Eindhoven university of technology
Department of mathematics and computer science

Master's thesis

Secure building blocks for privacy preserving reputation systems

J.E.J. (Joeri) de Ruiter

Eindhoven, August 2010

Supervisors:
Dr.ir. L.A.M. (Berry) Schoenmakers
Dr. J. (Jorge) Guajardo Merchan
Dr. S.S. (Sandeep) Kumar

Technische Universiteit Eindhoven University of Technology

PHILIPS sense and simplicity

Master's thesis

Secure building blocks for privacy preserving reputation systems

Eindhoven university of technology
Department of mathematics and computer science

Author:
J.E.J. (Joeri) de Ruiter

Supervisors:
Dr.ir. L.A.M. (Berry) Schoenmakers
Dr. J. (Jorge) Guajardo Merchan
Dr. S.S. (Sandeep) Kumar

Eindhoven, August 2010

Abstract

When performing online transactions it is important to be able to determine the trustworthiness of the users involved. To help in determining this, we can make use of reputation systems. In reputation systems, the opinions of users are used to compute a reputation score for a particular user. These ratings might encourage or deter people from performing transactions with a particular user. These actions can in turn result in a benefit or a disadvantage for both the user providing the ratings and the user receiving the ratings. For example, giving negative ratings might result in a retaliatory action by the user being rated. In this thesis, we investigate how multi-party computation techniques can be used to preserve the privacy of the users providing ratings in a reputation system. In particular, we introduce secure variants of two building blocks that are used in reputation systems.

The first building block that we discuss is a sorting algorithm. We introduce various new and privacy preserving sorting algorithms. For values from a limited domain, we design a new algorithm based on counting sort. We first give an algorithm for sorting values from a two-valued domain. Using this algorithm, we constructed an algorithm for values from a larger (but limited) domain. The complexity of the last algorithm increases linearly with the size of the domain for the input values.To sort values from a larger domain, we discuss the Randomized Shellsort algorithm by Goodrich [Goo10]. This algorithm is a comparison-based algorithm and sorts using only $O(n \log n)$ compare-exchange operations. However, it does this probabilistically. A variant by Wang et al., called Fast randomized Shellsort, uses less compare-exchange operations, but sorts less inputs correctly [WLG+10]. We introduce the Double brick randomized Shellsort that sorts more inputs correctly than the Fast randomized Shellsort. Our algorithm has a depth of only $O(\log n)$, compared to the $O(n)$ depth of the two other randomized Shellsort algorithms. This in turn allows for high degree of parallelization when implemented as a sorting network.

Secondly, we introduce a secure clustering algorithm. In reputation systems clustering is used to group users with similar votes. The idea behind this is that users with similar votes have comparable opinions about other users. Using these constructs results in more accurate or useful reputation scores. In this context, we first discuss how to convert any algorithm to a privacy preserving oblivious variant. After this we present the first secure instantiation of the distance-k clique algorithm by Edachery et al. [ESB99].

Table of Contents

1. Introduction

Nowadays, more and more interactions are performed online. When interacting with another user, we would like to know whether we can trust this user or what the quality is of the service he provides. In the offline world, we would ask friends and family about experiences with other people or companies. Online, there are a lot more possible users to interact with, who can be scattered throughout the world. Therefore, it is less likely that family and friends can give advice or experience about a particular user.

To still be able to determine the reputation of a user, we can make use of the opinions of other users that had an interaction with this user. This can be achieved by the use of reputation systems. Reputation systems can provide a means to assess the trustworthiness and quality of users, based on the opinions of other users. Examples of practical reputation systems can be found on eBay [1] and the Amazon Marketplace [2], where sellers are rated by buyers on quality of the service and goods. In a reputation system, the opinions of other users are collected, and based on this a reputation score is computed. A reputation score can be as simple as counting positive opinions or might involve complex calculations. The reputation score indicates how trustworthy the user is considered by other users. The opinions of other users can thus influence whether a user will interact with another user. Therefore, giving disappointing ratings might result in negative actions from the user that is rated. To prevent these negative actions, we thus would like to provide the users with privacy of their ratings. Though there exist a variety of different reputation systems, most of these do not take privacy of the ratings into account. The work that has been done on providing privacy is done mostly for simple additive or average based reputation metrics.

In this thesis we will introduce secure variants of two building blocks that are used in reputation systems. The first building block is sorting, that is, for example, used in the P2PRep system in the computation of the reputation score, where lower scores get a higher weight [ADD+06]. Secondly we introduce a secure clustering algorithm. In the knot-ware reputation system by Gal-Oz et al., clustering is used to group users with similar votes [GOGH08]. The idea behind this is that users with similar votes have comparable opinions about other users. Using these constructs results in more accurate or useful reputation scores.

For both sorting and clustering we present secure variants that do not leak information about the values of the ratings. To provide the privacy in the building blocks we use secure multi-party computation based on homomorphic encryption. Secure multi-party computation is a classic way to perform secure distributed computations. In secure multi-party computation, the participants jointly compute a function on private input data. The output of the protocol can be either public or private. In the computation no information about the private inputs is leaked, other than what can be learned from the output of the computation. In our case, we make use of homomorphic cryptosystems. These systems have the property that when performing certain algebraic operations on the ciphertext this results in a, possibly different, algebraic operation on the plaintext. This way, operations can be performed on the plaintext, without the need to decrypt the ciphertext. Using this property it is possible to compute any mathematical function.

Though we introduce the secure building blocks in the context of reputation systems, they can also be of use in various other types of systems and protocols.

[1] http://www.ebay.com/
[2] http://www.amazon.com/

1.1 Outline

In Chapter 2 we discuss the cryptosystems we use and secure multi-party computation. For secure multi-party computation, we give the building blocks to compute the multiplication of two private inputs, that are needed to compose the different protocols in the next chapters.
In Chapter 3 we present the basic secure multi-party protocols that we use for the protocols we introduce later. In Chapter 4 we discuss reputation systems. We give an introduction and discuss two existing systems for which we introduce secure building blocks in Chapters 5 and 6. The systems that are discussed are P2PRep and the knot-aware reputation system [ADD+06, GOGH08]. We also discuss work that has been done on privacy in reputation systems. An extension to the P2PRep reputation system by Nithyanand and Raman, called 3PRep, is discussed. Also several protocols to securely compute the weighted average are discussed.
In Chapter 5 we present two types of oblivious sorting algorithms. The first type is counting sort, where we introduce two new algorithms. One for sorting bits, and one for sorting elements from a larger domain. The second type is comparison-based sorting algorithms. We discuss the Randomized Shellsort recently introduced by Goodrich [Goo10], and a variant by Wang et al. [WLG+10]. We introduce a new variant that sorts better than the one by Wang et al., while using the same number of compare-exchange operations and having a lower depth compared to the variant by Wang et al. We also discuss the privacy issues regarding the randomized sorting algorithms.
In Chapter 6 we first show how to transform a simple algorithm to detect whether cycles exist in a graph to a secure oblivious variant. After this we discuss the algorithm by Edachery et al. for the distance-k clique problem [ESB99]. For this algorithm we again show how to transform it to a secure oblivious variant.

2. Preliminaries

In this chapter, the basic building blocks that we use in the next chapters are discussed. First we present the notations used in this thesis. Next we discuss the various cryptosystems that we use, and the special properties we require from these systems. After this we introduce secure multi-party computation and the building blocks we use.

2.1 Notation

We will denote by $[\![x]\!]$ the encryption of x. The number of bits that is needed to represent the value x, is denoted by $\ell(x)$. To convert a boolean value b to the integer domain $\{0, 1\}$, we use the notation $[b]$, i.e. $[true] = 1$ and $[false] = 0$.
To present values in unary notation, we make use of two different notations. The first we call Dirac notation. This notation has a 1 at the index that is equal to the value it represents, and 0 at the other positions. So, for the unary value of v the Dirac notation u is as follows:

$$u_{D_i} = \begin{cases} 1 & \text{if } i = v \\ 0 & \text{otherwise} \end{cases}$$

The second notation is what we call Heaviside notation. With this notation the value at the index of the value it represents is 1, but also the values at all lower indices are 1, the values at the higher indices are 0. The unary value of v in Heaviside notation is the following:

$$u_{H_i} = \begin{cases} 1 & \text{if } i \leq v \\ 0 & \text{otherwise} \end{cases}$$

If we encrypt a value x in unary notation, this is denoted by $[\![x]\!]_{\mathbb{U}}$.

2.2 Cryptosystems

In this section we will discuss the various cryptosystems that we use. A cryptosystem is used to encrypt messages, so the content of the message can only be retrieved if some private key is known. A cryptosystem consists of three algorithms: key generation, encryption and decryption. In this thesis we discuss two different cryptosystems, namely ElGamal and Paillier [Elg85, Pai99].

2.2.1 Homomorphic cryptosystems

All cryptosystems that are discussed are homomorphic. In general, a homomorphism is a function between two algebraic objects that respect the algebraic structure. A homomorphic cryptosystem is thus one, where the encryption operation preserves the algebraic structure of a plaintext operation, i.e. $[\![x]\!] \boxplus [\![y]\!] = [\![x \boxtimes y]\!]$. In our protocols, we need the cryptosystem to be additively homomorphic, i.e. performing some operation on two encrypted values results in the encryption of the sum of the two values.

2.2.2 Threshold cryptosystems

With a threshold cryptosystem, there are multiple parties needed to decrypt ciphertexts. There still is one public key that is used to encrypt messages. However, the secret key is shared among several parties. It is possible that all parties are required to participate in the decryption. It might also be sufficient if only a certain number of parties cooperate. Whenever the number of parties that cooperate is less than the threshold, they will not be able to learn anything about the encrypted message. If there are n participants in total, but only t required to perform a decryption, the system is a (t, n)-threshold cryptosystem.
Threshold encryption for the ElGamal cryptosystem is due to Pedersen [Ped91]. For the Paillier cryptosystem this due to Damgård and Jurik [DJ01].

2.2.3 ElGamal

In this section, the ElGamal cryptosystem is described [Elg85]. The cryptosystem is based on the discrete logarithm problem, and works with elements from a multiplicative cyclic group. Every party has a secret key x, from which the public key h can be derived, using $h = g^x$, where g is a generator of the group. A message m is encrypted as follows:

$$[\![m]\!] = (g^r, h^r m)$$

To decrypt, the private key x can be used to retrieve the original message:

$$m = \frac{h^r m}{(g^r)^x}$$

This system is multiplicative homomorphic. When multiplying the two ciphertexts of the messages m_1 and m_2, we get the following:

$$[\![m_1]\!]\,[\![m_2]\!] = (g^{r_1} * g^{r_2}, h^{r_1} m_1 * h^{r_2} m_2) = (g^{r_1+r_2}, h^{r_1+r_2} m_1 m_2) = [\![m_1 m_2]\!]$$

As can be seen, this is the encryption of the message $m_1 m_2$.
To get an additive homomorphic system, the message m can be encoded as g^m.

$$[\![m]\!] = (g^r, h^r g^m)$$

We then get the following when multiplying two ciphertexts:

$$[\![m_1]\!]\,[\![m_2]\!] = (g^{r_1} * g^{r_2}, h^{r_1} g^{m_1} * h^{r_2} g^{m_2}) = (g^{r_1+r_2}, h^{r_1+r_2} g^{m_1+m_2}) = [\![m_1 + m_2]\!]$$

This results in an encryption of the message $m_1 + m_2$. However, to decrypt a message m that is encoded as g^m, we need to solve a discrete logarithm problem, which is the problem that the cryptosystem is based on. We thus need to try all possible plaintexts, to see which one is in the encryption. On average we will need to try half of all possible plaintexts.
We want to be able to re-encrypt ciphertexts, such that we get a new ciphertext that cannot be linked to the original one, but still contains the same message. This can be done by using the homomorphic property, to add an encryption of 0 to the original ciphertext:

$$[\![m]\!]\,[\![0]\!] = (g^{r_1}, g^m h^{r_1})(g^{r_2}, h^{r_2}) = (g^{r_1+r_2}, g^m h^{r_1+r_2}) = [\![m]\!]$$

As can be seen, this is again an encryption of the message m, but with a different random value. If the encryption of m is already given, this requires two additional exponentiations.
To multiply an encrypted message m with a public message v we can use

$$[\![m]\!]^c = (g^r, g^m h^r)^c = (g^{cr}, g^{cm} h^{cr}) = [\![cm]\!]$$

This construction is also called private multiplier.

2.2.4 Paillier

Next we will discuss the cryptosystem that was introduced by Paillier in 1999 [Pai99]. This cryptosystem is based on the Decisional Composite Residuosity Assumption. On input of a security parameter k, a key generation algorithm outputs a composite number $n = pq$ of length k, with p and q safe primes. We consider the group $\mathbb{Z}_{n^2}$ and we have message space of size $m = n$. The public key is n, with as secret key a value μ satisfying $\mu = 1 \bmod m$ and $\mu = 0 \bmod lcm(p-1, q-1)$. The encryption function for message x and random value r is now defined as:

$$[\![x]\!] = (n+1)^x r^n \bmod n^2 \equiv (xn+1)r^n \bmod n^2$$

A ciphertext c can again be decrypted using the function that is defined as:

$$L(c^\mu \bmod n^2)$$

Where $L : (1+n)^{x\mu} \bmod n^2 \mapsto x \bmod n$ is a well-defined function [DJ01].
As can be seen in the following equation, this system is additively homomorphic:

$$[\![x_1]\!]\,[\![x_2]\!] = (n+1)^{x_1} r_1^n (n+1)^{x_2} r_2^n = (n+1)^{x_1+x_2}(r_1 r_2)^n = [\![x_1 + x_2]\!]$$

Like with ElGamal, we also have a private multiplier, to multiply encrypted message x with public message c:

$$[\![x]\!]^c = ((n+1)^x r^n)^c = (n+1)^{cx} r^{cn} = [\![cx]\!]$$

2.3 Secure multi-party computation

Secure multi-party computation is used to evaluate a function by a group of participants. The inputs to the computation can be private, either to all participants or to only a group of participants, for example everyone except the owner. The computation is performed in such a way that no information is leaked about the private inputs, except possibly the output of the computation. The protocol can be performed by all users providing an input. However, increasing the number of participants in the protocol also increases the communication and computational costs. An alternative to this would be the so called outsourcing model. In this model there exists a pool of trusted parties. These parties could be, for example, trusted peers or notaries. Each user selects a party he trusts, and all selected parties perform the protocol together. This way, as long as at least one of the computing parties is honest, the computation is still secure. The output can be re-encrypted using the keys of the users providing the input, so the trusted parties do not learn the plaintext output.

2.3.1 Security model

The security model defines the assumptions we have about the capabilities of the adversary. In our protocols we consider the semi-honest model, which is also called honest-but-curious. In this model, the adversary follows the protocol, but tries to learn information from the messages exchanged among the participants in the protocol. The adversary is in this case passive. Our protocols could be extended to also work with the malicious model, where the adversary is active. This can be done by adding zero knowledge proofs to the communications for the performed operations.

2.3.2 Building blocks

To be able to evaluate all possible functions we need to able to perform addition and multiplication. To be able to evaluate all possible functions in the encrypted domain we therefore only need a protocol to be able to perform multiplication, as addition is already provided by the homomorphic property of the cryptosystems. Next we introduce several protocols that can be used to compute the multiplication of two encrypted values.

2.3.2.1 Conditional gate

The conditional gate is introduced by Schoenmakers and Tuyls in [ST04]. The protocol, that is intended for ElGamal, computes the multiplication between two encrypted values, of one of which is from a two-valued domain. Since we only consider a passive adversary, we do not take the commitments into account that are part of the original protocol.

Protocol 2.3.1 (CONDITIONAL-GATE - Computing the multiplication of two values, where one is from a two-value domain)**.** Given two encrypted values $[\![x]\!]$ and $[\![y]\!]$, where x is from the domain $\{-1, 1\}$, the following protocol outputs $[\![xy]\!]$. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following computation:

1. Let $x_0 = x$ and $y_0 = y$
2. Every participant $\mathcal{P}_i$ takes $[\![x_{i-1}]\!]$ and $[\![y_{i-1}]\!]$ as input, and takes a random value s_i, where $s_i \in_R \{-1, 1\}$. $\mathcal{P}_i$ applies the private-multiplier to $[\![x_{i-1}]\!]$ and s_i, to get $[\![x_i]\!] \leftarrow [\![x_{i-1}s_i]\!]$. The same is applied to get $[\![y_i]\!] \leftarrow [\![y_{i-1}s_i]\!]$. $\mathcal{P}_i$ then sends $[\![x_i]\!]$ and $[\![y_i]\!]$ to $\mathcal{P}_{i+1}$.
3. Together, the parties decrypt $[\![x_n]\!]$ to get x_n.
4. Each participant can now individually compute the output by performing the private-multiplier on x_n and $[\![y_n]\!]$ to get $[\![x_n y_n]\!]$

2.3.2.2 Multiplication

We want to calculate the multiplication of two encrypted values $[\![x]\!]$ and $[\![y]\!]$. This can be done using the following protocols. Protocol 2.3.2 is optimized for communication complexity, and Protocol 2.3.3 for round complexity. Both protocols work with the Paillier cryptosystem, but not with the ElGamal cryptosystem. The protocols are based on the protocol by Cramer et al. [CDN01].

Protocol 2.3.2 (MULT - Multiplication of two encrypted values)**.** Given two encrypted values $[\![x]\!]$ and $[\![y]\!]$, the following protocol outputs the encrypted multiplication $[\![xy]\!]$. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following steps:

1. Participant $\mathcal{P}_1$ calculates $[\![sum_x]\!] \leftarrow [\![x]\!] [\![r_1]\!]$ and $[\![sum_y]\!] \leftarrow [\![y]\!]^{-r_1}$, for random r_1, and sends $[\![sum_x]\!]$ and $[\![sum_y]\!]$ to participant $\mathcal{P}_2$
2. Participant $\mathcal{P}_i$ receives $[\![sum_x]\!]$ and $[\![sum_y]\!]$, calculates $[\![sum_x]\!] \leftarrow [\![sum_x]\!] [\![r_i]\!]$ and $[\![sum_y]\!] \leftarrow [\![sum_y]\!] [\![y]\!]^{-r_i}$, and sends $[\![sum_x]\!]$ and $[\![sum_y]\!]$ to participant $\mathcal{P}_{i+1}$, for random r_i, for $i = 2, \ldots, n$
3. The participants jointly decrypt $[\![sum_x]\!]$ that is computed by participant $\mathcal{P}_n$ to get $x + \sum_{i=1}^{n} r_i$
4. Each participant individually computes the output $[\![y]\!]^{sum_x} [\![sum_y]\!] = [\![y(x + \sum_{i=1}^{n} r_i)]\!] [\![-y \sum_{i=1}^{n} r_i]\!] = [\![xy]\!]$

The round complexity of this protocol is $O(P)$, for P parties. The communication complexity is $O(Pk)$, where k is the size of an encrypted message.
The original x can only be restored from the decrypted value if all parties combine their random values together, in which case they also could just decrypt the values together.
We can also construct a protocol that has a constant round complexity at the cost of increased communication costs.

Protocol 2.3.3 (MULT - Multiplication of two encrypted values)**.** Given two encrypted values $[\![x]\!]$ and $[\![y]\!]$, the following protocol outputs the encrypted multiplication $[\![xy]\!]$. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following steps:

1. Each participant $\mathcal{P}_i$ individually computes $[\![sum_{x,i}]\!] \leftarrow [\![r_i]\!]$ and $[\![sum_{y,i}]\!] \leftarrow [\![y]\!]^{-r_i}$, for random r_i, and broadcasts the two encrypted values
2. Each participants individually computes $[\![sum_x]\!] \leftarrow [\![x]\!] \prod_{i=1}^{n} [\![sum_{x,i}]\!]$ and $[\![sum_y]\!] \leftarrow \prod_{i=0}^{n} [\![sum_{y,i}]\!]$
3. The participants jointly decrypt $[\![sum_x]\!]$
4. Each participant individually computes the output $[\![y]\!]^{sum_x} [\![sum_y]\!] = [\![y(x + \sum_{i=1}^{n} r_i)]\!] [\![-y \sum_{i=1}^{n} r_i]\!] = [\![xy]\!]$

This protocol has an increased communication complexity of $O(P^2k)$, for P parties. The round complexity is in this case $O(1)$. When referring to the MULT protocol in the next sections, this can be replaced by the CONDITIONAL-GATE.

3. Basic protocols

In this chapter we will introduce various basic secure multi-party protocols that are used in the protocols introduced in the next chapters.

3.1 Conditional-swap

Schoenmakers and Tuyls introduce the CONDITIONAL-SWAP protocol in [ST04]. This protocol makes use of the CONDITIONAL-GATE protocol to switch two input elements based on an encrypted condition bit. If the condition is equal to 1, the two values are swapped, otherwise they stay at their positions.

Protocol 3.1.1 (CONDITIONAL-SWAP - Swap two encrypted values based on an encrypted condition bit)**.** Given two encrypted values $[\![x]\!]$ and $[\![y]\!]$, and one encrypted condition bit $[\![b]\!]$, the following protocol outputs them either in the same or in swapped order. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following steps:

1. Each participant computes $[\![x]\!] [\![y]\!]^{-1}$ to get $[\![x - y]\!]$.
2. The participants jointly compute $[\![b(x - y)]\!]$ using the CONDITIONAL-GATE.
3. The participants compute the outputs $([\![b(x - y)]\!]^{-1} [\![x]\!], [\![b(x - y)]\!] [\![y]\!])$.

The last step in the protocol outputs $([\![x]\!], [\![y]\!])$ if $b = 1$, and $([\![y]\!], [\![x]\!])$ otherwise. The following equation shows why this holds:

$$
\begin{aligned}
&([\![b(x-y)]\!]^{-1}[\![x]\!], [\![b(x-y)]\!][\![y]\!]) \\
&\qquad = ([\![-b(x-y)+x]\!], [\![b(x-y)+y]\!]) \\
&\qquad = \begin{cases} ([\![-1(x-y)+x]\!], [\![1(x-y)+y]\!]) & \text{if b} = 1 \\ ([\![-0(x-y)+x]\!], [\![0(x-y)+y]\!]) & \text{if b} = 0 \end{cases} \\
&\qquad = \begin{cases} ([\![y]\!], [\![x]\!]) & \text{if b} = 1 \\ ([\![x]\!], [\![y]\!]) & \text{if b} = 0 \end{cases}
\end{aligned}
$$

3.2 Conditional select

The COND-SEL protocol is used to select a value from two given values, depending on a boolean condition. The function to be evaluated is:

$$\text{COND-SEL}([\![[b]\!]], [\![x]\!], [\![y]\!]) = \begin{cases} [\![x]\!] & \text{if } b \\ [\![y]\!] & \text{if } \neg b \end{cases} \tag{3.1}$$

The protocol is a special case of the COND-SWAP and can be implemented using the MULT or COND-MULT protocol.

Protocol 3.2.1 (COND-SEL - Conditional select of values)**.** Given the input $[\![x]\!]$, $[\![y]\!]$ and $[\![[b]\!]]$, the following protocol evaluates function 3.1. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following steps:

1. Each participant individually computes $[\![d]\!] \leftarrow [\![x]\!] [\![y]\!]^{-1} = [\![x - y]\!]$, to obtain the encrypted difference between x and y
2. The participants jointly compute $[\![m]\!] \leftarrow \text{MULT}([\![b]\!], [\![d]\!])$
3. Each participant individually computes the output $[\![y]\!] [\![m]\!] = [\![b(x - y) + y]\!]$

The COND-SEL protocol has the same round and communication complexity as the multiplication protocol that is used. If the Paillier cryptosystem is used, the MULT protocol can be used. The round complexity would then be $O(1)$. If the ElGamal cryptosystem is used, the CONDITIONAL-GATE is used. In this case the round complexity would be $O(P)$. The communication complexity is for both cryptosystems $O(Pk)$.

3.3 Compare-exchange

In, for example, comparison sort algorithms we need a secure *compare-exchange* operation. A secure *compare-exchange* operation sorts two values in ascending order without revealing where each value ends up. To achieve this, Protocol 3.3.1 could be used.

Protocol 3.3.1 (COMPARE-EXCHANGE - Performing a compare-exchange operation on two encrypted values)**.** Given two encrypted values $[\![x]\!]$ and $[\![y]\!]$, the following protocol outputs the two values in sorted order.

1. Use a secure integer comparison protocol to get $[\![[x > y]]\!]$.
2. Output the result of the CONDITIONAL-SWAP protocol on $[\![x]\!]$ and $[\![y]\!]$ as input values, and $[\![[x > y]]\!]$ as condition.

If we use n-bit numbers, we need for each *compare-exchange* operation, $2n - 1$ conditional gates for the comparison and n *conditional gates* for the *conditional swap*. The round complexity of the *compare-exchange* operation is $O(Pn)$, for P parties. The communication complexity is $O(Pnk)$, for security parameter k.

3.4 Generating random bits

For various applications, we might want to generate random bits. For example, these can be used to generate random values from a larger domain. The first protocol generates a public random bit.

Protocol 3.4.1 (RAND-BIT - Generate a public random bit)**.** To generate a random bit, the following protocol can be used. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following computation:

1. Every participant $\mathcal{P}_i$ commits to a random bit b_i, $\langle b_i \rangle$
2. Every participant $\mathcal{P}_i$ reveals his bit b_i
3. Each participant individually computes the output by taking the XOR over all bits, $\bigoplus_{i=1}^{n} b_i$

The following protocol gives an encrypted random bit. For this we make use of the XOR protocol by Schoenmakers and Tuyls [ST04].

Protocol 3.4.2 (SEC-RAND-BIT - Generate an encrypted random bit)**.** To generate an encrypted random bit, the following protocol can be used. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following computation:

1. Every participant $\mathcal{P}_i$ takes a random bit $[\![b_i]\!]$
2. Together the participants compute the output $[\![\bigoplus_{i=1}^{n} b_i]\!]$ using the XOR protocol on the encrypted bits

3.5 Generate random values

Using the secure random bit generating protocol, we can generate random values from a predefined domain, e.g. $[0, n)$. The following protocol was presented in [ST06].

Protocol 3.5.1 (RAND-INT - Generate a private random value from the domain $[0, n)$)**.** Given a domain $[0, n)$, the following protocol outputs an encrypted random value $[\![x]\!]$, such that $x \in_R [0, n)$. The participants $\mathcal{P}_i$ $(i = 1, \ldots, p)$ perform the following computation:

1. The participants jointly generate the random bits $[\![r_0]\!], \ldots, [\![r_{\ell(n)-1}]\!]$
2. The participants use a comparison protocol to compute $[\![[r < n]]\!]$, where $[\![r]\!]$ is private and n is public
3. The participants output $[r < n]$.
4. If $[r < n] = 0$, the protocol is repeated.

3.6 Generating a random permutation

There are several methods to construct random permutations. First we can use Knuth's method to construct a random permutation [Knu73]. For this, the parties have to generate random values from a specific range, which can be done using a protocol for generating random bits, such as Protocol 3.4.1. With Knuth's method, we start with the identity permutation of size n. For every position i, we swap the value with a the value at a random position between i and n.
In the second method every party constructs a random permutation in private and broadcasts a commitment to this permutation. After all parties committed to their permutation or some predefined deadline occurs, the permutations are revealed and applied in a sequential order. The commitment again can be performed in several ways. First, a hash can be calculated on the permutation, which is then used to perform the commitment on. The second method first maps the permutation to an integer value. If there are n values, then there are $n!$ possible permutations, so we can map all permutations to a unique value in the range $[0, n!)$. This integer value can be used in an integer commitment scheme to commit to the permutation.

3.7 Bit complement

On input of an encrypted bit $[\![b]\!]$, BIT-COMPL outputs complement of the bit. The function that has to be computed is:

$$\text{BIT-COMPL}([\![b]\!]) = \begin{cases} [\![1]\!] & \text{if } b = 0 \\ [\![0]\!] & \text{if } b = 1 \end{cases} \tag{3.2}$$

Protocol 3.7.1 (BIT-COMPL - Calculate the bit complement)**.** Given $[\![b]\!]$, the protocol calculates the function 3.2. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following computation:

- Each party individually computes $[\![1]\!] [\![b]\!]^{-1}$

In this protocol no communication is needed. When performing the protocol BIT-COMPL on a vector, the protocol can be performed in parallel on every component of the vector.

3.8 Integer to binary conversion

To convert encrypted integer values to their binary representations we make use of the protocol introduced by Schoenmakers en Tuyls in [ST06].

Protocol 3.8.1 (INT-TO-BIN - Convert an encrypted integer value to its binary representation)**.** Given an integer value $[\![x]\!]$, $0 \leq x < N$, the following protocol outputs $[\![x_0]\!], \ldots [\![x_{\ell(x)-1}]\!]$, where x_i is the binary representation of x. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following computation:

1. The participants together generate the encrypted bits $[\![r_0]\!], \ldots [\![r_{\ell(N)-1}]\!]$ using the RAND-INT protocol, where $r \in_R [0, N)$
2. Each participant computes $[\![x]\!] \prod_{j=0}^{\ell(N)-1} [\![r_j]\!]^{2^j}$, to get $[\![y]\!] = [\![x + r]\!]$
3. The participants jointly decrypt $[\![y]\!]$
4. Each participants computes $[\![x_0]\!], \ldots, [\![x_{\ell(N)}]\!]$, using $y_0, \ldots, y_{\ell(N)-1}$ and $[\![r_0]\!], \ldots [\![r_{\ell(N)-1}]\!]$, where $x_i = x$ or $x_i = x - N$ and $z_{\ell(N)}$ is the sign bit.
5. Each participant reduces the value of x modulo N by computing $[\![(Nx_m)_0]\!] [\![x_0]\!], \ldots,$ $[\![(Nx_m)_{\ell(N)-1}]\!] [\![x_{\ell(N)-1}]\!]$

3.9 Comparisons

In this section we will discuss several comparison protocols.

3.9.1 Schoenmakers and Tuyls

Schoenmakers and Tuyls present a comparison protocol in [ST04] for two bit-wise encrypted input values. The following equation is used to compute the result:

$$\begin{aligned} t_0 &= 0 \\ t_{i+1} &= (1 - (x_i - y_i)^2)t_i + x_i(1 - y_i) \end{aligned} \tag{3.3}$$

Protocol 3.9.1 (COMPARE - Compare two encrypted integer values [ST04])**.** Given two bit-wise encrypted values $[\![x]\!]$ and $[\![y]\!]$, the following protocol outputs $[\![1]\!]$, if $x > y$, and $[\![0]\!]$ otherwise. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following computation:

1. The participants assign $[\![t_0]\!] \leftarrow [\![0]\!]$
2. The participants jointly compute $[\![t_{i+1}]\!] \leftarrow [\![(1 - (x_i - y_i)^2)t_i + x_i(1 - y_i)]\!]$, for $i \in \{0, \ldots, \ell(x)-2\}$
3. The participants output $[\![t_{\ell(x)-1}]\!]$

If the protocol is performed by only two parties, and each party knows x or y, only private multiplications are needed. In this case, $2m - 1$ private multipliers are needed. If the input values would be private to both parties, then the private multipliers would need to be replaced by conditional gates.

3.9.2 Garay et al.

Garay et al. introduce a variant of Protocol 3.9.1, where the depth of the protocol is reduced [GSV07]. Instead of a linear depth, the new protocol only has a logarithmic depth. The number of multiplications is however increased.
The protocol makes use of the following observation. Assume we have two values x and y, that represented by bit strings. We divide the bit strings to get $x = X_1X_0$ and $y = Y_1Y_0$. Now the following equation holds:

$$[x > y] = [X_1 > Y_1] + [X_1 = Y_1][X_0 > Y_0]$$

Let $t_{i,j}$ denote the value for $>$, for the bit strings $x_{i+j-1}, \ldots, x_i$ and $y_{i+j+1}, \ldots, y_i$. For the value of $=$, we use $z_{i,j}$ for the bit strings $x_{i+j-1}, \ldots, x_i$ and $y_{i+j+1}, \ldots, y_i$. Now $[x > y]$ is given by $t_{0,m}$, which can be computed using the following equations:

$$t_{i,j} = \begin{cases} x_i - x_i y_i & j = 1 \\ t_{i+l,j-l} + z_{i+l,j-l} t_{i,l} & j > 1 \end{cases} \tag{3.4}$$

$$z_{i,j} = \begin{cases} 1 - x_i - 2x_i y_i - y_i & j = 1 \\ z_{i+l,j-l} z_{i,l} & j > 1 \end{cases} \tag{3.5}$$

Protocol 3.9.2 (COMPARE - Compare two encrypted integer values [GSV07])**.** Given two bit-wise encrypted values $[\![x]\!]$ and $[\![y]\!]$, the following protocol outputs $[\![1]\!]$, if $x > y$, and $[\![0]\!]$ otherwise. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following computation:

1. The participants jointly compute the output $[\![t_{t_{0,m}}]\!]$ using Equations 3.4 and 3.5, and the MULT protocol.

3.9.3 Damgård [DGK08]

In the protocol by Damgård et al. the input values are shared instead of encrypted [DGK08]. First we give a protocol to compute the XOR of two shared bits.

Protocol 3.9.3 (XOR - Compute the XOR of two shared bits)**.** Given two shared bits x and y, the following protocol computes $x \oplus y = x + y - 2xy$. The shares of $\mathcal{P}_A$ are denoted by x_A and y_A, and the ones of $\mathcal{P}_B$ by x_B and y_B. The participants $\mathcal{P}_A$ and $\mathcal{P}_B$ perform the following computation:

1. $\mathcal{P}_A$ sends x_A in encrypted form $[\![x_A]\!]$ to $\mathcal{P}_B$
2. $\mathcal{P}_B$ computes $[\![x_A]\!]^{y_B} [\![r_B]\!]$, where r_B is a random value
3. $\mathcal{P}_B$ sends $[\![x_A y_B + r_B]\!]$ back to $\mathcal{P}_A$
4. $\mathcal{P}_B$ sends x_B in encrypted form $[\![x_B]\!]$ to $\mathcal{P}_A$
5. $\mathcal{P}_A$ computes $[\![x_B]\!]^{y_A} [\![r_A]\!]$, where r_A is a random value
6. $\mathcal{P}_A$ sends $[\![x_B y_A + r_A]\!]$ back to $\mathcal{P}_B$
7. $\mathcal{P}_A$ decrypts $[\![x_A y_B + r_B]\!]$ and compute his share xy, as $xy_A = x_A y_A + x_A y_B + r_B - r_A$
8. $\mathcal{P}_B$ decrypts $[\![x_B y_A + r_A]\!]$ and compute his share xy, as $xy_B = x_B y_B + x_B y_A + r_A - r_B$
9. $\mathcal{P}_A$ computes his share of $x \oplus y$ by computing $x_A + y_A - 2xy_A$
10. $\mathcal{P}_B$ computes his share of $x \oplus y$ by computing $x_B + y_B - 2xy_B$

Using this protocol we can construct the actual comparison protocol.

Protocol 3.9.4 (COMPARE - Compare two shared values [DGK08])**.** Given two shared bit-wise values $x = x_A + x_B$ and $y = y_A + y_B$, the following protocols outputs a shared bit $[x > y]$. Participant $\mathcal{P}_A$ has the shares x_A and y_A and participant $\mathcal{P}_B$ the shares x_B and y_B. The participants $\mathcal{P}_A$ and $\mathcal{P}_B$ perform the following computation:

1. The participants compute together a shared random bit b
2. Each participant individually computes $x = x + (y - x)b$ and $y = y - (y - x)b$, for their respective shares of x and y
3. The participants compute $w_i = x_i \oplus y_i$ by performing the XOR protocol for the bits $i = 1, \ldots, \ell$
4. The participants individually compute their shares of $c_i = x_i - y_i + 1 + \sum_{j=i+1}^{\ell} w_j$, for the bits $i = 1, \ldots, \ell$

5. Participant $\mathcal{P}_A$ sends an encryption of his shares $[\![\alpha_i]\!]$ of c_i to participant $\mathcal{P}_B$, for the bits $i = 1, \ldots, \ell$

6. Participant $\mathcal{P}_B$ computes, using his shares β_i of c_i, $\gamma_i = ([\![\alpha_i]\!]\,[\![\beta_i]\!])^{s_i} = [\![(\alpha_i + \beta_i)s_i]\!]$, for the bits $i = 1, \ldots, \ell$

7. Participant $\mathcal{P}_B$ sends γ to $\mathcal{P}_A$ in randomly permuted order

8. Participant $\mathcal{P}_A$ uses his secret key to decide whether any $\gamma_i = 0$, for the bits $i = 1, \ldots, \ell$

9. If there was a $\gamma_i = 0$, then the participants jointly compute the output $1 \oplus b$

10. Otherwise the participants jointly compute the output $0 \oplus b$

3.10 Vector operations

3.10.1 Secure selection from a vector

We have an encrypted vector $[\![V]\!]$, from which we want to retrieve the value of the i-th component. i is given as an encrypted Dirac unary value, i.e. $[\![I]\!]_{\mathbb{U}}$.
We can retrieve the value by calculating the inner product of the two vectors, as can be seen in Protocol 3.10.1.

Protocol 3.10.1 (SELECT - Secure selection from a vector, using the inner product)**.** Given an encrypted vector $[\![V]\!]$ and an encrypted index in unary notation $[\![I]\!]_{\mathbb{U}}$, the following protocol outputs the encrypted value of I-th component of vector $[\![V]\!]$. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following steps:

1. The participants jointly compute $[\![s_i]\!] \leftarrow \text{MULT}([\![v_j]\!], [\![i_j]\!])$, for $i = 1, \ldots, |V|$

2. Each participant individually computes the output $\prod_{i=1}^{|V|} [\![s_i]\!]$

For this method we need $|V|$ secure multiplications. Since all multiplications are independent from each other, they can be performed in parallel. Depending on the protocol that is used, the round complexity is different. If we use the Paillier cryptosystem, we can use the MULT protocol. Since all multiplications can be performed in parallel, the round complexity of the SELECT protocol is the same as for the MULT protocol, i.e. $O(1)$. If the ElGamal cryptosystem is used, the CONDITIONAL-GATE protocol is used. This results in a round complexity of $O(P)$.
A second method to select the value from the vector would be by making use of mixing.

Protocol 3.10.2 (SELECT - Secure selection from a vector, using mixing)**.** Given an encrypted vector $[\![V]\!]$ and an encrypted index in unary notation $[\![I]\!]_{\mathbb{U}}$, the following protocol outputs the encrypted value of I-th component of vector $[\![V]\!]$. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following steps:

1. The participants are arranged in a ring, such that every participant has a predecessor and a successor.

2. Every participant individually mixes the vectors $[\![V]\!]$ and $[\![I]\!]$, using the same permutation for both of them, and forwards the mixed vectors to his successor

3. After all participants mixed the vectors they jointly decrypt the vector $[\![I]\!]$

4. Each participant can now individually select the element from $[\![V]\!]$ corresponding to the component that contained 1 in the decrypted vector I

Because the index vector contains only one value (namely $[\![1]\!]$), full mixing is not needed. Mixing based on rotation is enough to prevent the inputs to be linked to the output. In [dHSSV09], De Hoogh et al. give a method to provide an efficient zero-knowledge proof for the correctness of the rotation.
Using this method, no secure multiplications are required. If an active adversary would be considered, a zero knowledge proof is needed to prove that the permutation was correctly performed. After every peer mixes the values he has to blind them again. We need to do $|V|$ threshold decryptions at the end to decrypt the index value. The round complexity of this method is $O(P)$. The difference between the two selection methods is mainly in the distribution of the work. In Protocol 3.10.1, every party takes part in every round and also communicates in every round. In Protocol 3.10.2, every party only has to take part in one round for the mixing and in the final round to decrypt the index vector. The workload per round in Protocol 3.10.2 is however more computational expensive than a round in Protocol 3.10.1.

3.10.2 Secure updating of an encrypted vector

We want to set the $[\![i]\!]_{\mathbb{U}}$-th component of the vector $[\![V]\!]$ to $[\![x]\!]$. To achieve this, we present again two methods. First, component-wise multiplication of the vectors can be used.
To set the value $[\![x]\!]$ at the index that is depicted by unary value $[\![i]\!]_{\mathbb{U}}$. To do this, first we need a vector that contains the value $[\![x]\!]$ at the intended component. This vector can be calculated by the multiplication of the scalar value $[\![x]\!]$ with the vector $[\![i]\!]_{\mathbb{U}}$. Secondly, in the vector $[\![V]\!]$, the value at the index $[\![i]\!]_{\mathbb{U}}$ needs to be $[\![0]\!]$. This is done by multiplying $[\![V]\!]$ with the bit-wise complement of $[\![i]\!]_{\mathbb{U}}$.

Protocol 3.10.3 (UPDATE - Secure updating of a vector, using multiplications)**.** Given the vectors $[\![V]\!]$ and $[\![I]\!]_{\mathbb{U}}$, and the single value $[\![x]\!]$, the following protocol outputs the vector $[\![V']\!]$, where the $[\![i]\!]_{\mathbb{U}}$-th component contains $[\![x]\!]$ and the other components the corresponding values from $[\![V]\!]$. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following steps:

1. The participants jointly compute $[\![A]\!] \leftarrow \text{MULT}([\![x]\!], [\![i]\!]_{\mathbb{U}})$, to get the vector that contains $[\![x]\!]$ at the $[\![i]\!]_{\mathbb{U}}$-th component
2. The participants together compute $[\![C]\!]$ by performing the BIT-COMP protocol on $[\![i]\!]_{\mathbb{U}}$
3. The participants jointly compute $[\![B]\!] \leftarrow \text{MULT}([\![C]\!], [\![V]\!])$, to get the vector $[\![V]\!]$, where the i-th component contains $[\![0]\!]$
4. Each participant individually computes component-wise $[\![A]\!]\,[\![B]\!]$ to get the output of the protocol

A second method for secure updating would be by using mixing. First, the value at the index is retrieved using the method described in Protocol 3.10.2. Then the encrypted value is replaced by the new value x and the process is repeated backwards, thus reversing the permutations.

Protocol 3.10.4 (UPDATE - Secure updating of a vector, using mixing)**.** Given the vectors $[\![V]\!]$ and $[\![I]\!]_{\mathbb{U}}$, and the single value $[\![x]\!]$, the following protocol outputs the vector $[\![V']\!]$, where the $[\![I]\!]_{\mathbb{U}}$-th component contains $[\![x]\!]$ and the other components the corresponding values from $[\![V]\!]$. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following steps:

1. The participants are arranged in a ring, such that every participant has a predecessor and a successor.
2. Every participant i individually mixes the vectors $[\![V]\!]$ and $[\![I]\!]$, using the permutation π_i, and forwards the mixed vectors to his successor
3. After all participants mixed the vectors they jointly decrypt the vector $[\![I]\!]$

4. The last participant sets the component in $[\![V]\!]$, corresponding to the component that contained 1 in the decrypted vector I, to $[\![x]\!]$

5. Every participant individually performs the permutation π_i^{-1} on $[\![V]\!]$, and forwards the mixed vector to his predecessor

6. The first participants outputs $[\![V]\!]$ after he applied the permutation π_1^{-1}

The UPDATE-ADD protocol is the same as the UPDATE, except that it adds the value x to the old value instead of replacing it.
Both methods for updating require double the amount of round and communication of the corresponding selection methods.

3.10.3 Selecting the index of the maximum value

The protocol MAX-IND returns, on input of a component-wise encrypted vector $[\![V]\!]$ of size l, the index $[\![i]\!]$ of the maximum value $[\![V_i]\!]$ in the encrypted vector $[\![V]\!]$.

Protocol 3.10.5 (MAX-IND - Select the index of the maximum value from vector $[\![V]\!]$)). Given an encrypted vector $[\![V]\!]$, the following protocol outputs the index of the maximum value in $[\![V]\!]$. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following steps:

1. Each participant individually performs the assignment $M_0 \leftarrow [\![V]\!]$

2. The participants jointly construct the vector L_0 such that it contains the values of the indices of the vector, e.g. if decimal indices are used $L_0 \leftarrow ([\![0]\!], \ldots, [\![|V| - 1]\!])$

3. Repeat the following steps $\log |V|$ times. After each round i, the vectors M_i and L_i contain the output of that round.

   (a) The participants jointly perform the COMPARE protocol to get $b \leftarrow [\![[M_{i-1,2j} > M_{i-1,2j+1}]]\!]$, for $j = 1, \ldots, |V|/(2^i)$

   (b) The participants jointly perform the COND-SEL protocol to get $M_{i,j}$ on the inputs $(b, M_{i-1,2j}, M_{i-1,2j+1})$, for $j = 1, \ldots, |V|/(2^i)$

   (c) The participants jointly perform the COND-SEL protocol to get $L_{i,j}$ on the inputs $(b, L_{i-1,2j}, L_{i-1,2j+1})$, for $j = 1, \ldots, |V|/(2^i)$

4. Each participant selects the encrypted value $L_{\log |V|,1}$, that is the encrypted index of the largest component in vector $[\![V]\!]$

The comparisons are performed in a tree-like fashion. This results in the halving of the number of values after each round in the protocol. Therefore, the total number of comparisons needed in the protocol is $O(|V|)$.
The round complexity of the MAX-IND protocol is $O(\log |V|)$, for the Paillier cryptosystem. For the ElGamal cryptosystem, the round complexity is $O(P \log |V|)$.
The MAX2-IND protocol is an extension of the MAX-IND protocol to return the indices of the two largest components.
This can be done by performing $max([\![V]\!])$ two times. After the first time, you remove the maximum value from $[\![V]\!]$. Which can be done by multiplying $[\![V]\!]$ with the complement of the index of the maximum.

3.10.4 Distance

To calculate the distance between two vectors, the Euclidean distance can be used. Since this involves taking the square root of a value, it is more practical to use the squared Euclidean

distance in calculations in the encrypted domain. The squared Euclidean distance for vectors $V = (v_1, \ldots, v_n)$ and $W = (w_1, \ldots, w_n)$ is defined as:

$$\text{DIST}(V, W) = \sum_{i=1}^{n}(v_i - w_i)^2 = \sum_{i=1}^{n}(v_i^2 + w_i^2 - 2v_i w_i) \tag{3.6}$$

In [JW05], Jagannathan and Wright present a method to calculate the squared Euclidean distance, where one vector V is shared components wise and for vector W, the values of the components are shared. In the protocol, homomorphic encryption is only used in the SCALAR-PRODUCT protocol.

Protocol 3.10.6 (SHARED-DIST - Calculate the distance between two vectors, Jagannathan and Wright [JW05])**.** Given two vectors V and W the following protocol outputs the squared distance between the vectors. Without loss of generality, participant $\mathcal{P}_A$ knows the values $v_1, \ldots, v_m$ and $\mathcal{P}_B$ knows the values $v_{m+1}, \ldots, v_n$. For vector W, $\mathcal{P}_A$ and $\mathcal{P}_B$ know respectively the values w_i^A and w_i^B, such that $w_i = w_i^A + w_i^B$, for $i = 1, \ldots, n$. The participants $\mathcal{P}_A$ and $\mathcal{P}_B$ perform the following steps:

1. Participant $\mathcal{P}_A$ individually calculates $sv^A \leftarrow \sum_{i=1}^{m} v_i^2$
2. Participant $\mathcal{P}_A$ individually calculates $sw^A \leftarrow \sum_{i=1}^{n}(w_i^A)^2$
3. Participant $\mathcal{P}_B$ individually calculates $sv^B \leftarrow \sum_{i=m+1}^{n} v_i^2$
4. Participant $\mathcal{P}_B$ individually calculates $sw^B \leftarrow \sum_{i=1}^{n}(w_i^B)^2$
5. The participants jointly compute the following using a SHARED-SCALAR-PRODUCT protocol, $c^A + c^B = 2\sum_{i=1}^{n}(w_i^A w_i^B)$, $d^A + d^B = -2\sum_{i=1}^{n}(w_i^A v_i)$, $e^A + e^B = -2\sum_{i=1}^{n}(w_i^B v_i)$
6. Each participant individually computes $sv^P + sw^P + c^P + d^P + e^P$, for $P \in \{A, B\}$, to get their share of the distance

In the next protocol we want to calculate the encrypted distance between two vectors V and W, that are known by participants $\mathcal{P}_A$ and $\mathcal{P}_B$ respectively.

Protocol 3.10.7 (DIST - Calculate the distance between two vectors)**.** Given two vectors $V = (v_1, \ldots, v_n)$ and $W = (w_1, \ldots, w_n)$, known by participants $\mathcal{P}_A$ and $\mathcal{P}_B$ respectively, the following protocol outputs the encrypted squared Euclidean distance. The participants $\mathcal{P}_A$ and $\mathcal{P}_B$ perform the following steps:

1. Participant $\mathcal{P}_A$ individually computes $S_v \leftarrow [\![\sum_{i=1}^{n} v_i^2]\!]$, and sends S_v to $\mathcal{P}_B$
2. Participant $\mathcal{P}_B$ individually computes $S_w \leftarrow [\![\sum_{i=1}^{n} w_i^2]\!]$, and sends S_w to $\mathcal{P}_A$
3. The participants jointly perform the SHARED-SCALAR-PRODUCT protocol in V and W to get $S_{vw} \leftarrow [\![\sum_{i=1}^{n}(v_i w_i)]\!]$
4. The participants individually compute the output $S_v S_w S_{vw}^{-2}$

This protocol can be extended to n parties for two encrypted vectors $[\![V]\!]$ and $[\![W]\!]$.

Protocol 3.10.8 (DIST - Calculate the distance between two encrypted vectors)**.** Given two vectors $[\![V]\!] = ([\![v_1]\!], \ldots, [\![v_n]\!])$ and $[\![W]\!] = ([\![w_1]\!], \ldots, [\![w_n]\!])$ the following protocol outputs the encrypted squared Euclidean distance. The participants $\mathcal{P}_i$ $(i = 1, \ldots, n)$ perform the following steps:

1. The participants jointly perform the SCALAR-PRODUCT protocol to get $S_v \leftarrow [\![\sum_{i=1}^{n} v_i^2]\!]$
2. The participants jointly perform the SCALAR-PRODUCT protocol to get $S_w \leftarrow [\![\sum_{i=1}^{n} w_i^2]\!]$
3. The participants jointly perform the SCALAR-PRODUCT protocol to get $S_v w \leftarrow [\![\sum_{i=1}^{n}(v_i w_i)]\!]$
4. Each participant individually computes the output $S_v S_w S_{vw}^{-2}$

3.11 Set operations

In this section we will discuss various secure operations on sets. These operations will be used in the clustering algorithm presented in Chapter 6.
Oblivious sets can be represented using bit-arrays. For each possible element in the set there is a bit that indicates whether the element is in the set or not. Set S would be represented by vector $[\![V]\!]$ using the following:

$$[\![V_i]\!] = \begin{cases} [\![1]\!] & \text{if } i \in S \\ [\![0]\!] & \text{otherwise} \end{cases}$$

To test whether an element is part of a set, the corresponding component can be decrypted. If it is 1, the element is part of the set, and otherwise it is not. To add an element to a set, the corresponding component is set to 1.
Adding an element to a set can be done by setting the corresponding bit to $[\![1]\!]$. Removing an element can be done by setting the corresponding bit to $[\![0]\!]$. Checking whether an element is in the set can be done by taking the corresponding bit from the vector, if the element is known. If the element is not known, the secure selection method, described before, can be used.
The union of two sets $[\![A]\!]$ and $[\![B]\!]$ can be calculated as follows:

$$[\![A]\!] \cup [\![B]\!] = [\![A]\!] \,\text{MULT}(\text{COMP}([\![A]\!]), [\![B]\!])$$

The multiplication of $[\![B]\!]$ and the complement of $[\![A]\!]$, is to ensure that the values, for all elements that occur in $[\![A]\!]$, become $[\![0]\!]$. This way, the addition of $[\![A]\!]$ will not result in values other than $[\![0]\!]$ and $[\![1]\!]$. The communication complexity of this operation is $O(P|A|k)$ and the round complexity is $O(P)$, for P parties.
The intersection of sets $[\![A]\!]$ and $[\![B]\!]$ is given by the following:

$$[\![A]\!] \cap [\![B]\!] = \text{MULT}([\![A]\!], [\![B]\!])$$

The elements-wise multiplication will only result in $[\![1]\!]$ if an element occurs in both $[\![A]\!]$ and $[\![B]\!]$. The communication and round complexity of this operation is the same as for the union.
The difference between sets can be expressed as $A \setminus B = A \cap \overline{B}$, where $\overline{B}$ is the complement of B. This can also be used to calculate the difference of $[\![A]\!]$ and $[\![B]\!]$ using $[\![A]\!] \setminus [\![B]\!] = \text{MULT}([\![A]\!], \text{COMP}([\![B]\!]))$. Again, the communication and round complexity is the same as for the union and the intersection.

4. Reputation systems

If a user wants to perform a transaction with a service provider, he would like to know the quality of this service provider. A service provider could, for example, be a seller in an online marketplace or a physician. To help in assessing the quality of a service provider, we can use reputation systems. In reputation systems, the user can submit ratings about previous transactions. Based on these ratings, a reputation is calculated for the service provider.
Besides providing users a measure to help in selecting a service provider of good quality, a reputation system might also serve other purposes. It might also, for example, promote good behavior among service providers. If they do not provide good service, they will get a bad reputation and therefore less clients.
A reputation system can consist of the following entities:

- User
- Provider
- Peers
- Trusted party

The *User* wants to retrieve the reputation *score* of the *Provider* to decide whether he wants to perform a transaction with this entity. The *Provider* could be either another user (e.g. a seller) or an external entity (e.g. a physician). The *Peers* are other users of the system who may have had interactions with the *Provider* before. Based on these previous interactions, these *Peers* can send *votes*, which contain a rating about the previous transactions with the *Provider*, to the *User* or a *Trusted party*. The *Trusted party* is a party that all users trust to perform certain actions, such as, for example, storing votes. This party is not always part of a reputation system, i.e. if everything is done in a distributed way, the trusted party is not necessary. Communication can take place between all the different entities.

4.1 Metric

The core of a reputation system is its metric. When constructing a metric, there are a lot of possibilities. First, the type of the ratings has to be decided, e.g. the user can only say whether a transaction was successful or not, or the user can give a rating between 1 and 10. The same holds for the reputation score, this can, for example, be a single value or a vector containing the distribution of the ratings.
The source of information must to be also determined. Trust can be transitive, where also indirect experiences are taken into account, as opposed to only personal experiences. Time can also be taken into account, i.e. all ratings are weighted equally, independent of the time since they were generated, or newer ratings can have a higher weight.
Also, the party performing the calculation has to be determined. If the value of the votes has to stay private but the voter may be known, this has to be done either by a trusted party or by using secure multi-party computation. If the votes cannot be linked to peers, the target could also calculate his own score, but then he should provide a proof that his calculation was correct.
In the next section we will describe different types of metrics.

4.1.1 Distribution of scores

The reputation score can be a distribution of the values of the votes. This can be absolute, so the occurrences of the possible values are counted. The alternative is a relative distribution, where the percentage is given for the occurrences of the possible values.

4.1.2 Average

If the votes are numerical, the average of the votes can be calculated. In calculation of the average, also weights could be taken into account. Weights might depend on the voter, e.g. trusted peers get higher weight than unknown peers, or on the vote, e.g. votes with high values have a higher weight than votes with a lower value or recent votes get a higher weight.

4.1.3 Bayesian

Bayesian reputation systems make use of statistical calculations, like the beta probability density function, to calculate the reputation score [JsI02].
Bayesian systems can be divided in binomial reputation systems, i.e. only allowing two possible values (e.g. *good* and *bad*), or multinomial reputation systems, i.e. allowing multiple discrete levels, (e.g. *excellent*, *good*, *average* and *bad*) [JQ09].
The reputation score can be a list of probabilities, containing the expectation that a particular value will occur. Another possible reputation score can be an estimated value. This estimated value can be calculated by converting the different levels to corresponding numerical values and calculate the estimated reputation.

4.2 Privacy

When using a reputation system, users might not want their ratings to be public. If ratings are not private this might lead to negative actions by the service provider in case of bad ratings.
Users might also want to hide, with which service providers they had transactions with. For example, this could be the case when users do not want to reveal they visited a certain physician.
Next we will discuss two possible attacks on the anonymity of the users in a reputation system.

4.2.1 Δ-attack

With the Δ-attack the adversary tries to recover the value of a user's vote. If the adversary knows a target's reputation before and after a user voted, he might be able to calculate the individual vote of the user. To reduce this attack, votes could only be added in batches or randomization might be used in the calculation of the reputation score (e.g. dropping some random votes or adding noise to the reputation). When working with batches of size n, the adversary might try to add $n - 1$ votes to still be able to calculate the vote of the user.

4.2.2 Intersection attack

With the intersection attack the adversary again tries to learn the value of a vote. The attack is applicable if the users store their own votes about the targets they interacted with. If the adversary knows the group of users whose votes are used in calculation of the reputation score and this group is not static, he can retrieve the reputation score multiple times for different groups. Using these groups and their corresponding scores, he can try to reconstruct the votes of individual users.
If the adversary can choose the groups of users, that provide their votes, he can target an individual user. He requests the reputation score from a group including the user and from the same group excluding the user. Using these two scores he can calculate the vote of the individual user.

A possible countermeasure could be to not allow users to choose which peers provide votes. Instead this could be determined by some predefined method, such as, for example, pick all similar users, as with the knots-based reputation system, or pick a random subset of peers.

4.3 Existing systems

In this section we will discuss two existing reputation systems. In the next chapters we will provide privacy extensions to both systems.

4.3.1 Knot-aware reputation model

Gal-Oz et al. introduce a knot-aware reputation model, where users are divided into clusters based on their votes [GOGH08]. This results in knots of peers with similar votes. The user asks votes from peers in his cluster about the target to compute the reputation score. If there is no information within the knot, the complete system is polled for information. The idea behind this method, is that if users have similar votes for certain peers, they probably also think similar about other peers.
In the knot-aware model, users are encouraged to cast votes, as this results in becoming part of a knot, that contains users with similar opinions. Being part of a knot results in more tailored reputation scores for the user.
This system is particularly useful if the reputation is subjective. For example, different users might have different expectations from an expert depending on their own knowledge.
For the algorithm, the users are represented by vertices in a graph. The edges in the graph denote the similarity between the votes of the users. To cluster the users, the algorithm by Edachery et al. for distance-k cliques is used [ESB99]. With distance-k cliques, the distance between any two nodes in a cluster is at most k. This means that every node in the cluster is reachable by following at most k edges.
In the process of clustering the users, a lot of information is leaked about votes by the individual users. In the computation of the similarity, to determine whether there is an edge between two vertices, both users have to reveal all their votes. To provide better privacy, we will introduce oblivious clustering algorithms in Chapter 6, that can be computed using secure multi-party computation.

4.3.2 P2PRep

The P2PRep reputation system was introduced by Aringhieri et al. [ADD+06]. In the P2PRep system, there is a distinction between two different types of reputation. First there is local reputation. Local reputation is based only on personal experiences. The other type of reputation, is called network reputation. Network reputation is based on the experiences of other users.
Local reputation is used if the user has previous experiences with the target. The local reputation gives a higher weight to recent transactions, based on the order of occurrence. Absolute time is not taken into account to determine this weight. The local reputation R_j^n is calculated as follows, where v_j^n is the n-th vote for target j:

$$R_j^n = \begin{cases} v_j^n & \text{if } n = 1 \\ \alpha R_j^{n-1} + (1-\alpha) v_j^n & \text{otherwise} \end{cases}$$

α determines the decay of the old votes. If α is higher, the old votes have a higher weight in the reputation score.
If the user has no previous transaction with the target, global reputation is used. To determine the global reputation, the Ordered Weighted Average is used. The metric gives a higher weight to bad ratings and ratings that occur more often. The reputation R_j is defined as:

$$R_j = \frac{\sum_{x=1}^{d+1} \frac{x}{d+2} D_x |D_x|}{\sum_{x=1}^{d+1} \frac{x}{d+2} |D_x|}$$

There are d different values in the votes. D_x is the value of the x^{th} highest rating and $|D_x|$ is the number of votes received with rating D_x. D_{d+1} is the local trust value, that is based on previous personal transactions and therefore will have the highest weight. $\frac{x}{d+2}$ is the weight that is used for the values, and is higher for the lower votes.

4.4 Privacy in existing systems

In this section we will discuss some of the work that has been done on privacy in reputation systems.

4.4.1 3PRep

The 3PRep reputation system is an extension of the P2PRep system by Nithyanand and Raman [NR09]. Using homomorphic encryption the authors try to provide more privacy for the users.
A notion of trusted peers is introduced and a homomorphic cryptosystem is used. The process of calculating the network reputation score is now as follows:

- The peers send their votes, encrypted using the key of a trusted peer, to the user.
- The user calculates the encrypted differences of all votes (i.e. $\{[\![v_i]\!] [\![v_j]\!]^{-1} | 0 \leq i < j < n\}$ for votes $v_0, \ldots, v_{n-1}$).
- The user permutes the encrypted differences and sends them to the trusted peer.
- The trusted peer decrypts the differences and returns for each difference whether it is smaller, equal or larger than 0.
- Using the information from the trusted peer, the user sorts the encrypted values.
- The user computes the global reputation using the homomorphic properties of the encryption.
- The result of the computation sent to the trusted peer, who decrypts it and returns the result.

Remarks The proposed scheme is not suitable in the presence of a malicious user. Below two possible attacks are given that the user, requesting the vote, can perform by deviating from the protocol.
Because the user sorts the encrypted values himself, he can learn information about the value of the votes. He can insert his own vote into the list of encrypted votes and see which votes are higher, equal or lower than his own vote. Using binary search he can determine the exact value of the votes. If he is interested in the peers whose vote is below a certain value, he can insert a vote with this value and after the sorting he knows which votes are lower than his inserted vote.
Another possible attack can occur in the last phase. The user sends a single value to be decrypted to the trusted peer, but the origin of this value is not checked in any way. It is therefore also possible to send a single vote to be decrypted. This could be solved by sending a proof that the value to be decrypted is actually the sum of all received votes, i.e. it is the multiplication of all encrypted votes.
In Chapter 5, we will discuss other possible methods to sort the values.

4.4.2 Methods for computing trust and reputation while preserving privacy

In [GGOG09], Gudes et al. give three protocols to compute a weighted average while preserving the privacy of the votes. In the last two protocols, the weight is also private. This might be desired if, for example, the weight is based on the trust the requester has in the voter.

4.4.2.1 Protocol 1

In the first protocol the weight for the participant's votes is disclosed to the corresponding participants. A trusted party is used to anonymize the weighted votes.

1. The requesting user sends to every participant their corresponding weight.
2. Every participant multiplies its vote with the weight and encrypts it using the public key of the requesting user.
3. The participants send their encrypted values to the trusted party.
4. The trusted party shuffles the encrypted values and sends them to the requesting user.
5. The requesting user decrypts the value and computes the weighted average.

Remarks When using a homomorphic cryptosystem, the trusted party would already be able to calculate the encrypted sum. This could be done by multiplying the ciphertexts. Since he cannot decrypt the votes, he would not learn the individual votes this way. This modification would reduce the communication overhead, as the individual votes need not be sent to the requester. However, the computational cost for the trusted party increases.
Using a homomorphic cryptosystem, such as Paillier, the weight could also be kept private [Pai99]. The weight could be sent in encrypted form to the users. Using the private multiplication they could then compute the multiplication of the weight with their vote, without having to know the actual weight. Doing this however introduces a new possible attack. A dishonest requesting user could, for example, send to one selected user the weight 1 and to the other users the weight 0. This way the final reputation score will then be equal to the vote of the selected user.

4.4.2.2 Protocol 2

In the second protocol the weights are no longer disclosed to the participants. Again a trusted third party is used to provide privacy on the votes.

1. The participants encrypt their votes with the public key of the requesting user, and send them to the trusted party.
2. The trusted party randomly permutes the received encrypted votes and sends them to the requesting user.
3. The requesting user decrypts the votes.
4. The requesting user sends the weights to the trusted party.
5. The trusted party performs the same random permutation on the weights, as he did on the votes.
6. The trusted party and the requesting user compute together the scalar product for the votes and the weights, using secure multi-party computation, to get the weighted sum.

Remarks Since the trusted party knows the encrypted votes and the corresponding weights, he can already compute the weighted sum if a homomorphic cryptosystem is used.

$$\begin{aligned}
&\prod_{i \in Votes} (\llbracket Votes_i \rrbracket^{Weights_i}) \\
&= \prod_{i \in Votes} (\llbracket Weights_i * Votes_i \rrbracket) \\
&= \left\llbracket \sum_{i \in Votes} (Weights_i * Votes_i) \right\rrbracket
\end{aligned}$$

This way, no secure multi-party computation is needed anymore, which reduces the communication overhead.
To also keep the weights private, the encryption scheme by Boneh et al. could be used in the calculation [BGN05]. This scheme has homomorphic properties, however it does not only allow additions under the encryption, but also one multiplication. In this case, the weight could also be encrypted by the user, so the trusted party could calculate the reputation without interaction with the user and without learning the weights.

4.4.2.3 Protocol 3

The third protocol is completely distributed, so no trusted third party is needed. To hide the actual weights, the requesting user adds a random value to all weights, to randomize the weighted sum.

1. The sum of all votes is computed using secure multi-party computation ($\sum_i v_i$).
2. The participants are arranged in a random sequence.
3. The requesting user generates a random value r.
4. The requesting user sends to every participant his corresponding weight added with the random value r, $(w_i + r)$.
5. Every participant computes his weighted vote $(v_i(w_i + r))$.
6. The first participant sends his weighted vote to the second participant.
7. The other participants receive the value from the previous participant, add their own weighted vote and send it to the next participant.
8. The last participant sends the sum to the requesting user $(\sum_i v_i(w_i + r))$.
9. Using the received sum and the sum computed in the first phase, the requesting user computes the weighted sum $(\sum_i v_i(w_i + r) - r \sum_i v_i)$.

Remarks To increase privacy, homomorphic encryption could be used when calculating the randomized weighted sum, thus encrypting the intermediate results.

4.4.3 Efficient Private Multi-Party Computations of Trust

In [DGK09], Dolev et al. present efficient protocols for distributed calculation of sums and weighted sums. The first three algorithms are resistant to honest-but-curious users. The last two algorithms should also be resistant to k-listening curious adversaries. A k-listening curious adversary can compromise up to k users, where k is less than the total number of users. The protocols make use of a homomorphic cryptosystem. In the first three protocols the values are encrypted using the key of the requesting user.

4.4.3.1 Protocol 1

The first protocol computes the sum of the votes, which can be used to calculate the average. The user initializes the protocol by sending the encrypted value 1 to the first peer. This peer multiplies the received value with his own encrypted vote (thus adding values under the encryption). He sends this value to the next peer, who also multiplies the received value with his own encrypted vote. This is repeated until the last peer sends the encrypted sum to the user who can then decrypt it.

4.4.3.2 Protocol 2

The second algorithm is a variation of the first that also takes weights into account. In the initialization, the user sends the encrypted weights to the peers. Now every peer i calculates $[\![w_i]\!]^{v_i}$, were w_i is the assigned weight and v_i the peer's vote, to get the encrypted multiplication of his own vote with his assigned weight. He blinds this encrypted value and multiplies it with the received value. In the end, the user has the sum of all votes, multiplied with their respective weights.

4.4.3.3 Protocol 3

In the third protocol the peers do not perform the calculation, but only shuffle the votes. There are two rounds in the protocol. In the first round, every peer enters his encrypted vote in a vector. The requesting user does not receive this vector during this round. At the end of this round, the last peer sends the vector to the first peer. In the second round each peer shuffles and blinds the vector and sends it to the next peer. The last peer sends the vector to the user who can then decrypt it, thus retrieving all votes in a random order.

4.4.3.4 Protocol 4

In the previous protocols all votes were encrypted with the key of the user. Therefore, if the user would collude with other peers, these would leak information. In the next protocol, every peer divides his vote into shares, such that every peer gets a share encrypted with his own key. These encrypted shares are multiplied to get the sum of all received shares for each peer and decrypted by the corresponding peer. The sum of the shares are again encrypted, this time using the requesting user's key, and the shares are added using the first protocol.

4.4.3.5 Protocol 5

The last protocol is a variation of the fourth protocol, that takes weight into account. The vote is again calculated the same way as in the second algorithm. However, this time a random bias is added to the vote. This bias is then split into a share for each peer, that is encrypted with the key of the corresponding peer. In the second round of the protocol, every peer decrypts all the shares intended for him and calculates the sum of these shares. This sum is again encrypted using the key of the user. The sums are again added like in the first algorithm. The user can calculate the correct sum by subtraction of the sum of the biased weighted votes by the sum of the biases.

5. Sorting

In this section we will present and introduce several oblivious sorting algorithms. First we introduce a new oblivious sorting algorithm that sorts values from a limited domain. Next we will discuss a randomized comparison-based sorting algorithm, introduce a variation and discuss the privacy issues concerning these algorithms.
Because the algorithms are oblivious, they can be applied on sequences of encrypted values. These values can then be sorted, without revealing any information, additional to the fact that the values in the output are sorted.

5.1 Counting sort

The first sorting algorithm we will discuss is counting sort. Counting sort is an efficient method to sort values from a limited domain. For each possible value, the number of occurrences is counted. Using these counts the sorted sequence is then constructed.

5.1.1 Sorting bits

First we consider the scenario where we want to sort values from a two-valued domain, such as for example bit values. In Algorithm 1, we introduce an oblivious algorithm that sorts n input bits. This is done by first counting the bits having value 1. This can easily be done by computing the sum of all bits. Next this sum is converted from integer to binary notation using Protocol 3.8.1. This results in $\ell(n)$ bits, $[[\![cnt_0]\!], \ldots, [\![cnt_{\ell(n)-1}]\!]]$.
Without loss of generality we assume that there are $n - cnt$ zeroes and cnt ones. We want to construct the sequence where we have first $n - cnt$ zeroes and then cnt ones. This is in fact a conversion of cnt from binary to a Heaviside unary notation, where n is the size of the unary notation.

Example 5.1.1. We want to sort the sequence $[[\![0]\!], [\![1]\!], [\![0]\!], [\![1]\!]]$. First the number of ones are counted, using the homomorphic property of the cryptosystem. This results in $[\![cnt]\!] = [\![2]\!] = [\![0]\!]\,[\![1]\!]\,[\![0]\!]\,[\![1]\!]$. $[\![cnt]\!]$ is now converted to binary notation, $[\![0]\!]\,[\![1]\!]\,[\![0]\!]$. Now we iterate the for-loop:

- We start with the sequence containing the least significant bit, $seq = [[\![0]\!]]$.
- We add twice the next bit, $seq = [[\![1]\!], [\![1]\!], [\![0]\!]]$
- A conditional-swap is performed on elements 0 and 2 in the sequence, $seq = [[\![0]\!], [\![1]\!], [\![1]\!]]$
- We add four times the next bit, $seq = [[\![0]\!], [\![0]\!], [\![0]\!], [\![0]\!], [\![0]\!], [\![1]\!], [\![1]\!]]$
- A conditional-swap is performed on the elements 0 and 2, and 1 and 3 in the sequence, $seq = [[\![0]\!], [\![0]\!], [\![0]\!], [\![0]\!], [\![0]\!], [\![1]\!], [\![1]\!]]$

The final output is now retrieved by taking the last four elements from the sequence, giving the sorted sequence $[[\![0]\!], [\![0]\!], [\![1]\!], [\![1]\!]]$.

In Figures 5.1 and 5.2 we give the CONDITIONAL-SWAP operations for the first two rounds.

Algorithm 1 Counting sort for bits

Input: $[[\![b_0]\!], \ldots, [\![b_{N-1}]\!]]$, $b_i \in \{0, 1\}$
  $[\![cnt]\!] \leftarrow \prod_{i=0}^{N-1} [\![b_i]\!]$
  $[\![cnt_0]\!], \ldots, [\![cnt_{\log N}]\!] \leftarrow convertToBinary([\![cnt]\!])$
  $seq \leftarrow [[\![cnt_0]\!]]$
  for $i = 1$ to $\lfloor \log N \rfloor$ **do**
    $seq \leftarrow [[\![cnt_i]\!]]^{2^i} \parallel seq$
    for $j = 0$ to $2^i - 2$ **do**
      $(seq_j, seq_{2^i+j}) \leftarrow conditional - swap(seq_j, seq_j, seq_{2^i+j})$
    end for
  end for
  return $[seq_{2^{\lfloor \log N \rfloor + 1} - 2 - N}, \ldots, seq_{2^{\lfloor \log N \rfloor + 1} - 2}]$

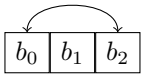

Figure 5.1: CONDITIONAL-SWAP operations for round 1

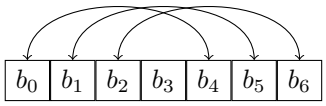

Figure 5.2: CONDITIONAL-SWAP operations for round 2

5.1.1.1 Complexity

The number of *swap* operations in Algorithm 1, and therefore the number of *conditional gates* that are needed, for $n > 0$, is:

$$\begin{aligned}\sum_{i=0}^{\lfloor \log n \rfloor} (2^i - 1) =& 2^{\lfloor \log n \rfloor + 1} - 1 - (\lfloor \log n \rfloor + 1) \\ =& 2^{\lfloor \log n \rfloor + 1} - \lfloor \log n \rfloor - 2 \\ \leq& 2^{\log n + 1} - \log(n) - 2 \\ <& 2n - \log(n)\end{aligned}$$

Algorithm 1 only uses $O(n)$ compare-exchange operations, where a comparison-based sorting algorithm needs at least $O(n \log n)$ compare-exchange operations.
We only need communication in the conversion to binary and the CONDITIONAL-SWAP operations.

5.1.1.2 Correctness

To prove the correctness of the protocol, first we prove that the correct number of ones is added to the output sequence. Then we prove that the output sequence is in sorted order and at last we prove that the sequence contains all ones.

Proof. For each bit at position i in *cnt* we add this bit 2^i times to the sequence. Therefore, the total number of ones that are added is equal to *cnt*. □

Next we use induction on the rounds to prove that the bits are sorted correctly.

Proof.
Case $i = 0$. Trivial.

Case $i > 0$. The sequence $[b_0, \ldots, b_{2^i-2}]$ is sorted. We add 2^i elements at the beginning of the sequence, that all have the same value, namely cnt_i. If the added elements have the value 0, the sequence stays the same, as the *swap* operation does not change the order of the elements in this case. If the added elements have the value 1, the *swap* operation does swap all the compared values. This means that all $2^i - 1$ previous values, are swapped to the front of the sequence. Since the previous sequence was already sorted, the zeroes that were at the front of the previous sequence, will again be at the front of the sequence. □

Finally, we prove that all the ones are actually output.

Proof. All ones are at the end of the sequence, as this is in sorted order. The sequence contains the correct number of ones, so it contains at most n ones. Since the last n elements are output, all ones are always output. □

5.1.1.3 Privacy

The underlying algorithm for our protocol is oblivious, so no information is leaked from the run of the algorithm. For the protocol we make only use of building blocks that are proven to preserve privacy. For both the COND-SWAP and the INT-TO-BIN protocols there exist simulators. Since the only communication in the protocol is done in these operations, a simulator can also be constructed for the complete protocol.

5.1.1.4 Lower bound

For every bit in the unsorted sequence there has to be at least one *conditional gate*, as the values are unknown and an action that depends on their value has to be performed.
If we use a bit-wise representation of the number of ones, we will need at least $2^{\lfloor \log n \rfloor} - 1$ *conditional gates*, for n bits.

5.1.2 Sorting in domains with > 2 elements

In this section we present an algorithm that sorts values from a domain of size t.
As input, n values in Dirac unary notation are given. We can now count the occurrences of each possible value by adding the corresponding encrypted bits. For the number of occurrences of the value with index i, we use the notation $[\![cnt_i]\!]$.
For each value with index i, use the counting sort for bits to calculate the sorted sequence with $\prod_{j=i}^{t-1} [\![cnt_j]\!]$ bits with the value $[\![1]\!]$. We denote the result of this sorting with $[\![ordseq_i]\!]$.
Using the calculated sequences we can calculate the following sequences: $[\![uniqseq_i]\!] = [\![ordseq_i]\!] \, [\![ordseq_{i-1}]\!]^{-1}$.
These sequences contain a $[\![1]\!]$ at the places where the value with index i should occur and $[\![0]\!]$ otherwise.
Now the final ordered sequence can be calculated as follows:

$$[\![seq_i]\!] = \prod_{j=0}^{t-1} ([\![uniqseq_{j,i}]\!]^j) = \left[\!\!\left[\sum_{j=0}^{t-1} j * uniqseq_{j,i} \right]\!\!\right], 0 \leq i < n \tag{5.1}$$

We need $(t-1) * (2^{\lfloor \log n \rfloor + 1} - \lfloor \log n \rfloor - 2) \leq (t-1) * (2n - \log(n) - 2)$ conditional gates and $t - 1$ conversions to binary representation for the intermediate bit sortings. Observe that for the final construction of the sequence (Equation 5.1), no conditional gates are needed. The complete algorithm is shown as Algorithm 2.

Algorithm 2 Counting sort to sort values $[[\![v_0]\!], \ldots, [\![v_{n-1}]\!]]$, $v_i \in [0, t)$ and $t > 2$

for $i = 0$ to $t - 2$ **do**
  $[\![cnt_i]\!] \leftarrow \prod_{j=0}^{N-1}([\![v_{j,i}]\!])$
  $[\![ordseq_i]\!] \leftarrow$ sorted bit list of length n with $\prod_{j=0}^{i} [\![cnt_i]\!] \left(= \left[\!\left[\sum_{j=0}^{i} cnt_i\right]\!\right]\right)$ $[\![1]\!]$s
end for
$[\![ordseq_{t-1}]\!] \leftarrow [[\![1]\!]]^n$
$[\![uniqseq_0]\!] \leftarrow [\![ordseq_0]\!]$
for $i = 1$ to $t - 1$ **do**
  $[\![uniqseq_i]\!] \leftarrow [\![ordseq_i]\!] \, [\![ordseq_{i-1}]\!]^{-1}$
end for
for $i = 0$ to $n - 1$ **do**
  Output $\prod_{j=0}^{T-1}([\![uniqseq_{j,i}]\!]^j)$, $\left(= \left[\!\left[\sum_{j=0}^{T-1}(j * uniqseq_{j,i})\right]\!\right]\right)$
end for

5.1.2.1 Complexity

This protocol is basically a repetition of the counting sort for bits for every possible value of T. Therefore, communication complexity is linear in the number of elements in T. The $|T|$ repetitions of the sorting protocol can be performed in parallel, as they do not depend on each other.

5.1.2.2 Privacy

Since no communication is added besides the bit-sorting, this algorithm also preserves privacy. All extra computations consist only of multiplications of encrypted values, and can be performed individually by each party.

5.2 Comparison-based sorting

In this section we will discuss comparison-based sorting algorithms. These algorithms are based on the *compare-exchange* operation. There are both oblivious and non-oblivious algorithms. With the non-oblivious algorithms, the execution of the algorithm is dependent on the input. In our setting, we will not consider these algorithms, as they might leak information on their input, based on the run of the algorithm.
Oblivious comparison based sorting algorithms can be represented by sorting networks. A sorting network consists of wires that contain the input values. These wires are connected by compare-exchange gates, that put the values on two wires in the correct order. An example of a sorting network can be found in Figure 5.3. In comparison-based sorting algorithms, there are various building blocks that can be used. In Figure 5.4, a bubble pass is shown. A bubble pass pushes the higher values in the direction of the higher positions or the lower values to the lower positions. When combining two bubble passes, we get a shaker pass, as can be seen in Figure 5.5. Since in both building blocks, the input of a comparison depends on the outputs of a previous comparison, the comparisons cannot be performed in parallel. To achieve high parallelism, a so called brick pass can be used, that is shown in Figure 5.6. A brick pass can only put elements into position that are at most two positions apart from their correct position. For all building blocks there are also variants called h-bubble, h-shake and h-brick pass. With these building blocks the elements that are compared are h positions away from each other instead of 1 position, as with the original building blocks.
If we want to sort n values, there is a lower bound of $n \log n$ *compare-exchange* operations that are needed. For the compare-exchange operation using Protocol 3.3.1, we need $3n-1$ *conditional gates*. Therefore, we will need at least $(3n-1)n \log n$ *conditional gates* for any secure comparison-based sorting algorithm. However, it is not trivial to find oblivious comparison-based sorting algorithms that need only $n \log n$ comparisons. Ajtai et al. found a sorting network that runs with $O(n \log n)$

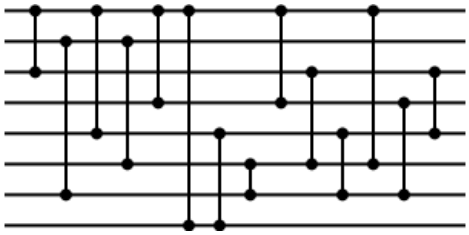

Figure 5.3: Example of a sorting network

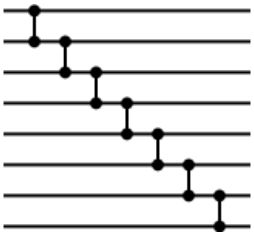

Figure 5.4: Bubble pass

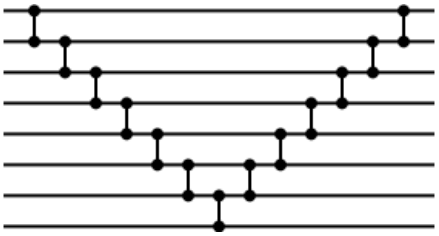

Figure 5.5: Shaker pass

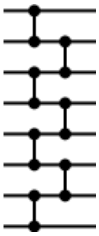

Figure 5.6: Brick pass

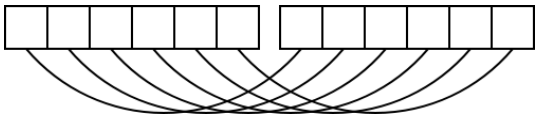

Figure 5.7: Comparisons for the region compare-exchange operation with the identity permutation

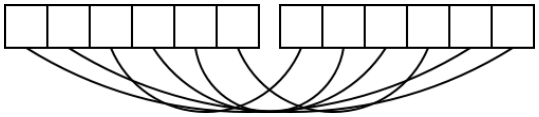

Figure 5.8: Comparisons for the region compare-exchange operation with a random permutation

comparisons [AKS83]. This network is unfortunately not very practical, as the hidden constant, that is masked by the O notation, is very large. A more practical sorting network with $O(n \log^2 n)$ comparisons is given by Batcher in [Bat68]. The round complexity of Batcher's sorting network is $O(\log^2 n)$.
In 1959, Shell introduced a new sorting algorithm that is known as Shellsort [She59]. In Shellsort the input is sorted in a number of rounds. Every round has a decreasing gap size. In one round, the gap size is used to group elements that are a number of position apart. Every group of elements is then sorted. As long as the last gap size is 1, the algorithm always sorts the input correctly.

5.2.1 Randomized Shellsort

To provide better efficiency, Goodrich recently introduced a randomized Shellsort algorithm [Goo10]. This randomized Shellsort algorithm sorts any input with very high probability and needs only $O(n \log n)$ comparisons. The complete algorithm is given in Algorithm 3. It is assumed, without loss of generality, that the number of elements is a power of 2. If the number of elements is lower, one can pad the sequence with either the lowest or the highest possible value. After the sorting these values can be removed again, as the will always end up at the beginning or the end respectively of the sorted sequence. The values are in each iteration grouped in regions. The size of the regions is equal to the gap size. The region compare-exchange operation performs a compare-exchange operation for each element in a region with an element in another region. In regular Shellsort, the first element of a region is compared to the first element of the other region, the second element is compared with the second etc. In the randomized version, a random permutation is used in these comparisons. In Figure 5.7 the comparisons for a region compare-exchange operation is shown with the identity permutation. In Figure 5.8 the same is shown with a random permutation.
For the analysis, in every region compare-exchange the permutations and comparisons are performed at least 4 times. However, in practice, performing only one permutation and comparison already gives a high success rate of over 99.9% [Goo10]. The last two for-loops in the algorithm are added for the proof of a high probability bound. According to Goodrich, this is probably an artifact of his analysis, rather than the algorithm in Algorithm 3. $\mathcal{S}$ is a deterministic sorting algorithm that can sort a subarray of size $2m$ as an atomic action. $\mathcal{S}$ can again, for example, be a deterministic Shellsort.
In [WLG+10], Wang et al. present several oblivious algorithms for use in secure two-party computation. One of the algorithms is an optimized version of the original randomized Shellsort algorithm, which they call Fast randomized Shellsort. In their algorithm they try to minimize the number of comparisons. Based on empirical studies, they designed the algorithm, as in Algorithm 4. They reduce the number of comparisons by performing only the h-bubble pass from Algorithm 3. Since according to their empirical studies, most of the misplaced elements are only one position out of place if only the h-bubble pass is performed, a single bubble pass is added at the end to

Algorithm 3 Randomized Shellsort [Goo10]

Input: Array A, containing the values to be sorted
  {Let A_i denote subarray $A[io \dots io + o - 1]$, for $i = 0, 1, 2, \dots, N/o - 1$}
  for $o = N/2, N/2^2, N/2^3, \dots, 1$ **do**
    Region compare-exchange A_i and A_{i+1}, for $i = 0, 1, 2, \dots, N/o - 2$
    Region compare-exchange A_{i+1} and A_i, for $i = N/o - 2, \dots, 2, 1, 0$
    Region compare-exchange A_i and A_{i+3}, for $i = 0, 1, 2, \dots, N/o - 4$
    Region compare-exchange A_i and A_{i+2}, for $i = 0, 1, 2, \dots, N/o - 3$
    Region compare-exchange A_i and A_{i+1}, for even $i = 0, 1, 2, \dots, N/o - 2$
    Region compare-exchange A_i and A_{i+1}, for odd $i = 0, 1, 2, \dots, N/o - 2$
  end for
  for $i = 0$ to $n - 2m$ incrementing by steps of m **do**
    Use $\mathcal{S}$ to sort $A[i \dots i + 2m - 1]$
  end for
  for $i = n - 2m$ to 0 decrementing by steps of m **do**
    Use $\mathcal{S}$ to sort $A[i \dots i + 2m - 1]$
  end for
  return A

put these elements into place. This bubble pass can also put elements into place that are further out of place but elements can only be moved one place to the left. Because h-bubble passes are used, the round complexity of this algorithm is still the same as for the original algorithm, namely $O(n)$. The number of comparisons that are needed, is reduced from $5n \log n$ to $2n \log n$.

Algorithm 4 Fast Randomized Shellsort [WLG+10]

Input: Array A, containing the values to be sorted
  {Let A_i denote subarray $A[io \dots io + o - 1]$, for $i = 0, 1, 2, \dots, N/o - 1$}
  for $o = N/2, N/2^2, N/2^3, \dots, 1$ **do**
    Region compare-exchange A_i and A_{i+1}, for $i = 0, 1, 2, \dots, N/o - 2$
    Region compare-exchange A_{i+1} and A_i, for $i = N/o - 2, \dots, 2, 1, 0$
  end for
  for $i = 0$ to $N - 2$ **do**
    compare-exchange $A[i]$ and $A[i + 1]$
  end for

To optimize the round complexity, we introduce a new variant we call Double brick randomized Shellsort in Algorithm 5. We use a double h-brick pass in each iteration and a single brick pass at the end. Because all comparisons in a brick pass can be performed in parallel, this algorithm has a depth of only $O(\log n)$. The number of comparisons needed is equal to the Fast randomized Shellsort (Algorithm 4).

5.2.2 Performance

In this section we compare the different randomized Shellsort algorithms we considered in Section 5.2.1. If we want to quantify the sortedness of an output, there are various measures we can use. First we can count the number of sequences that are sorted. We could either consider all binary or integer inputs, or take a number of random inputs. An alternative is to count the adjacent inversions. An adjacent inversion occurs if there exists in index i, such that for the output o it holds that $o_i > o_{i+1}$, i.e. two adjacent elements are out of order. A similar measure is the number of elements that are out of place. For this, you take the sorted sequence and compare the number of elements that are at a different position in the output. Instead of only looking at the number of elements that are out of place, one could also look at the distance that the elements are out of place and take the maximum of this.

Algorithm 5 Double Brick Randomized Shellsort

Input: Array A, containing the values to be sorted
{Let A_i denote subarray $A[io \dots io + o - 1]$, for $i = 0, 1, 2, \dots, N/o - 1$}
for $o = N/2, N/2^2, N/2^3, \dots, 1$ **do**
Region compare-exchange A_i and A_{i+1}, for even $i = 0, 1, 2, \dots, N/o - 2$
Region compare-exchange A_i and A_{i+1}, for odd $i = 0, 1, 2, \dots, N/o - 2$
Region compare-exchange A_i and A_{i+1}, for even $i = 0, 1, 2, \dots, N/o - 2$
Region compare-exchange A_i and A_{i+1}, for odd $i = 0, 1, 2, \dots, N/o - 2$
end for
Compare-exchange $A[i]$ and $A[i + 1]$, for even $i = 0, 1, 2, \dots, N - 2$
Compare-exchange $A[i]$ and $A[i + 1]$, for odd $i = 0, 1, 2, \dots, N - 2$

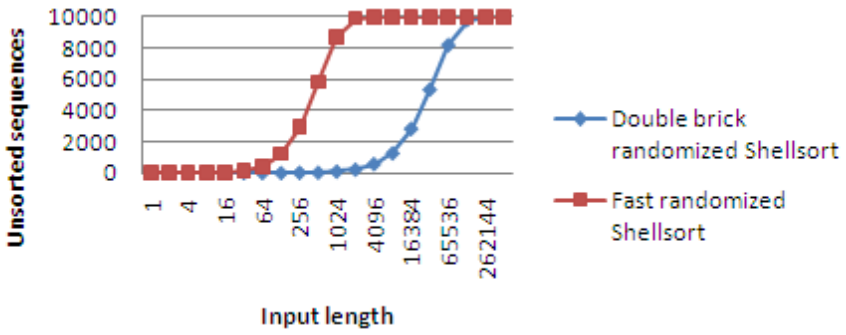

Figure 5.9: Number of unsorted sequences for 10000 random inputs per input length

In Figure 5.9, the number of unsorted sequences is given for the different algorithms considered in Section 5.2.1 and for 10000 random inputs per input length. As it can be seen, our Double brick randomized Shellsort, sorts more inputs correctly, than the Fast randomized Shellsort. As can be seen in Table 5.1, the average number of adjacent inversions is much larger for Fast randomized Shellsort than it is for Double brick randomized Shellsort. The maximum distance that elements are out of place, as can be seen in Table 5.2, is however much larger with Double brick randomized Shellsort than it is with Fast randomized Shellsort. The average distance that elements are out of place is however not very different, as can be seen in Table 5.3. With the average number of elements that are out of place, in Table 5.4, our Double brick randomized Shellsort performances again better than the Fast randomized Shellsort. It thus depends on the application which of the two algorithms would be preferable. With the Double brick randomized Shellsort more inputs are sorted correctly, and if they are not sorted correctly, on average, the number of elements that are not at their correct position is less than with the Fast randomized Shellsort. However, with Fast randomized Shellsort, the maximum distance that elements are out of place is much lower than with the Double brick randomized Shellsort.

5.2.3 Privacy

5.2.3.1 Preliminaries

In this section we will discuss the privacy that is provided by the randomized Shellsort algorithms. An algorithm that sorts every possible input correctly, provides perfect privacy, as nothing can be learned from the output about the input, except for the values that occur. However, since the randomized algorithms do not always sort correctly, information about the input might leak. For example, if the output is not sorted, the input was also not sorted. This is because as soon

Input length	Fast randomized Shellsort	Double brick randomized Shellsort
16	1	0
32	1.05	0
64	1.05	1
128	1.15	1
256	1.29	1
512	1.64	1.04
1024	2.66	1.02
2048	5.53	1.03
4096	12.43	1.05
8192	27.60	1.11
16384	60.56	1.21
32768	131.38	1.46
65536	281.60	2.17
131072	597.91	4.02
262144	1262.26	8.66
524288	2647.78	18.99

Table 5.1: Average adjacent inversions for 10000 random inputs per input length for unsorted sequences

Input length	Fast randomized Shellsort	Double brick randomized Shellsort
16	1	0
32	2	0
64	2	1
128	3	2
256	5	5
512	5	3
1024	6	6
2048	6	16
4096	10	121
8192	11	215
16384	13	545
32768	10	861
65536	14	2826
131072	13	5307
262144	18	16175
524288	19	29175

Table 5.2: Maximum positions out of place for 10000 random inputs per input length for unsorted sequences

Input length	Fast randomized Shellsort	Double brick randomized Shellsort
16	1	0
32	1.04	0
64	1.03	1
128	1.05	1.20
256	1.08	1.24
512	1.08	1.23
1024	1.10	1.20
2048	1.11	1.38
4096	1.12	1.48
8192	1.13	1.58
16384	1.14	1.69
32768	1.14	1.79
65536	1.15	1.87
131072	1.16	1.92
262144	1.16	1.95
524288	1.17	1.98

Table 5.3: Average distance that elements are out of place for 10000 random inputs per input length for unsorted sequences

Input length	Fast randomized Shellsort	Double brick randomized Shellsort
16	2	0
32	2.16	0
64	2.13	2
128	2.38	2.50
256	2.70	2.61
512	3.46	2.71
1024	5.69	2.55
2048	11.86	3.31
4096	26.82	4.00
8192	59.90	5.30
16384	131.95	7.79
32768	287.32	13.70
65536	618.17	32.41
131072	1316.88	95.17
262144	2787.96	351.75
524288	5863.07	1358.68

Table 5.4: Average elements out of place for 10000 random inputs per input length for unsorted sequences

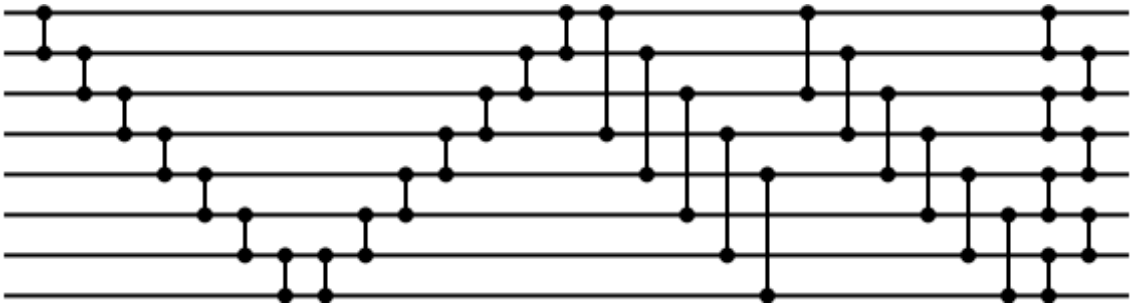

Figure 5.10: Fixed tail of the randomized Shellsort sorting network

as the input is sorted, it will not change any more. A sequence will never get unsorted, because only *compare-exchange* operations are used, which put two elements in the correct order. Also, a certain output might only be the result from a limited number of inputs. This again leaks information about the possible inputs.
We can identify classes of inputs that always get sorted, independent from the random permutations. This can be done, since, in the last iteration, the size of the regions is 1, and therefore the last iteration is always fixed. Also, the fast randomized Shellsort and the double brick randomized Shellsort both have an end pass that is always fixed. For the double brick randomized Shellsort, this results in a fixed tail of three brick passes. For the fast randomized Shellsort, the fixed tail consists of two bubble passes. The original randomized Shellsort has a bit more complicated tail, that can be seen in Figure 5.10, for input size 8.
Nearly sorted inputs, where the elements are only a few places apart from their position, always get sorted by the algorithms. A single brick pass can put elements into place, that are at most two places out of position. If elements are only one place out of position, the input will always be sorted correctly.
If the inputs of the algorithms can consist of distinct integer values, there are $n!$ possible inputs, where n is the number of input elements. Since the sorting only depends on the relative order of the values in the input, we can map the values of every possible input to the domain $[0, n)$. We denote the set of distinct integer sequences of length n with $\mathcal{I}_n$. The set of all possible binary sequences of length n is denoted by $\mathcal{B}_n$.
In what follows we make use of some known theorems in the theory of sorting.

Theorem 5.2.1 (Zero-one principle [Knu73])**.** *If a network with n input elements sorts all 2^n sequences in $\mathcal{B}_n$ into non-decreasing order, it will sort any arbitrary sequence of n integers into non-decreasing order.*

Proof. If $f(x)$ is a monotonic function, such that $f(x) \leq f(y)$ if and only if $x \leq y$, and a sorting network takes as input the sequence $x_1, \ldots, x_N$ and as output $y_1, \ldots, y_N$, then the network will transform the sequence $f(x_1), \ldots, f(x_N)$ into $f(y_1), \ldots, f(y_N)$. If, for some i, it holds that $y_i > y_{i+1}$, consider the monotonic function f, that is defined as

$$f(x) = \begin{cases} 0 & \text{if } x < y_i \\ 1 & \text{if } x \geq y_i \end{cases}$$

This defines a binary sequence $f(x_1), \ldots, f(x_N)$, that is not sorted correctly by the network. Thus, if all binary inputs are sorted correctly, we have $y_i \leq y_{i+1}$, for $1 \leq i < N$. □

Due to this theorem, the work is reduced that is needed to test whether a sorting networks sorts all possible inputs correctly. It is not necessary to test all $N!$ possible inputs, for a sorting network of size N. Instead it suffices to test only all 2^N binary input sequences.

Theorem 5.2.2 (Generalized 0-1 principle [RS05])**.** *Let S_k denote the set of length n binary strings with exactly k 0s, $0 \leq k \leq n$. Then, if a sorting circuit with n input lines sorts at least an α fraction of S_k for all k, $0 \leq k \leq n$, then the circuit sorts at least a $(1-(1-\alpha)(n+1))$ fraction of the input permutations of n arbitrary numbers.*

Proof.

Definition 5.2.3. A sequence $s \in \{0,1\}^n$ is a k-string, if it contains exactly k 0's. The set of all k-strings for a fixed k is called a k-set and will be denoted by S_k

Observation 5.2.4. If f is monotone, then the inverse is also monotone.

Theorem 5.2.5. *For any sorting network C and a monotone function f, $f(C(a)) = C(f(a))$.*

Given $I_n = \{1, 2, \ldots, n\}$, the only monotone function between I_n and sequences in S_k is given by $f_k(i) = 0$ for $j \leq k$ and 1 otherwise.
From the previous observation, f_k^{-1} is also monotone.

Lemma 5.2.6. *If a sorting circuit does not sort some $a \in S_k$ then it does not sort $f_k^{-1}(\alpha)$. If the circuit correctly sorts $f_k(\sigma)$ for all k, for an input permutation σ, then it correctly sorts σ.*

Consider the mapping between permutations of I_n and strings $a \in S_k$ such that $a_i = f_k(\Pi(i))$ for a permutation Π. This can be represented as a bipartite graph G_k with $n!$ elements in one set and $|S_k|$ on the other (for each k). Note that a single permutation can map to exactly one string in S_k. Using simple symmetry arguments, it follows that

Lemma 5.2.7. *Each set in the bipartite graph G_k has vertices with equal degree. In particular, the vertices representing S_k have degree equal to $\dfrac{n!}{|S_k|}$.*

If the circuit does not sort some permutation Π of I_n then it does not sort at least one sequence in $\{0,1\}^n$, say $a \in S_k$ for some k. Conversely, if the circuit does not sort some sequence $a \in S_k$ then it does not sort any of the permutations in the inverse map. Consider the graph G_k where we mark all the nodes corresponding to the permutations that are not sorted correctly and likewise we mark the nodes of S_k that are not sorted correctly. For a fixed k, if the circuit does not sort $\beta|S_k|$ ($\beta < 0$) sequences it does not sort $\beta|S_k|n!$ permutations. Therefore the total fraction of permutations (over all values of k) that may not get sorted correctly is $\beta(n+1)$. Setting $\alpha = 1-\beta$ completes the proof. □

Lemma 5.2.8 (Probabilistic 0-1 principle [Goo10])**.** *If a randomized data-oblivious sorting algorithm sorts any binary sequence of size n with failure probability at most ϵ, then it sorts any integer sequence of size n with failure probability at most $\epsilon(n+1)$.*

Proof. An integer sequence of size n has at most $n+1$ corresponding binary sequences. The integer sequence is only sorted if all corresponding binary sequences are sorted. So if one binary sequence is sorted with failure probability at most ϵ, then the binary sequences corresponding with the integer sequence are sorted with failure probability at most $\epsilon(n+1)$. □

Definition 5.2.9. To denote the number of zeroes or ones in a binary array b we use the notations $\#_0 b$ and $\#_1 b$ respectively.

To compute certain privacy measures we need to determine the number of occurrences of each possible output sequence, for all possible input sequences. This could be done by trying all possible input sequences, and counting the corresponding outputs. To reduce the work that is needed, we try to reduce the number of inputs that we need to test while still getting the desired result. To do this we try to reduce the case of integer inputs to the case of binary inputs, which would lead to a reduction from $n!$ possible inputs to 2^n possible inputs.

We define a mapping of sequences from $\mathcal{B}_n$ to classes of sequences from $\mathcal{I}_n$. Assume we have a binary sequence $b \in \mathcal{B}_n$, then the class of integer sequences it corresponds to is defined as

$$\{s \in I_n | (\forall i \in [0, N) : (b_i = 0 \Rightarrow s_i < \#_0 b) \wedge (b_i = 1 \Rightarrow s_i \geq \#_0 b))\}$$

For input sequences in the corresponding class there is a value $< m$, at the position of a 0 in the binary input, and there is a value $\geq m$, at the position of a 1 in the binary input.
From Theorem 5.2.1 the following lemma follows.

Lemma 5.2.10. *Assume we have a binary input that is not sorted correctly by a sorting network. Then the inputs in the corresponding class of integer inputs will also not be sorted correctly.*

The same reasoning does not apply to correctly sorted binary inputs. If a binary input is sorted correctly, this does not mean that the corresponding class of integer inputs is also sorted correctly. Only if all binary inputs corresponding to an integer input are sorted correctly, then the integer input is also sorted correctly.
The elements from $\mathcal{I}_n$ can again be represented by $n + 1$ different binary sequences. Given an integer sequence $s \in \mathcal{I}_n$, the set of corresponding binary sequences is defined as

$$\{b \in \mathcal{B}_n | 0 \leq i \leq n \wedge (\forall j | (s_j < i \Rightarrow b_i = 0) \wedge (s_j \geq i \Rightarrow b_i = 1))\}$$

Assume an integer sequence is given that sorted correctly. The relative order of the integer sequence is the same as the relative order of the corresponding binary sequences with respect to the order $\geq$. Therefore, if the integer sequence is sorted correctly, then the corresponding binary sequences are also sorted correctly.
Assume a binary output sequence is given for which, without loss of generality, it can be determined that the value for a particular input position is 1. Then the number of possible values at this position for an input sequence from $\mathcal{I}_n$ is equal to $\#_1 b$.

Theorem 5.2.11. *If the input for any binary output can be determined with probability at least ϵ, then the input position for one output value x for an integer output can be determined with probability at least ϵ^2.*

Proof. We construct two binary outputs. The first contains 0's at the positions where the value in the integer output is $\leq x$, and 1's at the other position. The second contains 0s at the positions where the value in the integer output is $< x$, and 1s at the other position. The two binary outputs thus only differ at the position of x. For both outputs, a corresponding input can be determined with probability at least ϵ. Between both binary inputs only one bit is different. At the position of this bit, x occurred in the arbitrary input. We thus need to be able to determine the input for two binary outputs, which can be done with a probability of at least ϵ^2. □

Definition 5.2.12. Binary sequence a, of length N, dominates b if the following holds:

$$(\forall i \in [0, N) : a_i = 0 \Rightarrow b_i = 0) \tag{5.2}$$

To denote that a dominates b we use the notation $a \sqsubseteq b$. This relation is a partial ordering.

Lemma 5.2.13. *Two binary sequences only have overlapping corresponding integer sequences, if one of the binary sequences dominates the other.*

Proof. This follows from the definition of the binary sequences that correspond to an integer sequence. Assume we have two binary sequences a and b that correspond to an integer sequence $i \in \mathcal{I}_n$. For a and b there is an m_a and m_b, respectively, such that for every position i_j, there is a 0 at a_j if $i_j < m_a$ resp. if $i_j < m_b$ for b_j, otherwise there is a 0 at this position, where $j \in [0, n)$. Assume w.l.o.g. that $m_a \geq m_b$. At every position where $i_j < m_b$ there is a 0, both at a_j and b_j. Since these are the only 0's in b, a dominates b. Therefore, for every two possible binary sequences that correspond to an integer sequence, one binary sequence dominates the other. Therefore it is not possible to have two binary sequences where neither one dominates the other, but that do have a common corresponding integer sequence. □

Lemma 5.2.14. *If we have two binary sequences $a, b \in \mathcal{B}_n$, such that $a \sqsubseteq b$, the number of overlapping corresponding integer sequences from $\mathcal{I}_n$ is*

$$\#_0 a!(\#_0 a - \#_0 b)!\#_1 b!$$

Proof. We can divide a and b in three areas, the 0-area where there is a 0 in both a and b, the 1-area where there is a 1 in both a and b, and the mixed area where a and b differ. The 0-area will have length $\#_0 a$, since a contains the least 0s. The same holds for the 1s and b. The mixed area will have length $\#_0 a - \#_0 b$. In the overlapping integer sequences, they will have values $< \#_0 a$ in the 0-area, as a only contains $\#_0 a$ 0s. In 1-area, the values will be $\geq \#_0 b$, as b contains $N - \#_0 b$ 1s. The mixed area will contain the other values $\geq \#_0 a$ and $< \#_0 b$. In total we can create $\#_0 a!(\#_0 a - \#_0 b)!\#_1 b!$ different permutations using these values in the different areas. □

The overlapping corresponding integer sequences for a and b, where $a \sqsubseteq b$, are given by

$$\{i \in I_n | (\forall j : 0 \leq j < n \wedge ((a_j = 0 \wedge b_j = 0) \Rightarrow i_j < \#_0 a) \wedge$$
$$((a_j = 1 \wedge b_j = 0) \Rightarrow \#_0 a \leq i_j < \#_0 b) \wedge ((a_j = 1 \wedge b_j = 1) \Rightarrow \#_0 b \leq i_j < n))\}$$

If we have $S \subset \mathcal{B}_n$, we can compute a lower bound on the number of corresponding integer sequences using the following:

$$\sum_{b \in S} \#_0 b!\#_1 b! - \sum_{a,b \in S \wedge a \neq b \wedge a \sqsubseteq b} (\#_0 a!(\#_0 a - \#_0 b)!\#_1 b!) \tag{5.3}$$

To get the actual number of corresponding integer sequences, we can construct a set containing the integer sequences corresponding to the sequences in S. This way we get all the integer sequences corresponding to S, without duplicates.
Another method would be to construct a multiset containing all overlapping integer sequences for all pairs $a, b \in S$, such that $a \sqsubseteq b$. Remove from all sequences in the multiset one occurrence. The size of the resulting multiset then has to be added to Equation 5.3 to get the number of integer sequences corresponding to S. This method would require less work then the first method, as the number of overlapping integer sequences is smaller than the number of all corresponding integer sequences.

Definition 5.2.15. The number of integer sequences corresponding to a set $S \subset \mathcal{B}_n$ is denoted by $NrCorInt(S)$.

5.2.3.2 Anonymity measures

When considering measures of privacy we can divide these into two categories. First we have the global anonymity. Global anonymity is a measure about the total information leakage. The second measure is local anonymity. Local anonymity is about the privacy of a particular party, i.e. whether information about the value at a particular position is leaked.

Global anonymity For global anonymity, we make use of conditional entropy as a measure. Conditional entropy is based on Shannon entropy. Shannon entropy gives the expected number of bits of uncertainty and is defined as:

$$H(X) = -\sum_i p(x_i) \log p(x_i)$$

Conditional entropy is then defined as:

$$\begin{aligned} H(X|Y) &= \sum_i H(X|Y = y_i) \\ &= -\sum_i p(y_i) \sum_j p(x_j|y_i) \log p(x_j|y_i) \end{aligned}$$

	Entropy	Unsorted sequences
Randomized Shellsort	12.950	17
Fast randomized Shellsort	12.851	834
Double brick randomized Shellsort	12.948	34
Correct sorting	12.954	

Table 5.5: Conditional entropy for $n = 16$ on binary input for identity permutations

Randomized Shellsort	12.95345044
Fast randomized Shellsort	12.95338379
Double brick randomized Shellsort	12.95345044
Correct sorting	12.9535

Table 5.6: Average conditional entropy for $n = 16$ on binary input for random permutations for 100 runs

For distinct inputs from $\mathcal{I}_n$ of size n, the conditional entropy when sorting correctly is equal to

$$\log n!$$

If all inputs are sorted correctly, the output for every input sequence from $\mathcal{I}_n$ is the same.
Conditional entropy is the expected number of bits of uncertainty given some known condition. In our case, X is the input and the condition Y is the output of the algorithm. The conditional entropy thus is a measure for the uncertainty of the input given that the output is known. We now do not look at what information exactly leaks, like for example particular input positions, but the measure only takes into account the number of inputs corresponding to the outputs.
The correct sorting will always give the maximum value of entropy. In this case there will be the minimum number of possible outputs and every output has the maximum number of possible inputs. The number of outputs can only be increased if the number of possible inputs for the outputs of the sorting are decreased.
We ran the three algorithms from Section 5.2.1 with all possible inputs from $\mathcal{B}_n$ to compute the conditional entropy for $n = 4$ and $n = 8$. For $n = 4$ and $n = 8$, all three algorithms sort correctly. The conditional entropy for the correct sorting is 1.969 and 5.456 respectively. In Table 5.5, the values for conditional entropy and the number of unsorted inputs can be found for $n = 16$ for binary inputs. For the permutations inside the region-compare operations we used the identity permutation. In Table 5.6, the average conditional entropy can be found for 100 runs and $n = 16$ for binary inputs, where the permutations are random. With the identity permutations, the original and the double brick randomized Shellsort both sort almost all inputs correctly. With the random permutation, they both sort all inputs correctly.
Let the conditional entropy of a given network for binary inputs be denoted by $H(X_b|Y_b)$, and for distinct integer inputs by $H(X_i|Y_i)$. For every network it holds that $H(X_b|Y_b) \leq H(X_i|Y_i)$.
The number of possible binary inputs of length n containing i ones is equal to

$$\binom{n}{i}$$

The conditional entropy for binary input sequences of length n when sorting correctly is given by the following equation:

$$H(X_b|Y_b) = \frac{1}{2^n} \sum_{0 \leq i < n} \binom{n}{i} \log(\frac{1}{2^n}\binom{n}{i}) \tag{5.4}$$

Conjecture 5.2.16. *The following holds for $n > 1$:*

$$(\log(n) + 1)H(X_b|Y_b) > H(X_i|Y_i)$$

Local anonymity The global anonymity gives an average or worst-case loss of anonymity. However, we might only be interested in one particular input or output of the algorithm. For example, we might only want to know the input of one of the parties or want to know who gave a particular output value as input. We consider both the case where the output is public and where it is private.
As measure we use the one proposed by Golle and Juels for measuring anonymity in mix-nets [GJ04]:

$$Anon(j) = \left(\max_{0 \leq k < n} p(i_k \to o_j) \right)^{-1},$$

where $p(i_k \to o_j)$ is the probability that the value of the input element at position k will occur in the output at position j. The measure $Anon(j)$ gives a lower bound on the number of elements that can occur as input for a given output position. This can be seen as the minimal privacy for a particular output position.
If the output is public we could also construct a conditional variant of this measure for position j and output y:

$$Anon(j|y) = \left(\max_{0 \leq k < n} p(i_k \to o_j|y) \right)^{-1}$$

This measure could be useful if the output is known, and we want to know for one particular output value the amount of information that is leaked about the possible input positions.
The conditional anonymity gives the anonymity for a particular output, whereas the regular anonymity gives the worst case result for all possible outputs. These measures can be applied to a particular sorting network, but also to a group of sorting networks. For example, they can be applied for all possible random permutations in the randomized Shellsort algorithms, thus giving a lower bound on the privacy provided by the algorithm.
A variant of this measure can be constructed to determine the privacy for a particular input position. The following gives a lower bound on the possible outputs for a given input position:

$$AnonInput(j) = \left(\max_{0 \leq k < n} p(i_j \to o_k) \right)^{-1}$$

The same way as before, we can construct a conditional variant of this measure, to apply it for a given output y:

$$AnonInput(j|y) = \left(\max_{0 \leq k < n} p(i_j \to o_k|y) \right)^{-1}$$

These measures could again be used for just a single network or a group of networks.

5.2.4 Pre-mixing

To prevent information leakage based on the output, the algorithm could be preceded by applying a random permutation to the input values. If this permutation is secure, the attacker can no longer retrieve information about the input, only based on the output. However, performing a secure permutation is an expensive operation. A secure permutation could, for example, be performed by using mix-nets. When using mix-nets, every party has to perform a permutation, and provide proof that the values are not changed, but only the positions of the values.
A public permutation might also be useful, but not in all cases. Since a public permutation can be seen as a bijective function from and to all possible inputs, this might not prevent the amount of information leakage if there is a uniform distribution on the inputs. Every possible input is still as likely to occur as input to the algorithm as before. Because the permutation is public, the original input can be constructed from the input to the algorithm, by applying the permutation in reverse order. Thus, if information can be learned about the input of the algorithm, this can be traced back to the original input.
If the attacker has some apriori knowledge about the distribution of the inputs, this knowledge could be rendered useless by applying a permutation on the input before performing the algorithm.

For example, if there is one input that occurs often and that has some characteristic output, applying a permutation might result in a different output that is less distinguishable.
Applying a public permutation will not influence the entropy, if the inputs are uniformly distributed.

6. Clustering

In the knot-based reputation system by Gal-Oz et al., the users are divided into clusters, depending on their votes [GOGH08]. The membership of a certain cluster thus leaks information about a user's votes. In this section we will transform several clustering algorithms to oblivious variants. These oblivious variants can then easily be performed using secure multi-party computations to provide privacy for the users.
The users can be represented as nodes in a graph. There exists an edge between two nodes, if the similarity between the votes of the two corresponding users is above a specified threshold.
An extension to this setting could be the introduction of negative edges. A negative edge would be introduced if the similarity between two users is below the threshold. Edges could also contain the actual similarity between two users. This would give a graph with weighted edges, which might result in better results. When using unweighted edges the is no distinction between users that are almost completely similar and users where the similarity is just above the threshold.
The graph is represented by a $n \times n$ adjacency matrix E, for n nodes. There is an edge from node i to node j if $E_{i,j} = 1$, for an unweighted graph, or $E_{i,j} > 0$, for a weighted graph. If the graph is undirected then $E_{i,j} = E_{j,i}$. The vector containing the nodes that node i is connected to, is denoted by E_i, which corresponds to row i in the adjacency matrix.

6.1 Oblivious algorithms

The original algorithm makes use of sets. In the oblivious variant of the algorithm, we thus use the secure sets, as discussed in Section 3.11, where all sets are represented by encrypted vectors. Using secure sets does mean that we always need the maximum amount of storage for each of the sets we use.
When converting an algorithm to an oblivious variant, we have to remove the if- and while-statements. The run of the algorithm gives the values of the conditions of these statements, which might leak information about private inputs. For the if-statements we can use the conditional gate. We perform both the if and the else clause and assign the values based on the condition of the if-statement.
If we, for example, have the following algorithm:

if b **then**
  $a \leftarrow 5$
else
  $a \leftarrow 7$
  $c \leftarrow 10$
end if

We can convert this algorithm to the following oblivious variant:

$a_{if} \leftarrow 5$
$a_{else} \leftarrow 7$
$c_{else} \leftarrow 10$
$a \leftarrow switch(b, a_{if}, a_{else})$
$c \leftarrow switch(b, c, c_{else})$

The while-statement is a bit more complicated to convert. First we need an upper bound on the number of iterations. However, this might not always be possible. If an upper bound is retrieved, we can convert the while-statement to a for-loop with an if-statement inside. This if-statement can then be transformed using the construction above.
Take the following algorithm:

while b **do**
  $a \leftarrow a + 10$
end while

Assume we can determine that b will hold for at most 10 iterations. We then convert this algorithm to the oblivious variant using the construction described above:

for Repeat 10 times **do**
  if b **then**
    $a \leftarrow a + 10$
  end if
end for

Using this method, always the maximum number of iterations is performed. It might also be considered to leak some information in exchange for better performance. This could be done by weakening the condition for the while-loop and add an if-statement inside to evaluate the original condition. This way, the while-loop is still used, but the condition leaks less information than before. This could also be useful if no upper bound for the number of iterations for the condition can be computed, but it can be weakened such that it leaks less information.

6.2 Detecting cycles

First we will show with a simple algorithm how to transform this to an oblivious variant. The algorithm we discuss is an algorithm to detect cycles in a directed graph. A cycle in a graph occurs if there is a path $v_1, v_2, \ldots, v_k$, where $v_1 = v_k$ and the path contains at least one edge.
In this algorithm we will not represent the graph with an adjacency matrix, but instead with a list e containing all edges. Every edge from node x to node y is represented by a pair (x, y).
The idea behind the algorithm, that can be seen in Algorithm 6, is that for every node we determine the number of children. If the number of children is larger than the total number of nodes, there exists a cycle in the graph. For every node we initialize the value with the number of outgoing edges. At every iteration we set the new size to the number of outgoing edges plus the sum of the sizes of the nodes at the outgoing edges. After at most n iterations all nodes contain the total number of children, if no cycles exist in the graph. If a cycle does exist, at least one node will have a value $\geq n$. This effect is due to the fact that if a cycle exist, the value of the nodes will be 'amplified'.

Algorithm 6 Determine whether cycles exist in a directed graph

for $i = 0$ to $n - 1$ **do**
  $s_i \leftarrow (\#j : (d, j) \in e)$
end for
for $i = 0$ to $n - 1$ **do**
  for $j = 0$ to $n - 1$ **do**
    $s'_j \leftarrow \sum_{k \wedge (l,k) \in edge} s_l + (\#m : (d, m) \in e)$
  end for
  $s \leftarrow s'$
  if $(\exists m : s_m > n)$ **then**
    return True
  end if
end for
return False

To transform this algorithm to an oblivious variant, first we need to be able to determine the number of incoming edges for each node. We can do this by using Dirac unary notation for the values in the pairs in e. This way we can take the summation of the second element in the pairs. For every component in the unary values the summation is computed, so every component in the sum contains the number of incoming edges of the corresponding node. As we need these values in every iteration we will compute this once and store it for future use.
To compute s', we need for every node the summation of s for the incoming edges. This can be done by computing, for every pair, the scalar product between s and the first value in the pairs in e. This scalar product is then multiplied with the second value of the pair. Since both values in the pairs are in unary notation, this results in a list of values that contain the value of s, for every incoming edge at the position corresponding with the node. By taking the summation of all these values we get a vector containing for every node the summation of s for all incoming edges at the corresponding position. By adding the number of incoming edges, which we computed in the initialization of the algorithm, we get the new value for s'.
The if-statement is taken outside the for-loops to get an oblivious algorithm. The algorithm now returns $(\exists i : s_i > n)$. In Algorithm 7 the new algorithm can be seen.

Algorithm 7 Oblivious algorithm to determine whether cycles exist in a directed graph

for $i = 0$ to $n - 1$ **do**
  $c_i \leftarrow s_i \leftarrow \sum_{(x,y)\in e} x_i$
end for
for $i = 0$ to $n - 1$ **do**
  $s \leftarrow c + \sum_{(x,y)\in e}(x \cdot s)y$
end for
return $(\exists i : s_i > n)$

The resulting algorithm can now easily be transformed to a secure multi-party computation protocol. The first part of the algorithm consists only of additions and multiplications, which we can compute using homomorphic cryptosystems. In the computation of the inner product, the second vectors contains a 1 at only one position. So instead of computing the scalar product we can make use of a secure selection protocol. The return value can be rewritten to:

$$\sum_{x\in s}[x > n] \neq 0$$

We now only need a comparison protocol to be able to compute the return value.

6.3 Distance-k cliques

In their reputation system, Gal-Oz et al. make use of distance-k cliques. The distance-k cliques problem is a generalization of the regular cliques problem, where all nodes in a cluster are directly connected. In distance-k cliques, the diameter of all clusters is at most k. The diameter of a cluster is the maximum of the length of shortest path between any two nodes in a cluster. The goal is to minimize the number of clusters while still satisfying the distance condition. If we take $k = 1$, we get the original cliques problem.
The algorithm that is used by Gal-Oz et al. is the algorithm presented by Edachery et al. in [ESB99]. This algorithm works on unweighted undirected graphs. In Algorithm 8 the algorithm is given. First we will discuss this original algorithm. In the next sections we will transform it to an oblivious variant.
For every cluster i the members of the cluster are denoted by M_i. Every node v belongs to one cluster i, which is given by C_v, i.e. $C_v = i$. We make use of the notion of bridge nodes. Bridge nodes are nodes that have an edge to another cluster that they are not part of themselves. Every bridge node v has, at the start of the algorithm, a list of external clusters L_v it is connected to.

An external cluster is a cluster that a node is not part of itself. The size of L_v, denoted by $|L_v|$, is the total number of nodes in the clusters in L_v. The set of neighbors of a node v is denoted by N_v, which is the set of all nodes that v has an edge to. In the algorithm an estimated diameter of a cluster C_i is stored as D_i. The estimated diameter is an upper bound for the actual diameter of a cluster.
At the beginning of the algorithm, the initial clusters are constructed. The simplest method is to let every node have it's own cluster. In this case, initially, the number of clusters is equal to the number of nodes.
The protocol merges clusters, until no clusters can be merged without getting a diameter larger than k. At every iteration a bridge node u is selected. This bridge node u is selected such that merging all clusters in L_u, results in the largest possible cluster for all possible bridge nodes.
A new tentative cluster T is then created by merging C_u with all clusters in L_u. The two clusters with the largest estimated diameter in T are selected. An upper bound for the diameter is computed for the merge of these two clusters. If this is smaller or equal than k, T is added to the set of clusters, and the individual clusters are removed. Also all corresponding values in are updated, removing all occurrences of the individual clusters and replacing them by the cluster T. If the new estimated diameter is larger than k, the largest cluster in T is removed from L_u, unless this is the cluster that u belongs to. In this case, the second largest cluster is removed. Also the L for other bridge nodes is updated, such that the two clusters will not again be tried to merge. This process is repeated until the L for all bridge nodes is empty, i.e. there are no more clusters that can be joined together.

6.3.1 Similarity

To construct the graph containing all users we need to determine the edges in this graph. To do this we need to be able to determine the similarity between two users. If the similarity between two users is above a certain threshold, there exists an edge between the corresponding nodes in the graph. To be able to define similarity we need the notation of difference, which is the average distance between the votes of the two users A and B:

$$\mathit{difference}(A,B) = \frac{\sum_{i=1}^{|A|} \mathit{distance}(A_i, B_i)}{|A|}$$

If the distance is from the domain $[0,1]$, then similarity is defined as:

$$\mathit{similarity}(A,B) = 1 - \mathit{difference}(A,B) \tag{6.1}$$

Every user has a set of t votes, one for each target. The votes consist of ratings for m aspects of the transactions. A vote is represented by a vector, that contains m encrypted components. If a user did not have an interaction with a particular target, then the vector corresponding with the vote for the target will be undefined, noted by $\perp$. A 0 indicates that a position has no value.
Since the cryptosystems we use do not work with real numbers, we multiply the values with a large constant to get an approximate value for the integer numbers. The value of the distance will be in the domain $\{0, 1, \ldots, C^2 * M\}$, where C defines the precision of the input values, e.g. $C = 100$ will give a precision of two decimal places.
Using Equation 6.1, we can express similarity between two users A and B as $C^2 * M - difference(A,B)$. To compute the difference we can use the following, where $X_{A,i}$ and $X_{B,i}$ are the votes for A resp. B for target i and $Y = \{i | X_{A,i} \neq \perp \wedge X_{B,i} \neq \perp\}$:

$$\begin{aligned} \mathit{difference}(A,B) &= \left[\!\!\left[\frac{\sum_{i \in Y} (\mathit{distance}(X_{A,i}, X_{B,i}))}{|Y|} \right]\!\!\right] \\ &= \left[\!\!\left[\sum_{i \in Y} (\mathit{distance}(X_{A,i}, X_{B,i})) \right]\!\!\right]^{|Y|^{-1}} \end{aligned}$$

Algorithm 8 Distance-k clustering

Form initial clusters (suppose there are q bridge nodes $u_0, \ldots, u_{q-1}$)
for $i = 1$ to q **do**
  $L_i \leftarrow \{j | j \in N_i \wedge j \neq i\}$
end for
while $\bigcup_{0 \leq i < q} L_i \neq \emptyset$ **do**
  Find u_x such that $|L_x|$ is the largest among all L_i, $0 \leq i < q$
  $CList \leftarrow L_x \cup \{M_x\}$
  $T \leftarrow \{v | v \in M_i \wedge i \in CList\}$
  Find the clusters LC_1 and LC_2 with the largest and second largest estimated diameter in the $CList$
  $D_T \leftarrow D_{LC_1} + D_{LC_2} + 2 - [LC_1 = C_x \vee LC_2 = C_x]$
  if $D_T \leq k$ **then**
    for $v \in T$ **do**
      $C_v \leftarrow T$
    end for
    for $i = 0$ to $q - 1$ **do**
      for $y \in CList \cap L_i$ **do**
        $L_i \leftarrow L_i \setminus y$
        if $D_T < k$ **then**
          $L_i \leftarrow L_i \cup \{T\}$
        end if
      end for
    end for
    if $D_T = k$ **then**
      for $u_i \in T$ **do**
        $L_i \leftarrow \emptyset$
      end for
    end if
  else
    if $LC_1 \neq C_x$ **then**
      $R \leftarrow LC_1$
    else
      $R \leftarrow LC_2$
    end if
    if $LC_1 = C_x \vee LC_2 = C_x$ **then**
      for $u_i \in T$ **do**
        $L_i \leftarrow L_i \setminus R$
      end for
      for $u_i \in R$ **do**
        $L_i \leftarrow L_i \setminus T$
      end for
    else
      $L_x \leftarrow L_x \setminus R$
    end if
  end if
end while

Y could be calculated using a secure matchmaking algorithm. This would have to be performed t times. Since $|Y|$ would be public in this case, we would not need the secure division protocol, but instead we can make use of private multiplications.
Using the difference we can populate the matrix that defines all the edges, $E(i,j) = E(j,i) = difference(i,j), \forall i,j \in \{0,1,\ldots,t-1\}$.
Using this method to calculate the difference leaks the overlapping set of targets that the users have votes defined for. To resolve this the following could be used to compute the difference:

$$difference(A,B) = \frac{(\prod_{i=0}^{t-1} [\![[X_{A,i} \neq \perp][X_{B,i} \neq \perp] distance(X_{A,i}, X_{B,i})]\!])}{\prod_{i=0}^{t-1} [\![[X_{A,i} \neq \perp][X_{B,i} \neq \perp]]\!]}$$

$[X_{A,i} \neq \perp]$ and $[X_{B,i} \neq \perp]$ can be determined by A resp. B. This method can introduce a significant overhead, as for every possible targets the distance between the votes is calculated.
The difference is only used, when comparing to a threshold to determine whether there exists an edge between two nodes. Therefore it is possible to remove the division from the computation of the difference. Instead, when comparing the difference to the threshold, the threshold is multiplied by $\prod_{i=0}^{t-1} [\![[X_{A,i} \neq \perp][X_{B,i} \neq \perp]]\!]$, to get the same result.
Both methods require communication between all possible pairs of users. This might be reduced by involving the targets of the votes in the process. The targets know who they had a transaction with, so they could send this information to the users. This way the users know with what other users they share targets, and thus have to compute the distance with.
Another method could be by using a pair consisting of the identity of the target of the vote and the vote itself. Calculate, for each pair of votes of the two parties A and B, the distance and multiply this with the boolean indicating whether the identities of the votes are the same. $(I_{A,x}, V_{A,x})$ is the pair of the identity and the vote of user A for target x. X is the set of targets that node A has a vote for, Y is the set of targets for user B.

$$difference(A,B) = \frac{\prod_{x \in X} \prod_{y \in Y} [\![[I_{A,x} = I_{B,y}] distance(V_{A,x}, V_{B,y})]\!]}{\prod_{x \in X} \prod_{y \in Y} [\![[I_{A,x} = I_{B,y}]]\!]}$$

This method requires $O(|X||Y|)$ multiplications and comparisons. This is worse than the first method, but does not leak the set of overlapping targets.

6.4 Oblivious distance-k cliques

Next we will discuss the oblivious algorithm in detail. The complete algorithm can be found in Algorithm 9. This variant is not yet completely oblivious, as the while and if statements are still included. In order to get a completely oblivious algorithm, the methods described in Section 6.1 can be applied.
We use the following vectors in the algorithm:

- E, containing for each node a set with the nodes it has an edge to.
- C, containing for each node the id of the cluster that it belongs to.
- M, containing for each cluster a set containing the members of the cluster.
- L, containing for each node a set that depicts to what clusters it is connected to via an edge, but that it is not part of itself.
- D, containing the estimated diameter for each cluster.

The ids in the vectors are stored in unary format, as they are used in the secure selection protocol described in Section 3.10. Vectors C and M are stored in one matrix, where one of the vectors can be seen as the rows and the other as the columns. This way updating one of the vectors would

also update the other. The vectors will stay synchronized, so updating the cluster a node belongs to, also changes the members of that cluster and vice versa.
Another advantage is that less memory is required, since we can store one of the vectors for free, as the elements in both vectors are the same.

Initialization (1-6) Initially there are n clusters each corresponding, where every node is a member of it's own cluster. Therefore, the clusters that a node is connected to, are the edges from that node. Since at this point every cluster consists of one node, the estimated diameter of the cluster is initially 0.

Selecting the bridge node (8) The body of the protocol is repeated until there are no more nodes associated with a bridge node. The bridge node is selected, where the total number of nodes in the corresponding clusters is the largest.

Create the tentative cluster (9-10) A list N is created consisting of the cluster of the selected bridge node and the associated clusters. Secondly, a list is created that contains the estimated diameters of the clusters in the first list. If cluster i occurs in the first list, the value of N_i is 1 and otherwise it is 0. Therefore the new list will only contain the estimated diameters of the clusters in the list. For the clusters that are not in the list, the estimated diameter in Q is 0.

Estimating the diameter (11-13) The indices of the largest two clusters are selected. The value of d depends on whether A is in one of the two largest clusters. If it is in one of the two cluster, either SELECT(SELECT(M, P_1), A) or SELECT(SELECT(M, P_2), A) is 1. They cannot both be 1, as a node can be in only one cluster at a time. So, d is 1 if A is in one of the two largest clusters and 2 otherwise. d is used in the calculation of the estimated diameter of the possible new cluster, when combining all clusters in N. The maximum diameter of this new cluster is the estimated diameter of the two largest cluster plus d.

Add the new cluster (14-21) The new diameter is smaller than or equal to the maximum path length. The list of members of the cluster to which the bridge node belongs will contain the new cluster. It is updated to contain the members of all clusters in N. $N_i * M_i$ will result in a 0-vector if cluster i is not in N, and a vector with the members otherwise. We can use summation because every node will be a member of at most one cluster. All members from the clusters in N, except the new cluster, are removed from the cluster member lists. After this, we assign the index of the new cluster to a temporary value S, if there is at least one cluster both in N and L_i, and a 0-vector otherwise. We then remove all occurrences of the clusters in N from L_i, and add the temporary value, if the estimated diameter of the new cluster has not yet reached its maximum diameter.

Remove the two largest clusters from each other (24-31) The new diameter is greater than the maximum path length, and the bridge node A is in one of the two largest clusters in N. First four temporary lists are constructed. U is an the same as L and V consists of all members of the clusters in U. W is the cluster that does not contain the bridge node, and X consists of the members of W. Next we remove W from the bridge nodes in V, and we remove U from the bridge nodes in X. If bridge node i is in V, then BIT-COMPL($V_i * W$) will give a vector with 0s at the places of the cluster in W. Multiplying this vector with the vector of associated clusters will set all the clusters in L_i that occur in W to 0. The same holds for BIT-COMPL($X_i * CC$). If node i is in X, than the cluster in U are removed from L_i.

Remove the largest cluster from bridge node A (33) The new diameter is greater than the maximum path length, and the bridge node is not in one of the two largest clusters in N. The largest cluster is removed from the associated clusters of the bridge node.

Algorithm 9 Secure algorithm to find distance-k cliques

for $i = 0$ to $n-1$ **do**
  $[\![C_i]\!] \leftarrow [\![i]\!]_{\mathbb{U}}$
  $[\![M_i]\!] \leftarrow [\![i]\!]_{\mathbb{U}}$
  $[\![L_i]\!] \leftarrow [\![E_i]\!]$
  $[\![D_i]\!] \leftarrow [\![0]\!]$
end for
while $\prod_{i=0}^{n-1} \prod_{j=0}^{n-1} [\![L_{i,j}]\!] \neq 0$ **do**
  $[\![A]\!]_{\mathbb{U}} \leftarrow \text{MAX}(\{\left[\!\left[\sum_{j=0}^{n-1} L_{i,j}\right]\!\right] \mid 0 \leq i < n\})$
  $[\![N]\!] \leftarrow \text{SELECT}([\![L]\!], [\![A]\!]_{\mathbb{U}})\text{SELECT}([\![C]\!], [\![A]\!]_{\mathbb{U}})$
  $\forall 0 \leq i < n : [\![Q_i]\!] \leftarrow (\text{MULT}([\![N_i]\!], [\![D_i]\!]), [\![i]\!]_{\mathbb{U}})$
  $([\![P_1]\!]_{\mathbb{U}}, [\![P_2]\!]_{\mathbb{U}}) \leftarrow \text{MAX2}([\![Q]\!])$
  $[\![d]\!] \leftarrow [\![2]\!]\,\text{SELECT}(\text{SELECT}([\![M]\!], [\![P_1]\!]_{\mathbb{U}}), [\![A]\!]_{\mathbb{U}})^{-1}\text{SELECT}(\text{SELECT}([\![M]\!], [\![P_2]\!]_{\mathbb{U}}), [\![A]\!]_{\mathbb{U}})^{-1}$
  $[\![R]\!] \leftarrow \text{SELECT}([\![D]\!], [\![P_1]\!])\text{SELECT}([\![D]\!], [\![P_2]\!])\,[\![d]\!]$
  if $R \leq k$ **then**
    $\text{UPDATE}([\![M]\!], \text{SELECT}([\![C]\!], [\![A]\!]), \prod_{i=0}^{n-1} \text{MULT}([\![N_i]\!], [\![M_i]\!])$
    for $i = 0$ to $n-1$ **do**
      $[\![M_i]\!] \leftarrow \text{CONDITIONAL-SWAP}([\![N_i]\!]\,[\![A_i]\!]^{-1}, [\![0]\!], [\![M_i]\!])$
      $[\![S]\!] \leftarrow \text{MULT}(\left[\!\left[[\prod_{j=0}^{n-1}(\text{MULT}([\![L_{i,j}]\!], [\![N_j]\!])) \neq 0]\right]\!\right], \text{SELECT}([\![C]\!], [\![M]\!]))$
      $[\![L_i]\!] \leftarrow \text{MULT}(\text{BIT-COMPL}([\![N_j]\!]), [\![L_i]\!])(\text{MULT}(\text{BIT-COMPL}([\![[R == k]]\!]), [\![S]\!]))$
    end for
    $\text{UPDATE}([\![D]\!], \text{SELECT}([\![C]\!], [\![M]\!]), [\![R]\!])$
  else
    if $[[\![P_1]\!] = \text{SELECT}([\![C]\!], [\![A]\!])] + [[\![P_2]\!] = \text{SELECT}([\![C]\!], [\![A]\!])] = 1$ **then**
      $[\![U]\!] \leftarrow [\![N]\!]$
      $[\![V]\!] \leftarrow \prod_{i=0}^{n-1} \text{MULT}([\![N_i]\!], [\![M_i]\!])$
      $[\![W]\!] \leftarrow \text{MULT}([\![[[\![P_1]\!]_{\mathbb{U}} = \text{SELECT}([\![C]\!], [\![A]\!])]]\!], \text{SELECT}([\![M]\!], [\![P_2]\!]))$
        $\text{MULT}([\![[[\![P_2]\!] = \text{SELECT}([\![C]\!], [\![A]\!])]]\!], \text{SELECT}([\![M]\!], [\![P_1]\!]))$
      $[\![X]\!] \leftarrow \text{SELECT}([\![M]\!], [\![W]\!])$
      for $i = 0$ to $n-1$ **do**
        $[\![L_i]\!] \leftarrow \text{MULT}(\text{MULT}(\text{BIT-COMPL}(\text{MULT}([\![V_i]\!]\,[\![W]\!])), \text{BIT-COMPL}(\text{MULT}([\![X_i]\!], [\![U]\!]))), [\![L_i]\!])$
      end for
    else
      $\text{UPDATE-ADD}([\![L]\!], [\![A]\!], [\![P_1]\!]^{-1})$
    end if
  end if
end while

All operations can be performed in the encrypted domain using homomorphic properties, secure multiplications and the secure selection and updating of vectors. The calculation can be performed by all parties, or a group of trusted parties.
We will now determine the upper bound on the number of iterations. We follow the idea by Edachery et al. that they use in determining the performance of the algorithm. Since we start with all nodes being bridge nodes, we have n bridge nodes. In the true part of the first if-statement, at least one bridge node is removed. If the false part is executed d times, at least one bride node is removed, according to Edachery et al., where d is the maximum node degree in the graph. The maximum possible node degree is n. An upper bound on the number of iteration is thus given if we have to perform the false part for all bridge nodes with the maximum possible node degree. Therefore, the while-loop has an upper bound of n^2 iterations. We can use this upper bound to make the algorithm oblivious. Performing n^2 iterations instead of using the while-loop with its condition will not give strange outputs, as nothing will happen anymore after the condition is satisfied, as no more clusters can be joined after this without exceeding the threshold k for the diameter.

7. Conclusions

In this thesis we introduced several secure building blocks that can be used in existing reputation systems. We introduced reputation systems in general and two specific systems, namely P2PRep and the knot-aware reputation system. We also discussed some of the previous work on anonymity in reputation systems. We showed several protocols to securely compute the weighted average, which is used in various reputation systems when computing the reputation score.

In Chapter 5 we discussed secure sorting algorithms. These could be used in, for example, the P2PRep reputation system by Aringhieri et al. [ADD+06]. They make use of the ordered weighted average, where the votes are first sorted in order to compute the reputation score. First we introduced a new oblivious counting sort algorithm, for both values from a two-valued domain and from a t-valued domain, where $t > 2$. These algorithms need less compare-exchange operations compared to the comparison-based algorithms. For the algorithm for values from a two-valued domain and from a t-valued domain, we need only $O(n)$ and $O(nt)$ compare-exchange operations respectively, compared to the lower bound of $O(n \log n)$ compare-exchange operations for comparison-based algorithms.

In the same chapter we also presented the Randomized Shellsort algorithm by Goodrich [Goo10]. This algorithm is oblivious and only needs $O(n \log n)$ compare-exchange operations. This is better than any previous sorting network. However, it sorts with very high probability, so not all inputs are sorted correctly. The depth of this sorting network is $O(n)$.

Wang et al. introduced a variant called Fast randomized Shellsort, that needs less compare-exchange operations [WLG+10]. The depth of their variant is still $O(n)$. The reduction of compare-exchange operations comes at the cost of less inputs being sorted correctly. We introduced a new variant we call Double brick randomized Shellsort. This variant uses the same number of compare-exchange operations as the Fast randomized Shellsort, but has a depth of only $O(\log n)$. In our experiments on random inputs, it sorts more inputs correctly compared to the Fast randomized Shellsort.

With regard to the randomized Shellsort algorithms, we considered the privacy issues that occur due to the fact that the inputs are not always sorted correctly. Since the algorithms and the random permutations are public, an adversary can try all possible inputs and learn their corresponding output. If a certain output is given, the adversary can look up the corresponding input, thus reducing the privacy of the users.

We present two different types of measures of anonymity for the algorithms. First we discuss the global anonymity. For this we introduce the conditional entropy, that gives an average on the anonymity that is provided by the algorithms for all users. The second type of anonymity is local anonymity. For this we introduce a measure that gives a lower bound on the anonymity that is provided for a certain input or output position.

In Chapter 6 we discussed secure clustering. Clustering is, for example, used in the knot-aware reputation system by Gal-Oz et al. [GOGH08]. In their system users are clustered based on their votes. Using these clusters when computing the reputation score results in more accurate scores. We start by presenting an algorithm to detect cycles in a graph and showed how to transform this algorithm to an oblivious variant. To transform algorithms to a secure and oblivious variant, we introduce a method to convert if- and while-statements to oblivious constructs. After this we presented the distance-k clustering algorithm by Edachery et al. [ESB99]. We also introduce a method to compute the similarity between users, based on their votes, in order to construct a

graph containing all users. Finally we introduced the secure and oblivious variant of the distance-k clustering algorithm. This variant makes use of a homomorphic cryptosystem and secure multi-party computation for providing the security.
In both the secure sorting and clustering algorithms we make use of secure multi-party computation. Though these techniques are proven secure, they might not be practical in all situations. Every protocol involves communication between the parties which results in a significant overhead with regard to time. It is possible for all parties providing input to take part in the computation. However, increasing the number of participants in the protocol also increases the communication and computational costs. An alternative would be to have a pool of trusted users. Each party selects a user he trusts, and all selected users compute the protocol together. This way, as long as at least one of the computing users is honest, the computation is still secure. The output can be re-encrypted using the keys of the parties providing the input, so the trusted users do not learn the plaintext output.

7.1 Future work

Based on our work we will now give some topics that are of interest for future work.

- Though the algorithm by Goodrich performs very well, it is still of interest to get a deterministic $O(n \log n)$ sorting network.
- The Double brick randomized Shellsort algorithm has a depth of only $O(\log n)$. The Randomized Shellsort algorithm sorts however more inputs correctly. Better performing randomized Shellsort algorithms of depth $O(\log n)$ might be possible.
- We gave two types of quantification for the privacy that is given by sorting networks. Computing these requires to test all possible inputs. Future work might be directed at finding quantification of privacy in sorting networks that are less computationally expensive.

Bibliography

[ADD+06] R. Aringhieri, E. Damiani, S. Di Vimercati, S. Paraboschi, and P. Samarati. Fuzzy techniques for trust and reputation management in anonymous peer-to-peer systems. *Journal of the American Society for Information Science and Technology*, 57(4):528–537, 2006.

[AKS83] M. Ajtai, J. Komlos, and E. Szemeredi. An 0 (n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9. ACM New York, NY, USA, 1983.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*, page 307, New York, New York, USA, 1968. ACM Press.

[BGN05] D. Boneh, E.J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Proceedings of 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 325–341, Berlin, 2005. Springer.

[CDN01] R. Cramer, I. Damgård, and J.B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology - EUROCRYPT 2001*, Lecture Notes in Computer Science, pages 280–300. Springer, 2001.

[DGK08] I. Damgård, M. Geisler, and M. Kroigard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography*, 1(1):22–31, 2008.

[DGK09] S. Dolev, N. Gilboa, and M. Kopeetsky. Efficient Private Multi-Party Computations of Trust. Technical Report 10-02, Ben-Gurion University of the Negev, Beer-Sheva, 2009.

[dHSSV09] S. de Hoogh, B. Schoenmakers, B. Skorić, and J. Villegas. Verifiable Rotation of Homomorphic Encryptions. In *Public Key Cryptography PKC 2009*, number 5443 in Lecture Notes in Computer Science, page 410, Berlin, 2009. Springer.

[DJ01] I. Damgård and M. Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. *Public Key Cryptography - PKC 2001*, pages 119–136, 2001.

[Elg85] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[ESB99] Jubin Edachery, Arunabha Sen, and Franz Brandenburg. Graph clustering using distance-k cliques. In Jan Kratochvyl, editor, *Graph Drawing*, volume 1731 of *Lecture Notes in Computer Science*, pages 98–106. Springer, 1999.

[GGOG09] E. Gudes, N. Gal-Oz, and A. Grubshtein. Methods for Computing Trust and Reputation While Preserving Privacy. In *Proceedings of the 23rd Annual IFIP WG 11.3 Working Conference*, volume 5645 of *Lecture Notes in Computer Science*, pages 291–298, Berlin, 2009. Springer.

[GJ04] Philippe Golle and Ari Juels. Parallel mixing. In *Proceedings of the 11th ACM conference on Computer and communications security - CCS '04*, page 220, New York, New York, USA, 2004. ACM Press.

[GOGH08] N. Gal-Oz, E. Gudes, and D. Hendler. A Robust and Knot-Aware Trust-Based Reputation Model. In *IFIP International Federation for Information Processing*, volume 263 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2008.

[Goo10] M.T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.

[GSV07] J. Garay, B. Schoenmakers, and J. Villegas. Practical and secure solutions for integer comparison. *Lecture Notes in Computer Science*, 4450:330, 2007.

[JQ09] A. Jøsang and W. Quattrociocchi. Advanced Features in Bayesian Reputation Systems. In *Proceedings of the 6th International Conference on Trust, Privacy and Security in Digital Business*, volume 5695 of *Lecture Notes in Computer Science*, pages 105–114, Berlin, 2009. Springer.

[JsI02] A. Jø sang and R. Ismail. The beta reputation system. In *Proceedings of the 15th Bled Electronic Commerce Conference*, pages 324–337, 2002.

[JW05] Geetha Jagannathan and Rebecca N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining - KDD '05*, page 593, New York, New York, USA, 2005. ACM Press.

[Knu73] D.E. Knuth. *The art of computer programming - Volume 3 - Sorting and searching.* Addison-Wesley, 1973.

[NR09] R. Nithyanand and K. Raman. Fuzzy Privacy Preserving Peer-to-Peer Reputation Management, 2009.

[Pai99] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT 99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Ped91] T. Pedersen. A threshold cryptosystem without a trusted party. *EUROCRYPT'91: Proceedings of the 10th annual international conference on Theory and application of cryptographic techniques*, 162(3):1119–1134, March 1991.

[RS05] S. Rajasekaran and S. Sen. PDM Sorting Algorithms That Take A Small Number of Passes. *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10–10, 2005.

[She59] D.L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 3(1):20–22, January 1959.

[ST04] B. Schoenmakers and P. Tuyls. Practical two-party computation based on the conditional gate. In *Advances in Cryptology - ASIACRYPT 2004*, volume 3329, pages 119–136. Springer, 2004.

[ST06] B. Schoenmakers and P. Tuyls. Efficient binary conversion for Paillier encrypted values. In *Advances in Cryptology-EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 522–537, 2006.

[WLG+10] G. Wang, T. Luo, M.T. Goodrich, W. Du, and Z. Zhu. Bureaucratic protocols for secure two-party sorting, selection, and permuting. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security - ASIACCS '10*, page 226, New York, New York, USA, 2010. ACM Press.