**IT UNIVERSITY OF COPENHAGEN**

*Bachelor Project Spring 2020*

*Digitisation of Elections*

# Oblivious transfer in voting protocols

**Supervised by:** Bernardo Machado David & Carsten Schürmann

AUTHORED BY:

**International Democratic Girlscouts Workers Party of ITU**

Sebastian N. Behrndtz . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . sbeh@itu.dk

Oliver E. Astrup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . olas@itu.dk

# Contents

# *Abstract*

*Digitisation of elections comes with at lot of problems, from security to trust in the digital solution. How can a voter trust that their vote has been correctly submitted and how do we securely give the voter this kind of confirmation, without the system knowing what was voted? The problem is old, but the digital setting is new. Voting with ballots has not changed a lot over the year. We have added anonymity, but the basic premise remains the same, even when utilising digital solutions.*

*In this paper we will explore the history of voting and examine the experiences from Norway and Switzerland with digital voting systems. We will also discuss a potential solution to the problem regarding submitting votes as stated above. We suggest that the problem can be solved by utilising* Oblivious Transfer (OT) *and we have implemented a prototype system as proof of concept based on the* Efficient Composable Oblivious Transfer (ECOT)

*Based on our results and the experiences from Switzerland, we believe that* OT *could be part of the solution, but it cannot stand alone. Several problems still remain and having a secure system does not guarantee a voter's trust, which is paramount for a system to be successful.*

# 1. Introduction

Digitalisation has been a buzzword for quite a few years now, and the digitisation of our society leads to a lot of interesting possibilities - among them the election processes in our democratic systems. Digital voting have several advantages, from improved vote counting and the possibility of sharing results faster to *I-Voting*. By using electronic voting machines, computers or other medias for voting, we can make the act of voting easier for people who suffer from different disabilities such as visual impairments, physical impairments etc. and thereby improve their ability to exercise their political rights [16]. Another aspect of digital voting is the prospect of making *Direct Democracy* feasible or using the system to more easily let citizens weigh in on important matters.

Experimentation with *I-Voting* has been going on for several years in eg. Estonia, Namibia, Norway and Switzerland [3] [16] [31] [26] and their experience shows us that, even though there are a lot of possibilities, several problems arise, for example with security, trust and ensuring anonymity of the voters.

One of the recurring issues is to ensure the voter's confidence in the vote they just cast. If a voter does not have confidence in the correctness of the digital vote, it does not matter how smart and secure it is. Either the voter will not make use of the digital solution, or worse, it could even damage voters' trust in the democratic system in general. When utilising digital solutions to vote online, how can we ensure that the voter's vote is counted correctly, that the voter can only vote once and finally give the voter confidence that the vote is counted correctly? While ensuring these things, the digital solution must also ensure the voter's anonymity and that the system is strong against coercion from a third party. And last, but not least, how do we ensure a voter's trust in the system and that it actually does what it is intended to do?

In this paper we intend to highlight the issue concerning *"voter's confidence in the correctness of their vote"* and look at a potential way of solving it. Our general idea is to give the user feedback on their voting so that they will know that they cast their vote successfully and that they voted for whom they intended. To do this we are proposing to make use of *OT* to generate a receipt as part of the voting process.

We will start out by trying to create a common understanding of what digital voting is in chapter 2, by taking a look at the historical perspective of voting with ballots, the experiences from the Norwegian trials with *I-Voting* and finally looking at problems and constraints. The technologies we intend to use for our proposed solution, will be introduced in chapter 3. We will also take a closer look at the discontinued project *CHVote* in chapter 4 and how they used *OT* in their *I-Voting* solution. As proof of concept we have implemented a prototype receipt generator using an *OT* protocol. The implementation details can be seen in chapter 5. Finally in chapter 6, we will discuss the results of our implementation and our thoughts on the solution. This will be followed by our conclusion in chapter 7.

# 2. Digital voting

In this paper we refer to digital voting instead of *Electronic Voting*. Firstly to differentiate between "digital" and "electronic". According to the Cambridge dictionary "digital" can be defined as: *recording or storing information as a series of the numbers 1 and 0, to show that a signal is present or absent,* [10] and "electronic" can be defined as: *(especially of equipment), using, based on, or used in a system of operation that involves the control of electric current by various devices,* [11]. The important factor here is the focus on storing data, since this is an important aspect of *I-Voting*. Secondly *Electronic Voting* includes both *E-Voting* and *I-Voting* which are very different things.

Since we are discussing a full digital solution, where every part of the process is digitised and using the internet, a more precise wording would be *Digital Remote Voting (DRV)*. But why would we even want to have a *DRV* process?

We have already mentioned some of the reasons in chapter 1, and to reiterate and expand on them, we find the main advantages are:

- Improved and faster vote counting and sharing.
- Voting will become more accessible, especially in rural areas with long and/or difficult transport routes to the nearest polling station.
- A faster process for the voter by avoiding unnecessary queuing and travel time, making it more attractive to vote.
- It will be easier for people with visual impairments, physical impairments or other disabilities to exercise their political rights.
- Elections will be easier to facilitate, since fewer people are involved directly in the process.
- It can pave the way for *Direct Democracy* and make it easy for politicians to include voters' opinions on specific topics.

In short, *DRV* could increase participation and make participants more involved in politics through better inclusion or even *Direct Democracy*.

In the following sections we explore the concept of trust in section 2.1 and how it is important for *DRV*, and give the reader a a brief overview of the history of voting using ballots, see section 2.2, and how voting has evolved over the years. This is followed by section 2.3 where we look at how *DRV* has been used in Norway and what we can learn from their trials. Finally we take a short look at constraints and possible problems that can arise with *DRV* in section 2.4.

## 2.1 Trust

How do we define trust? According to the Cambridge dictionary, trust can be defined as,

*"to believe that someone is good and honest and will not harm you, or that something is safe and reliable"*, *"to hope and expect that something is true"*, *"the belief that you can trust someone or something"* or if we take the American version as, *"to have confidence in something, or to believe in someone"*, *"to hope and expect that something is true"* - [12]

From this, it can be gathered, that the absence of trust, results in distrust in something, we believe it is bad, with ill intent and wishes to do harm. In a world without trust, the world can be presented like this:

*"We would never enter a taxi, never pay with coin or believe in what our doctor says. Furthermore, we would not know when and where we are born and might even still believe that the sun rotates around the earth."* - [34]

In relation to democracy and voting; We assume, if voters do not trust in the election, their participation will decline. If people trust in something, they are willing to work with it and accept the results. If they do not trust in something why should they make use of it or believe the results?

So why is this important? If we look at democracy as a machine, trust is one of the important cogs for making it run. Trust influences the participation of people and a democracy's legitimacy depends on people's participation. When the legitimacy falls, people might perceive election results as wrong or manipulated. This can result in making especially minority groups in society feel suppressed and misrepresented and thus lead them to opt out of society [40]. As such, the result might lead to the voter's feeling of having political opponents elected going from *"I did not vote for you!"*, to *"the majority did not vote for you!"*, an thus undermine the official authority, whether this is the truth or not.

A lack of trust can undermine the entire democratic system. Why follow our elected leaders if we do not believe in the system that placed them there as leaders? Trust is essential for the cogs of democracy to keep turning. Furthermore democracy is one of the cornerstones of the western world. If one of these cornerstones breaks down it could greatly influence our society. It might lead to deep rifts between groups of people, create instability, hurt the economy, lead to civil unrest or worse.

## 2.2 History

Voting dates back many years and has taken several forms, from the simple show of hands, voting with ballots, remote voting, secret ballots and today both *E-Voting* and *I-Voting*. Some of the earliest laws for voting with ballots, can be dated back to at least 140-130 B.C [41]. The process from back then, is very close to how voting is done today, with the exception of secrecy:

> *"The mode of voting from this time forward was as follows: As the voters entered the booths they were given ballots (tabellac). These ballots differed according to the subject under discussion ; if the vote to be taken was on some law the tablets were marked VR (uti rogas) and A (antiquo); if the assembly was gathered for an election, the tablets were plain, and the voters inscribed on them the name of the candidate for whom they wished to vote. Each citizen as he passed out of the booth of his voting unit deposited his ballot under the supervision of tellers (rogatores) and watchers (custodes) in baskets. When the voting was completed the baskets were carried off to some special place called the diribitorium, where they were emptied. Here the ballots were sorted and counted, and the results recorded. When this was done the result was announced [...]"*
>
> - Wolfson [41]

An early form of remote voting was postal voting, which can be traced back as far as the Roman Empire [6]. This type of voting is still in use today. More recent examples of remote voting includes telephone, FAX and internet and nowadays we can even vote from space, which was done in 1997 by an American astronaut [17].

Secret ballots are taken for granted in western democracies today and was first introduced in 1848 in France and in 1872 in Britain [8]. They weren't completely secure until 1913 with the addition of polling booths and envelopes.

Today secret ballots are the norm in the western world, but apart from that the process is still very similar to that of the ancient Greeks, which becomes very visible if we compare it to the current Danish system. According to the official homepage for the Danish parliament [13], parliamentary elections are done in the following way:

*"On election day, everybody who is entitled to vote in a general election can go to one of the polling stations located throughout Denmark and cast their vote. Voters will receive a poll card by post well in advance of the election day. The poll card will tell them where and when to vote. On election day, polling stations will have been established throughout the country, usually at town halls, schools and sports centres. Returning officers are responsible for overseeing that elections are conducted according to the rules. They also count the votes afterwards Voters hand in their poll cards at the polling station and receive a long ballot paper listing the names of the parties and the candidates running for election. The ballot is secret and votes are cast in polling booths so that nobody can see for whom people vote. Voters can put a cross either beside the name of a person or a party. [...] When the polling stations close, the votes are counted and the 175 seats can be distributed."*

It is clear that the core act of ballot voting has not changed a lot over the last two millennia: We have added secrecy and are a bit more explicit in the description, but apart from that, it is the same process and it still works. This means we can have a high degree of confidence in the system and that the weaknesses that the system might have can be accounted for.

## 2.3 Digital voting in Norway

The voting process in Norway [32] is very similar to the one used in Denmark, with the exception of being able to split up ones vote. The splitting of a vote in the Norwegian voting process, makes it possible to vote on more than one candidate. Norway initiated trials for *DRV* in 2011, which was discontinued again in 2013 [31]. In the following subsections we will describe the system itself and what we find to be the most relevant takeaways for our system.

### Simplified protocol

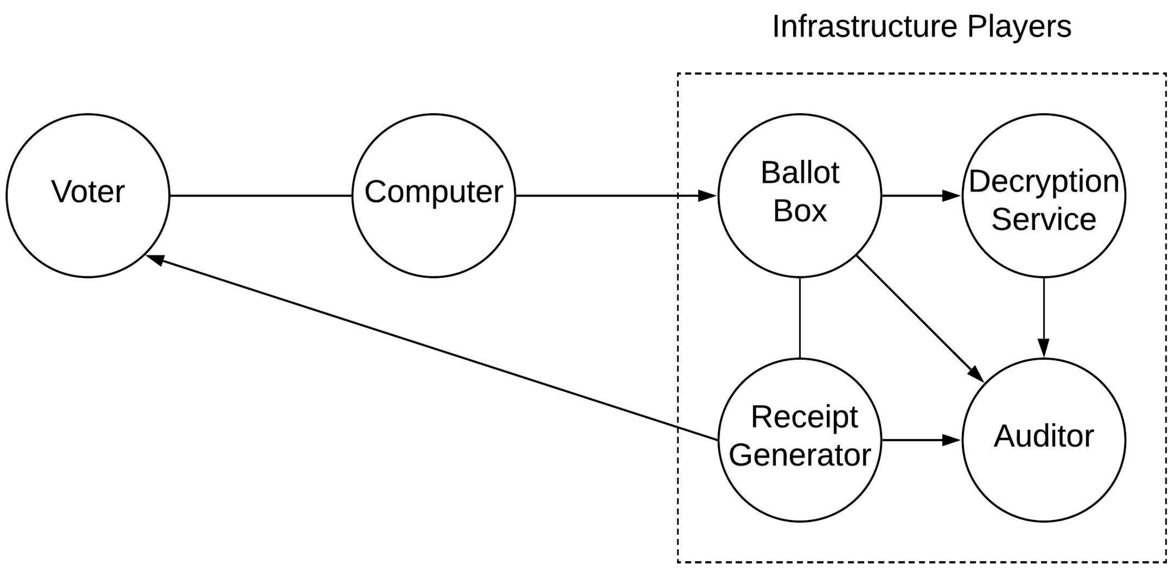


**Figure 2.3.1:** *Communication between parties in the Norwegian protocol.*

The parties communicate as shown in Fig. 2.3.1, and in this simplified version the ballot box knows which voter is communicating through the computer. The voter chooses a ballot, based on his choices from the given set of options, which the computer then encrypts using the election encryption key. This encrypted ballot is transferred to the ballot box, which then generates a receipt in cooperation with the receipt generator.

The receipt is sent directly to the voter, who can now check if the codes corresponds to the choices made. If the receipt codes match the choice, the voter accepts and otherwise the voter will know that something is wrong.

When the ballot box closes, all the encrypted ballots are transferred to the decryption service, which decrypts the ballots and publishes the result.

The auditor receives input from the ballot box, receipt generator and decryption device. This is used to verify the different parts of the process. Should the verification fail, the election will be cancelled.

### Security goals

According to Kristian Gjøsteen [18], four security goals can be identified. As might be obvious, every voter can only vote once, so only one vote per voter must be counted. If the computer used to submit the vote is honest, the vote must remain confidential, unless the information is leaked through receipt codes. The auditor will not cancel the election if no infrastructure parties are corrupt. Lastly - if the auditor does not cancel the election, the votes cast by honest voters must be counted, unless the voter submits a new ballot or complains about forgery.

### Trust

Trust is a major factor of elections [23] and even more so in *DRVs*, see section 2.1 for more details. In the Norwegian protocol, several steps are taken to improve voters' trust in the system.

The authentication service is based on *MinID*, which is a well-established service for authentication. The authentication system has been used for some time now and has this far proven itself to be safe [37]. This improves the users trust by "borrowing" the credibility of *MinID* and ensuring a higher level of confidence. Another step is a high level of transparency, where documents describing guidelines, responsibilities and the administrative context can be accessed publicly through the project website, along with the source code [37].

### Compromised computers

Since many computers are compromised, users will have to detect ballot tampering themselves. This is very difficult, especially since the average voter will not be able to do even simple cryptography, especially without computer assistance [18].

### Coercion

Another big problem, that no amount of cryptography can alleviate, is coercion. To combat this, the protocol allows the user to vote several times, with the last vote being the one counted. By allowing re-voting, a potential coercer can never be sure if the coercion worked and thus, this assists honest voters in actually voting as they want. On top of this, the voters are also allowed to vote once using a paper ballot, which will always be counted instead of any digital votes given [18].

## 2.4 Constraints and problems

There are many constraints related to a *DRV*, as is evident from the above sections, both as a result of the voting process itself and of the security perspectives and laws of the country in which the voting takes places[1]. For the security part, as already mentioned above, there are also lots of other things to consider, like *Denial of service* attacks, viruses, physical attacks, privacy and many more [1]. To stay focused on this paper's main subject, *"receipt generation with* OT*"*, we will focus on the constraints and problems specific to this part of the system, and we find the following:

[1]Which we have made a conscious choice not to go into.

- Receipts can be used as proof of one's vote, making it easier to sell the vote or to ensure coercion is successful.
- Since the receipt is digital, it gives an adversary an additional way to potentially find out what people vote.
- If the system itself is malicious, it may be possible to learn what all voters vote thereby removing anonymity.
- Every vote needs to be stored along with user details, so voters can change what they vote. This must be done without anyone else being able to figure out, what any individual's vote is.

# 3. Technologies

To implement receipt generation using *OT*, we use several technologies and intend to give the reader an overview of the most important ones in the following sections. First of, in section 3.1 we explain what *OT* is and then describe the specific *OT* extension we intend to use in section 3.2. In the implementation we will make use of *Elliptic Curve Cryptography (ECC)*, which we will give a short overview of in section 3.3. For the actual security implementation we will make use of the open source *Library Bouncy Castle (BC)*. Details about this can be found in section 3.4. The program itself is coded in Java and a short description of important uses from the language can be found in section 3.5

## 3.1 Oblivious transfer

*OT* is an important building block in cryptography and can be used for the construction of secure protocols [20]. *OT* is a protocol between two parties, Alice as the sender and Bob as the receiver. The objective is for Bob to receive and read one or more messages from a range of messages given by Alice, without Alice being able to gain knowledge of what message(s) Bob has chosen. Bob should also only be able to read messages he has chosen and no others [19].

A simple version of this is called the 1-out-of-2 *OT*, and is defined as $((x_0, x_1), \sigma) \mapsto (\lambda, x_\sigma)$ where $\lambda$ is the empty string. In this protocol, Alice has two different messages $x_0$ and $x_1$ and Bob has a single bit $\sigma$, that is his choice, so this can be 0 or 1. Then the purpose of the *OT* protocol is for Bob to receive and read $x_\sigma$, without Alice knowing the value of $\sigma$ [20].

Another more complex version is called the k-out-of-n *OT*, which in many ways works the same way as the 1-out-of-2 protocol. In this protocol, Alice has $n$ messages instead of two $(x_0, x_1, ..., x_n)$. Bob can chose $k$ messages, where it holds that $k \leq n$, which he can open and read. It is important to note that the messages Bob chooses, do not have to be in sequel. And like the 1-out-of-2 protocol, it should hold true that Alice does not know any of the k values that Bob has chosen, and Bob is not able to open and read messages that he has not chosen [19].

A combination of the two is also possible and would be called 1-out-of-n *OT*. It would be close to the k-out-of-n protocol, where Alice has $(x_0, x_1, ..., x_n)$. The difference here lies in Bob only having a single choice $\sigma$, and only receives $x_\sigma$, without Alice knowing what $\sigma$ is.

## 3.2 Efficient composable oblivious transfer

The *ECOT* is an *OT* protocol constructed by Bernardo David and Rafael Dowsley [9]. This protocol is *Universally Composable*, secure against active static adversaries based on the *Computational Diffie-Hellman Assumption* and proven to be secure in the *Global Random Oracle Model*. Two versions are described in their paper, one with selective failures and one without.

It builds on the 1-out-of-2 *OT*, described in section 3.1, where Bob has two different choices: $c_0$ or $c_1$. Depending on his choice, he will receive a corresponding message from Alice without Alice having any knowledge of what message Bob has chosen.

In subsection 3.2.1 we first describe the protocol extension with selective failures and then in subsection 3.2.2 we describe the one without. We have made use of a shortened version of the protocol, courtesy of Bernardo David, where we handle session id and randomness (R) differently from the original paper. In our version, session id is part of the underlying communication and the randomness is derived from p in the encryption scheme. The following shorthand's is used in the subsections:

- RANDOM - A random generator

- k - The security parameter
- $H_{Key}$, $H_{Ch}$, $H_{Pad}$ - Different hashing algorithms
- x - Can be both 0 and 1 (eg. $p_x$ means $p_0$ and $p_1$)
- c - The choice of 0 or 1
- ENC - An encryption algorithm
- DEC - A decryption algorithm

### 3.2.1 With selective failures

The following sections describe the protocol with selective failures, which is vulnerable to a malicious Alice, that has a 50% chance of guessing Bob's choice without alerting Bob. For a description of the protocol without selective failures see chapter 3.2.2.

#### Key pair generation

At the start of this protocol, Bob generates a *Key Pair* ($pk_c$ and $sk_c$), that corresponds to his choice ($c$) and will be used in the encryption of his vote. With a *Key Pair* Bob is able to send a *Public Key* to Alice, which she can then use to do encryption that only Bob can decrypt.

Bob also needs to generate a *Public Key* without a corresponding *Secret Key*, to obscure his choice from Alice. This is achieved through the following equations:

$$s = RANDOM(k) \tag{3.1}$$

$$q = H_{Key}(s) \tag{3.2}$$

$$pk_0 \star pk_1 = q \tag{3.3}$$

By using equation (3.3), Bob creates the new *Public Key* that is related to the *Public Key* in his *Key Pair*[1], and makes it infeasible to calculate which one is from the *Key Pair* and which is not. To obscure Alice, Bob will always send her $pk_0$ and $s$. She can then calculate $pk_1$ using (3.1), (3.2) and (3.3), but she will not know which key has a corresponding *Secret Key*.

#### Challenge

After the *Key Pair* generation, Alice creates a *Challenge* for Bob, so that they are able to exchange two secrets ($p_0$ and $p_1$) which are randomly generated by Alice. This challenge makes it possible for Bob to receive the secrets, which will later be used to encrypt his message and at the same time validate him as the correct recipient. Alice will generate a challenge using the following equations:

$$p'_x = H_{Ch}(pk_x, p_x) \tag{3.4}$$

$$p''_x = H_{Ch}(p'_x) \tag{3.5}$$

$$ch = p''_0 \oplus p''_1 \tag{3.6}$$

[1]eg. If Bob's choice is 0, then his *Public Key* from the *Key Pair* is $pk_0$, and he generates $pk_1$

She will then encrypt her secrets using the *Public Keys* that she received and calculated in the *Key Pair* generation.

$$ENC(pk_x, p_x) \tag{3.7}$$

Alice can now send the challenge and the two encrypted secrets to Bob, since he only has the *Secret Key* for the secret corresponding to his choice, using the following equation:

$$p_c = DEC(sk, ENC(pk_c, p_c)) \tag{3.8}$$

Using equations (3.4) and (3.5), Bob is able to calculate $p''_c$ which should be identical to the value calculated by Alice. Using this value, Bob is able to calculate $p_{1-c}$ using equation (3.6):

$$p''_{1-c} = ch \oplus p''_c \tag{3.9}$$

Bob still needs to obscure his choice to Alice and does this by always responding to her challenge with the value of $p''_0$. Therefore he will only make use of equation (3.9), when his choice is 1. To do this he makes use of a modified version of (3.9) equation that will always calculate to $p''_0$:

$$chr = p''_c \oplus (c \cdot ch) \tag{3.10}$$

Using this value as a response to her challenge, Alice can now verify that $chr = p''_0$. If the verification succeeds, she will know that he was able to decrypt one of her secrets and calculate a valid response, which means Bob is the correct recipient. If the check fails, Alice will abort.

### Sending the message

With a valid *Challenge* response, Alice can now send her messages ($m_0$ and $m_1$) to Bob. She encrypts them using her own generated secrets with the following equations:

$$\widetilde{p_x} = H_{Pad}(pk_x, p_x) \tag{3.11}$$

$$\widetilde{m_x} = \widetilde{p_x} \oplus m_x \tag{3.12}$$

Alice then sends the encrypted messages $\widetilde{m_0}$, $\widetilde{m_1}$ and the values calculated in (3.4) for $p'_0$ and $p'_1$ to Bob. Bob then verifies that the $p'_c$ value received from Alice corresponds to the one he calculated earlier using equation (3.4). He also computes $p''_{1-c}$ using equation (3.11) and then uses (3.6) with $p''_{1-c}$ and $p''_c$ to verify that this $ch$ corresponds to the previously calculated one. If any of these verifications fail, Bob aborts.

After validation, Bob is able to use equations (3.11) and (3.12) to calculate the messages, and since he only knows $p_c$ and not $p_{1-c}$ he is only ever able to decrypt one of the two messages sent by Alice.

## 3.2.2 Without selective failures

This version of the protocol protects Bob against a selective failure attack from Alice, so that Bob's choice stays hidden. This chapter will closely resemble chapter 3.2.1, as many of the equations are used in both protocols. However, this protocol has additional equations after the challenge is resolved.

### Key pair generation

The key pair generation is almost identical to the one used for selective failures. The only change is that Bob's choice ($c$) is not used. Instead Bob generates a random bit $c'$ to use instead of $c$.

### Challenge

The *Challenge* generation and response is identical to the challenge with selective failures. In addition to the challenge, Alice also generates the two random values $\hat{p}_0$ and $\hat{p}_1$ which she encrypts using (3.7) and sends to Bob with the encryption of $p_0$ and $p_1$.

### Sending the message

When the *Challenge* response is verified by Alice, she computes two random messages ($\hat{m}_0$ and $\hat{m}_1$) of the same length as the real messages, that she needs to send to Bob. Using (3.11) and (3.12) with these random messages along with $\hat{p}_0$ and $\hat{p}_1$ she is able to create the two messages to be sent to Bob along with the values of $p_0$ and $p_1$.

When Bob receives these values from Alice, he is able to verify her by using (3.7) with $p_0$ and $p_1$ and compare them to the values that he received from her for the challenge. He then verifies her values by using $p_0$ and $p_1$ to recreate the challenge using equations (3.4), (3.5) and (3.6) and compare if this challenge is identical to the one he received from Alice.

If all verifications are complete without problems, Bob is able to decrypt the random messages sent by Alice. By decryption of $\hat{p}_c$ using equation (3.8), he is able to replicate (3.11) and (3.12) and get the values of $\hat{m}_c$. As in the previous protocol, Bob is still only able to decrypt one of Alice's values because he only has the secret key for one of them.

To get the messages from Alice, Bob computes the value d, using the following equation:

$$d = c \oplus c' \tag{3.13}$$

Since $c$ and $c'$ are bit values, the value of d can be calculated using the following matrix:

| $\oplus$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Bob then sends his $d$ value to Alice. She then encrypts her messages with her random message using the following equations, which she sends to Bob:

$$m'_0 = \hat{m}_d \oplus m_0 \tag{3.14}$$

$$m'_1 = \hat{m}_{1-d} \oplus m_1 \tag{3.15}$$

When Bob receives these values from Alice, he is able to compute $m_c$ using the following equation:

$$m_c = m'_c \oplus \hat{m}_c \tag{3.16}$$

Since Bob is only able to gain the knowledge of $\hat{m}_c$ and not any knowledge of $\hat{m}_{1-c}$ he is not able to use (3.16), to compute $m_{1-c}$. He can therefore only get the value of $m_c$, which is the message equal to his choice.

## 3.3 Elliptic curve cryptography

*ECC* is a technique to generate secure encryption keys which are short when compared to other well used techniques like *RSA* and *Diffie-Hellman*, while still keeping the same level of security. Especially for sites that conduct large amount of secure transactions, *ECC* can be very useful, since a reduced key length will directly reduce the processing needed for each transaction [35].

The implementation of this protocol should work with any elliptic curve, and in our implementation we have used the *secp256r1* curve. This curve is *NIST*-approved and can be used in both *TLS* and *SSH* protocols. Working with the curve is faster than comparable methods, without compromising security [21]. However, we must take into consideration that *ECC* has not been around for as long as other techniques like *RSA* and might still contain undiscovered flaws or backdoors that have yet to be discovered [35].

*ECC* builds on the mathematical concept of elliptic curves. However, a description of this it out of scope for this paper.

## 3.4 Bouncy castle

Since we are not security experts, we make use of the *BC Library* to lower the risk of security problems in the protocol. By relying on a *Library* created by security experts who have been providing a free and open-source *Library* for almost 20 years [22], we can be more confident in our solution. The *Library* has been implemented to support the Java and C# programming languages.

The *Library* contains many different security protocols and elements and in our project we have made use of the *Library* for implementing our hash functions and elliptic curves. Details can be seen in the following subsections.

### Hash function

For hashing elements in the protocol, we make use of the *SHA3-256* algorithm provided by *BC*. It operates as a one-way hash function, which takes any length input and provides a 256 bit output. The produced output will always be the same when given the same input, but it will be infeasible to get the input from the output.

*BC* offers a plethora of hash functions, ranging from *SHA-1* to *SHA3-512*, and one could argue that using the *SHA3-512* would be more secure than the chosen *SHA3-256*. However, a longer hash function also results in longer computations and bigger packets to be sent over the network. In this protocol, we also make use of *ECC* that allows for lower key lengths and therefore we will not need long hash values, since our keys are already short and secure. Another reason is that in a real voting protocol, with lots of people voting at the same time, a reduction of computation time would be beneficial for the users, as they will get a faster response from the server.

### Elliptic curve cryptography

We make use of *BC* to do all operations related to *ECC*, like addition, subtraction and scalar multiplication. The *Library* also allows us to turn points on the elliptic curve into binary strings and to create points from binary strings. As mentioned in section 3.3, we make use of the *secp256r1* curve. This can be changed at any time by instantiating the program with a different curve, without changing any code.

All this mitigates the risks when trying to implement an elliptic curve, as the underlying mathematical calculations are very complex. With this *Library* we can be confident that the underlying mathematical calculations are done correctly and focus on aspects we are good at.

For more about *ECC* read section 3.3.

## 3.5 Java

The protocol is implemented in the Java programming language. We choose to use Java because *BC* as explained in section 3.4 is available for Java and C#, and we have more experience with Java in general. The following subsections will describe some of the technologies we have used from the Java *Library*.

### Secure random

The protocol currently makes use of the Secure Random implementation supplied by Java's standard security *Library*. This random implementation makes use of the underlying operating system to generate a *Seed* for the random generator. This is in contrast to other non-secure random generators' implementations, which often make use of the current time for randomness. The Secure Random uses process and thread id's, keystrokes, etc. at the time of the *Seed* generation, making it infeasible to guess or recreate the *Seed* [29].

### Mockito

For creating our unit tests, we have made use of the Mockito *Library* to mock interfaces. Mockito is a framework created for mocking and assisting with clean, simple and readable tests [25].

# 4. CHVote

Remote voting has been used in different countries in the last decade, one of these is the country of Switzerland, which has developed its own voting protocol called *CHVote*. This chapter is dedicated to studying the protocol, with section 4.1 explaining a brief history of the protocol and where it comes from. Then section 4.2 will explain some of the key elements of the protocol, handling what parties are involved and what phases are in an election. This is then followed by section 4.3 that is a brief explanation of the voting cards involved in an election, since they are an integral part of both creating trust and limiting the power adversaries. Since our work revolves around an *OT* protocol, section 4.4 will explain how OT is used in *CHVote*.

This chapter is written primarily using the paper *"CHVote System Specifications - Version 3"* from Bern University [19]. This paper is an extension of the *CHVote* 2.0 system that was developed earlier in a collaboration between the state of Geneva and Bern University but was discontinued in 2018. Therefore this version is developed independently by Bern university and is not currently in use by the Swiss state.

## 4.1 History of CHVote

The *CHVote* project started back in 2001, where the State of Geneva started developing the building blocks to implement a *DRV* system for its citizens. This resulted in several referendums to allow for *DRV* and resulted in the 2009 constitutional provision that was approved by a 70,2% majority, which made is possible to use *DRV* in Switzerland. The first version of the *CHVote* system did get its fair share of criticism, mainly from its lack of transparency and verifiability and due to the *Insecure platform problem*. Inspired by the Norwegian *DRV* project from 2011-13, where both high transparency and verifiability was achieved, an evolution of *CHVote* began, resulting in *CHVote* 2.0 in collaboration with Bern University in 2016. This collaboration aimed at adjusting the problems of the earlier protocol. However, in 2018 the state of Geneva stopped the project, and released the source code to the general public [14].

## 4.2 Overview of the protocol

This chapter will give an overview of two key elements in the *CHVote* protocol: Firstly, the different parties involved in the election. Secondly, a description of each of the 3 phases that the protocol has to go though to successfully complete a full election.

### 4.2.1 Parties

The *CHVote* system consists of 5 different parties/entities that all need to collaborate to ensure the success of the election. In this section we will briefly explain each of them and what their roles are in the protocol.

#### Election administrator

The administrator is the start and the end of the election. They are responsible for setting everything up correctly, so that the *election authorities* can do their job. This involves elements such as: Creating elections, adding candidates to elections and adding voters that are eligible to the elections. Then when the election is finished, it is the administrator who will tally the votes from the *election authorities* and publish the result.

#### Election authorities

The election authorities are used to ensure the integrity and privacy of the voters. They generate voting cards and the public key used for vote encryption. During the vote they are responsible for all vote confirmations, validation and submissions, and after they are responsible for mixing, tallying and verification. Overall they are a integral part of the system, and if all of them become corrupted then the system is broken. Therefore a lot of trust needs to be placed in these authorities, and a system always need at least one not corrupted authority, who can tell if others have become corrupted.

#### Printing authority

The printing authority is only used in the setup of the system, however, it is important that this authority is trusted. The printing authority is responsible for the printing of the voting cards described in section 4.3. These cards contain all the information that is needed for a vote to be valid and therefore a corrupt printing authority could vote on behalf of a voter in an election.

Another part of this process is the transportation of the cards from printing to voter, this process will likewise need a high level of trust to work.

#### Voting client

The voting client is the machine that the voter uses to cast their vote. It could be anything from a phone to a computer. The important thing is that voting clients can be infested by a malicious opponent, and can therefore not always be trusted, and it would be infeasible to be able to detect opponents in all voting clients. To counteract this *CHVote* makes use of the voting card, as it would be almost impossible for an opponent to try and guess the randomised values on a physical the voting card. It is also important to have a strong voting client, as it has to do several cryptographic computations and have communications with the election authorities.

#### Voter

The voter is the user that wants to make use of the system. They make use of the voting client to cast their vote and use the voting card to validate and confirm their vote.

### 4.2.2 Phases

An election in *CHVote* consist of three different phases: *Pre-election*, *Election* and *Post-election*. Each phase needs to be complete in order to complete a full election using *CHVote* and each contains a small list of subphases that also need to be completed in the right order.

#### Pre-election phase

This phase is necessary to set up an election and prepare it for the voters and ensure that they are able to actually vote in election. This is done using the voting cards generated during this phase. All communication in this phase is mainly between the election administrator and the election authorities and is the only phase in which the printing authority is involved.

The phase starts by **election preparation** where all voting cards are generated by the election authorities and all public credentials are saved so they can be used to identify voters during the election phase.

After the generation of the voting cards, the printing authority needs to start **printing the voting cards**. This is done by sending all voting cards from the voting authorities to the printing authority using a secure channel using symmetric encrypting. However, the rest of the printing relies on trust, since the voting cards contain secret information that could be used by others to vote on behalf of a voter. Therefore both printing authority and the delivery methods from print to voter need to be trusted and secure.

The last part of this phase is **key generation**, since each voter needs to encrypt their vote using a *Public Key*. This key is generated between all the election authorities, where they share their public keys, but keep their secret keys. When the votes needs to be read, each election authority will use their secret key, to decrypt part of the vote.

**Election phase**

The election phase is the core of the protocol. It is in this phase that the voters can cast their votes, and the system needs to make sure that votes are valid, while being able to keep voter anonymity ans also being able to verify that votes are correct. For each voter, there are three distinct steps to go though in this phase. What is important in this phase, is that it is all or nothing; if anything goes wrong during a vote, then the vote session aborts and needs to be started again.

The first step is the **Candidate Selection** step. This is the most simple step as the voter gets all the candidates that they are allowed to vote for in the elections, and makes their decisions. Many of the parameters for this step are already handled during election preparation by the election administrator.

The second step is the **Vote Casting** step. Here the voting client encrypts the voter's selections, and sends them to the election authorities, who need to both confirm that the ballot is valid and that the voter is allowed to vote before responding with the correct verification codes, so the voter knows that their selections are what is being tallied later.

The last step is the **Vote Confirmation** step. The voter only ends here when the verification codes from the previous step are correct. The voter uses this step to confirm to the voting authorities that their vote was correct and then receives a last code to make sure that everything is correct. The voting authorities then need to save the vote so it can be tallied, but also save the voter to make sure the voter doesn't vote again.

**Post-election phase**

The post-election phase is the last phase in the protocol. It is only run once when the election is over and the main focus of this is the tallying of the election result. However, we also need to make sure that no vote is able to be backtracked to a specific voter.

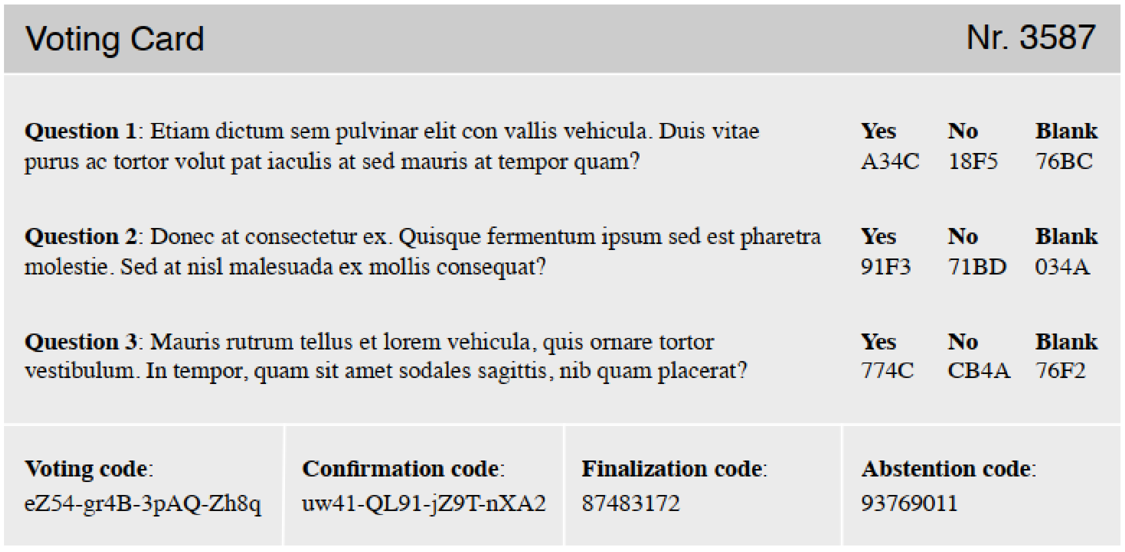
The phase starts with a cryptographic **mixing**. This is important, since it removes the link between a voter and their vote. The mixing is done by each election authority in turn, by shuffling the votes, before handing the votes to the next authority. After the mixing is completed, the **decryption** of the votes begin. This is again done by each of the election authorities in turn, so that after decryption each authority sends their partial decryption to the election administrator. With the partial decryptions from each authority, the election administrator is able to obtain the decrypted plaintext of each vote and then tally the result of the election. When the tally is complete, the administrator will publish the result.

After this there is an optional subphase called **inspection**. This is where voters can inspect the result and their votes to ensure that the election was run correctly. The subphase is optional, since it is not needed for the election, but it is there to build up trust that the election was correct. During the inspection, all the election authorities publish their relevant finalization and abstention codes, which each voter can use to check against their own voting card.

## 4.3 The voting card

The *CHVote* protocol makes use of voting cards. These are used by the voters to validate their vote in a given election. It contains a lot of different codes, as show in figure 4.3.1, and all the codes are secret and used for different purposes throughout the protocol. The small codes next to each question/election are **Verification codes**. The user will use the codes to check against the response they received, to see if they are correct. The codes are written using a base 16 system, giving the system 65.536 unique verification codes. The **Voting code** is used to identify the voter. However, the code is never used directly. Instead your public identification can be derived by the voting client using the voting code. The **Confirmation code** is used during vote confirmation. Here voters respond to the verification codes with their confirmation code to tell the election authorities that the codes were correct. Like with the voting code, the Confirmation code is never sent directly, instead the response is derived from this code. The **Finalization code** is the final code used during vote confirmation, and is only shown to the voter by the voting client. The voter can then respond if the code matches the code they have on their voting card and when an election is completed, all **Finalization codes** are published.

The **Abstention code** is a different from the other codes, as it is not used directly in the protocol. Instead, if the voter decides not to vote at all. Then when the election is over, they will be able to find their abstention code on a public list and can verify that their vote was not used in the election.



Voting Card | Nr. 3587

| | | | |
|---|---|---|---|
| **Question 1**: Etiam dictum sem pulvinar elit con vallis vehicula. Duis vitae purus ac tortor volut pat iaculis at sed mauris at tempor quam? | **Yes** A34C | **No** 18F5 | **Blank** 76BC |
| **Question 2**: Donec at consectetur ex. Quisque fermentum ipsum sed est pharetra molestie. Sed at nisl malesuada ex mollis consequat? | **Yes** 91F3 | **No** 71BD | **Blank** 034A |
| **Question 3**: Mauris rutrum tellus et lorem vehicula, quis ornare tortor vestibulum. In tempor, quam sit amet sodales sagittis, nib quam placerat? | **Yes** 774C | **No** CB4A | **Blank** 76F2 |

| **Voting code**: | **Confirmation code**: | **Finalization code**: | **Abstention code**: |
|---|---|---|---|
| eZ54-gr4B-3pAQ-Zh8q | uw41-QL91-jZ9T-nXA2 | 87483172 | 93769011 |

**Figure 4.3.1:** *An example of a* CHVote *Voting Card, that contains validation codes for 3 different elections. It contains a plethora of different codes, that is needed for the completion of the protocol. - [19] p. 14*

## 4.4 How CHVote makes use of oblivious transfer

It is only during the Election phase of the *CHVote* protocol that *OT* is utilised for vote casting and confirmation. The OT protocol used is a k-out-of-n protocol as described in chapter 3.1. This allows for multiple elections to be held at the same time. An example of this can be seen in figure 4.3.1, where it contains 3 different elections and its corresponding codes for **Yes**, **No** and **Blank**, so this essentially becomes a 3-out-of-9 *OT* protocol.

The OT protocol sends the voters' encrypted votes to the election authorities. Using these and the pre-generated vote matrix for the voter, the election authority is able to partially respond to the voters' encrypted vote. To form the full response, the voter needs this partial response from each election authority. With the full response from each different election authority, the voting client is then able to recreate the verification codes, and the voter can check that they are identical to the ones on their voting card. The protocol is an extension of the OT-Scheme by Chu and Tzeng, which has a overall asymptotic running time for the sender of $O(nk)$.

The same principle holds true for the finalization code, where each of the election authorities will respond with the finilizations they each have, which together form the voters' finalization code that confirms that the vote was a success.

# 5. Implementation

The implementation of the prototype *OT* protocol for receipt generation is based on the Efficient Composable Oblivious Transfer with and without selective failures, described in section 3.2. In a real system a 1-out-of-n[1] *OT* extension should be used instead, but for the purpose of creating a prototype and using it as proof of concept, we find the chosen *OT* extension, with or without selective failures, to be okay. Since this is only a prototype, we have taken a naive approach and do not take malicious input or actions into account. We have therefore not implemented proper error handling and we simply close the program if any exceptions are thrown.

Our overall thoughts on the system architecture can be found in 5.1. The implementation details for receipt generation using *OT* can be found in section 5.3. This section will be more in depth compared to other sections. Each method is described to make it clear where and how *OT* is used. To get a better understanding of how the receipt generation could work in a real voting system, we have implemented a simple voting simulator. The details of this can be seen in section 5.4. The simulation system is meant as a visualisation of the concept, so there is no testing or error handling beyond the bare necessities. Section 5.2 provides an overview of the combined systems and the general ideas behind it. The testing strategies for the receipt generation system are explained in section 5.5. If the reader wants to run the source code, an explanation on how to do this is given in section 5.6.

## 5.1 Architecture

When designing the architecture we have tried to adhere to the *SOLID-principles* and we had a large focus on being able to switch out subsystems. We chose to do this, in order to make it easier to do testing, which is written about in section 5.5, and to make it easier to switch out the implementations. This enables the use of both *ECOT* with and without selective failures, but also easier testing of different types of *OT* in the future, and to implement improvements to different parts of the system, without having to make changes to more than a single class. To achieve this we have made extensive use of the *Bridge Pattern*.

To handle data transportation we have created several *Data Transfer Objects (DTOs)*. These includes the *Ciphertext* and each of the steps that are transmitted during the *OT* protocol as well as messages transmitted over *TCP*.

## 5.2 Overview of the combined systems

The idea behind the combined system is to show that receipt generation could work in an actual system. In figure 5.4.1 it is shown how the combined system is meant to work, when a **Voter** submits a vote to a **BallotBox**. Both the **Voter** and the **BallotBox** use the **ReceiptGeneration** system. Upon receiving a vote, a receipt is generated on the **Server** and is sent to the **Client**. Both sides are dependent on the same underlying **CryptoSystem**. When an election has ended, all votes from a **BallotBox** are sent to the decryption service, which then tallies and announces the result.

[1]Or k-out-of-n if several choices can be made

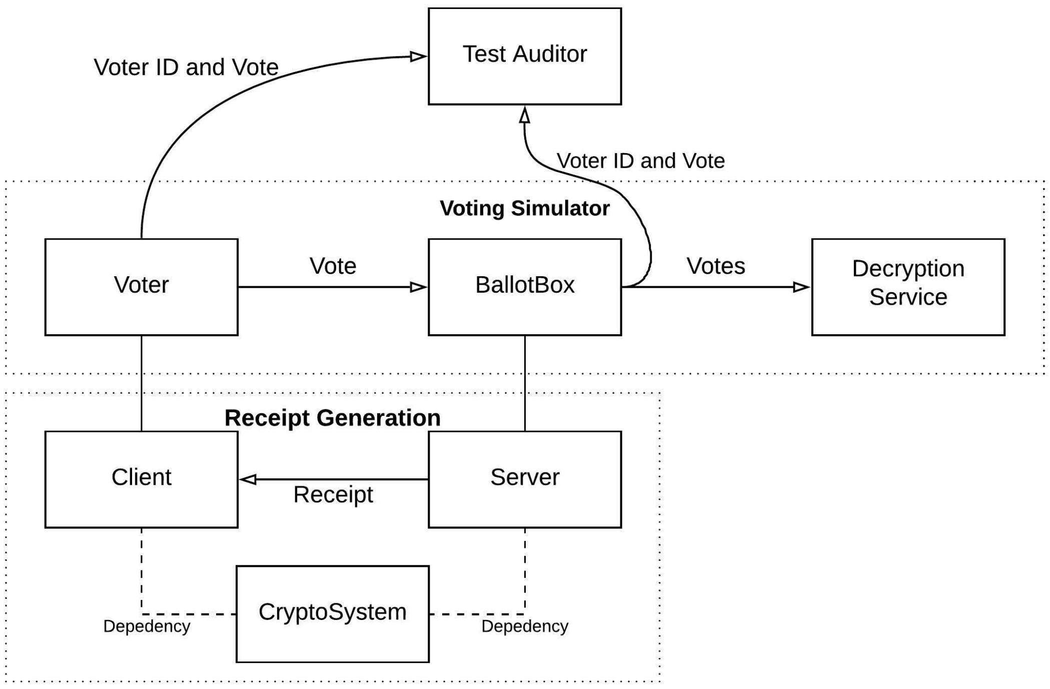


**Figure 5.2.1:** *Overview of the combined system*

In our model we added an **TestAuditor**. The purpose of the **TestAuditor** is to verify that the final tally is correct. The **TestAuditor** gets the user id and vote from every voter, once they have accepted the result of a vote. When the election is over the **TestAuditor** receives a list of all user ids and corresponding votes from the **BallotBox**. If the list from the **BallotBox** corresponds to all the information gained from the voters, the **TestAuditor** declares the election a success. It also prints a full list of every user, their final vote and number of times they voted.

The idea for this setup is based on the Norwegian model in figure 2.3.1, where we merged the voter and computer into one entity and made the receipt generator a part of both the client and server side. The **TestAuditor** is a stand-in for the auditor. We chose the name **TestAuditor** instead since it actually monitors every voter, with no secrecy involved and we really want to emphasise that this kind of auditor should never be used in an actual system with secret ballots.

The combined system makes use of two global classes. The first is **Constants**, which provides an easy place to change various variables throughout the program, like the names of algorithms, curves, security providers, security parameters, etc. The second is the **Provider**, this class is intended as an easy place to change between implementations, by changing the return value to the class currently being used, for example changing between *ECOT* with or without selective failures.

All communication is sent using *TCP*, by transmitting *DTOs*. Contrary to a real system, all information in the simulation is unencrypted and sent in plain text.

## 5.3 Receipt generator

The receipt generator has three subsystems, which can be seen in figure 5.3.1. They each represent different parts of the *OT* protocol. The **Client** subsystem represents Bob, where all steps are initiated in **RGClient**, the computations are handled in **RGClientLogic** and communication with Alice is done through **RGClient-Communicator**. As the reader might have guessed, the **Server** subsystem represents Alice, where all steps are initiated in **RGServer**, the computations are handled in **RGServerLogic** and communication with Bob is done through **RGServerCommunicator**. Lastly the **CryptoSystem** is used as the underlying building blocks of the protocol, to encrypt, decrypt, hashing and key generation.

The **RGClient** is supposed to be used by a **ReceiptHandler** in the voting system and the **RGServer** is used by the **ReceiptGenerator**. By providing a standardised interface for both of them we allow the underlying logic to be switched out with different types of *OT* extensions, as long as they are 1-out-of-2 or 1-out-of-n, without the overall voting system having to change as a result. In our original design we forgot to take this into account, which means that in our current implementation the steps of the protocol are part of the **RGClient** and **RGServer**. This should be separated at a later stage to improve our adherence to the *SOLID-principles* and improve our testing. Two types of **RGClient**/**RGServer** have been implemented based on the *ECOT* described in section 3.2, one with and one without selective failures.

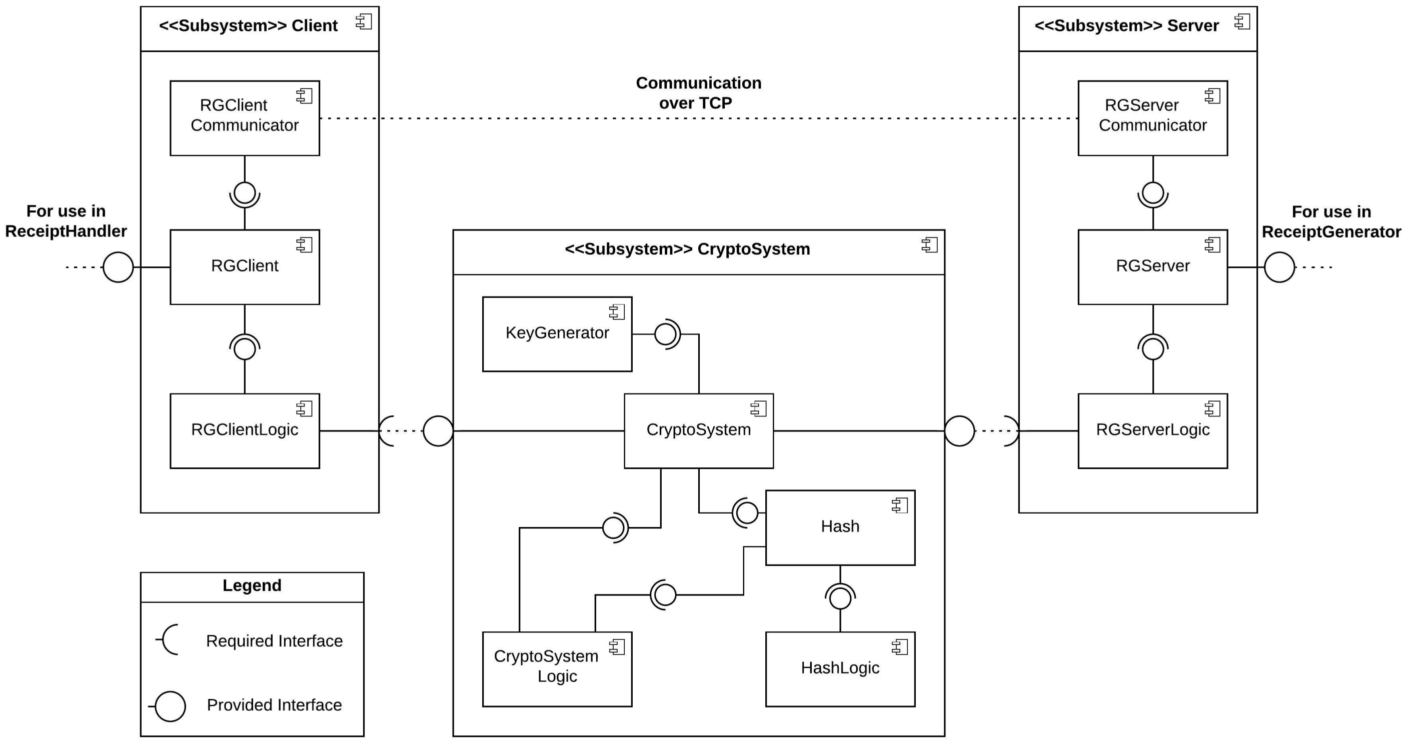


**Figure 5.3.1:** *Breakdown of the receipt generation system, showing inbound and outbound communication channels and class relations.*

Both the **RGServerLogic** and **RGClientLogic** make use of the same underlying **Cryptosystem**, separated by an interface so it can be changed as needed by the chosen *OT* extension.

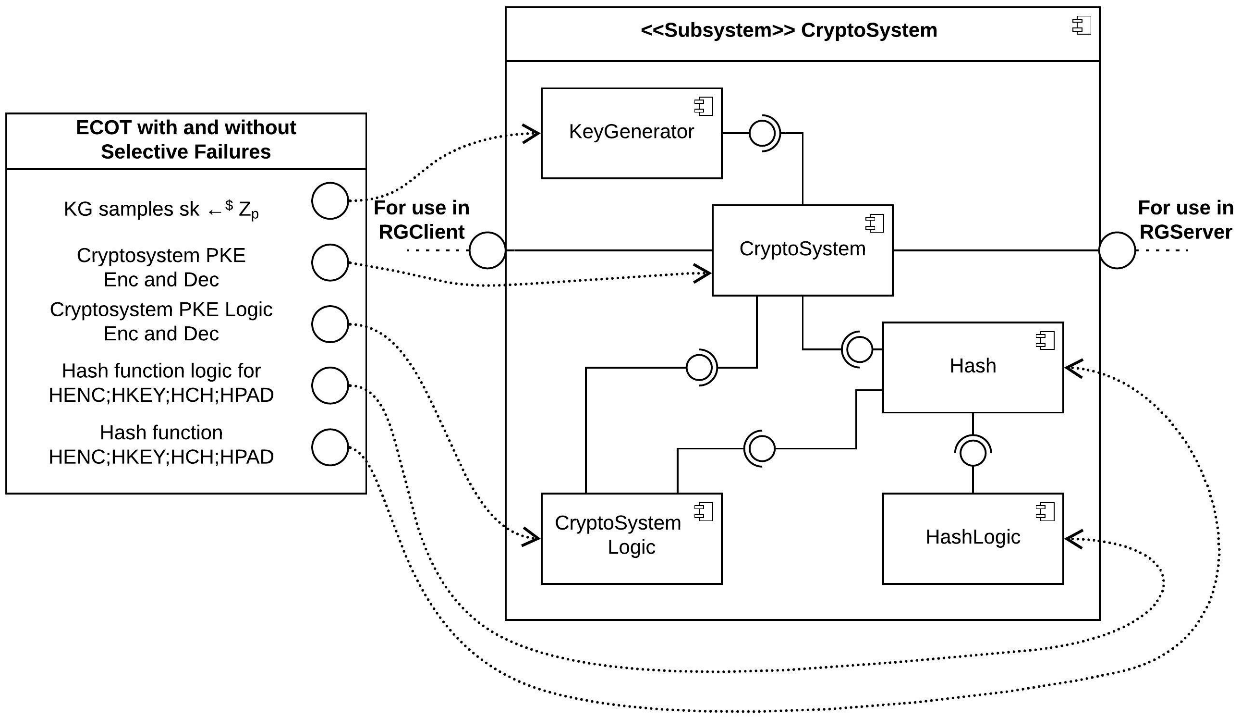
More in depth details about each subsystem and how their classes are implemented can be found in the following subsections. See subsection 5.3.1 for details on the **Cryptosystem**, subsection 5.3.2 for details on the **RGClient** and subsection 5.3.3 for the details on **RGServer**. In all of these subsections we mention binary strings and ciphers, which are implemented as byte arrays and *Elliptic Curve Points (ECPoints)* respectively.

Apart from the three subsystems, the receipt generator also makes use of the following helper classes:

- **RandomBigInteger**, which is used to securely generate a random big integer between 0 and n by utilising Java's secure random. See section 3.5 for details on secure random.
- **XOR**, which takes two byte arrays and does an XOR operation on them. It has two methods, one for byte arrays of the same length and one for different lengths. In the second method the second input is the message that has to be scrambled and the first is the scrambler. The second method works by shortening or increasing the length of the scrambler so the result is both arrays being of equal length and then making use of the first method to do the actual XOR.

### 5.3.1 Cryptosystem

The Cryptosystem contains two important building blocks for the *ECOT* protocol, the cryptosystem PKE and hash functions. For an overview of where different parts of *ECOT* is implemented see Fig. 5.3.2. Both versions of the *ECOT* use the same Cryptosystem.



<<Subsystem>> CryptoSystem
ECOT with and without Selective Failures
KG samples sk ←$ Z_p
Cryptosystem PKE Enc and Dec
Cryptosystem PKE Logic Enc and Dec
Hash function logic for HENC;HKEY;HCH;HPAD
Hash function HENC;HKEY;HCH;HPAD
For use in RGClient
For use in RGServer
KeyGenerator
CryptoSystem
Hash
CryptoSystem Logic
HashLogic

**Figure 5.3.2:** *OT usage in the **CryptoSystem***

#### CryptoSystem

The **CryptoSystem** implements the Enc and Dec algorithms and provides indirect access to key generation and hash functions. Dependency for both **Hash** and **CryptoSystemLogic** are set in the setup method. The method for key generation returns a new key generator which has been set up. **hkey**, **hpad** and the two different **hch** methods, are used to get access to different hash functions. This is done by calling the corresponding hash function in the **Hash** class. Enc and Dec are described in depth below:

- **Enc** takes two *ECPoints*, the *Public Key* and the message to be encrypted, and returns a *Ciphertext* containing two ciphers. This is done by computing randomness based on the message, and then using the randomness to compute cipher1 and then computing cipher2 by using the *Public Key*, message and randomness. These two ciphers are then used to create a new *Ciphertext*. All computations are done using **CryptoSystemLogic**.

- **Dec** takes an *ECPoints*, the *Public Key*, a *BigInteger*, the corresponding *Secret Key* and the *Ciphertext* to be decrypted and returns the decrypted message. The message is first extracted and then the same steps are done as in enc. If the newly calculated ciphers equal the equivalent ciphers in the *Ciphertext*, the message is returned. In case they are not equal, the program will abort. All computations are done using **CryptoSystemLogic**.

**CryptoSystemLogic**

This class handles all the underlying computations done in Enc and Dec. It is dependent on **Hash**, which is set in the setup method, for using hash functions that are part of the computations. Methods are described in depth below:

- **ComputeRandomness** takes an *ECPoint* and then computes randomness by giving the point to the hash function *HENC* along with the order of the elliptic curve provided by **Constants**. The result of this is returned.

- **ComputeCipher1** takes a *BigInteger* and computes a new *ECPoint* by multiplying the elliptic curve provided by **Constants**′ base entry and multiplying it with the *BigInteger*. The result of this is returned.

- **ComputeCipher2** takes two *ECPoint* and a *BigInteger*. A new *ECPoint* is computed by multiplying the first given *ECPoint* with the *BigInteger*. The result of the addition of the second *ECPoint* with this new *ECPoint* is then returned.

- **ExtractMessage** takes *BigInteger* and a *Ciphertext*. An *ECPoint* is calculated by multiplying the first cipher with the *BigInteger*. This point is then subtracted from the second cipher and the result of this is returned.

**Hash**

Implementations of *HENC, HKEY, HCH* and *HPAD* can be found in **Hash**. These implementations are dependent on *HashLogic* to do the actual computations. Dependencies are set in the setup method. Methods are described in depth below:

- **HENC** takes an *ECPoint* and a *BigInteger*. First a binary string is generated from the string "HENC", which is then used together with the *ECPoint* to compute an array of binary strings. Next the combined length of all the binary strings is computed. The combined length and array of binary strings is then used to compute a concatenated binary string of all the binary strings, with the binary string of "HENC" as the first part. Following this the concatenated binary string is hashed into a new binary string. All these computations are done using **HashLogic**. Finally a positive *BigInteger* is computed based on the hashed binary string and the modulo of the *BigInteger* received in the beginning. The value of this is then returned.

- **HKEY** takes a binary string. First a binary string is generated from the string "HKEY", which is then added to an array together with the given binary string. Next the combined length of all the binary strings is computed. The combined length and array of binary strings is then used to compute a concatenated binary string of all the binary strings, with the binary string of "HKEY" as the first part. Following this the concatenated binary string is hashed into a new binary string. All these computations are done using **HashLogic**. The hashed binary string is then used to generate a positive *BigInteger* which is used to compute an *ECPoint*, by multiplying the elliptic curve provided by **Constants**' base entry with the generated *BigInteger*.

- **HCH** takes two *ECPoints*. First a binary string is generated from the string "HCH", which is then used together with the two *ECPoints* to compute an array of binary strings. Next the combined length of all the binary strings is computed. The combined length and array of binary strings is then used to compute a concatenated binary string of all the binary strings, with the binary string of "HCH" as the first part. Following this the concatenated binary string is hashed into a new binary string and this value is returned. All these computations are done using **HashLogic**.

- **HCH** takes a binary string. With the exception of adding the given binary string and the generated one to an array of binary strings, instead of computing it using **HashLogic**, it is identical to the **HCH** described above.

- **HPAD** takes two *ECPoints*. This method is identical to the first **HCH** described above, with the exception that the generated binary string uses the string "HPAD" instead.

### HashLogic

The computations that are needed for *HENC, HKEY, HCH* and *HPAD*, are done using **HashLogic**. Upon initialisation the security provider and hash algorithm is set up by using the value set in **Constants**. Methods are described in depth below:

- **ComputeBinaryStrings** takes a binary string and any number of *ECPoint*. It converts every *ECPoint* to a binary string by using the built in encoded method. An array of all the binary strings are then created, with the given binary string in place 0, and returned.

- **ComputeConcatenatedBinaryStringLength** takes an array of binary strings. The length of each binary string in the array is added together and the result are then returned.

- **ComputeConcatenatedBinaryString** takes an array of binary strings and an int. A new binary string is created of the given int's length. Every binary string in the array is then concatenated by merging them into the new binary string in succession from entry 0 to entry n. The concatenated binary string is then returned.

- **Hash** takes a binary string and hashes it using the set algorithm and security provider. The output of the hashing operation is then returned.

### KeyGenerator

A generate method can be called to create a *Secret Key* and the corresponding *Public Key*. The *Secret Key* is generated by using **RandomBigInteger** to generate a random *BigInteger* within the order for the elliptic curve set in **Constants**. The *Public Key* can then be generated by multiplying the base entry of the previously used curve with the *Secret Key*. Both values can be retrieved with getter methods.

### 5.3.2 Client

This subsystem contains the implementations of Bob's actions, as described in the *ECOT* protocol. For an overview of where different parts of *ECOT* is implemented see Fig. 5.3.3. When describing the implementation of each class we will first describe how it is done for the version with selective failures, and then how the version without selective failures differentiate.

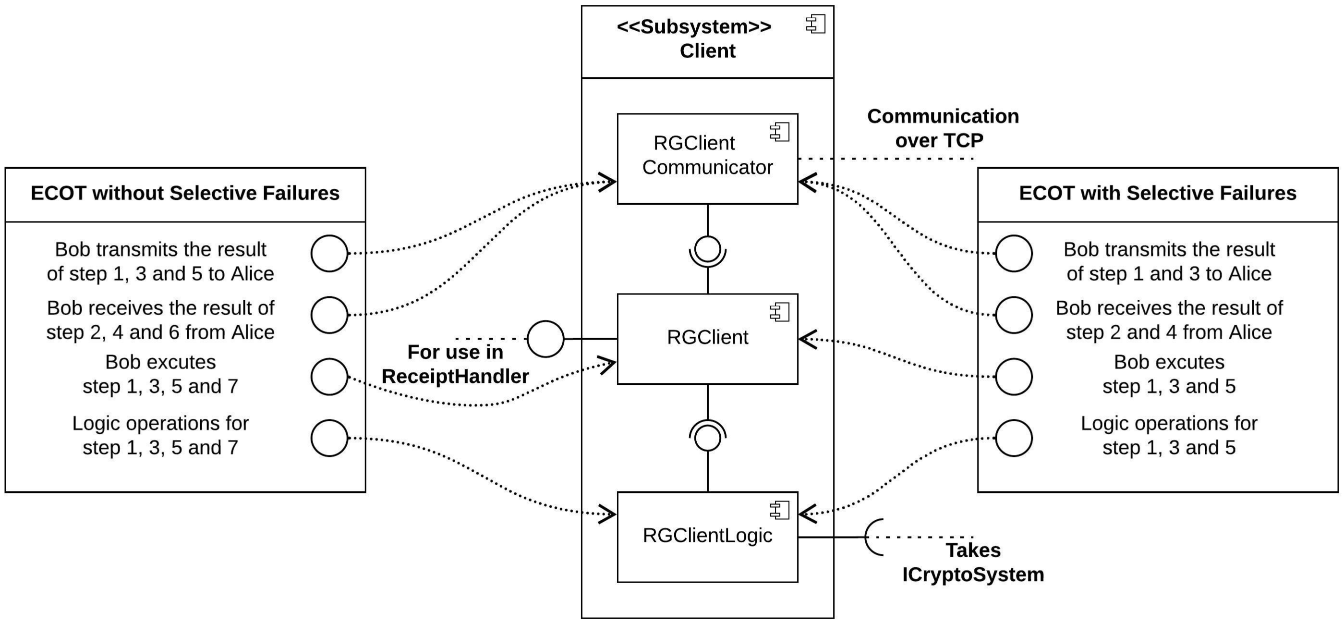

<<Subsystem>>
Client
RGClient Communicator
RGClient
RGClientLogic
Communication over TCP
For use in ReceiptHandler
Takes ICryptoSystem
ECOT without Selective Failures
Bob transmits the result of step 1, 3 and 5 to Alice
Bob receives the result of step 2, 4 and 6 from Alice
Bob excutes step 1, 3, 5 and 7
Logic operations for step 1, 3, 5 and 7
ECOT with Selective Failures
Bob transmits the result of step 1 and 3 to Alice
Bob receives the result of step 2 and 4 from Alice
Bob excutes step 1, 3 and 5
Logic operations for step 1, 3 and 5

**Figure 5.3.3:** *OT usage in the* ***Client***

#### RGClient

When setting up the **RGClient** using the setup method, Bob's choice $c$ is given as an int and the dependencies for both **RGClientLogic** and **RGClientCommunicator** are set. Everything that is computed, generated or verified in the step methods uses **RGClientLogic** to do the actual computation, generation or verification. Methods are described in depth below and if nothing else is written the first time a value is mentioned, the reader can assume that it is a binary string.

- **getReceipt** is a method for going through each step of the protocol in succession, by calling the internal step methods in ascending order, eg. step1 then step3 and so on. In between each step, the **RGClientCommunicator** is used to get the result of the **RGServer**'s steps. The information from each step is set to fields via the receiveStep's methods. The final step produces a binary string which is returned. The only difference for the version without selective failures is that there are more steps to go through.

- **step1** is the implementation of step1 from the *ECOT* protocol. Firstly, a *Secret Key* $sk$ in the form of a *BigInteger* and the *Public Key* $pk_c$ represented by an *ECPoint* is generated and the random binary string $s$ is sampled. The binary string is used to compute the *ECPoint* $q$, which is then used together with $pk_c$ to compute a new *ECPoint Public Key*, called $pk_{1-c}$. When this is done, **RGClientCommunicator** is used to transmit step1 by using the parameters $c$, $s$, $pk_c$ and $pk_{1-c}$. In the version without selective failures a random bit $c'$ is sampled and in the call to transmitStep1 $c$ is replaced with $c'$.

- **receiveStep2** sets the fields for $ch$, and the two *Ciphertexts* $ct_0$ and $ct_1$. In the version without selective failures the fields for two additional *Ciphertexts* $\hat{ct}_0$ and $\hat{ct}_1$ are also set.

- **step3** is the implementation of step3 from the *ECOT* protocol. The *Ciphertext*, corresponding to the choice $ct_c$, is retrieved and used along with $pk_c$ and the $sk$ to compute a value $p_c$. Next $pk_c$ and $p_c$ is used to compute $p'_c$, which in turn is used to compute $p''_c$. With all these computations done, $chr$ can be computed with the $c$, $ch$ and $p''_c$. Lastly $chr$ is transmitted by using **RGClientCommunicator**. This step is almost identical in both *ECOT* versions. The only variation is that instead of making use of Bob's $c$ it uses $c'$

- **receiveStep4** sets the fields $\widetilde{m}_0$, $\widetilde{m}_1$, $p'_0$ and $p'_1$. $p'_0$ and $p'_1$ is replaced by $p_0$ and $p_1$ in the version without selective failures

- **step5** is the implementation of step5 from the *ECOT* protocol. This step depends a lot on the version being used. Therefore both will be explained in full.

  - **step5** with Selective Failures. First it is verified that Alice's $p'_c$ and Bob's $p'_c$ are equal, by using $c$, $p'_c$, $p'_0$ and $p'_1$. Should this check fail the program will abort. Following this $p''_{1-c}$ is computed by using $c$, $p'_0$ and $p'_1$, and now CH can be verified by using $ch$, $p''_{1-c}$ and $p''_c$. As before, if the check fails the program will abort. Next $\widetilde{p}_c$ is computed by using $pk_c$ and $p_c$ and the message $m_c$ can now be obtained by using the $c$, $\widetilde{p}_c$, $\widetilde{m}_0$ and $\widetilde{m}_1$. The value of $m_c$ can then be returned.
  - **step5** without Selective Failures. $pk_0$ is computed by using $c'$, $pk_c$ and $pk_{1-c}$ and $ct_0$ is then verified using $ct_0$, $p_0$ and $pk_0$. This process is then repeated to verify $ct_1$, by taking the same steps just with variables of 1 instead of 0. Next $pk_0$ and $p_0$, and $p_1$ and $pk_1$ are used to compute $p'_0$ and $p'_1$ respectively. $p'_0$ and $p'_1$ are then used to compute $p''_0$ and $p''_1$. After these computations $ch$, $p''_0$ and $p''_1$ are used to verify $ch$. When the last verification is done, $\hat{ct}_c$ can be retrieved by using $c'$, $\hat{ct}_0$ or $\hat{ct}_1$. Next, $\widetilde{p}_c$ is computed using $pk_c$, $sk$ and $\hat{ct}_c$, and then using $c'$, $\widetilde{m}_0$ and $\widetilde{m}_1$ to obtain $\hat{m}_c$. Lastly, $d$ is calculated using $c$ and $c'$ and then transmitted with **RGClientCommunicator**. Should any verifications fail, the program will abort.

- **receiveStep6** is only applicable for the version without selective failures. It sets the fields $m'_0$ and $m'_1$.

- **step7** is only applicable for the version without selective failures and is the implementation of step7 from the *ECOT* protocol. The message $m_c$ is obtained by using the $c$, $\widetilde{m_c}$, $m'_0$ and $m'_1$ and then returned.

**RGClientLogic**

Both versions of *ECOT* use the same **RGClientLogic**, since they have many methods in common. **RGClient-Logic** has a dependency on the **CryptoSystem**, which is set in the setup method. Below is a short description of every method, except setup.

- **getKeyGenerator** gets a key generator from the **CryptoSystem**.

- **sampleRandomBinaryString** creates a binary string of the security parameter length given in **Constants** and randomises the content using Java's secure random. The resulting binary string is then returned.

- **computeQ** takes a binary string, forwards it to the hkey method in **CryptoSystem** and returns the *ECPoint* received.

- **computePK1MinusC** takes two *ECPoints* and subtracts the second from the first. The result is then returned.

- **getCTC** takes an int and two *Ciphertexts*. If the int is zero, the first *Ciphertext* is returned. Otherwise the second is returned.

- **computePC** takes an *ECPoint*, *BigInteger* and *Ciphertext*, and forwards them to the decrypt method in **CryptoSystem**. Finally it returns decrypted *ECPoint*.

- **computePCMark** takes two *ECPoints*, forwards them to the hch method in **CryptoSystem** and returns the resulting binary string.
- **computePCMark2** takes a binary string, forwards it to the hch method in **CryptoSystem** and returns the resulting binary string.
- **computeCHR** takes an int and two binary strings. If the int is 0, the first binary string is returned. Otherwise an XOR operation is done on the two binary strings and the result is returned.
- **verifyPCMark** takes an int and three binary strings. It checks whether the first binary string is equal to one of the other two binary strings. It checks against the second binary string if the int is 0 otherwise it checks against the third. A boolean value of true is returned if they are equal. Otherwise false is returned.
- **abort** takes a string and throws a run time exception with the string as the error message.
- **computeP1MinusCMark2** takes an int and two binary strings. If the int is 0, the first binary string is returned. Otherwise the second binary string is returned.
- **verifyCH** takes three binary strings. It verifies that the XOR result of the second and third binary string equals the first binary string. It returns a boolean value of true if they are equal and false if they are not.
- **computePCWave** takes two *ECPoints* and forwards them to the hpad method in **CryptoSystem**. The resulting binary string from hpad is then returned.
- **obtainMessageC** takes an int and three binary strings. If the int is 0, the XOR result of binary string one and two is returned. If the int is not 0, the XOR result of binary string one and three is returned.
- **sampleRandomBit** uses Java's secure random to return a random bit value, as an int.
- **computePk0** takes an int and two binary strings. If the int is 0, the first binary string is returned. Otherwise the second binary string is returned.
- **computePk1** takes an int and two binary strings. If the int is 1, the first binary string is returned. Otherwise the second binary string is returned.
- **verifyCT** takes a *Ciphertext* and two *ECPoints*. A new *Ciphertext* is computed by calling encrypt in **CryptoSystem** with the two *ECPoints*. It is then verified that the given *Ciphertext* and the newly computed *Ciphertext* are equal. If they are equal, a boolean value of true is returned and if not the false is returned.
- **calculateD** takes two ints and throws a runtime exception if they are not 1 or 0. The XOR of the input is then returned.

**RGClientCommunicator**

Two implementations have been made of **RGClientCommunicator**, one for each version of *ECOT*. Both have a number of methods for transmitting steps equal to the amount of steps Bob makes in the protocol and a number of methods for receiving equal to how many step results Alice sends to Bob. Each transmit and receive functions the same way, the only difference is the data transmitted. For details on what data is being transmitted and received see the corresponding steps and receive method in **RGClient**. The setup is used to set the connection to the **RGServer** and details of the other two types of methods found in **RGClientCommunicator** can be seen below.

- **transmitStep** takes data and uses it to create a step *DTO* corresponding to the name of the method (eg. transmitStep1() creates a step1 *DTO*). This *DTO* is then transmitted to the **RGServer**. If any errors occur, a run time exception is thrown. An exception to the general rule is in transmitStep1(), where a choice is given, which is used to ensure $pk_0$ is transmitted.
- **receiveStep** listens for a response from the **RGServer**. Upon receiving it, the response is cast to a corresponding step *DTO* (eg. receiveStep2() returns a step2 *DTO*) and returned. If any errors occur, a run time exception is thrown.

### 5.3.3 Server

Alice's actions described in the *ECOT* protocol is handled in the server subsystem. For an overview of the where different parts of *ECOT* are implemented, see Fig. 5.3.4. As we did with the client in subsection 5.3.2, we will describe the implementation of each class. For each class we will first describe how it is done for the version with selective failures and then how the version without selective failures differentiate.

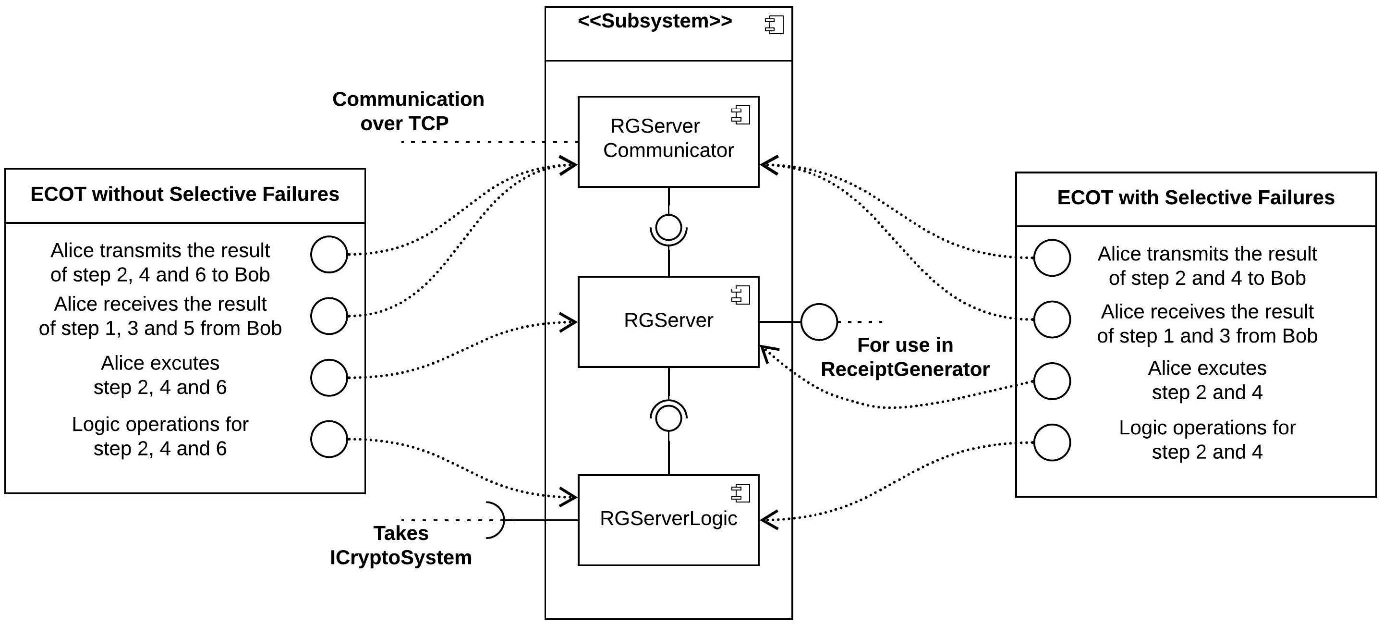

<<Subsystem>>
Communication over TCP
RGServer Communicator
ECOT without Selective Failures
Alice transmits the result of step 2, 4 and 6 to Bob
Alice receives the result of step 1, 3 and 5 from Bob
Alice excutes step 2, 4 and 6
Logic operations for step 2, 4 and 6
RGServer
For use in ReceiptGenerator
ECOT with Selective Failures
Alice transmits the result of step 2 and 4 to Bob
Alice receives the result of step 1 and 3 from Bob
Alice excutes step 2 and 4
Logic operations for step 2 and 4
RGServerLogic
Takes ICryptoSystem

**Figure 5.3.4:** *OT usage in the* ***Server***

#### RGServer

The setup method for **RGServer** takes the two receipt messages $m_0$ and $m_1$ as an array of *ECPoints* and sets the dependency for **RGServerLogic** and **RGServerCommunicator**. If there are less or more than two *ECPoints* in the array, a runtime error is thrown. Everything computed, generated or verified in the step methods uses **RGServerLogic** to do the actual computation, generation or verification. Methods are described in depth below and if nothing else is written the first time a value is mentioned, the reader can assume that it is a binary string.

- **generateReceipt** is a method for going through each step of the protocol in succession. The **RGServer-Communicator** is called to get the result of the steps done in the **RGClient**, eg. receiveStep1 then receiveStep3 and so on. The information from each step is set to fields via the receiveSteps methods. In between each received step, internal step methods are called in ascending order, eg. step2 then step4 and so on. This continues until the protocol is done. The only difference for the version without selective failures is that there are more steps to go through.

- **receiveStep1** sets the fields $s$ and the *ECPoint* $pk_0$.

- **step2** is the implementation of step2 from the *ECOT* protocol. The step starts by using $s$ to compute the *ECPoint* $q$, which is then used with $pk_0$ to calculate the *ECPoint* $pk_1$. Following this, $p_0$ and $p_1$ are sampled and used to create two *ECPoints* $p_0point$ and $p_1point$. These do not appear in the *ECOT* protocol and are an addition we have added as a result of how our implementation of encryption works. Since encryption needs the $p$ value to be an *ECPoint*, we chose to convert all $p's$ to *ECPoints*. After this $p_0$ and $p_1$ is used with $pk_0$ and $pk_1$ respectively to compute $p'_0$ and $p'_1$. The values $p'_0$ and $p'_1$ are then used to compute $p''_0$ and $p''_1$, which are used to compute $ch$. Two *Ciphertexts* $ct_0$ and $ct_1$ are then computed from two values, $pk_0$ and $p_0$, and $pk_1$ and $p_1$ respectively. The **RGServerCommunicator** is used to transmit $ch$, $ct_0$ and $ct_1$.

  In the version without selective failures the following is done as well as all the above mentioned: Two more p's are sampled, $\hat{p}_0$ and $\hat{p}_1$, which are then used to create two *ECPoints* for the same reasons as mentioned above. $\hat{p}_0$ is used to compute $\hat{p}_0point$ and $\hat{p}_1$ is used to compute $\hat{p}_1point$. $pk_0$ and $\hat{p}_0point$ will then be used to compute $\hat{ct}_0$ and $pk_1$ together with $\hat{p}_1point$ will then be used to compute $\hat{ct}_1$. At the end of the step, $\hat{ct}_0$ and $\hat{ct}_1$ will be added to the things already being transmitted.

- **receiveStep3** sets the field $chr$.

- **step4** is the implementation of step4 from the *ECOT* protocol. Firstly, $chr$ is verified by using both $chr$ and $p''_0$. Should the verification fail, the program will abort. If the verification is a success the values $\widetilde{p}_0$ and $\widetilde{p}$ will be computed from $pk_0$ and $p_0point$, and $pk_1$ and $p_1point$ respectively. Next $\widetilde{m}_0$ is computed from $\widetilde{p}_0$ and $m_0$, followed by $\widetilde{m}_0$ computed from $\widetilde{p}_1$ and $m_1$. Finally $\widetilde{m}_0$, $\widetilde{p}_1$, $p'_0$ and $p'_1$ are transmitted with the **RGServerCommunicator**.

  Two extra values are computed in the version without selective failures. Firstly, $\hat{m}_0$ is computed as a binary string of the same length as $m_0$'s binary string length and then $\hat{m}_1$ is computed as a binary string of the same length as $m_1$'s binary string length. Finally different values are transmitted with the **RGServerCommunicator**. The ones transmitted are $\widetilde{m}_0$, $\widetilde{p}_1$, $p_0point$ and $p_1point$

- **receiveStep5** is only applicable for the version without selective failures. It sets the field for the int $d$.

- **step6** is only applicable for the version without selective failures and is the implementation of step6 from the *ECOT* protocol. First the value $\hat{m}_d$ is computed from $d$, $\hat{m}_0$ and $\hat{m}_1$, followed by the computation of $\hat{m}_{1-d}$, using the same values. By using $\hat{m}_d$ and $m_0$, $m'_0$ can be computed. Similarly, the value $m'_1$ can be computed from $\hat{m}_d - 1$ and $m_1$. Finally, $m'_0$ and $m'_1$ will be transmitted using **RGServerCommunicator**.

**RGServerLogic**

As with the **RGClientLogic**, both version of *ECOT* use the same **RGServerLogic**. In the setup method a dependency for the **CryptoSystem** is set. Below is a short description of every method, except setup.

- **computeQ** takes a binary string, forwards it to the hkey method in **CryptoSystem** and returns the *ECPoint* received.

- **computePK1** takes two *ECPoints* and subtracts the second from the first. The result is then returned.

- **sampleP** creates a binary string of the security parameters length given in **Constants** and randomises the content using Java's secure random. The resulting binary string is then returned.

- **pToECPoint** takes a binary string, forwards it to the hkey method in **CryptoSystem** and returns the *ECPoint* received.

- **computePCMark** takes two *ECPoints*, forwards them to the hch method in **CryptoSystem** and returns the resulting binary string.

- **computePCMark2** takes a binary string, forwards it to the hch method in **CryptoSystem** and returns the resulting binary string.
- **computeCH** takes two binary strings. An XOR operation is done on the two binary strings and the result is returned.
- **computeCipherText** takes two *ECPoints*, forwards it to the encrypt method in **CryptoSystem** and returns the *Ciphertext* received.
- **verifyCHR** takes two binary strings and verifies if they are equal. If they are equal a boolean value of true is returned. Otherwise the boolean value of false will be returned.
- **abort** takes a string and throws a runtime exception with the string as the error message.
- **computePWave** takes two *ECPoints*, forwards them to the hch method in **CryptoSystem** and returns the resulting binary string.
- **computeMWave** takes a binary string and an *ECPoint*. An XOR operation is then done on the binary string and the binary representation of the *ECPoint*. The result of this is returned.
- **sampleRandomBinaryString** takes an int and creates a binary string of the int value's length. The content of the binary string is randomised using Java's secure random. The resulting binary string is then returned.
- **computeMWave** takes two binary strings, performs an XOR operation on the two binary strings and returns the result.
- **computeMdH** takes an int and two binary strings. If the int is 0, the first binary string is returned. Otherwise the second binary string is returned.
- **computeM1dH** takes an int and two binary strings. If the int is 1, the first binary string is returned. Otherwise the second binary string is returned.
- **computeMMark** takes a binary string and an *ECPoint*. An XOR operation is then done on the binary string and the binary representation of the *ECPoint*. The result of this is returned.

**RGServerCommunicator**

The **RGServerCommunicator** is completely identical in the way the methods work as in the **RGClient-Communicator**. The only difference, is that communication is from Alice to Bob, not Bob to Alice.

## 5.4 Voting simulator

Where the combined system is based on the Norwegian example, the underlying specifics of the voting simulator is based on our interpretation of how Danish elections are held, which is shown as we describe the **Voter** in subsection 5.4.2. The **BallotBox** will be described in subsection 5.4.1 and the **DecryptionService** will be described in subsection 5.4.3. A full overview of the system as a whole is shown in Fig. 5.4.1.

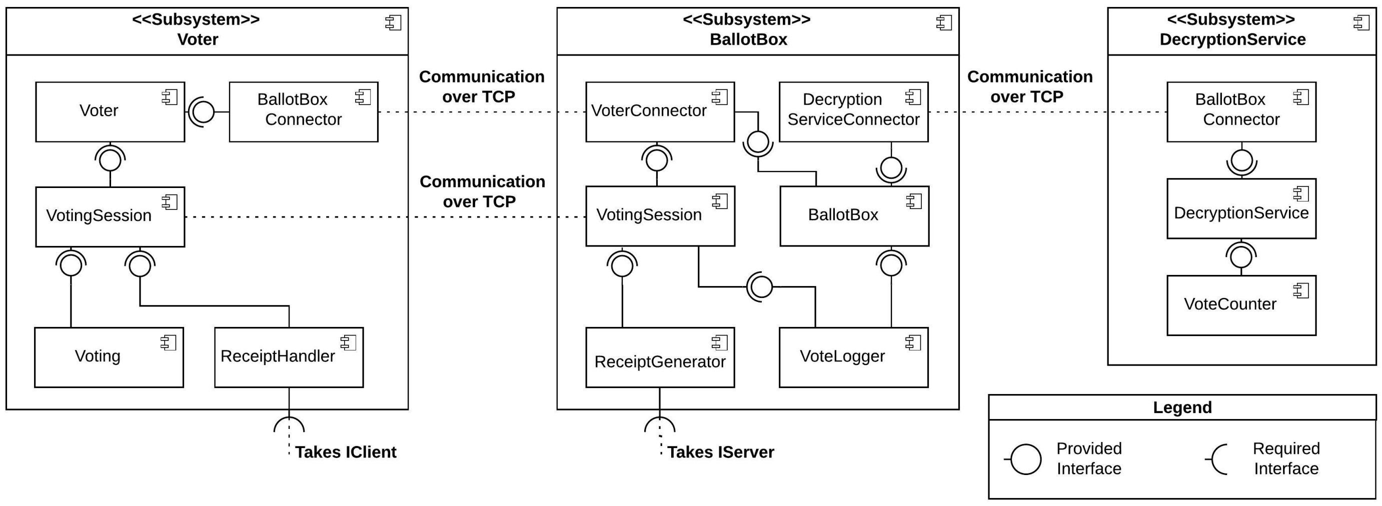


**Figure 5.4.1:** *Breakdown of the receipt voting simulator system, showing class relations.*

### 5.4.1 Ballot box

The **BallotBox** is set up to allow several connections for different voters, who connect through the **VoterConnector**. In a real system an authentication process would then take place and a queue system would be in place to handle a large volume of requests, but for simplicity we assume that all voters are authenticated and do not run tests with more users than the system can handle. When this step is complete a **VotingSession** is started for each voter connected.

During the voting session, the voter will be given a ballot with two choices and a unique *ECPoint* for each choice. It will then receive the vote from the voter and start the **ReceiptGenerator**. If the receipt is accepted the vote will be stored in the **VoteLogger** and the voter will be informed that it is stored.

The **ReceiptGenerator** sets up and runs the server side of the receipt generator and returns the result. Either the receipt is accepted or not.

When the election is over, the **BallotBox** closes all open connections and stops accepting new ones. In a real system remaining users should properly be allowed to finish their vote within a time frame. The system then gets all the votes from the **VoteLogger** as an ArrayList and transmits them using the **DecryptionServiceConnector** to the **DecryptionService**. This process should include mixing to ensure that voters cannot be traced back to their own vote.

Lastly, the **VoteLogger** transmits what each voter has voted to the **TestAuditor**, which of course must never be done in an actual system.

### 5.4.2 Voter

In real life, a voter goes to a polling station to gain access to a ballot box. In our system the **Voter** does this by establishing a connection through the **BallotBoxConnector**. Upon establishing connection and getting authenticated, the **Voter** starts a **VotingSession**. This can be seen as a voter being allowed into a voting booth.

Thereafter the **VotingSession** will request a ballot for the user. The act of voting is simulated by **Voting**, which in our system returns a random bit. A vote is then sent to a ballot box's **VotingSession** and the **ReceiptHandler** is then used to ensure the correct vote is tallied. This corresponds to when a voter in a voting booth has set their mark on the ballot and is now double-checking that the mark is in the correct place.

Upon receiving an accept from the **ReceiptHandler**, the **VotingSession** transmits an acceptance to the ballot box's **VotingSession** and if it's a confirmation that the vote is being stored on the server, the voting is over. In a real life example this is equivalent to the voter having checked the ballot, folded it and put it into the a ballot box. Should any of these steps fail, or should the receipt not be accepted, the process is terminated.

The final step is only done for testing purposes. When the **VotingSession** is over, it returns the vote to the **Voter**, which then transfers it to the **TestAuditor** along with the user id.

### 5.4.3 Descryption service

The **DecryptionService** starts a listening server **BallotBoxConnector**, which accepts incoming connections from a **BallotBox** and stores the vote result. In our simplification we only use one **BallotBox**, so there is no checks to ensure all **BallotBoxes** have delivered their results.

When all the results have been delivered, the **DecryptionService** will use **VoteCounter** to tally the votes and print the results.

## 5.5 Test strategy

As already mentioned we did not test the simulation system. We chose not to do this, because if we were to properly test the simulation system it would take us a disproportionate amount of time, since our focus is on the *OT* and the simulation system is only used to show it could work. Our test strategy is therefore only applicable for the receipt generator. Below we will discuss unit testing in subsection 5.5.1, integration testing in 5.5.2 and system testing in 5.5.3.

### 5.5.1 Unit testing

The purpose of our unit testing is to make sure every individual part of the program works as intended. The idea being that if every part works as intended in isolation and we add all parts correctly together, then we can have a high degree of confidence that our program actually works.

We intended to do this by ensuring that each class is only dependent upon interfaces, this would make testing a lot easier since interfaces can easily be mocked. At the same time by ensuring that every class and method has only one responsibility, we get code that is easily testable. Since every method only handles one logic computation or calls sub-parts of the system, we can test every piece of logic separately and verify that correct methods are called to ensure the flow of data is correct. We do this by making use of boundary testing and path testing [5].

As mentioned in section 5.3 we did not consider how the system should be connected to the simulation system and the design constantly changed as we learned new things. This means that the way we have done our unit testing sometimes differs a little from how we imagined doing unit testing. This is especially noticeable in the **RGClients** and **RGServers**, where proper unit testing could not be done for the method

**generateReceipt()** since it was dependent on methods in the same class, which cannot be mocked, at least without dubious practices. Also the communicators were completely redone towards the end, which means we did not have time to redo the tests for them.

### 5.5.2 Integration testing

Integration tests would have been a very good thing in addidtion to our unit tests to ensure everything worked properly. This would be to insure that all parts are added correctly. Integration tests were however scrapped as a result of the unit testing demanding a lot more time than we had anticipated.

If we had more time we would have implemented tests similar to our unit test, with one of the dependent interfaces using the actual implementation instead of a mocks. For example if we were testing the **CryptoSystem** we would have one group of tests to test the integration with **CryptoSystemLogic** where only **Hash** is mocked. A second group of tests would then be made to test the integration with **Hash**. In this example **CryptoSystemLogic** and **HashLogic** in **Hash** would be mocked.

### 5.5.3 System testing

We decided that our system testing would be done indirectly by making use of the **FullSystemMain**. Our reasoning for this is mainly time. It would have been nice to have automated tests that could fail or succeed, but for our purpose we found it to be sufficient that we can see that the program runs hundreds of times and prints results. By interpreting the results we can make a confident claim that the system works as intended, while showcasing it.

## 5.6 Running the program

We suggest compiling the program through the reader's preferred IDEA, we recommend something like *IntelliJ*, and then run **FullSystemMain**. Dependencies for the program are supplied with the source code in the folder *libs*. Depending on the settings setup in the top fields of **FullSystemMain**, it will either compare the two version of *ECOT* or run an election simulation. The compare works by running each version with n **Voters**, connecting to the **BallotBox** and voting sequentially, and then simply timing them. The standard setting are to compare the two versions of *ECOT* with 1000 voters.

Running the election simulation will start a **BallotBox** and start n sessions of between x and y **Voters**. The sessions start with five second intervals. Each **Voter** in a session will start on a separate thread and attempt to vote. A timeout of between 0,5 and 1,5 seconds is set between starting each thread. There is a random 50/50 chance that a session will be set as "Rush Hour", which means that there will be no timeout on starting the **Voters**. The simulated election ends after some predefined set time, new Voters can still be started after this, if the election time is set too low, but they will not be able to vote. When the election is closed, the election results will be printed.

The parameters that can be changed in the **FullSystemMain** file and their standard setting can be seen below.

- minVotersPerSession = 15; The minimum amount of voters per session in a simulated election.
- maxVotersPerSession = 50; The maximum amount of voters per session in a simulated election.
- numberOfVoters = 5000; Number of potential unique voters in the election, used for creating user ids. The fewer unique voters, the higher the chances of a voter voting multiple times.
- votingSessions = 20; The number of sessions to run when simulating the election.
- electionLengthSeconds = 120; How many seconds to wait before closing the election.
- voteAttempts; Should not be edited! Used for seeing how many voters that have been started.

- runElectionSimulation = false; Set to true if the election simulator should be run. Set to false if the compare should be run.
- useECOTWithSelectiveFailures = false; Set to true to use *ECOT* with selective failures for the election simulator, false if the compare should be run. Set to false to use without selective failures.
- compareVoters = 1000; Number of voters to use when comparing *ECOT* versions.
- sessionTimeoutInSeconds = 5; Number of seconds to wait in between sessions.

# 6. Discussion

It is tempting to dive right into discussing how well or bad *OT* worked for generating receipts for voters, but several things need to be examined before we do this. We will start by examining if it is a good idea to generate the verification using only the computer used to vote. We will do this by discussing the differences between having a physical card with verification codes and just having the verification written on a computer screen, this will be done in section 6.1. Another part of this discussion is about receiving the verification on the same device as used for voting or on another device, which will be discussed in 6.2. We will reflect upon the problems of authentication in section 6.3 and then the problems with communication in section 6.4. We also need to consider network security if we were to finish a *DRV* system and we will discuss this in section 6.5. Next we will examine performance and bottlenecks in our implementation in section 6.6. Lastly, we will digress a bit and discuss the problem of building trust in section 6.7.

## 6.1 Physical card verification vs. computer only verification

In both the Norwegian and Swiss *DRV* systems, the voters receive a physical card that they use as part of the verification process. We find the main disadvantage of this to be the need of a central system for generating the receipts beforehand and the added complexity of securely transmitting these to the voter and the voting system when voting occurs. This would increase the cost of the whole process and slightly inconvenience the voter by making them check the received numbers against some physical card. But it is in no way a deal breaker and it might actually have some extra benefits, as compared to how we addressed this, where the system just informs the voter on screen, "you voted this".

The most obvious problem, to us at least, with computer only verification is that the computer tells you what you voted. The first issue with this is that if the computer is monitored in some way, then the reply could be seen by a malicious third party, compromising secrecy of the election. One could easily imagine people offering to buy votes and having voters install third party software to actually monitor the vote response, making it a lot easier and more safe to buy votes. Potential buyers can be more confident that they figuratively speaking "get what they pay for" and they could potentially reach many more customers. The same goes for coercers. The second issue, which relates to the previous ones, is what if someone, like foreign powers, political parties, companies or others, could infect several computers and know what specific people voted? This could be used to target specific people based on what they voted, ranging from harassment, exclusion, public ridicule, influencing job opportunities, targeting advertisement or propaganda and so on. Another way to misuse this information would just be to make it public and use it to fuel mistrust in the system in general and potentially undermine the entire process.

A second problem is related to the last issue mentioned above, trust. How can a user truly believe that the system does not know what was voted and believe that secrecy is intact, when the system actively writes it on the screen. If you have enough technical expertise, you could read through the source code and see for yourself that this is true, but most voters would not have that technical expertise. Some might argue that the experts could assure the public that the system is fine, but people do not always trust experts: One can just google flat earth and anti-vaxxers to get a good idea of how experts are not trusted. This could again fuel mistrust in the system. If a malicious third party as mentioned above gets information about specific people's votes and leaks them, this problem could become even worse. How would you convince anyone to trust this system again, or maybe even trust the election result?

If trust takes a hard enough hit, some people might even begin to mistrust the underlying democratic process. And this is the true problem that we must avoid. If the trust in the *DRV* system is broken, then we can just replace it or drop it for now. But if trust in the democratic process is broken or just damaged, then it could have far reaching consequences for a democratic society.

We also see these problems with the physical cards, but they are not as pronounced. In the Swiss and Norwegian examples third parties can only get a code and would need either access to the system that generated the cards or the cards themselves. This makes it a lot harder for third parties not in contact with the voter to use the information, since they only know they voted, not what they voted. It also means that when buying or coercing you now have to get a photo of the verification card and compare this to the code, making the process require more effort.

As for the problems with trusting in the secrecy, we would argue that it is easier for people to believe their vote is secret when they receive a code they have to verify. In fact this might actually give them a feeling of being part of the security and therefore make the system more trustworthy, this is however just speculation and research should be done to see if this claim can be substantiated.

This discussion has so far been focused on the receipt, but the problem with a computer being monitored, is that the whole process can be monitored, not just the receipt. This means that all the points made in the beginning can be made for when a voter is selecting what to vote for. So the biggest difference so far ends up being whether the voter can trust that the ballot is actually secret.

A bigger problem might be if we consider the risk of *Man-in-the-Middle Attacks*, where some malicious party attempts to vote on the voter's behalf. If there is no verification card, then the user will have no way of knowing something is wrong, but with a verification card they will be able to see something is wrong using the codes.

## 6.2 Verification on the used device vs. another device

Another difference is the one between receiving the verification on the same device that was used for voting and on another device like they did in Norway. We find that both solutions have advantages and disadvantages.

When having another part of the verification process happen on a different device, we increase the likelihood of the voter actually being who they claim to be. But as the voter is already using some kind of personal identification like *MinID* and a personal voter card with the verification codes, this seems like a small improvement. But it makes identity theft for voting purposes harder, since the voter will know if someone else voted on their behalf.

At first glance it might also seem like an advantage that malicious parties would need to monitor two devices. However, as we saw above malicious parties only need to monitor a computer to see what people actually vote. If they can monitor things like mobile devices, it might be easier to link specific people to specific votes. When using another device, it can also be used to assist coercers or buyers, since they will know by monitoring the mobile device whether the voter votes again. This means that a voter will have to go to a physical voting station to vote if they want to keep their vote as their own, which could be monitored by a potential coercer or buyer. All this would not be an issue if the voter only use one device, since they could "just" go to an unmonitored computer and vote from there.

## 6.3 Authentication

In a *DRV* system authentication will always be of great concern. When voting conventionally, you authenticate yourself by turning up to the election and showing your id. However, when using a computer for voting, who is there to authenticate you? And how can the system make sure that you are actually the person who you claim to be? In a system like *CHVote* from chapter 4, each voter is given a unique secret code on their voting card, which is used for authentication. However, this solution is not fully secure since the voting cards are both calculated, printed and transported by entities other then the voter, and it would be possible for any of these parties to gain access to the codes from the voting cards, and use them to authenticate as other voters. Such a system requires trust in the entities or at least a way to see if any of the entities have become corrupted and are exploiting their power. This results in a system that should be able to effective cancel elections if there is any sign of corrupted entities, since corruption could have dire consequences for the final result.

If a system like *CHVote* were to be used in Danish elections, we would also have to add authentication to the system, which should be doable for every Danish voter. Using the secret code as described above is one solution, however there might also be other possible solutions. Many elements of Danish online security are already linked to the *NemID* system a *Multi-Factor authentication* tool available for all Danes, which they have used for several years. This could add trust to a system, instead of having to build up trust in a whole new medium of authentication. However, having dependencies like this as a part of your system, can limit your knowledge of problems and flaws that is happening in your system. There might be backdoors and holes in a system like *NemID*, specially when they are not following the open design principle. These flaws would be transferred into your system, or they might come later, when *NemID* is updated. Dependencies like this can grant relieve and help the project come along, but can also bring in problems both now and later, so they should be used with care and they should be changed, if problems arise.

## 6.4 Problems with communication

Transport of messages from the system to users would also need to be thought through. As outlined in the discussion about computer verification, users might not trust messages coming from their own computer and for good reason. This is what is called the *Insecure platform problem* where it is infeasible to make sure that all computers, phones etc. that are used for voting are secure. This is why validation in *CHVote* is done using voting cards or as in the Norwegian protocol, where phones are used during validation. Using a phone or voting card brings *Multi-Factor authentication* to the system, and if or when a voting system is adopted in Denmark, there should be a similar system to transmit messages between system and voter, that can avoid the *Insecure platform problem*.

## 6.5 Network security

The web is not built to be secure, therefore we need to use cryptography and network security when exchanging secrets using web protocols. For *OT* there is already built-in cryptography as described in chapter 3, but we could still add something like the *HTTPS* protocol. This would provide an extra layer of security to the system and make it even harder for adversaries to hack into the system. Using such a trusted protocol could also have a positive effect on the trust in the system. To setup *HTTPS* you need a trusted *certificate authority* to sign the certificates used to distribute the *Public Key*, this again requires trust in 3rd parties that might or might not be trustworthy. However, the gain could offset the problems, as there are already many trusted certificate authorities and no messages in the protocol would ever have to be transmitted in plaintext.

## 6.6 Performance and bottlenecks

Running the protocol with 1000 sequential voters returns the following average running times:

With selective failures: 29184 ms
Without selective failures: 50858 ms

The difference between with and without selective failures is therefore 57%, which is not surprising since the without selective failures protocol contains more computations. In a real world application, you would also have to work with network latency. This would however be dependant on the distance between the client and the server. Estimating a latency of between 20-30 ms would result in the following transmitting time for a user:

With selective failures: 50-75 ms
Without selective failures: 70-105 ms

Since the latency of using the network is between 2-4 times greater than the computation times, it would seem that even a reduction in computation times would not have a great effect on the protocol as a whole.

The results from scaling the system to a greater number of sequential voter′ shows that the protocol scales in linear time with the following results from 10000 sequential voters:

With selective failures: 282352 ms
Without selective failures: 505862 ms

Looking into the computations, many of them are computed using the *BC Library*, limiting the work we can do to optimise the performance of the functions. An example of this is the use of ECC in the implementation. Since we have not made the implementation, we can't control when doing point multiplication and other functions, and just have to trust that *BC* has implemented secure and fast functions.

### Point multiplication

Point multiplication is quite a big part of our implementation, and running several point multiplications in sequence shows an average computation time of 1.5-2.0 ms pr. multiplication. The protocol with selective failures contains 14 multiplications. Compared to the total computation time of about 29 ms for a single user, about *85%* of the total computation time is used on point multiplication. The protocol without selective failures contains 25 multiplications. Compared to the total computations time of about 51 ms for a single user, about *92.5%* of the total computation time is used on point multiplication.

This shows that the point multiplication is a bottleneck of this protocol. Since we use the *BC Library* for the implementation of the point multiplication, we are fully dependant on the computation time of their implementation. This could be removed by using another form of encryption, that does not make use of point multiplication, though this might just bring in another bottleneck with another type of computation, used for that encryption technique.

### Subtraction and addition

Other functions on Elliptic points that we use are subtraction and addition. However, testing the average computation time for these are about 0.01 ms each, so they are negligible compared to point multiplication.

### XOR

The XOR functionality we have built using the java *Library*, our implementation runs in linear time proportional to the length of the inputs. However, since the length of the byte arrays are at maximum 33 bytes long, this have very little influence on the computation time.

#### Hashing

We have the same problem with hashing as we have with the scalar multiplication, where we make use of *BC* to preform the computations and can't change much in the way of implementations. We can test the running time for the hashing algorithms and this results in a running time between 0.001 ms and 0.03 ms. This probably does not have much of an impact on the protocol's performance.

## 6.7 The problem with building trust

Trust is a very important factor. Even with a fully secure system it would not matter if the voters do not trust the system. We can add several layers of security, have experts review the system, use open source etc, but this will not necessarily convince the voters or be enough to counter potential misinformation. So how do we get around this? Our best tool to build trust could be the same we use for the physical voting system: the test of time. One of the reasons we can be so confident in the security and anonymity of elections, using conventional voting, is our long experience of using it. Time has shown us that it works and keeps working. But relying on this tool is a long process, especially with elections being years apart. To get around this, a voting system could be built to support *DRV* in several countries. If for example the EU built a voting system that could be used by all member countries, the voting system would be used frequently and therefore be tested in the real world more often. This could help to increase our confidence faster and help us to find weaknesses in the system. Another step that could be taken is to make the system open for other types of *DRVs* and not just elections, like it has been done with *CHVote*. The more we use the system, the more secure it will become as we identify and fix weaknesses and the more confident we can be that the system works, thereby increasing our trust in the system overall.

# 7. Conclusion

Over the last couple of years the idea of having *DRV* has been on the rise. It has been seen as a solution to many of the problems with the current systems and would only be natural in a society that becomes more and more digitised. There have been many attempts at implementing *DRV* other then the two attempts in Norway and Switzerland described in this paper, though we have yet to see any of them becoming a real contender to replace conventional voting. One step in this direction could be some kind of receipt generation system that voters can trust.

Generating a receipt for a voter that provides the voter with evidence that their vote has been received correctly, while remaining anonymous, can be done using *OT*. Based on our prototype and how it is used in *CHVote*, it seems a feasible solution for a real world system. If we assume that it is possible to create a k-out-of-n protocol that has the same asymptotic running time of $O(kn)$, as the k-out-of-n *OT* protocol, which *CHVote*'s *OT* protocol is extended from. While also assuming that running a 1-out-of-2 problem with this imagined protocol results in the same real world running times as we found for our implementation.

We can now theorise on how long it would take to get a receipt. If we imagine that there are 200 candidates to choose from, we would then have a 1-out-of-200 problem where n has been raised from 2 to 200. This would result in an increase in computation by a factor of 100, resulting in around 5085 ms pr. user for generating a receipt, if we use our results for the version of *ECOT* with selective failures. Disregarding latency, a user would have to wait around 5 seconds for a receipt, using this logic. This would be okay in our opinion, but it is not optimal and handling more choices than this could become problematic, especially in systems where the voter can vote multiple times. The next step would be to create a k-out-of-n protocol and see if we can ensure security and reasonable running times. Until this is done we cannot be sure whether *OT* would be a good solution, at least from the technical perspective in a *DRV* system.

The biggest problem might however not be the technical aspects, but rather if a receipt actually improves the voter's confidence in the system's correctness and if the voter trusts that anonymity is intact. As shown in this paper, if we can not earn the voter's trust then everything else is wasted work. We argue that receipts would help, but there are other methods than using an *OT* protocol to solve this problem. So is *OT* actually the best choice or could it make the voters suspicious of the system and question the anonymity when they receive the verification response on screen. This needs to be researched and if we were to continue this project, we would suggest making a thorough study of how different types of receipts / verifications influence the voter's trust and confidence in the voting process.

*OT* is a solution for generating a receipt in a *DRV* system, while keeping the voter anonymous. Should we use it nonetheless? We believe more data is needed before we can answer this question, but based on our current results it is a contender.

# References

[1] Abdalla Al-Ameen and Samani Talab. "The Technical Feasibility and Security of E-Voting". In: *The International Arab Journal of Information Technology* 10.4 (2013).

[2] BigInteger. *Oracle*. URL: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/math/BigInteger.html (visited on 08/11/2020).

[3] Nadja Braun Binder et al. "International Standards and ICT Projects in Public Administration: Introducing Electronic Voting in Norway, Estonia and Switzerland Compared". In: *Halduskultuur* 19.2 (2019), pp. 8–22.

[4] Encyclopædia Britannica. *Direct democracy*. URL: https://www.britannica.com/topic/direct-democracy (visited on 06/17/2020).

[5] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns and Java*. Third Edition. Pearson, 2014. ISBN: 978-1-292-02401-1.

[6] B. McM. Caven. "E.S. Staveley, Greek and Roman Voting and Elections. (Aspects of Greek and Roman life). London: Thames amp; Hudson. 1972. Pp. 271. 9 text-figs." In: *Journal of Roman Studies* 63 (1973), pp. 263–263. DOI: 10.2307/299193.

[7] George Coulouris et al. *Distributed Systems Concepts and Design*. Fifth Edition. Pearson, 2012. ISBN: 978-0-273-76059-7.

[8] Malcolm Crook and Tom Crook. "The Advent of the Secret Ballot in Britain and France, 1789–1914: From Public Assembly to Private Compartment". In: *History* 92.308 (2007), pp. 449–471. DOI: 10.1111/j.1468-229X.2007.00403.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1468-229X.2007.00403.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1468-229X.2007.00403.x.

[9] Bernardo David and Rafael Dowsley. "Efficient Composable Oblivious Transfer from CDH in the Global Random Oracle Model. Personal Communication, 2020". In: (2020).

[10] Cambridge Dictionary. *Digital*. URL: https://dictionary.cambridge.org/dictionary/english/digital (visited on 07/25/2020).

[11] Cambridge Dictionary. *Electronic*. URL: https://dictionary.cambridge.org/dictionary/english/electronic (visited on 07/25/2020).

[12] Cambridge Dictionary. *Trust*. URL: https://dictionary.cambridge.org/dictionary/english/trust (visited on 05/07/2020).

[13] Folketinget.dk. *Folketingsvalg*. URL: https://www.ft.dk/da/folkestyret/valg-og-afstemninger (visited on 05/08/2020).

[14] State of Geneva. *CHVote 2.0 project release*. URL: https://chvote2.gitlab.io/ (visited on 08/17/2020).

[15] State of Geneva. *E-voting system - CHVote*. URL: https://republique-et-canton-de-geneve.github.io/chvote-1-0/index-en.html (visited on 06/17/2020).

[16] Jan Gerlach and Urs Gasser. "Three case studies from switzerland: E-voting". In: *Berkman Center Research Publication No* 3 (2009).

[17] J. Gibson et al. *A review of E-voting: the past, present and future*. Springer, 2016. URL: https://link.springer.com/article/10.1007/s12243-016-0525-8#Sec2 (visited on 07/22/2020).

[18] Kristian Gjøsteen. "Analysis of an internet voting protocol". In: (July 2010).

[19] Rolf Haenni et al. "CHVote System Specification". In: (2019).

[20] Carmit Hazay and Lindell Yehuda. *Efficient Secure Two-Party Protocols*. Springer, 2010. ISBN: 978-3-642-14302-1.

[21] Azine Houria, Bencheif Mohamed Abdelkader, and Guessoum Abderezzak. "A comparison between the secp256r1 and the koblitz secp256k1 bitcoin curves". In: (2018).

[22] Legion of the bouncy castle inc. *Bouncy castle webpage*. URL: https://www.bouncycastle.org/index.html (visited on 07/08/2020).

[23] Randi Markussen, Lorena Ronquillo, and Carsten Schürmann. "Trust in Internet Election Observing the Norwegian Decryption and Counting Ceremony". In: International Conference on Electronic Voting, EVOTE2014, E-Voting.CC GmbH, 2014.

[24] Robert C. Martin. *The Principles of OOD*. URL: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (visited on 08/10/2020).

[25] mockito.org. *Mockito*. URL: https://site.mockito.org/ (visited on 07/29/2020).

[26] N. Mpekoa and D. van Greunen. "E-voting experiences: A case of Namibia and Estonia". In: *2017 IST-Africa Week Conference (IST-Africa)*. 2017, pp. 1–8.

[27] nemid.nu. *NemID*. URL: https://www.nemid.nu/dk-da/ (visited on 08/13/2020).

[28] norge.no. *MinID*. URL: https://www.norge.no/en/service/minid (visited on 07/27/2020).

[29] Oracle. *Secure Random*. URL: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/security/SecureRandom.html (visited on 07/22/2020).

[30] René Peralta. *Electronic voting*. URL: https://www.britannica.com/topic/electronic-voting (visited on 07/08/2020).

[31] Regjeringen.no. *Internet voting pilot to be discontinued*. URL: https://www.regjeringen.no/en/aktuelt/Internet-voting-pilot-to-be-discontinued/id764300/ (visited on 05/08/2020).

[32] Regjeringen.no. *The main features of the Norwegian electoral system*. URL: https://www.regjeringen.no/en/topics/elections-and-democracy/den-norske-valgordningen/the-norwegian-electoral-system/id456636/ (visited on 05/08/2020).

[33] *Secure Shell*. URL: https://www.ssh.com/ssh/ (visited on 07/27/2020).

[34] Judith Simon. "Trust". In: *Pritchard, D. (Ed.): Oxford Bibliographies in Philosophy. New York:* (2013). URL: https://www.oxfordbibliographies.com/view/document/obo-9780195396577/obo-9780195396577-0157.xml (visited on 05/22/2020).

[35] William Stallings and Lawrie Brown. *Computer Security, Principles and Practice*. Fourth Edition. Pearson, 2018. ISBN: 978-1-292-22061-1.

[36] National Institute of Standards and Technology. *NIST*. URL: https://www.nist.gov (visited on 07/27/2020).

[37] Melanie Volkamer, Oliver Spycher, and Eric Dubuis. "Measures to Establish Trust in Internet Voting". In: (Sept. 2011).

[38] *What is the the random oracle model and why is it controversial*. URL: https://crypto.stackexchange.com/questions/879/what-is-the-random-oracle-model-and-why-is-it-controversial (visited on 07/19/2020).

[39] Wikipedia. *Data transfer object*. URL: https://en.wikipedia.org/wiki/Data_transfer_object (visited on 06/08/2020).

[40] Rima Wilkes and Cary Wu. "Ethnicity, Democracy, Trust: A Majority-Minority Approach". In: *Social Forces* 97 (1 2018), pp. 465–494.

[41] Arthur M. Wolfson. "The Ballot and Other Forms of Voting in the Italian Communes". In: *The American Historical Review* 5.1 (1899), pp. 1–21. ISSN: 00028762, 19375239. URL: http://www.jstor.org/stable/1832957.

# A. Glossary and Acronyms

## Glossary

**BigInteger** *Immutable arbitrary-precision integers. All operations behave as if BigIntegers were represented in two's-complement notation (like Java's primitive integer types). [...] The range must be at least 1 to 2500000000* - [2]
23–26

**Bridge Pattern** *"This pattern decouples the interface of a class from its implementation. It serves the same purpose as the Adapter pattern except that the developer is not constrained by an existing component."* - [5]
19

**Challenge** A compilation of equations that is used for verification from one party to another. A challenge is met with a challenge response, and if the response is correct, then verification is a success. An example of a challenge is to send a value, and then the recipient should create a new value that is expected by the sender.
10–12

**CHVote** *"CHVote : a public system, Swiss and open source. The electronic voting system CHVote is a concrete answer and a real breakthrough in digital technology and e-government."* - [15]
3, 15–18, 36, 37, 39, 40

**Ciphertext** *This is the scrambled output of a message produced as output. It depends on the plaintext and the secret key. For a given message, two different keys will produce two different ciphertexts.* - [35]
19, 23, 25–27, 29, 30, 45

**Computational Diffie-Hellman Assumption** Even for an adversary with all publicly exchanged values, group and group elements in the Diffie-Hellman key-exchange, it would still be infeasible to correctly find-/guess the secret key exchanged between the partners. - [35]
9

**Data Transfer Object** *"In the field of programming a data transfer object (DTO) is an object that carries data between processes. The motivation for its use is that communication between processes is usually done resorting to remote interfaces (e.g., web services), where each call is an expensive operation. Because the majority of the cost of each call is related to the round-trip time between the client and the server, one way of reducing the number of calls is to use an object (the DTO) that aggregates the data that would have been transferred by the several calls, but that is served by one call only."* - [39]
19, 28, 45

**Denial of service** *"The denial of service prevents or inhibits the normal use or management of communication facilities. This attack may have a specific target; for example, an entity may suppress all messages directed to a particular destination"* - [35]
7

**Diffie-Hellman** The first published public-key algorithm created by Diffie and Hellman from 1976 and is used as a key exchange technique. The algorithm enables two users to exchange a secret key, which can be used to encrypt subsequent messages. - [35]
13

**Direct Democracy** *"Direct democracy, also called pure democracy, forms of direct participation of citizens in democratic decision making, in contrast to indirect or representative democracy. Direct democracies may operate through an assembly of citizens or by means of referendum and initiatives in which citizens vote on issues instead of for candidates or parties."* - [4]
3, 4

**E-Voting** *"Because of security and access concerns, most large-scale electronic voting is currently held in designated precincts using special-purpose machines. This type of voting mechanism is referred to as e-voting. There are two major types of e-voting equipment: direct recording electronic (DRE) machines and optical scanning*

*machines."* - [30]
4, 5, 44

**Electronic Voting** *"Electronic voting, a form of computer-mediated voting in which voters make their selections with the aid of a computer. The voter usually chooses with the aid of a touch-screen display, although audio interfaces can be made available for voters with visual disabilities.*

*There are two quite different types of electronic voting technologies: those that use the Internet (I-Voting) and those that do not (E-Voting)."* - [30]
4

**Global Random Oracle Model** The Random Oracle model is a function that takes an input and returns a truly random output. Two different inputs can never return the same result, but the same input will always return the same output. - [38]
A Global Random Oracle instance is accessible by all parties. - [9]
9

**HTTPS** A secure protocol to communicate over networks that uses *TLS* to ensure the security.
37

**I-Voting** *"As use of the Internet spread rapidly in the 1990s and early 21st century, it seemed that the voting process would naturally migrate there. In this scenario, voters would cast their choices from any computer connected to the Internet—including from their home. This type of voting mechanism is sometimes referred to as I-voting."* - [30]
3–5, 44

**Insecure platform problem** A common problem in electronic voting, where computers can be infected by malicious users and therefor cannot be trusted. This is resolved by for example using voting cards or other forms of *Multi-Factor authentication* to make sure that the computer is not the only party involved.
15, 37

**IntelliJ** IntelliJ IDEA is an integrated development environment written in Java for developing computer software. It is developed by JetBrains, and is available as an Apache 2 Licensed community edition, and in a proprietary commercial edition.
33

**Key Pair** A set of keys containing one *Secret Key* and its corresponding *Public Key*.
10, 11

**Library** A collection of functions and methods in a program. Libraries can be either internal, where they have been written by the programmer or external where the programmer uses code that has been written by another programmer, group or institute. An example of external libraries could be Bouncy Castle, that is code already written and tested by others, and offer a vast collection of functions for a programmer.
9, 13, 14, 38

**Man-in-the-Middle Attack** *"[...] this attack involves persuading a user and an access point to believe that they are talking to each other, when in fact the communication is going through an intermediate attacking device. - [35]*
36

**MinID** *"MinID is an an electronic ID which provides access to public services at a medium-high level of security (level 3)."* - [28]
7, 36

**Multi-Factor authentication** MFA is used by a user in a system to authenticate themself with more means then one. This could be using both a password and a fingerprint.
37, 44

**NemID** *"NemID is a secure joint log-in service for both public and private selfservice solutions."* - [27]
37

**NIST** *"The National Institute of Standards and Technology (NIST) was founded in 1901 and is now part of the U.S. Department of Commerce. NIST is one of the nation's oldest physical science laboratories. Congress established the agency to remove a major challenge to U.S. industrial competitiveness at the time—a second-rate measurement infrastructure that lagged behind the capabilities of the United Kingdom, Germany, and other economic rivals."* - [36]
13

**Oblivious Transfer** *"Oblivious transfer (OT) is a fundamental cryptographic primitive that serves a building block for a number of interesting applications, such as secure two-party and multi-party computation."* - [9]
2, 3, 7, 9, 15, 18, 19, 21, 32, 35, 37, 40, 45

**Public Key** A key used for encryption of plaintext that can be decrypted by its corresponding *Secret Key*.
10, 11, 16, 23–25, 37, 44, 45

**RSA** *"One of the first public-key schemes was developed in 1977 (...). The RSA scheme has since that time reigned supreme as the most widely accepted and implemented approach to public-key encryption."* - [35]
13

**Secret Key** A key used for decryption of a *Ciphertext* that has been encrypted by its corresponding *Public Key*.
10, 11, 23–25, 44, 45

**Seed** A value used to generate other values. If two seeds are identical, they will generate the same values. An example of seeding could be for a random generator that will generate different random values depending on the seed that is used.
14

**SOLID-principles** SOLID has five principles that if followed ensures that the code is flexible, robust, and reusable. This is achieved by adhering to the following:
Single responsibility: *"a class should only have one, and only one, reason to change."*
Open/closed: *"You should be able to extend a classes behaviour, without modifying it."*
Liskov substitution: *"Derived classes must be substitutable for their base classes."*
Interface segregation: *"Make fine grained interfaces that are client specific."*
Dependency inversion: *"Depend on abstractions, not on concretions."* - [24]
19, 21

**SSH** *"The SSH protocol uses encryption to secure the connection between a client and a server. All user authentication, commands, output, and file transfers are encrypted to protect against attacks in the network."* - [33]
13

**TCP** TCP is used to provide the communication capabilities of the internet in a form that is useful for applications.

*"It provides reliable delivery of arbitrarily long sequences of bytes via stream-based programming abstraction. The reliability guarantee entails the delivery to the receiving process of all data presented to the TCP software by the sending process, in the same order."* - [7]
19, 20

**TLS** *"TLS is designed to make use of TCP to provide a reliable end-to-end secure service."* - [35]
13, 44

**Universally Composable** *"The Universal Composability (UC) framework is one of the most widely used methodologies for analysing protocol security under arbitrary composition."* - [9]
9

## Acronyms

**BC** Bouncy Castle 9, 13, 14, 38, 39

**DRV** Digital Remote Voting 4, 6, 7, 15, 35, 36, 39, 40

**DTO** Data Transfer Object 19, 20, 28,
*Glossary:* Data Transfer Object

**ECC** Elliptic Curve Cryptography 9, 13, 38

**ECOT** Efficient Composable Oblivious Transfer 2, 9, 19–22, 25–29, 33, 34, 40

**ECPoint** Elliptic Curve Point 21, 23–31

**OT** Oblivious Transfer 2, 3, 7, 9, 15, 18, 19, 21, 32, 35, 37, 40,
*Glossary:* Oblivious Transfer