application of two; the algorithm proposed by Franklin and Haber in [FH96], and Paillier’s cryptosystem [Pai99].
4.2 Application to Secret Santa

With secure computations, we can achieve a solution to the Secret Santa problem. We need to create an algorithm that can be securely evaluated using one of the protocols described in this chapter. Since virtually any function can be written as a boolean circuit, this will not prove to be a difficult task.

One thing that needs special attention is the complexity of the protocol. All described protocols have either a very high round complexity, or a very high message complexity. If we perform Secret Santa with a large group, we need to be sure that the computation is implementable. We would prefer a low round complexity to a low message complexity, since it would allow for a bit more asynchronous implementation.

4.3 Solution

To develop a solution to the Secret Santa problem using Secure Computations, we need to create a boolean circuit that can shuffle the names. This circuit can then be evaluated with any one of the mentioned protocols. We will do this step by step: first we create an algorithm in very high-level pseudocode. We then transform this algorithm to one containing boolean and arithmetic expressions. This algorithm will then be transformed into a boolean circuit.

Recall that we have to generate a permutation of ballots, where no ballot may be located on its original position. The following algorithm creates a derangement by first generating a permutation, and subsequently testing this derangement for fixed points. Approximately one in $e$ permutations is a derangement\[Knu97\], so we will approximately need $e$ runs of the algorithm to succeed.

Algorithm 1 Calculate a derangement of $n$ values
1: Initialise set $S$ with all numbers, copy this set to $B$
2: Initialise set $R$ as empty
3: while $S \neq \emptyset$ do
4: choose random $p$ between 0 and #$S$
5: put $S[p]$ in $R$
6: put $S[\#S]$ in $S[p]$
7: remove last element from $S$
8: end while
9: Check for all elements $R[i]$ in $R$ that $R[i] \neq B[i]$
10: If failed, return to 1

The next step in our approach is to transform this algorithm to one containing only boolean and arithmetic operations. The result of this transformation is the algorithm 2. Some notes about this algorithm:

- The $bs(i)$ function creates a bit representation for a given integer $i$. 

4.3. Solution

- In line `derang:infor`, bitstrings $S[l]$ and $S[i]$ are being multiplied by the result of an equality operator. This notation should be read as a bitwise AND of the bitstrings with either 1 in case the comparison evaluates to TRUE, 0 otherwise.

Algorithm 2 Calculate a derangement of $n$ values

1: for $i = 1$ to $n$ do
2:     $S[i] = bs(i)$
3: end for
4: $R = \{\}$
5: $B = S$
6: for $i = n$ downto 1 do
7:     $i = \#S$
8:     choose random $p \in \{1...i\}$
9:     $j = \sum_{k=1}^{i} S[k] \times (p = k)$
10:    for $l = 1$ to $i$ do
11:        $S[l] = S[l] \times (p \neq l) + S[i] \times (p = l)$
12:    end for
13:    $R[n - i + 1] = j$
14:    $S = \{S[1]...S[i - 1]\}$
15: end for
16: $c = 0$
17: for $i = 1$ to $n$ do
18:     $c = c \text{ OR } (B[i] = R[i])$
19: end for
20: if $C[i] = \text{ a list of 0’s}$ then
21:     accept $R$ as a valid derangement
22: else
23:     return to line 1 and start over
24: end if

The last step to this solution is to rewrite all arithmetic operations in the algorithm to a boolean equivalent. We will demonstrate this procedure by example: we will generate a boolean circuit for line 11 of algorithm 2.

\[ S[l] = S[l] \times (p \neq l) + S[i] \times (p = l) \]

This line is the equivalent to line 6 from algorithm 1, it is part of a loop over $l$ from 1 to $i$, where $i$ is upperbound of the array $S$. The value $p$ is a randomly chosen integer between 1 and $i$, but since it is garbled the value is unknown.

The idea is quite simple: only one value of $l$ will equal $p$, so this line of code will put the last element of the array at position $S[p]$. The equality test $(p = l)$ translates to evaluation of the NOT of the XOR of $p$ and $l$, followed by an AND of all the bits of this result. The test $(p \neq l)$ can be evaluated in a similar way. What remains is the multiplication of this result with the bitstrings $S[i]$ and
$S[l]$ respectively, which consists of a bitwise AND. The addition is simply an OR of the two results.

When applying this methodology go algorithm 2, we get something with only loops and binary operations: algorithm 3. While this isn’t technically a boolean circuit, it does allow to be evaluated using a secure computation method. All boolean values in this algorithm must be garbled, encrypted or shared; boolean operations between them can be implemented using any of the described secure computation protocols.

A next step could be to rewrite this algorithm to one that does not contain variables or loops. This circuit would however be extremely large, while there are no apparent advantages to that approach. All loops in the given algorithm can be *inlined*: copying the loop body multiple times to allow for sequential evaluation.

### 4.3.1 Generation of Random Bits

There is one tricky part left: the generation of a random integer in a bit representation. For this we propose to use a *Pseudo-Random Number Generator* (PRNG), since these can be entirely written as boolean circuits also. This PRNG circuit can then be evaluated using secure computation, and be initialised with a garbled seed.

**TODO:** add the compare-two-bitstrings research

### 4.3.2 Complexity

Analysing the complexity of algorithm 3 is very straightforward. With $n$ players, the size of the bitstrings is $\log(n)$. The most expensive loops are the for loops on line 12-19 and 21-27. Both these loop from 1 to $i$, where $i$ is decreasing from $n$ to 1 in the outer while loop. This means the body code of these two for loops will be executed $n + (n - 1) + ... + 1$ times, this equals $\frac{n^2}{2}$.

The body code of mentioned for-loops consists of a number of bit-operations linear in the length of the bitstrings: $O(\log(n))$. Combining this with the $O(n^2)$ complexity we just calculated, we arrive at a complexity of $O(n^2\log(n))$ for the main while loop.

The calculation of fixed points (lines 30 and further) takes $O(n\log(n))$ time, but we also have the risk of needing to restart the algorithm. Fortunately, the chance that a given random permutation has any fixed points is $1/e$, as proven by Knuth\[Kn97\]. This means we need (by approximation) a constant number of rounds, the total complexity of this circuit thus equals $O(n^2\log(n))$ bit-operations.

### 4.3.3 Implementation

When we wish to evaluate the boolean circuit, we must choose which secure computation protocol to use. This is a question of practical application, every protocol discussed has its own advantages and disadvantages. Some have very low round complexity, others have very low communication complexity.
Algorithm 3 Calculate a derangement of \( n \) values

\( n \), number of participants.

\( \text{bitsize} \), length of the bitstrings, \( \lceil \log_2 n \rceil \).

\( S[i][j] \), \( j \)-th bit of the \( i \)-th element of set \( S \).

\( \text{bs}(i) \), gives a bit representation of size \( \text{bitsize} \) for integer \( i \).

1: for \( i = 1 \) to \( n \) do
2: \( S_i = \text{bs}(i) \);
3: \( R = \{ \} \);
4: \( B = S \);
5: while \( S \neq \{ \} \) do
6: \( \text{size} = \# S \);
7: choose random \( p \in \{1...i\} \);
8: // Begin of line 9
9: for \( k = 1 \) to \( \text{bitsize} \) do
10: \( \text{value}_k = 0 \)
11: \( j = \text{bs}(0) \)
12: for \( \text{index} = 1 \) to \( i \) do
13: \( k = \text{bs}(\text{index}) \)
14: \( eq = 1 \)
15: for \( q = 1 \) to \( \text{bitsize} \) do
16: \( eq = eq \text{ AND } (\text{NOT } (p_q \text{ XOR } k_q)) \)
17: // The value of \( eq \) now equals the outcome of \( p == k \)
18: for \( q = 1 \) to \( \text{bitsize} \) do
19: \( j_q = j_q \text{ OR } (S[\text{index}]_q \text{ AND } eq) \) // Summation and multiplication
20: // End of line 9
21: for \( \text{index} = 1 \) to \( i \) do
22: \( l = \text{bs}(\text{index}) \)
23: \( eq = 1 \)
24: for \( q = 1 \) to \( \text{bitsize} \) do
25: \( eq = eq \text{ AND } (\text{NOT } (p_q \text{ XOR } l_q)) \)
26: for \( q = 1 \) to \( \text{bitsize} \) do
27: \( S[\text{index}]_q = (S[\text{index}]_q \text{ AND } eq) \text{ OR } (S[\text{index}]_q \text{ AND } (\text{NOT } eq)) \)
28: \( R = R \cup \{j\} \);
29: \( S = \{S[1]...S[i-1]\} \);
30: for \( i = 1 \) to \( n \) do
31: for \( q = 1 \) to \( \text{bitsize} \) do
32: \( C[i]_q = B[i]_q \text{ AND } R[i]_q \)
33: \( \text{fixedpoints} = 1 \)
34: for \( i = 1 \) to \( n \) do
35: for \( q = 1 \) to \( \text{bitsize} \) do
36: \( \text{fixedpoints} = \text{fixedpoints} \text{ AND } C[i]_q \)
37: // The following needs to be tested in public
38: if \( \text{fixedpoints} = 0 \) then
39: accept \( R \) as a valid derangement
40: else
41: return to line 1 and start over
Some instances of Secret Santa could require low round complexity, for instance when participants live in different timezones and are never online at the same time. Others may require low communication complexity because of bandwidth limitations.
5. Mix-Networks

5.1 Existing Protocols

5.1.1 Encryption Chains

The notion of Mix-Networks was first used by Chaum [Cha81], he describes a method to allow emails to be sent anonymously. Key to his idea are Mixes, computers that decrypt an encrypted message that was sent to them, and send it to another mix.

To send a message anonymously, one can use a chain of mixes. The message has to be encrypted with the public key of the last mix, then with the last but one, etc: \( E(m) = E_1(E_2(...E_n(m))) \). This message is then sent to the first mix, who decrypts it and sends the result to the next one. When the last mix decrypts the message, there is no way for it to find out where it originated from.

One major application of mix networks is in electronic voting. Consider a scenario where \( n \) people vote, but they do not want anybody to know what they voted for. When using a mix-network, they can encrypt their vote and submit it to a mix network. That way nobody can find out where the message originated from.

A big drawback of Chaum’s system is that it requires additional encryption for every mix that is used; this means that messages get bigger when more mix servers are used. Secondly, if a deterministic encryption algorithm like RSA is used (as proposed by Chaum), one could re-encrypt the final result and match it to the submitted votes. This would disclose the identity of the voter and the entire permutation.

Chaum’s protocol was broken in 1989 by Andreas and Birgit Pfitzmann in [PP89].

5.1.2 Re-encryption

Because Chaum’s method results in a size-blowup of the messages (every encryption layer adds to its size), several algorithms were developed that use re-encryption of messages. This means that instead of removing one decryption layer from a chain, a mix will do something to the ciphertext that will change the exterior but not the encapsulated plaintext. If an encryption algorithm has this property, it is called to be self-reducible. It is obvious that this re-encryption must be verifiable, otherwise a mix could remove messages and add new ones without anybody noticing.

The first algorithm that had these properties was that of Park, Itoh and Kurosawa [PIK94]. This protocol is locally verifiable, allowing a voter to verify...
that his vote has been counted. Unfortunately the protocol was shown to be insecure by Pfitzmann\cite{Pfi94}, but the vulnerability was later fixed by Ogata et. al.\cite{OKST97}.

Sako and Kilian presented a protocol\cite{SK95} that is universally verifiable, meaning that anybody can verify that all votes were counted correctly. Especially for large scale elections this is a great improvement, since in these elections it is impractical to ask all voters to verify the outcome. However, Michels and Horster showed\cite{MH96} that an adversary must not collude with any mix and that at least two mixes need to be honest, in order to guarantee the privacy of votes.

The problem with mentioned protocols is the verification step, this step tends to be very expensive since it usually relies on cut-and-choose zero-knowledge proofs (or others that have equivalent complexity). Some algorithms were proposed that make use of more efficient zero-knowledge proofs\cite{Abe99, AH01, JJ99}, but they still have complexity larger than linear in the number of participants. When applying a Mix-Network to large scale elections, it is obvious that the procedure should be efficient. Recent research\cite{FS01, Nef01} resulted in algorithms with very low complexity, the number of modular exponentiations is linear in the number of votes and mixes.

5.2 Application to Secret Santa

It is obvious that a Mix-Network can be applied for the Secret Santa problem. Instead of votes, ballots contain names of the participants, or maybe wish lists for gifts. When we choose to make every participant a mix, we are assured that at least one mix is honest if at least one participant is honest. After verification that all participants shuffled correctly, they use a group decryption protocol to decrypt all ballots, slightly altered to allow only the intended recipient of the ballot to see the result.

There is one problem with this protocol that also exists in real life. What is somebody received a ballot containing his own name? In real life we ask this person to come forward with this so that a new shuffle can be made. Can we maybe do better than this when using cryptographic protocols? We give away that this is indeed possible, the detailed solution can be found in the next section.

5.3 Solution

The next subsections will describe all steps necessary to solve the problem using the Mix-Network of Furukawa and Sako\cite{FS01} as it’s basis. This Mix-Network mixes a list of ElGamal encrypted texts. The choice for this Mix-Network was one based on suggestions of experts, since security proofs are very well reviewed and it allows for very easy implementation. Another suitable protocol would have been the one of Neff\cite{Nef01}, but it proved to be so hard to understand that an implementation would have fallen out of the scope of this research.
5.3. Solution

Initialisation

During initialisation, all users need to participate in a distributed protocol to create an ElGamal keypair that will be used during the mix. Here we have a choice between a veto- and a threshold scheme. A threshold scheme is more powerful (if participants cheat, the protocol can still terminate correctly), but we there is no use for such a complicated scheme for Secret Santa. With other, more serious applications of this research it might be needed to use a threshold scheme, but in our setting we only have the requirement that cheating players are caught. A veto scheme suffices here.

Generation of a group-shared public key using a veto scheme for ElGamal is very simple. All participants generate their own keypair and publish the public key part. The product of those public keys (modulo $p$) is the group public key.

After generation of the group key, all the names of the users can be encrypted. One can choose between two settings here. When encrypting the publicly known names of the participants, it become impossible to lie about one’s identity. It is also possible to allow people to submit their own private data to the server (for example a wish list). In that case one can lie about his identity, but in our setting that would mean he would not receive a present himself. Since we allow cheating which only puts the cheating player at a disadvantage (see chapter 2, this is no security problem.

Mixing

The second step in the protocol is the mixing of the encrypted inputs. This consists of the following parts for every user:

- The user receives the previous data from the bulletin board.
- The user creates a ‘Permutation Matrix’: a matrix with dimensions $(n \times n)$, where every row and every column has exactly one ‘1’ and $n-1$ ’0’$s$. When multiplying this matrix with the vector of original ballots, one gets a permutated vector.
- The user creates a list of random integers and uses them to re-randomise the ElGamal encrypted ballots.
- Zero knowledge proofs to prove correct use of the randomisation-integers and the permutation-matrix are placed on the bulletin board, together with the resulting mixed data.

We note that while the zero knowledge proofs are interactive (they require a challenge sent by the group, followed by a response by the user that is proving the correctness of his shuffle), they can be easily turned into non-interactive using the Fiat-Shamir heuristic $[FS87]$.

Furthermore, it is not important to know the order in which the participants mix, it can be implemented using a first-come-first-serve mechanism, as long as every participant acts as a mix server once.
Plaintext Equality Tests

After mixing, we need to make sure that nobody receives his own wishlist or name: we need to check for fixed points in the generated permutation. To do this without the need of openly decrypting ciphertexts, we need to perform a series of Plaintext Equality Tests. These consist of a protocol to compare two ciphertexts without actually decrypting them both.

Creating a Plaintext Equality Test for the ElGamal cryptosystem seems to be quite easy at first glance. Since ElGamal is multiplicative homomorph, the division of two ciphertexts is the encrypted version of the division of the original plaintexts. Using this property, a PET for ElGamal seems to be nothing more than decryption of the division of two ciphertexts. If this yields 1, the two plaintexts are equal.

Unfortunately it is not as easy as this. If the decryption of the devision does not yield 1, but another number, this number will reveal some information about the original plaintexts. Therefore, we must use a trick to blind the value of the division before decrypting it. By raising the quotient to an unknown random exponent $u$, any other value than 1 will be randomised.

A formal description of the Plaintext Equality Test described here can be found in a paper from Jakobsson and Juels[JJ00]. While the mix network proposed in this paper has been broken, the Plaintext Equality Test proof still holds.

The PET described by Jakobsson and Juels consists of the following steps:

- The two ciphertexts $(\alpha, \beta)$ and $(\alpha', \beta')$ are divided into $(\epsilon, \zeta) = (\frac{\alpha}{\alpha'}, \frac{\beta}{\beta'})$.
- Each player selects a random $z_i \in \mathbb{Z}_q$.
- Each player computes $(\epsilon_i, \zeta_i) = (\epsilon^{z_i}, \zeta^{z_i})$.
- Each player computes the commitment $C_i = h(\epsilon_i, \zeta_i)$ and broadcasts it to the rest of the group.
- After all players sent out their commitments, all players send out $(\epsilon_i, \zeta_i)$.
- All players verify the commitments, and then jointly decrypt $(\gamma, \delta) = (\prod_{i=1}^n \epsilon_i, \prod_{i=1}^n \zeta_i)$.
- If the resulting plaintext is 1, then the players conclude that $(\alpha, \beta) \equiv (\alpha', \beta')$. Otherwise they conclude that $(\alpha, \beta) \not\equiv (\alpha', \beta')$.

The protocol of Jakobsson and Juels originally used very extensive zero-knowledge proofs to allow players to prove correct exponentiation. They do however recommend using the hash function $h$ under the random oracle assumption, to greatly reduce message and computation complexity.
5.3. Solution

Group Decryption

If all PET’s succeeded, the participants will need to decrypt their message, in a way that only the supposed recipient of the ballot can read it. When using a veto scheme for ElGamal, this is very simple since it only requires all other participants to send a partial decryption to the participant. The recipient combines these partial decryption with his own partial decryption, and can then reconstruct the plaintext.

If the secret key is shared in a threshold scheme, it becomes more difficult since the information placed on the bulletin board will allow anybody to generate a decryption. One possibility to ensure that only one participant can read the plaintext, is that partial decryptions are instantly re-encrypted using the public key of the intended recipient. This way only this recipient can combine all shares. Another method is to allow the recipient to raise the ciphertext to a random power. This new ciphertext will then be decrypted, and the recipient uses his knowledge of the exponent to recover the plaintext. This of course requires a proof that the participant knows the exponent, and that he did not use any of the other ciphertexts instead of the one intended for him.

When looking at the friendly nature of the Secret Santa setting, it might not be a problem at all to use a veto scheme. This will make implementation a lot easier.

5.3.1 Security

For security of the protocol outlined before, we must not only trust on the security of the different components, but also that the composition itself is secure. While one can easily check the security proofs of the components, proving a composition secure is very hard. Recent research [Can00] in security of multi-party protocols shows that there are frameworks in which composition can be proven to be secure.

Correctness

Correctness of our protocol follows from correctness of all components: Group ElGamal encryption, Verifiable Mixing and the Plaintext Equality Test. If all participants follow the protocol, we create a random permutation of the encrypted inputs with no fixed points.

Public Verifiability

For both veto- and threshold ElGamal group encryption, there exist variants that are public verifiable. Both the Plaintext Equality Test and the Verifiable Mix come with zero knowledge proofs that allow for public verifiability.

Privacy

Privacy is guaranteed by using ElGamal encryption, which is considered to be secure if Diffie-Hellman decision is hard.
5.3.2 Complexity

The complexity of the mix network depends on the complexity of the components used. We can reasonably assume that the number of participants is smaller than the prime $p$ used for ElGamal encryptions. Therefore we can conclude that the size of the ballots is constant and not dependent on the number of participants. Since the most expensive operation in our protocol is the modular exponentiation, we will calculate complexity in the number of modular exponentiations.

Encryption of the names of the users consists of two modular exponentiations. When using either the Furukawa-Sako [FS01] or Neff [Nef01] for mixing, the number of modular exponentiations for mixing $n$ ciphertexts is linear with a small constant (respectively $18n + 18$ and $8k + 5$). The amount of modular exponentiations required to verify a proof is also linear in the number of ciphertexts.

The description of the Plaintext Equality Test we gave shows that every participant needs to apply 2 modular exponentiations. After revealing the exponentiated ciphertexts, they need to collaborate in decrypting them to test for equality. This joint decryption stage consists of 2 modular exponentiations for every participant. Verification of these partial decryptions takes the coordinator 4 modular exponentiations for every partial decryption. In case all participants wish to coordinate (and verify), they will need to do $6n$ modular exponentiations for every plaintext equality test.

After the Plaintext Equality Tests, everybody will need to decrypt their ballot, we just saw that this will cost all participants $6n$ modular exponentiations.

We can see that all steps in the protocol require a participant to do a number of modular exponentiations, linear in the number of participants, except for the mixing verification step. If a participant wishes to verify that a user mixed correctly, he will need to do a linear number of modular exponentiations. There are however $n$ proofs to check, so the total amount of work is quadratic in $n$. This results in a total cost for the protocol of $O(n^2)$ modular exponentiations for every participant.

5.3.3 Implementation

When implementing this solution to Secret Santa, we note that there is a heuristic that will greatly improve round complexity. When applying the Furukawa-Sako mixing network, participants should verify the proof of correct mixing before the next user mixes. This means that with $n$ participants, there are $n$ mixing rounds where communication is required with all users.

If there is no security impact when somebody mixing with faulty input, we may choose to verify all proofs after mixing stage. This means that somebody who cheated will still be caught, but his output was used during the rest of the mixing protocol. If this poses no threat, then this heuristic greatly improves round complexity.
6. Implementation

This chapter describes the implementation details regarding our mixing solution. It consists of requirements on our implementation, some design decisions we made, and the communication protocol used between the clients and the bulletin board.

While a good software design usually consists of a separate functional and technical design, this chapter features both these intertwined. Considering the fact that this document is not a design document but a thesis, we chose to combine the two designs to improve readability.

6.1 Libraries

To implement the Mix-Network described in chapter 5, we will need a library that allows for ElGamal Group encryption. Since no such libraries exist yet, we need to create our own. Furthermore, we need to implement the Furukawa-Sako mixing protocol and the Plaintext Equality Test.

6.2 Environment

Our implementation is based on a protocol for mixing[FS01] that makes use of a public bulletin board, which is authenticated, tamper-proof, and resistant to denial-of-service attacks. All calculations are done locally by the participants, this bulletin board is used only for communication between the users. These requirements are not mentioned explicitly in the description of the protocol, but other similar protocols have used this same kind of communication channel.

To mimic this behaviour, we create a database on a server with some very lightweight pages that allow the database to be queried. We will encrypt all communication between the bulletin board and the clients, but we stress that this part of our implementation is vulnerable to a man-in-the-middle attack. Solving this vulnerability requires separate communication channels which would make the protocol far less user-friendly. We therefore assume that a malicious user can read and inject messages all the time, except for the phase where the client receives the public key of the bulletin board.

6.2.1 Choice of programming language

So far, we have not spoken about the actual implementation in terms of programming languages and specific tools. In our opinion, it is very important that the software should be accessible to virtually anybody, we therefore choose to use the Java programming language for our client-side implementation.
When creating a client program, it would be preferable if the client could run the program without installation of any software. An applet would be a very efficient way of achieving this, but unfortunately this means that we are very limited in the operations we can execute. An applet lives in a so-called sand-box, a protected environment in which it is restricted to operations that cannot influence the computer it runs on. We however need to read and save local files, in order to allow the applet to store a keypair and other configuration settings. For this we need to escape the sand-box. While this is technically possible, we run into the problem that different browser vendors implement different ways of granting special privileges to applets.

The first browser that allowed applet execution (Netscape Navigator 4.0) had its own set of classes \texttt{(netscape.security.*)} that can be used for obtaining privileges. One does need to have access to these classes to be able to compile the applet (calling the API using reflection is prohibited). Furthermore, we will need to sign the resulting JAR-file with a valid certificate, one that can only be obtained from a commercial party like VeriSign or Thawte.

The Microsoft browsers that include a JVM (Microsoft Internet Explorer 4.0 and up) have their own API for obtaining privileges, the \texttt{(com.ms.security.*)} package. Apart from using this API, one also needs to use some Microsoft tools to create a 'Microsoft Cabinet File', which is then signed using a certificate that can be self-made. The HTML syntax to load a signed applet designed for Microsoft Internet Explorer is different than the syntax for a Netscape browser, but some javascript tricks can be used to allow both browsers to be supported. We note that the latest version of the Microsoft Operating System (Windows XP) does not include a Java Virtual Machine, it needs to be be downloaded separately.

While both described browsers differ in their security implementation, they are similar in the fact that they support an old version of Java. The new Java standard (Java 2, from the Java Software Development Kit 1.3 and up) contains some great enhancements, but they can only be used by browsers who have an up-to-date JVM. To circumvent this problem, Sun Microsystems released a 'Java Plug-in' for all major platforms, which allows Java 2 code to be executed. When using this plug-in, it is fairly easy to support multiple browsers at once since obtaining privileges is now standardised (it is as easy as signing the jar with a self signed certificate).

Sun’s latest technology is called 'Java Web Start', which allows complete Java programs to be initiated over the web. The latest JDK release (1.4.1) already contains Java Web Start, for those who have 1.3 installed there is a separate browser plug-in. Unfortunately, this technology has not found its way to many client systems, meaning that a client needs to install Java Web Start before he can run any code.

After evaluation of these facts, we choose to create a Java 1.3 applet, for which there is support for all major platforms (including virtually all flavours of Windows, Linux and other Unix variants).
6.2.2 Choice of server environment

Since we use Java for the client, it is logical to use Java for the bulletin board code too, we can then reuse all libraries written for the client. There are several mechanisms that can be used, ranging from creating your own HTTP server in Java to using Java Servlets or Java Server Pages. It is not terribly important which of these technology we choose, but since JSP allows for very fast debugging we chose this technology.

One very popular free server to run JSP pages is 'Tomcat' from the Apache Software Foundation. For storage of information, we use the popular MySQL database.

6.3 Design

As noted in the previous section, we use a client-server model for our implementation. We recall that the server only needs to be an authenticated bulletin board, all computation is done locally by the clients.

The following subsections describe which steps are taken by the clients and the bulletin board to execute the Secret Santa protocol. These steps include the messages sent between the bulletin board and clients. Messages originating from the clients have the following format:

```
johannes@verelst.net 32412123 newgroup test
```

The first string contains the user-id of the user sending the message. The second strings is the cookie that identifies this user. This cookie is currently being implemented as a random string issued by the bulletin board, but an signature based on knowledge of the private key of the user is also possible. After the cookie, the command follows, which may include command parameters. This message asks the bulletin board to create a new group "test", for which "johannesverelst.net" will be the owner.

A bulletin board response message has the following format:

```
newgroup test OK
```

It starts with the command that was issued, followed by return data. In case an error occurred, the command will be replaced by the string "error".

There are two communication modes between the bulletin board and client: encrypted and plain. When a party sends a plain message the message itself is prefixed with the string 1|. Encrypted messages start with 2|, followed with the encrypted message encoded using the base64 encoding standard.

During implementation of the protocol, we noticed that there are several problems regarding coordination. In all papers, there are no coordinators, all participants have the same role and should be treated as equals. This implies that a majority of the participants choose what happens. They all use the bulletin board to communicate and to form a majority.

Unfortunately there are some problems with this when implementing this in client software. Looking at the Plaintext Equality Test, our protocol requires
that the group decides whether or not the mixing should start over. We could implement this using some kind of voting system, but that would require a huge amount of work.

Another possibility is to let the bulletin board make decisions for the group, based on the input sent by the participants. For instance, when the participants send in their partial decryptions for the Plaintext Equality Test, we could let the bulletin board combine these and decide whether or not there are fixed points. Unfortunately this 'extended' bulletin board surpasses it’s original purpose a great deal. Instead of being a passive participant, the bulletin board would now have an active role and the players need to trust the bulletin board: it becomes a Trusted Third Party.

The difficulty during implementation is finding a compromise between a bulletin board that only relays messages, and a full-fledged Trusted Third Party that makes decisions. In this design, we will elaborate whenever our bulletin board surpasses it’s original purpose.

6.3.1 Initialisation

Whenever a client sends data to the bulletin board, it needs to be encrypted to maintain privacy. For this, the client must know the public key of the bulletin board. We will assume that this key is already available at the client, or has been sent to the client in a secure way.

6.3.2 Registration

When a user wishes to communicate with the bulletin board, communications needs to be secure in both directions. The user already knows the public key of the bulletin board from the initialisation stage, but the bulletin board needs to know the public key of the user in order to be able to encrypt the communication to him. The process in which a user submits his public key is called registration.

When a user wishes to be registered, he sends his email address to the bulletin board. The bulletin board will generate a random challenge and send this challenge to the provided address. We note here that this challenge can also be intercepted by a man in the middle. A proper system should use PGP (or something equivalent) encryption and signing to distribute the challenge, but this also means the user should be in possession of PGP tools and the public key of the sender.

While PGP has grown to have a big userbase in the technology-aware community, most normal users do not have PGP installed. Therefore we do not use encryption of emails in our code yet, but it can be easily extended to do so.

The following messages are sent in this stage:

C: johannes@verelst.net COOKIE requestaccount

... bulletin board sends an authentication code to the specified email address ...

S: requestaccount johannes@verelst.net
Communication from the client to the bulletin board is encrypted, communication from the bulletin board to the client is plain since the bulletin board has not yet received a public key from the user.

### 6.3.3 Submitting a Public Key

Before a user can execute any commands, he must submit his public key to the bulletin board. To be able to do this, the user must identify himself using the cookie sent to him by email.

The bulletin board will accept the public key, and send a message to the user requesting the user to send a reply. Only when the user successfully faces this challenge, the public key is accepted. The bulletin board sends the user a new cookie to use in further protocol rounds.

This interaction is necessary to let the client prove to the bulletin board that he knows the private key to the corresponding public key. A client cannot send just a random public key to the bulletin board, it must be a valid one.

Our protocol does not feature a login phase, since identity of the user has been proven by knowledge of the cookie.

```plaintext
C: johannes@verelst.net 14917128938 submit 1235..
S: challenge 248123748912
C: johannes@verelst.net acknowledge 248123748912
S: acknowledge ok
```

The first step consists of the user sending the cookie (in this case the string 14917128938) to the bulletin board, together with a public key. The bulletin board stores this public key internally, and generates a new cookie for the user. This cookie is then sent to the user in an encrypted message. When the user correctly acknowledges this cookie, the bulletin board is convinced that the user knows the private key of the submitted public key.

### 6.3.4 Forming a Group

One of the requirements for our bulletin board is that it can be used for several different Secret Santa sessions simultaneously. This means that the bulletin board needs to have a notion of *groups*, which allows participants to only see data relevant to their instance of Secret Santa.

When the first user of a group has logged in, he can create a group area in the bulletin board dedicated for his instance of Secret Santa. He submits the email addresses of all other participants, the bulletin board will then send email invitations to these users. When these users register with the bulletin board they will be asked whether or not they wish to join the group.

In case the user was already member of another group, he will not receive an email but he will directly see a popup asking him to join the new group also. Users can be member of an unlimited amount of groups.

```plaintext
C: johannes@verelst.net 248123748912 newgroup mygroup
S: newgroup OK
```
When a user starts his application, it checks whether there are any pending invitations. This goes as follows:

C: johannes@verelst.net 248123748912 getinvitations
S: getinvitations 1,some description for group 1|2, ...

Invitations are separated by a ‘|’ character, and consist of a group id followed by the description of the group.

When a user wishes to join a group he was invited to, he sends the following message:

C: johannes@verelst.net 248123748912 join 4
S: join OK

It is also possible for a user to get a list of groups. This 'getgroups' command has one parameter: '0' in case a list of all groups should be returned, '1' to return those groups of which this user is a member.

C: johannes@verelst.net 248123748912 getgroups 1
S: getgroups 3,Name of group 3,17,Description of 17

The last groups related command is the 'getgroupinfo' command. This will return the status for all members of the group. This allows other members to receive the state of the execution of the Secret Santa protocol.

When a user sends the 'getgroupinfo' he will receive a message containing the following information:

- Whether or not he owns the group ('1' if so, '0' otherwise)
- The id’s of all users (email addresses)
- The status of the users

The status can be one of the following numbers:

- 1, The user has received an invitation, and is asked to join
- 2, The user has joined, but has to submit his public key
- 8, The user has to shuffle the wishlists
- 16, The user has shuffled, but needs to verify the proofs

C: johannes@verelst.net 248123748912 getgroupinfo 3
S: getgroupinfo 1|ID-of-user1|STATUS-of-user1|...
6.3.5 Key Download

Our mixing solution demands that all data used in the protocol is encrypted with a group key, which is the product of the public keys of the participants. Our bulletin board needs to assist users in downloading this group key.

When the creator of the group verified that all users have joined, he closes the group. This means that no more participants may enter, and that the public key of the group can be downloaded as soon as all participants submitted their public key.

C: johannes@verelst.net 248123748912 closegroup 3  
S: closegroup OK

When the group is closed, the group key can be downloaded. The bulletin board will calculate this group key for the user by multiplying all public keys of the participants.

C: johannes@verelst.net 248123748912 getgroupkey  
S: getgroupkey 4234132367981030819662...

6.3.6 Mixing

As soon as a user downloaded the group key, he can participate as a Mix-server. This consists of retrieving the previously mixed names, mixing them, creating a proof of correct mixing, and uploading all this data back to the bulletin board.

Special care must be taken that no two participants mix at the same time. The bulletin board should disallow anybody to retrieve the mixing data when somebody else mixes. This is implemented with a lock that is granted to the mixing user, which will expire after a configurable amount of time.

The first message from the user asks the bulletin board to send him the data to mix. The bulletin board will respond with two sets of integers. The first set contains the encrypted names as they were placed on the bulletin board by the last player that mixed. The second set contains a (static) list of integers which represent trivial encryptions of the names of the participants: they are the ciphertexts that were created and then used by the first participant. Every user should verify that these ciphertexts are indeed correct, to conclude that the first participant didn't cheat.

C: johannes@verelst.net 248123748912 getmixdata 3  
S: getmixdata 87294...,2839423...,1967...,3413...,...

When a user finishes mixing, he needs to upload the new ciphertexts to the bulletin board, together with the proof that his mixing was correct. If this user is the first to submit mix-data, he must also submit the initial ciphertexts that he created.

Both kinds of submissions of mix-data are executed through the 'setmixdata' command, which takes three arguments: the id of the group, the new mix-data and the corresponding proof of correct mixing. If the proof is absent, the mix-data is being treated as the list of initial ciphertexts.
6.3.7 Proof Verification

When all users mixed, the proofs that were generated need to be verified. Therefore we ask all users to download these proofs, check them for validity and report back on the bulletin board whether or not the proof was correct.

First the participants need to download the proofs:

C: johannes@verelst.net 248123748912 getproofs 3
S: getproofs userid1|data|proof|userid2|data|...

Both the data and proof fields are of the same format as the list of integers before: integers separated by comma’s.

When all proofs are verified, the user sends the verifyproofs command to the bulletin board. Possible arguments are:

- OK, When the user sends the string 'OK', he indicates that all proofs were valid.
- userid, When a userid is given, this indicates that the user cheated according to this participant.

C: johannes@verelst.net 248123748912 verifyproofs OK
S: verifyproofs OK

6.3.8 Plaintext Equality Test

Implementation of the Plaintext Equality Test is straightforward: the participants download all the original ciphertexts and the final mixing output. They create a random exponentiation of all divisions, and hash the output. This hash is placed on the bulletin board.

C: johannes@verelst.net 248123748912 commitpet MD5-3fa6e3...
S: commitpet OK

When all commitments have been placed on the bulletin board, they continue by uploading the random exponentiations.

C: johannes@verelst.net 248123748912 submitpet 125572...,...
S: johannes@verelst.net submitpet OK

All users now download the commitments and the exponentiations and check if they are correct.

C: johannes@verelst.net 248123748912 getpetcommitments
S: getpetcommitments MD5-3fa6e3..|MD5-8cbb73...
C: johannes@verelst.net 248123748912 getpetvalues
S: getpetvalues 125572...,....
The users now combine all exponentiated ciphertexts to a combined ciphertext, and participate in a joint decryption protocol to decrypt it. Therefore they must all upload a partial decryption of this ciphertext to the bulletin-board.

C: johannes@verelst.net 248123748912 submitpetdecryptions ...
S: submitpetdecryptions OK

At this stage we are left with a challenge: the users need to download the partial decryptions and reach consensus about how to proceed. They need to post back their full decryption, wait for a majority to have come to the same conclusion, and then a majority needs to instruct the bulletin board how to proceed (either all mix data should be purged, or users should be able to continue with decryption of the ballots).

Instead of implementing this troublesome scenario, we chose to give the bulletin board a bit more functionality. The bulletin board will combine the partial decryptions, and then decide how the group proceeds. To maintain verifiability, we allow all participants to download the partial decryptions to verify the choice made by the bulletin board.

After receiving the last batch of partial decryptions, the bulletin board will change the group status to either one of these values:

- 8, Waiting for users to shuffle the data again (in case there were fixed points).
- 64, Waiting for users to submit partial decryptions for the ballots.

### 6.3.9 Joint Decryption

The final step in the protocol is the decryption of the ballots. All participants $i$ create partial decryptions of all ballots $j$ where $i \neq j$, and upload these to the bulletin board. When they are all uploaded, the intended recipient of the ballot downloads all partial decryptions and combines them with the partial decryption he made.

The user sends out a message with the following format:

C: johannes@verelst.net 248123748912 submitdecryptions ...
S: submitdecryptions OK

As soon as all partial decryptions for a user are submitted, he can download them to combine them with the partial decryption he made himself:

C: johannes@verelst.net 248123748912 getdecryptions
S: getdecryptions 195184...,57235...,..

### 6.4 Evaluation

Writing cryptographic software proved to be harder than we estimated at first. After implementing the Secret Santa protocol, we noticed that there are several areas that delayed our implementation. We elaborate on those in the following paragraphs.
Padding All computer information consists of bits and bytes, but the protocols we used for our implementation consisted of calculations with big integers. We note that all calculations have to be done modulo $p$, which we consider to be of bitlength $n$. When converting bitstreams to integers, we must now use blocks of $n - 1$ bits, since all values need to be smaller than $p$.

When converting an integer to a bitstring, we must pad this bitstring with zeroes until it reaches size $n - 1$.

Trying not to invent the wheel again, we chose for implementing padding using the PKCS#1 standard. Unfortunately, this proved not to be a good idea, since PKCS#1 requires messages to be in a certain format. After re-randomisation of our ciphertexts (during mixing stage), this format was lost, and our messages were no longer in PKCS#1 format. After some investigation we decided to write our own padding routines, at the cost of not being in compliance with standards.

Modular Arithmetic We chose Java as our programming language, and it features a nice rich API for modular arithmetic. Unfortunately, this API must be implemented on a very wide range of platforms, with different conventions on storing integers in memory. While the first Java Development Kit (1.0) contained errors on some platforms, it’s modular arithmetic was considered to be quite fast. The errors were fixed in subsequent versions, but at the expense of a serious drop in performance. This means that modular exponentiations (which we use constantly) are rather slow.

Secure Programming Another problem during our implementation was the fact that cryptographic software needs to be 100% secure. This means that all bounds need to be checked, all incoming values need to be validated, etc. While it is good practice to keep security in mind during implementation, it also adds to development time.
A. Bibliography


B. Implementation Details

This section contains some background information about the mixing implementation: the development of an extendible ElGamal library.

B.1 Libraries

In our search for a suitable library for our implementation, we encountered the problem that there appear to be no free Java libraries that allow all of the following operations:

- Generation and import of ElGamal keys, both with known parameters
- Multiplication of ElGamal keys to generate a group ElGamal key
- Encryption of plaintext
- Multiplication of ciphertext
- Division of ciphertext

These operations are required to be able to perform ElGamal group encryption, one of the requirements for our mixing solution. There are however several free libraries that come close:

B.1.1 JCE

When using Java as programming language, it seems obvious to use the Java Cryptographic Extensions for cryptographic operations. Unfortunately, the JCE does not allow all required operations in the API. Generation of keys with known parameters is not possible, neither is creation of PublicKey and PrivateKey objects with known parameters.

B.1.2 Cryptix

The Cryptix library is available from http://www.cryptix.com. It allows cryptographic manipulations through the JCE (it acts as a cryptographic provider), but it is also possible to directly call the Cryptix api. Cryptix is licensed under the Cryptix Public License, with bears great resemblance with the well-known GNU Public License.

Cryptix makes use of known strong primes, which are precomputed and hard-coded in the software. This has a big advantage for us, since we can trust on the fact that all generated ElGamal keys have the same modulus. This means that all generated keys can be combined for ElGamal group encryption.
Unfortunately, after some correspondence with the authors, it became clear Cryptix does not allow partial decryptions of a ciphertext. This means that Cryptix is unsuitable for us since we need partial decryptions and recombination of them for group cryptography.

B.1.3 Flexiprovider

While FlexiProvider (http://www.flexiprovider.de) is a very extensive library, it also does not suffice our requirements. Besides the fact that it does not allow for group encryption, it has another weak point: it generates a new prime modulus for every generated keypair. Besides the fact that will make these keys unsuitable for group encryption, there is another caveat. For every keypair, FlexiProvider will try to generate a new prime $q$ and test if $p = 2q + 1$ is a prime. This generation of strong primes is a very intensive process, it may take over 90 minutes to generate a strong prime.

B.1.4 Bouncy Castle

The third major library is Bouncy Castle (http://www.bouncycastle.org), which is also released under an Open Source license. Just like Cryptix it allows for ElGamal operations with precomputed parameters. Unfortunately this library also lacks group encryption, and the API does not allow for easy extension to add group encryption.

B.2 Software Design

Since none of the mentioned libraries contains all functionality needed for this project, we decided to create our own. The Java language has a very powerful API for manipulations of big integers, which allows us to quickly create a library for ElGamal encryption.

B.2.1 Classes

The net.verelst.crypto.elgamal package contains the following classes:

- ElGamalPublicKey, represents a public key.
- ElGamalPrivateKey, represents a private key.
- ElGamalKeyPair, contains a public and a private key.
- ElGamalKeyPairGenerator, generate a new keypair using a precomputed strong prime.
- ElGamalText, abstract interface defining some general properties of both plaintexts and ciphertexts.
- ElGamalPlainText, contains a plaintext before encryption or after decryption. It includes methods for converting byte-arrays to big integers, including padding issues.
• *ElGamalCipherText*, contains the ciphertext, split into *ElGamalCipherTextElements*. It contains routines to convert this ciphertext to and from byte-arrays, and methods to allow for multiplication of ciphertexts.

• *ElGamalCipherTextElement*, represents a tuple \((u, v)\) that is part of an *ElGamalCipherText*.

• *ElGamalCipher*, decryption and encryption routines, including partial decryption and re-encryption function.

The following sample code shows how to encrypt and decrypt a message:

```java
import net.verelst.crypto.elgamal.*;
public class CryptTest {
    public static void main(String[] args) {

        /* Initialise the parameters */
        ElGamalParams params = new ElGamalParams();
        ElGamalCipher cipher = new ElGamalCipher();

        /* Generate a new keypair */
        ElGamalKeyPairGenerator kpg =
            new ElGamalKeyPairGenerator(params);
        ElGamalKeyPair kp = kpg.generateKeyPair();
        ElGamalPrivateKey privkey = kp.getPrivate();
        ElGamalPublicKey pubkey = kp.getPublic();

        /* Encryption with the public key, and decryption with the private key */
        ElGamalPlainText plain =
            new ElGamalPlainText(args[0].getBytes(), params);
        ElGamalCipherText encrypted =
            cipher.encrypt(plain, pubkey);
        ElGamalPlainText decrypted =
            cipher.decrypt(encrypted, privkey);
    }
}
```

Full description of the API, you can download and view the API documentation on the download site of this software package.

The *net.verelst.crypto.mix* package contains the following classes:

• *BaseIntegers*, represents the list of *base integers* needed for the proof that mixing was done correctly.

• *PermutationMatrix*, represents the permutation that is done on the ciphertexts.
• RandomizationIntegers, contains the integers that were used to re-randomise the ciphertexts.

• Proof, contains the entire proof that a shuffle was done correctly.

• Shuffle, a class containing methods that can generate a new permutation of a list of ElGamal encrypted ciphertexts. Return values contain the new shuffled ciphertexts, and a zero-knowledge proof to show that the shuffle was correct.

• ProofVerifier, contains methods that allow a proof to be checked.

The following example code shows how to use this API:

```java
import net.verelst.crypto.mix.*;
import net.verelst.crypto.elgamal.*;

...

/* ... we assume to have the following variables defined:
   - params, of type ElGamalParams
   - original, an array of ElGamalCipherText objects
   - groupKey, an ElGamalPublicKey
 */

BaseIntegers baseints = new BaseIntegers(n);
Shuffle shuffle = new Shuffle(params, baseints, original, groupKey);
shuffle.shuffle();

newballots = shuffle.getBallots();
proof = shuffle.getProof();
ProofVerifier verifier = new ProofVerifier(params, baseints, groupKey);
System.out.println("Proof valid? " +
   verifier.verify(original, newballots, proof));

B.3 Database Design

Our bulletin board is built upon a database that contains all information. There are several tables in this database, which we will describe here.

B.3.1 User

The user table contains all information known about a user on the bulletin board.
CREATE TABLE user (
  id int(11) NOT NULL auto_increment,
  email varchar(255) default NULL,
  publickey text,
  authcode varchar(255) default NULL,
  PRIMARY KEY (id)
);

The authcode field contains the cookie known only by the server and the client, all other fields speak for themselves.

**B.3.2 Usergroup**

When users wish to participate in Secret Santa, they need to create a group. The usergroup table contains group information:

CREATE TABLE usergroup (
  id int(11) NOT NULL auto_increment,
  name varchar(255) default NULL,
  owner int(11) NOT NULL default '0',
  closed int(11) NOT NULL default '0',
  PRIMARY KEY (id)
);

The owner field contains the id of the owner of the group, which is a foreign key to the user table. The closed field indicates whether or not the group is closed.

**B.3.3 Membership**

To be able to connect users to groups, we have a membership table. This table has the following layout:

CREATE TABLE membership (  
  userid int(11) NOT NULL default '0',
  groupid int(11) NOT NULL default '0',
  invitation int(11) NOT NULL default '0',
  position int(11) default '0'
);

The invitation field is used to invite people to a group. There is an extra field position that can be used for ordering purposes. The first member of the group has position '0', the second '1', and so on.

**B.3.4 Mixdata**

All data regarding to mixing is saved in the following table:
CREATE TABLE mixdata (
    userid int(11) NOT NULL default '0',
    groupid int(11) NOT NULL default '0',
    position int(11) NOT NULL default '0',
    data text,
    proof text
);

The position fields holds the position of this data entry in the entire protocol. The first data submitted will have position '0', and it will increase as more data is submitted.

### B.3.5 Plaintext Equality Test

The following table creation SQL command creates a table for holding the intermediate values for the Plaintext Equality Test.

CREATE TABLE pet(
    userid int(11) NOT NULL default '0',
    groupid int(11) NOT NULL default '0',
    position int(11) NOT NULL default '0',
    commitment varchar(255),
    data text,
    decryption text
);

The userid and groupid fields speak for themselves. The position field points to the ciphertext index, counting from 0. All data regarding the divisions of ciphertexts for the 3rd user will have position 2.

The data field contains the exponentiated division of ciphertexts, the commitment field contains a commitment to this value. The partial decryption for the PET is stored in the decryption field.

### B.3.6 Joint Decryption

The final stage in the Secret Santa protocol is the joint decryption of the ciphertexts. We have the following table definition to store the data:

CREATE TABLE decryption(
    userid int(11) NOT NULL default '0',
    groupid int(11) NOT NULL default '0',
    position int(11) NOT NULL default '0',
    decryption text
);

The fields have the same semantics as for the pet table.