UCLouvain

École polytechnique de Louvain

Efficient Implementation of Cryptographic Protocols for Electronic Voting

Author: **Razvan Alexandru Ursu**
Supervisors: **Olivier Pereira, Edouard Cuvelier**
Readers: **Olivier Pereira, Edouard Cuvelier, François-Xavier Standaert, Damien Giry**
Academic year 2018–2019
Master [120] in Electrical Engineering

Contents

Chapter 1

Introduction

The arrival of the *era of digitalization* has allowed our world to evolve towards newer, more ingenious electronic solutions to problems relating to our everyday lives. One of the many fields in which technology proposed an alternative was *voting*. Thus, *electronic voting* solutions are currently gaining more and more popularity against the older paper-based scenario.

However, one could argue that there are advantages and disadvantages on both sides [23]. Some of the advantages of electronic voting include:

- **Freedom of usability**: Electronic voting solutions allow for various scenarios that are different from the usual "one out of n candidates". Indeed we can easily adapt the interface of electronic voting for cases where one may need to choose a higher or even a variable number of candidates. We can also make weighted votes where people may choose their preferences. Although this is not impossible for paper-based voting, it is highly unpractical.

- **Monitoring**: Since every vote is encoded and stored into a database, the voter can make sure at any time that his unmodified vote is present on the *bulletin board* along with the votes of all the other people who voted. This is done publicly and via the internet. Verifying that one's vote is being accounted for and hasn't been modified is impossible in paper-based elections.

- **Counting**: In paper-based elections one of the most difficult tasks is the counting of the votes at the end of the election. The process is hard both in its nature but also because one must make sure that there is no cheating. Electronic voting overcomes this issue by being publicly open for audit. Thus anyone can verify that the vote counting was done properly.

Some of the disadvantages include:

- **Black box**: Probably the biggest disadvantage of electronic voting is its complexity. Indeed the whole process is based on mathematical concepts that are far more complicated than the ones behind paper-based voting. As a result people may not trust that the whole process is correct since not anyone has the capacity to understand the procedure. At a certain point people must trust someone to be unbiased.

- **Volatile memory and display**: A machine remains a machine, and even though nobody is trying to temper with the voting process there is always a risk of having practical issues when casting one's vote.

- **Content leaking** The actual voting process could leak side-channel information that are linked to the practical aspects of voting. Thus in Netherlands [6] an attack was found where the voting machine makes different sounds[1] depending on the chosen candidate.

As mentioned, the setup for such an electronic election may vary. Thus we can split the elections along two axis:

- **Browser vs Dedicated machine**: Voting on a `dedicated machine` resembles to the old paper-based solution in the way that in order to vote, one must travel to one of the voting centers and make their choice. The other option is to make use of the internet and allow people to vote from wherever they are as long as they are connected to the internet. In order to do so they must submit their vote via a `browser`. Although the second choice is more comfortable, it raises two main problems. First of all, the authority in charge of the election must verify that the voter is allowed to vote and that they don't cast multiple votes. Second of all, a browser is orders of magnitude slower than a dedicated machine and this speed decreases even further with the power of the device on which runs the browser. The reason is that browsers are limited to executing their code in JavaScript which is a rather slow, not very compatible with cryptography programming language. while a dedicated machine can and will deploy a compiled version of the same operations which will make it a lot faster.

[1] detectable only with a microphone

- **Homomorphic vs Mixnet**: `Homomorphic voting` accounts for all the voting structures in which *homomorphic encryption* is used. The homomorphic property of an encryption scheme is that one can manipulate the ciphertexts in a way that will modify the plaintexts in a desired manner, without having to decrypt any of the ciphertexts. In the field of electronic voting this allows to perform operations such as vote counting without having to decrypt every individual ballot, but instead make a sum of all the encrypted votes and decrypt the final result. On the other hand `Mixnet voting` decrypts each individual ballot, but only after the ballots have been well *shuffled* (or *mixed*). Counting the votes in this case takes more time but one can also have more freedom towards what can contain the ballot.

In the literature there are implementations of both homomorphic [7] and mixnet approaches [11] as well as browser-based [7] and dedicated machine implementations [13].

However, this document only focuses on the **homomorphic browser-based voting**. It aims at finding faster ways to perform encryption in the browser and at comparing the results with already existent implementations. The homomorphic encryption methods deployed in the final result are the El Gamal Encryption and the Perfectly Private Audit Trail (PPAT) Commitment Consistent Encryption (CCE) scheme [14]. The JavaScript implementation is based on the Verificatum library [28] which creates a framework for working with large integers in a language which would otherwise be limited to 53-bit integers.

Chapter 2

Theoretical aspects

In this section are presented all the mathematical aspects implemented during the project. It is going to be divided in two sub-sections. First of all the voting procedure based on the first homomorphic encryption called El Gamal will be explained. This section aims at describing this method but also at explaining how the entire election procedure works. The second subsection will delve deeper into the realm of elliptic curves and pairings. It will explain how they work but also how we can perform homomorphic encryption using these techniques and finally how the voting procedure has to be modified to account for this technique.

2.1 A first voting prototype

2.1.1 El Gamal homomorphic encryption

In order to understand this form of encryption one must start with some basic concepts of modular arithmetic.

Definition 2.1 *Let $\mathbb{Z}_q$ define the ring of integers modulo q:*

$$\mathbb{Z}_q = \{a \in \mathbb{Z} \mid 0 \leq a < q\} \tag{2.1}$$

Over this group we can define the following properties:

1. **Addition and Subtraction**: $\forall a, b \in \mathbb{Z}_q$ we can define $\mathbf{a} \pm \mathbf{b}$ as $a \pm b \ mod \ q$. Thus $\mathbf{a} \pm \mathbf{b} \in \mathbb{Z}_q$
2. **Multiplication**: $\forall a, b \in \mathbb{Z}_q$ we can define $\mathbf{a} \cdot \mathbf{b}$ as $a \cdot b \ mod \ q$. Thus $\mathbf{a} \cdot \mathbf{b} \in \mathbb{Z}_q$
3. **Additive and multiplicative identity**: $\forall a \in \mathbb{Z}_q$ we define 0 as the additive identity $\mathbf{a} + \mathbf{0} = \mathbf{a}$ and 1 as the multiplicative identity $\mathbf{a} \cdot \mathbf{1} = \mathbf{a}$

One may notice that although we may perform addition, subtraction and multiplication over this ring we can not perform the division for any two elements. This is due to the presence of the element 0 in the set. Let us define another set:

Definition 2.2 *Let $\mathbb{Z}_q^\times$ define the multiplicative group modulo q:*

$$\mathbb{Z}_q^\times = \{a \in \mathbb{Z} \mid 1 \leq a < q\} \tag{2.2}$$

which is followed by the theorem (proof at [1] p.19 Proposition I.3.1):

Theorem 2.1 *Let q be a prime integer, then $\forall a \in \mathbb{Z}_q^\times, \exists b \in \mathbb{Z}_q^\times$ s.t. $a \cdot b = 1 \ mod \ q$. b is then called the inverse of a mod q.*

The main advantage of working in $\mathbb{Z}_q^\times$ is that, **if q is prime**, we may always find an inverse for any element in $\mathbb{Z}_q^\times$ and this element is also going to be in $\mathbb{Z}_q^\times$. Let us now define the concept of group order and Euler's totient function:

Definition 2.3 *The order of a group is defined by the number of elements it comprises. The order of $\mathbb{Z}_q^\times$ is given by Euler's totient function $\varphi(q)$. In case q is prime we have $\varphi(q) = q - 1$ (proof at [1] p.15).*

which leads to the Euler's totient theorem (proof at [1] p.22 Proposition I.3.5):

Theorem 2.2 *Let q be a prime integer, then $\forall a \in \mathbb{Z}_q^\times$ we have:*

$$a^{\varphi(q)} = a^{q-1} = 1 \ mod \ q \tag{2.3}$$

Let us now discuss the concept of subgroups:

Definition 2.4 *Let q be a prime integer and $g \in \mathbb{Z}_q^\times$ be any element in $\mathbb{Z}_q^\times$. We define a subgroup of $\mathbb{Z}_q^\times$ as the set:*

$$\{g,\ g^2,\ g^3\ ...\ g^r\} \tag{2.4}$$

where r the first integer such that $g^r = 1 \ mod \ q$

There are several things to be noted about this sub-group:

- all the elements inside this subgroup are different
- In order to respect Euler's totient theorem the order of the subgroup, in this case r, must divide the order of $\mathbb{Z}_q^\times$ which is $\varphi(q)$. That is $r \mid \varphi(q)$.
- every element g^i has an inverse g^{r-i} which belongs to the same subgroup.

We call g the generator of the subgroup since repeatedly multiplying it by itself generates all the elements inside the subgroup. Note that since $r \mid \varphi(q)$ it means that the order of any sub-group must be a factor of $\varphi(q)$.

Throughout this section we are interested in finding a **prime order subgroup** of a **multiplicative group modulo a prime q**. Let us illustrate this by a small example:

Example 2.1 *Let $q = 11$ be the modulus of the multiplicative group $\mathbb{Z}_q^\times$. Since q is prime, we have $\varphi(11) = 10$. This implies that the order of the subgroup generated by any element in $\mathbb{Z}_q^\times$ is either 1, 2, 5 or 10. This is verified in the following table where we have the result of g^i mod $11, \forall g, i \in \mathbb{Z}_q^\times$. The order of the subgroup is given by the first occurrence of 1 in each line:*

g \ i	1	2	3	4	5	6	7	8	9	10	order
1	①	1	1	1	1	1	1	1	1	1	1
2	2	4	8	5	10	9	7	3	6	①	10
3	3	9	5	4	①	3	9	5	4	1	5
4	4	5	9	3	①	4	5	9	3	1	5
5	5	3	4	9	①	5	3	4	9	1	5
6	6	3	7	9	10	5	8	4	2	①	10
7	7	5	2	3	10	4	6	9	8	①	10
8	8	9	6	4	10	3	2	5	7	①	10
9	9	4	3	5	①	9	4	3	5	1	5
10	10	①	10	1	10	1	10	1	10	1	2

Table 2.1: g^i *mod* 11 where the subgroup order is the index i of the first occurrence of 1 on each line

From Table 2.1 and the respective orders we may conclude that 3, 4, 5 and 9 are generators of the same prime-order subgroup which is {1, 3, 9, 5, 4}. The reason why this subgroup is interesting is that any of its elements (except for 1) can generate the entire subgroup and of course because all of its elements have an inverse. This subgroup is called **a cyclic subgroup of order 5**.

However, there is another very interesting property that these subgroups have, which is the fact that the mapping between the exponents i and the elements of the sub-group g^i is **one-way**. This means that by looking at a random element g^i, although we know g there is no efficient method that allows to extract i other than trying every possible value. This is called the **Discrete logarithm (DL) problem** and all the cryptographic primitives in the rest of this section reside on

its difficulty. Note that we know efficient polynomial-time methods[1] that compute g^i from g and i. It is only the inverse function which has exponential complexity.

We now have all the tools to present the El Gamal Encryption method:

Definition 2.5 (El Gamal Encryption) *Let $\mathbb{G} \subset \mathbb{Z}_q^\times$ be a cyclic subgroup of prime order r with ($||r|| = n$ bits) and let g be a generator of this group. Let m be the message to encrypt, $m \leftarrow \mathbb{Z}_r$.*

- $\mathbf{Gen}(\mathbf{1^n})$ *runs polynomial-time algorithm to find $\mathbb{G}, q, g$ as described above and then chooses random $x \xleftarrow{\$} \mathbb{Z}_r$ and compute $h = g^x$. The public key is $pk = \langle \mathbb{G}, q, g, h \rangle$ and the private key is $sk = \langle \mathbb{G}, q, g, x \rangle$.*
- $\mathbf{Enc_{pk}}(\mathbf{m})$ *chooses a random $y \xleftarrow{\$} \mathbb{Z}_r$ and outputs the ciphertext $\langle c1, c2 \rangle = \langle g^y, h^y \cdot m \rangle$*
- $\mathbf{Dec_{sk}}(\langle \mathbf{c1}, \mathbf{c2} \rangle)$ *computes $m := (c_2 / c_1^x)$*

The privacy of the secret key is guaranteed by the DL problem. However, the security of the encryption can not reside only on this assumption. Indeed, if knowledge of $\langle g, h = g^x, c1 = g^y \rangle$ can leak any information about $h^y = g^{xy}$ then information about the message will be leaked. Thus the encryption is secure under what is called the **Decisional Diffie-Hellman (DDH) assumption** [2]:

Definition 2.6 (Decisional Diffie-Hellman problem) *Let $\mathbb{G} \subset \mathbb{Z}_q^\times$ be a cyclic subgroup of prime order r with ($||r|| = n$ bits) and let g be a generator of this group. Let $\mathbb{A}$ be the adversary. We denote the following experience $DDH_{\mathbb{A},\mathbf{G}}(n)$*

- $\mathbf{G}(\mathbf{1^n})$ *runs polynomial-time algorithm to find $\langle \mathbb{G}, q, g \rangle$*
- *Choose $(x, y, z) \leftarrow \mathbb{Z}_r^3$*
- *Flip a coin $b \xleftarrow{\$} \{0, 1\}$ and set $h_1 := g^{xy}$ and $h_0 := g^z$*
- *Set $b' \leftarrow \mathbb{A}(\mathbb{G}, q, g, h_b)$*
- *If $b = b'$ then the experience succeeds (e.g. $DDH_{\mathbb{A},\mathbf{G}}(n) = 1$)*

The security of the El Gamal method thus resides in the hypothesis that the DDH problem described above is difficult. Mathematically this assumption is written as $Pr[DDH_{\mathbb{A},\mathbf{G}}(n) = 1] = \frac{1}{2} + \epsilon(n)$ where $\epsilon(n)$ is a negligible function.

[1] in the number of bits of the exponent

The encryption method used throughout the rest of the section is a variation of the standard El Gamal Encryption method in which the message is encoded as g^m instead of simply m:

Definition 2.7 (El Gamal Homomorphic Encryption) *Let $\mathbb{G} \subset \mathbb{Z}_q^\times$ be a cyclic subgroup of prime order r with ($||r|| = n$ bits) and let g be a generator of this group. Let m be a relatively small positive integer.*

- $\mathbf{Gen}(\mathbf{1^n})$ *runs polynomial-time algorithm to find $\mathbb{G}, q, g$ as described above and then chooses random $x \xleftarrow{\$} \mathbb{Z}_r$ and compute $h = g^x$. The public key is $pk = \langle \mathbb{G}, q, g, h \rangle$ and the private key is $sk = \langle \mathbb{G}, q, g, x \rangle$.*
- $\mathbf{Enc_{pk}}(\mathbf{m})$ *chooses a random $y \xleftarrow{\$} \mathbb{Z}_r$ and outputs the ciphertext $\langle c1, c2 \rangle = \langle g^y, h^y \cdot g^m \rangle$*
- $\mathbf{Dec_{sk}}(\langle \mathbf{c1}, \mathbf{c2} \rangle)$ *computes $m := log_g(c_2/c_1^x)$*

This type of encryption is indeed homomorphic. Given two ciphertexts:

$$\langle c_{11} = g^{y_1}, c_{21} = h^{y_1} g^{m_1} \rangle \tag{2.5}$$
$$\langle c_{12} = g^{y_2}, c_{22} = h^{y_2} g^{m_2} \rangle \tag{2.6}$$

one may simply multiply the two ciphertexts element by element to obtain:

$$\langle C_1 = g^{y_1+y_2}, C_2 = h^{y_1+y_2} g^{m_1+m_2} \rangle \tag{2.7}$$

which decrypted in the same manner leads to $m_1 + m_2$.

Although the last step of this method involves the computation of a discrete logarithm, this is not an issue as long as the message is relatively small ($m < 2^{16}$) compared to the size of the key. When performing such an encryption the range of the message space is one of the most important aspects that needs to be taken into account.

2.1.2 The voting procedure

The voting process is composed of 5 main steps, which are presented in Figure 2.1[18]:

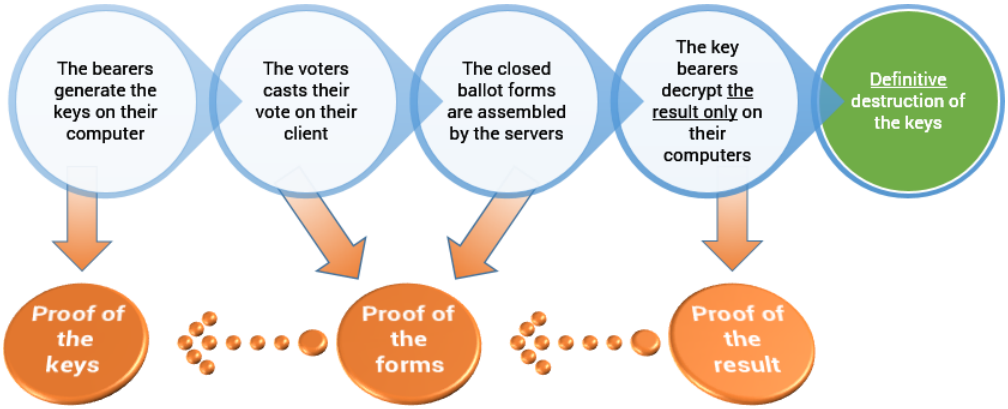


Figure 2.1: Voting procedure for browser-based elections

1. First of all the public key of the election must be computed homomorphically between a fixed number of trustees, each one of them holding a share of the secret key.

2. The actual election begins, each candidate receives the public key, encrypts their vote and sends it to the server. A signature of each voter's ballot is published on a public bulletin board for the voter to be able at any time to verify that their vote is being accounted for and hasn't been modified.

3. Once the election is over, the ballots are added homomorphically by the server and are sent to the trustees.

4. The trustees decrypt the final ballot with their share of the secret key

5. Finally the results are published and the secret keys are destroyed

In this case, the El Gamal Homomorphic Encryption was used to create the ballots. Let's analyze how it was implemented in each of these steps.

The **first step** of the voting process begins with the server loading a `JSON file` which contains all the information about the election. Parameters inside this file include: names of the candidates, group generator, group order, group modulus, name of the election, description, start date, end date, election ID etc. One of the key information missing from this file is the public key. This is normal since it

has not yet been computed at this stage. Once this file is loaded it is stored in a database for trustees to query. Note that the cryptographic parameters such as the group generator and the group order don't have to be different from one election to another.

Then the public key used for the encryption must be computed, so the trustees ask the server for the election parameters. With these parameters one might think it is enough for someone to pick a random private key x, compute the public key g^x and make it public so that people can cast their votes. However the main issue with this approach is that the person who knows the private key would be able to decrypt any ballot and look inside individual votes. This is a problem since the goal of this type of encryption is to find out the result only. Thus, the private key has to be shared between several *trustees*.

The key element here is that the public key can be computed without ever knowing the private key with the help of homomorphic addition. The protocol works as follows:

1. Each trustee i picks $x_i \xleftarrow{\$} \mathbb{Z}_q^\times$
2. Each trustee computes their share of the public key as $pk_i = g^{x_i}$
3. Each trustee computes a non-interactive zero knowledge proof (ZKP) proving that they know the private key[2]
4. Each trustee sends their share of the public key pk_i and the correspondent ZKP to the central server which stores each share of the public key after verification of the correspondent ZKP.
5. After all the shares have been received the server performs a homomorphic addition of the public keys as such: $pk = \prod pk_i = \prod g^{x_i} = g^{\sum x_i} = g^x$ where x is the private key that nobody knows.
6. The server updates the election parameters in the database with the public key.

The shares must remain secret and the trustees mustn't collude in order to compute the private key. Revealing the actual private key is impossible as long as there is at least one trustee that doesn't cheat.

The **second step** of the election is the actual voting. During this step the voter connects via a browser to the web page of the election, receives the election parameters (which now contains the public key as well) and encrypts their choice

[2]The zero knowledge proofs are to be explained in subsection 2.1.3

after authentication[3]. Depending on the election rules the voter may need to choose one or more candidates or he may even choose to vote blank. The ballot encryption process is the following:

1. The voter encrypts using the El Gamal Homomorphic Encryption a 1 for each candidate they chose and a 0 for each candidate they didn't.
2. For each encryption they perform a Disjunctive ZKP (e.g. ZKP proving that they encrypted either a 0 or a 1 but not something else)
3. Once all the ballots are created the voter sums up homomorphically the ballots he created for each candidate. The encrypted message in this ballot contains the total number of candidates the voter chose. On this encrypted ballot the voter computes a Conjunctive ZKP that proves that this number of candidates lies in the interval of accepted number of votes. This proof is done even if there is a fixed number of candidates that the voter has to pick.
4. The encrypted ballots and proofs are sent to the server who checks the proofs and stores each ballot in the database.

The drawback of homomorphic encryption in voting is that the number of encryptions increases linearly in the number of candidates, and so does the time required to perform all the encryptions. This is one of the reasons why homomorphic encryption is unpractical when the number of candidates is very big.

The **third step** of the process occurs after the voting period is over. At this stage the server has all the ballots from all the voters. It then proceeds to homomorphically adding the ballots corresponding to the same candidate and creating a set of ciphertexts each containing the total number of votes for one candidate. This result is saved in the database along with the other election parameters.

The **fourth step** requires the trustees to come together again and help decrypt each candidate's ballot. The procedure works as follows:

1. For each candidate the server sends the final ballot, $\langle c1, c2 \rangle$ to each trustee
2. Each trustee then computes their decryption factor, which is $decF_i = c_1^{x_i}$ for each candidate's ballot, where x_i is their share of the private key.
3. Each trustee computes a Chaum-Pederson ZKP prooving that they used the same x_i to compute the decryption factor as they did to compute their share of the public key

[3]Authentication is out of the scope of this work

4. The server receives the decryption factors and the proofs and stores them in the database if the proofs hold

5. Finally after all the trustees sent the decryption factors the server homomorphically adds all the decryption factors as such: $\prod decF_i = \prod c_1^{x_i} = c_1^{\sum x_i} = c_1^x$ where x is the private key of the election. The server then proceeds to perform the decryption as usual: $\sum v_i = log_g(c_2/c_1^x)$ for each candidate. This sum of votes represents the number of people who voted for candidate i.

It is very important that the trustees do not lose their share of the private key. Otherwise it is impossible to decrypt the ballots and thus to obtain the result of the election.

The **fifth and final step** consists in the publication of the election results but also in the erasing of the private keys by the trustees in order to make sure that the election remains safe even against an *a posteriori* attack.

2.1.3 Zero-Knowledge Proofs

In this subsection we're going to present the need for Zero-Knowledge Proofs (ZKPs) and explain how they work.

A ZKP, as its name suggests, is a way one party called **the prover** proves to another party called **the verifier** that some hypothesis is correct, without revealing any information. Such a proof must fulfill three criteria:

1. **Completeness**: an honest verifier is always convinced by the prover if the hypothesis is correct.

2. **Soundness**: if the hypothesis is not correct, the prover can't prove the correctness to the honest verifier even if he doesn't follow the protocol and tries to cheat.

3. **Honest verifier zero-knowledge**: the honest verifier doesn't learn anything from the messages exchanged during the proving procedure.

An interactive ZKP is a protocol in which the prover and the verifier exchange a series of messages and perform a series of computations whose goal is to convince the verifier of the hypothesis. In order to describe this procedure let us describe the most basic ZKP known as Schnorr's protocol:

Definition 2.8 (Schnorr's protocol) *Let g be the generator of a cyclic subgroup $\mathbb{G}$ of prime order r and let $x \in \mathbb{Z}_q$ with q a prime modulus. The prover knows x and thus can compute $h = g^x$. He then publishes g and h and wants to prove knowledge of x in a Zero-Knowledge manner. The following protocol is used:*

1. *The prover picks $a \xleftarrow{\$} \mathbb{Z}_r$ and computes $A = g^a$. This is called **the commitment**. The prover then sends the commitment to the verifier.*

2. *The verifier picks $e \xleftarrow{\$} \mathbb{Z}_r$. This is called **the challenge**. The verifier sends the challenge back to the prover.*

3. *The prover computes $f = a + e \cdot x$. This is called **the response**. It then sends the response to the verifier and the protocol is over.*

Let us see how this protocol respects the properties cited above, thus proving knowledge of x by the prover:

- **Completeness**: If the prover is honest and knows x then they will respect the protocol and provide a real transcript $\langle A, e, f \rangle$. The verifier can verify the correctness by comparing g^f with $A \cdot h^e$. They should be equal.

- **Soundness**: Let's assume the prover doesn't know x but is still capable of producing a correct transcript. That means that for two different challenges e_1 and e_2 coming from the verifier he would be able to produce two correct responses f_1 and f_2. But if he can do that then he can find out what x is by computing $\frac{f1-f2}{e1-e2} = x$ or this contradicts the initial hypothesis that the prover doesn't know x.

- **Honest Verifier Zero-Knowledge**: In order to prove this property it is only required to show that the verifier can produce a valid transcript itself without knowledge of x. He can do so by picking $e, f \leftarrow \mathbb{Z}_r$ and then computing A as such: $A = \frac{g^f}{h^e}$. The resulting $\langle A, e, f \rangle$ is a valid transcript

In practice Schnorr's protocol is used in a slightly modified version. The main issue with this kind of protocol is that the server and the client must exchange a series of messages in order to compute the proof. In browser-based elections the proofs are required to be computed locally without any connection to the internet. In this case the interactive ZKP must become **non-interactive** and we do that by replacing the only message comming from the server, the challenge, by something else. On the verifier's side the challenge is nothing but a random element in $\mathbb{Z}_r$ so the client must be able to simulate this randomness and be able to prove that it is really random. It does so by applying a **hash function** over a series

of parameters[4] and assuming that this is the actual challenge, since the output of a hash function is pseudo-random. The verifier then performs the verification as usual, but only after computing the challenge using the same hash function over the same parameters. This variant is called the **Non-interactive Schnorr's protocol**. Note that replacing the challenge by the output of a hash function will be used to make other ZKPs non-interactive as well.

The Non-intercative Schnorr's protocol is actually used during the first step of the election (the key generation step) where each trustee has to provide a ZKP of their share of the key x_i given their share of the public key pk_i.

The second and third types of ZKP, which are the disjunctive and the conjunctive proofs, are quite similar in the way that the disjunctive proof is a particular case of the conjunctive one. Thus let us explain the way the conjunctive proof wroks:

Definition 2.9 (The conjunctive proof) *Let g be the generator of a cyclic subgroup $\mathbb{G} \subset \mathbb{Z}_q^\times$ a cyclic subgroup of prime order r with q a prime modulus as well, let $x \in \mathbb{Z}_r$ be the private key of the election, let $h = g^x$ be the public key of the election and let $\langle c1, c2 \rangle$ be an El Gamal Homomorphically encrypted ciphertext of message m. The prover wishes to prove that the message m they encrypted lies between a minimum value min and a maximum value max. It does so by creating a simulated proof transcript for each value in the set $\{min, min+1, ...max-1, max\}/m$ and then computes a real proof on the real encrypted value m. The following protocol is used:*

1. *For each message $i \in \{min, min+1, ...max-1, max\}/m$ the prover picks $e_i, f_i \xleftarrow{\$} \mathbb{Z}_r$, computes $A_i = \frac{h^f}{\beta^e}$ where $\beta = \frac{c_2}{g^i}$ and stores all the transcripts $\langle A_i, e_i, f_i \rangle$.*

2. *For the real message the prover picks $a_m \xleftarrow{\$} \mathbb{Z}_q$ and computes the commitment $A_m = h^{a_m}$. It then proceeds by applying a hash function to a series of important parameters (that include the commitments of all the transcripts in the conjunctive proof among other important information) and then subtracts from this number e the sum of all the challenges used in the simulated proofs $e_m = e - \sum_{i \neq m} e_i$. The result will be the challenge used in the proof of the real message. Finally it computes the result as usual $f_m = a_m + e_m \cdot y$ where y is the randomness used to perform the El Gamal Homomorphical Encryption.*

3. *The prover then outputs the proofs ordered from min to max with the real proof inserted at the right place.*

[4] the choice of these parameters that will be explained at the end of the subsection

The verification of this ZKP is performed in the same manner as the verification of the Schnorr's protocol with a few extra steps:

1. First the verifier must re-compute the hash function and verify that it is equal to the sum of all the challenges $e = \sum e_i$
2. Then, for each transcript the verifier has to compute $\beta_i = \frac{c_2}{g^i}$ where $i \in \{min, min+1, ...max-1, max\}$
3. Finally the proof is accepted if $h^{f_i} = A_i \cdot \beta_i^{e_i},\ \forall i \in \{min, min+1, ...max-1, max\}$.

This proves the **Completeness** character of the conjunctive proof. The **Soundness** of the conjunctive proof can be reduced to the soundness of the real sub-proof, which was demonstrated for Schnorr's protocol. Concerning the **Honest Verifier Zero-Knowledge** character the verifier could simulate the real sub-proof in the same manner the others were simulated, thus obtaining a correct set of transcripts.

In the second step of the election process, which is the actual voting, the disjunctive proof is a variant of the conjunctive proof with $min = 0$ and $max = 1$ while in the actual conjunctive proof *min* and *max* are represented by the minimum and respectively maximum number of candidates a voter can vote for.

The final ZKP to be discussed here is the one provided by the trustees during the decryption process in step 4. The kind of ZKP needed here must prove that the same x_i was used to go from g to $pk_i = g_i^x$ as to go from c_1 to $decF_i = c_1^{x_i}$. The proof that does this is called **Chaum-Pedersen proof** and the protocol, made non-interactive, is the following:

Definition 2.10 (Non-interactive Chaum-Pedersen proof) *Let g_1 and g_3 be two elements of a cyclic subgroup $\mathbb{G} \subset \mathbb{Z}_q^\times$ of prime order r with q a prime integer as well and let $x \in \mathbb{Z}_r$. The prover knows x and thus can compute $g_2 = g_1^x$ and $g_4 = g_3^x$. He then publishes $\langle g_1, g_2, g_3, g_4 \rangle$ and wants to prove knowledge of x in a Zero-Knowledge manner as well as that the same exponent was used in both exponentiations. The following protocol is used:*

1. *The prover picks $a \xleftarrow{\$} \mathbb{Z}_r$ and computes the commitments $A = g_1^a$ and $B = g_3^a$.*
2. *The prover applies a hash function over a series if inputs that includes the commitments. The output of this hash function is the challenge $e \in \mathbb{Z}_r$.*
3. *The prover finally computes the response $f = a + e \cdot x$.*

The verification of this ZKP must respect three conditions:

1. First of all the output of the hash function[5] must be consistent with the challenge of the proof
2. Then one must verify the equality between g_1^f and $A \cdot g_2^e$
3. Finally one must verify the equality between g_3^f and $B \cdot g_4^e$

These steps prove the **Completeness** property of the ZKP. The proof's **Soundness** is given by the same argument as for Schnorr's protocol. That is if a non-honest prover who doesn't know x can produce a right response f_1 for a challenge e_1 then they can produce a right response f_2 for a second challenge e_2. But if they do so, then they can extract $x = log_{g_1}(g_2) = log_{g_3}(g_4) = \frac{f_1 - f_2}{e_1 - e_2}$ which means that the non-honest prover actually knows x which contradicts the initial hypothesis. Finally the **Honest Verifier Zero-Knowledge** property is given by the verifier being able to pick $e, f \xleftarrow{\$} \mathbb{Z}_r$ and then compute $A = \frac{g_1^f}{g_2^e}$ and $B = \frac{g_3^f}{g_4^e}$. This produces a correct transcript without the need of knowing x.

One last aspect that should be discussed in this section regards the parameters that should be fed to the hash function in order to produce a correct challenge. There are three main things that should be included in the parameters to be hashed:

- First of all **the commitment** of the ZKP must be present as it is one of the main parameters of that specific proof. It is what makes the hash different from the one of other proofs.
- Then **the ciphertext that corresponds** to the ZKP must be present as well. This avoids being able to compute the same hash for the same kind of proof made on another ciphertext, in the case someone who cheats might use the same randomnesses to compute the commitments.
- Finally the hash parameters must contain **details that are specific to the election** such as: the public key, the group generator, the group modulus, the election date, the election description, the number of candidates etc. These parameters fix the ZKP to this election and avoid the same proof to be used in other future elections.

[5] as computed by the verifier

2.2 A second voting prototype

In the first section of this chapter we described a first homomorphic voting prototype based on the El Gamal Homomorphic encryption. Although this form of encryption which is at the core of the voting procedure is easy it is also less secure. There are ways to achieve a more secure way of performing homomorphic encryption at the price of it being slightly slower. In this second section such a method will be described. This method is called **Perfectly Private Audit Trail for simple messages (PPATS) Commitment Consistent Encryption (CCE)**.

In order to do so one must first understand the basics of **Elliptic Curve (EC)** cryptography and **Pairings**. We will start this section by describing these concepts and will end it by showing how the voting procedure in the first section can be adapted to allow for this homomorphic encryption to replace El Gamal.

2.2.1 Elliptic Curves

The technical aspects described in this subsection are inspired from [20].

Let us begin by describing what an elliptic curve is in general:

Definition 2.11 *We define the elliptic curve E as the set of points (x, y) with $x, y \in K$ where K is a general field, that respect the following equation:*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.8}$$

*with $a_1, ..., a_6 \in K$. To this set of points we add one extra point, called **point at infinity**[6] $\mathcal{O}$.*

The formula of the elliptic curve in the definition above is called *the general Weierstrass equation*. This form can be simplified in what is called *the short Weierstrass equation*:

$$E : y^2 = x^3 + ax + b \tag{2.9}$$

This equation is to be used throughout this chapter. Figure 2.2 shows some examples of elliptic curves defined over $\mathbb{Q}$ [26]. Let us now define the operations we can perform over elliptic curves, namely the **addition** and **doubling**.

Elliptic curves of this form have the interesting property that any line in the plane intersects the curve in at most 3 points. This creates 4 possible scenarios

[6]The point at infinity will be described later

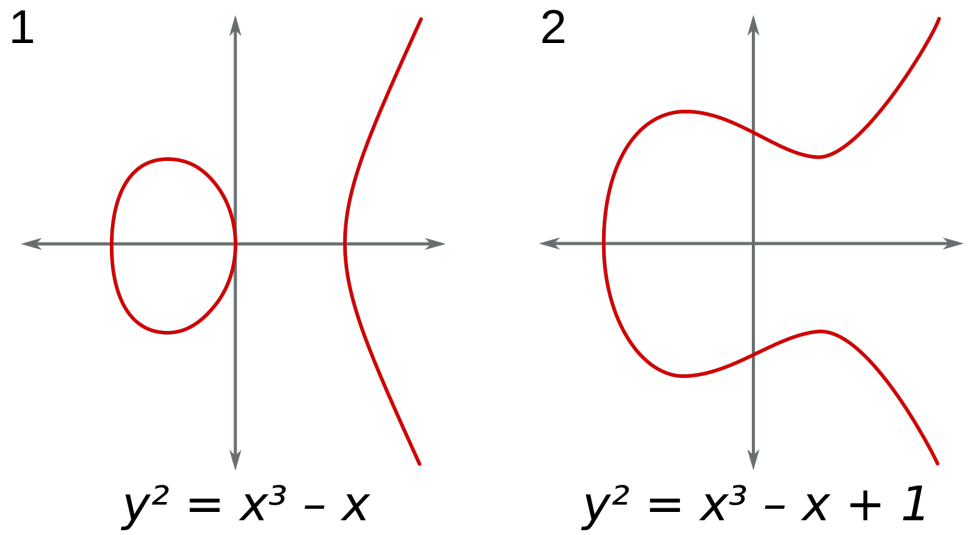


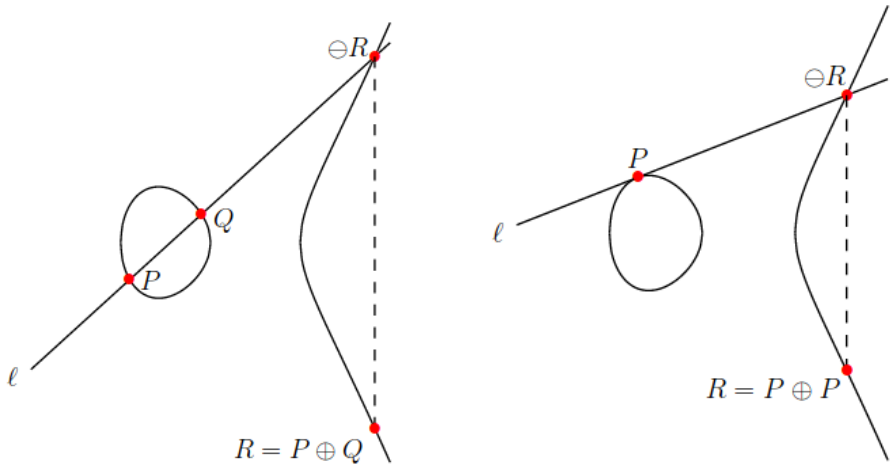
Figure 2.2: Elliptic curve examples over $\mathbb{Q}$

and depending on each case we can define what the addition rule on the curve is. Let us analyze these cases:

1. **The line intersects 3 points**: Let's suppose the situation in Figure 2.3a[20]. We define the addition between P and Q as R, the symmetrical point with respect to the X-axis of the point $-R$, where $-R$ is the third intersection between the line passing through P and Q. This is called **Point addition**

2. **The line intersects 2 points**: Here we have two possible cases. The first case is depicted in Figure 2.3b[20]. Here the line is non-vertical and tangent to the curve, in which case it intersects a second point $-R$ on the curve. The result is R, the symmetrical of $-R$ with respect to the X-axis. This is called **point doubling**. The second option is depicted in Figure 2.4a where the line is vertical but not tangent. In this case the line intersects the curve in two points P and $-P$. We define the third point as **the point at infinity** $\mathcal{O}$ which is situated towards infinity along the Y-axis. The reason why the points are called P and $-P$ is because they are symmetrical with respect to the X-axis. Thus we have that $P + (-P) = \mathcal{O}$. The point at infinity can be viewed as the *zero* of the addition. From the same scenario we can extract as well that $P + \mathcal{O} = P$ which is a particular case of the addition.

3. **The line intersects one point**: This case occurs only when the line is vertical and tangent to the curve. This is a particular case of doubling when $[2]P = \mathcal{O}$. The situation is depicted in Figure 2.4b



(a) Addition with $P \neq Q$ (b) Doubling with $y_P \neq 0$

Figure 2.3: Addition and doubling on elliptic curves

4. **The line doesn't intersect the curve at all**: In this case no addition and doubling can be performed. This case can never occur.

Let us now provide the formulas that allow to compute the result of the addition and doubling operations algebraically rather than geometrically. We're focusing on the non-trivial cases depicted in Fig. 2.3a and Fig. 2.3b. Suppose we want to find the intersection between the elliptic curve in equation 2.9 and the line: $l : y = \lambda x + \nu$. Suppose we have $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ both belonging to the curve E.

- If $P \neq Q$ then the addition formulas give us $P + Q = R = (x_R, y_R)$:

$$\lambda = \frac{y_Q - y_P}{x_Q - x_P} \tag{2.10}$$

$$\nu = y_P - \lambda x_P \tag{2.11}$$

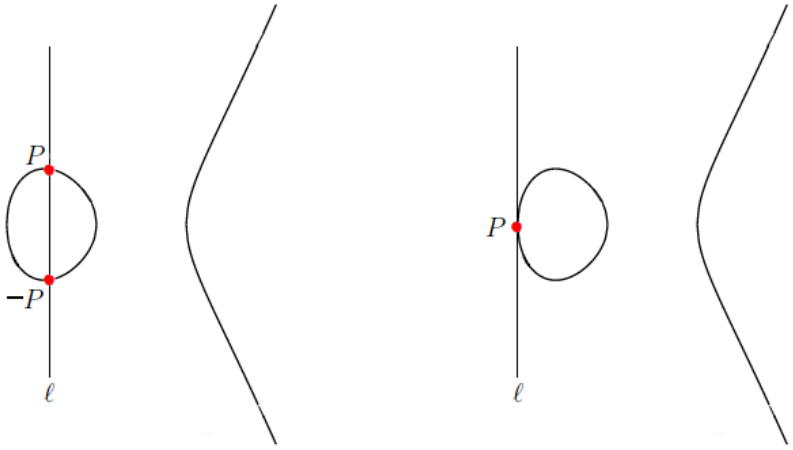
$$R = (x_R, y_R) = (\lambda^2 - x_P - x_Q, -(\lambda x_R + \nu)) \tag{2.12}$$

- If $P = Q$ then the doubling formulas give us $[2]P = R = (x_R, y_R)$:

$$\lambda = \frac{3x_P^2 + a}{2y_P} \tag{2.13}$$

$$\nu = y_P - \lambda x_P \tag{2.14}$$

$$R = (x_R, y_R) = (\lambda^2 - 2x_P, -(\lambda x_R + \nu)) \tag{2.15}$$



(a) Addition with P and $-P$ (b) Doubling with $y_P = 0$

Figure 2.4: Particular cases of addition and doubling on elliptic curves

Combining these two formulas one is able to compute $R = [k]P$ for any $k \in \mathbb{Q}$. There are polynomial-time algorithms that help compute this value. However, there are is no polynomial-time algorithm that allows the computation of k from R and P. This problem is similar to the DL problem seen in the previous section and is called **Elliptic Curve Discrete Logarithm (ECDL)** problem and it is considered as hard to break as the DL problem.

So far we assumed the geometrically intuitive curves were defined over $\mathbb{Q}$, but in practice we are going to work with curves defined over finite fields such as $\mathbb{F}_p = \mathbb{Z}_p^\times$ with p a prime modulus. Over such a field the equations above hold as well, only all the computations are reduced modulo p and we define the division by a number x as the multiplication by the inverse of x modulo p.

The fact that we are defining the curve over a finite field means that the number of points belonging to the curve is finite. The **Hasse's bound** [9] stipulates that the number of points on $E/\mathbb{F}_p$ differs from $p+1$ with at most $2\sqrt{p}$. If we denote $\#E(\mathbb{F}_p)$ the number of points on the curve then we have:

$$|\#E(\mathbb{F}_p) - (p+1)| \leq 2\sqrt{p} \tag{2.16}$$

or if we write it depending on the **trace of Frobenius** t:

$$\#E(\mathbb{F}_p) = p + 1 - t, \quad |t| \leq 2\sqrt{p} \tag{2.17}$$

One last parallel to make between elliptic curves and finite groups is the notion of **r-torsion** which is the subset of points on the elliptic curve who have an order r, that is they are sent to the point at infinity $\mathcal{O}$ when multiplied by r. Indeed, there exists a k for which if we compute $[k]P$ we obtain $\mathcal{O}$. For any point on $E/\mathbb{F}_p$ we have that $[\#E(\mathbb{F}_p)]P = \mathcal{O}$ which is the equivalent of raising an element to the power $\varphi(p)$ in finite fields. But here again, $\#E(\mathbb{F}_p)$ is not the smallest scalar by which if we multiply the point we obtain $\mathcal{O}$. We define r to be the smallest scalar such that $[r]P = \mathcal{O}$ and this r has the property that $r|\#E(\mathbb{F}_p)$.

2.2.2 Extension fields

In our pairing-based implementation we're going to work with r-torsions where r is a prime integer. In that case the subset of points which constitute the r-torsion is cyclic. Thus any point in the r-torsion generates the entire subgroup.

The issue with pairing-based implementations is that we are required to have two different subgroups of order r, so how can this be achieved if there is only one such cyclic subgroup over $\mathbb{F}_p$? The solution is to extend $\mathbb{F}_p$ to $\mathbb{F}_{p^d}$ via an irreducible polynomial of degree d.

One can do that by choosing an irreducible polynomial of degree d and represent the elements in $\mathbb{F}_{p^d}$ as polynomials of degree at most $d-1$ with coefficients in $\mathbb{F}_p$ as such:

Definition 2.12 *Let p be a prime integer and $P(u)$ an irreducible polynomial of degree d. We define $\mathbb{F}_{p^d} = \mathbb{F}_p(u)$ with $P(u)$ as the set of all the elements of the form:*

$$a_0 + a_1 u + a_2 u^2 + ... + a_{d-2} u^{d-2} + a_{d-1} u^{d-1} \tag{2.18}$$

with $a_0, ..., a_{d-1} \in \mathbb{F}_p$

The addition and subtraction operations over such an extension field are done element-wise which makes its complexity linear in the extension degree d. However, multiplication and division are slightly more complicated.

Concerning the multiplication between two random elements in $\mathbb{F}_{p^d}$ there are two steps. First there is the actual multiplication between each pair of components of the two elements which makes the complexity of the operation increase quadratically in the extension degree d. Then there is the reduction of the terms with degree higher than or equal to d. This reduction is done by keeping the remainder of a simple euclidean division between the non-reduced element and the irreducible polynomial.

Finally the division between two random elements can be performed via the extended euclidean algorithm [27] which requires multiple euclidean divisions and multiplications.

The Tutorial in [20] Chapter 4.1 shows that extending a field to a certain degree k would effectively allow us to find more cyclic sub-groups of the same prime order r. It also shows that this is precisely the case when the extension order reaches k and not before. One of the definitions of k is that of the smallest d such that $r|(p^d - 1)$. k is called **the embedding degree**.

However, extending a group via a k^{th}-degree irreducible polynomial is not the only way one can achieve the embedding degree extension. Indeed successive extensions could be performed each with smaller degree polynomials. This operation is called **Towered Extension fields**. The easiest way to explain how they work is to give an example[7]

Example 2.2 *Let p be a prime order modulus. One wishes to compute the extension field $\mathbb{F}_{p^{12}}$ using towered extension fields:*

- $\mathbb{F}_{p^2} = \mathbb{F}_p/(i^2 + 1)$
- $\mathbb{F}_{p^6} = \mathbb{F}_{p^2}/(Y^3 - \xi)$
- $\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}/(X^2 - \xi)$

with ξ neither a square root nor a cubic root in $\mathbb{F}_{p^2}$. Thus an element in $\mathbb{F}_{p^{12}}$ can be expressed as:

$$((a_0+a_1i)+(a_2+a_3i)Y+(a_4+a_5i)Y^2)+((a_6+a_7i)+(a_8+a_9i)Y+(a_{10}+a_{11}i)Y^2)X \tag{2.19}$$

The reason why one might want to use the towering instead of the direct extension is that for the former there are optimized methods to perform multiplications in $\mathbb{F}_{p^{12}}$ that take less then the naive $12^2 = 144$ multiplications in $\mathbb{F}_p$.

We said that the reason behind making an extension is to find other cyclic subgroups in the r-torsion. But how many such cyclic subgroups emerge when extending a field to its embedding degree? A result by [4] shows that there are precisely r^2 elements in the r-torsion and since the point at infinity is in all of them then it means there are $r + 1$ cyclic sub-groups of order r. One of these sub-groups

[7]This example is not random. It represents the actual towering that will be used in our pairing-based implementation

is entirely in $E(\mathbb{F}_p)$ (the original cyclic sub-group). We call it $E(\mathbb{F}_p)[r]$ while the others are entirely in $E(\mathbb{F}_{p^k})/E(\mathbb{F}_p)$.

In our pairing-based implementation we are going to use two out of these $r+1$ sub-groups. Although the choice is easy for the first sub-group which is going to be the one in which the computations are the easiest to perform, $\mathbb{G}_1 = E(\mathbb{F}_p)[r]$, the choice of the second sub-group can vary. In literature there are different types of pairings depending on the choice of this second group. In our case we chose a **type 3 pairing** which implies that the second group is what we call the **trace zero sub-group**, which is defined as:

Definition 2.13 *Let p, r be prime integers. Let E be an elliptic curve defined over $\mathbb{F}_{p^k}$ with k the embedding degree of the curve. The trace zero sub-group $\mathbb{G}_2$ is a prime order sub-group of order r in which all the elements are mapped to the point at infinity via the Trace map:*

$$Tr(P) = \sum_{i=0}^{k-1} (x^{p^i}, y^{p^i}) \tag{2.20}$$

That is $P \in \mathbb{G}_2$ if $Tr(P) = \mathcal{O}$

Al the points respecting this property lie in the same subgroup. Furthermore, $TR(P)$ maps all the points from the other subgroups to a point in $E(\mathbb{F}_p)[r]$. This is why the trace-zero subgroup is special.

2.2.3 Divisors

Another building block of pairings is the one of **divisors**. This section aims at briefly explaining what divisors are and at giving some useful definitions and properties that will come in useful in the next section.

Definition 2.14 (Divisor) *A divisor is a way of denoting a multi-set of points on an elliptic curve E as:*

$$D = \sum_{P \in E(\mathbb{F}_{p^k})} n_P(P) \tag{2.21}$$

where $n_P \in \mathbb{Z}$ is the multiplicity of each point P on the curve.

Note that this is not point multiplication as there are no brackets around n_P. Instead we add parentheses around the point P to avoid confusion. The following list enumerates a series of definitions related to divisors:

- The set of all the divisors of an elliptic curve is denoted by $Div(E)$.
- The **degree** of a divisor in denoted $Div(D)$ and is represented by the sum of the multiplicities of all the individual points: $Div(D) = \sum_{P \in E(\mathbb{F}_{p^k})} n_P$
- The **support** of a divisor, denoted $supp(D)$ represents the set of points with multiplicities different from 0 in D: $supp(D) = \{P \in E(\mathbb{F}_{p^k}) | n_P \neq 0\}$

The concept of divisors allows us to approach the concept of functions. Mainly we define the **divisor of a function** f as:

$$(f) = \sum_{P \in E(\mathbb{F}_{p^k})} ord_P(f)(P) \tag{2.22}$$

where $ord_P(f)$ is positive if the function f has a zero at P and negative if it has a pole at P. In the case of a function we always have $Deg((f)) = 0$ so even if a function has a number of poles that is different from the number of zeros than an appropriate number of points at infinity must be added or subtracted to make the degree equal to 0. The following example illustrates this property:

Example 2.3 *Let us reconsider the example in Figure 2.3a. In this case the line intersects the curve in P, Q and $-(P+Q)$. Thus in order to respect the condition above the divisor of the line function becomes:*

$$(f) = (P) + (Q) + (-(P+Q)) - 3(\mathcal{O}) \tag{2.23}$$

However, the fact that the divisors of functions have degree 0 it doesn't imply that all divisors with degree 0 are divisors of functions. There is one other condition that must be respected for a divisor to be the divisor of a function [5]. That is:

$$\sum_{P \in E(\mathbb{F}_{p^k})} [n_P]P = \mathcal{O} \tag{2.24}$$

A divisor is the divisor of a function if and only if these two conditions are satisfied. Such a divisor D belongs to a subset called $Prin(E)$ which naturally obeys $Prin(E) \subset Div(E)$.

The next concept that will come in useful in the pairing-based implementation is the evaluation of a function f at a divisor D. This can be done as long as the divisor of the function (f) and D have different supports. The evaluation follows the following formula:

$$f(D) = \prod_{P \in E} f(P)^{n_P} \tag{2.25}$$

The final properties described in this subsection are linked to the recursive computation of complicated functions. Let us assume one might want to compute:

$$(f_{r,P}) = r(P) - ([r]P) - (r-1)\mathcal{O} \tag{2.26}$$

where r is a large integer. In this case two properties[8] will come in useful:

$$f_{m+1,P} = f_{m,P}\frac{l_{[m]P,P}}{v_{[m+1]P}}$$
$$f_{2m,P} = f_{m,P}^2\frac{l_{[m]P,[m]P}}{v_{[2m]P}}$$

where $l_{P,Q}$ is the equation of the line passing through P and Q while v_P is the equation of the vertical line passing through P. They come in useful because one may decompose r in its binary form and then reconstruct it using these two formulas.

2.2.4 Pairing

Finally let us explain the concept of pairings and which of its properties are important. A pairing is a bilinear map from two groups $\mathbb{G}_1$ and $\mathbb{G}_2$ to another group $\mathbb{G}_T$. We denote the pairing e by:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T \tag{2.27}$$

A trivial property is that $\forall g \in \mathbb{G}_1$ and $\forall h \in \mathbb{G}_2$:

$$e(g, 0) = e(0, h) = 1 \tag{2.28}$$

but the most important property of the pairing is the bilinearity. That means that $\forall g_1, g_2 \in \mathbb{G}_1$ and $\forall h_1, h_2 \in \mathbb{G}_2$:

$$e(g_1 g_2, h_1) = e(g_1, h_1) \cdot e(g_2, h_1)$$
$$e(g_1, h_1 h_2) = e(g_1, h_1) \cdot e(g_1, h_2)$$

from which we obtain $\forall g \in \mathbb{G}_1$, $\forall h \in \mathbb{G}_2$ and $\forall a, b \in \mathbb{Z}$:

$$e(g^a, h^b) = e(g, h^b)^a = e(g^a, h)^b = e(g, h)^{ab} = e(g^b, h^a) \tag{2.29}$$

In order to achieve this desired properties we used as previously stated a type 3 pairing, which sets $\mathbb{G}_1$ to the cyclic subgroup of order r $E(\mathbb{F}_p)[r]$ and $\mathbb{G}_2$ to the

[8] that are not proved here but here [20] Chapter 5.3

trace zero subgroup of order r. The output group is the multiplicative subgroup $\mathbb{F}_{p^k}$ where k is the embedding degree of the extension.

There are several pairings that have these desired properties, such as the Weil pairing ([20] Chapter 5.1) and the Tate pairing ([20] Chapter 5.2). In this paper we are going to focus on the latter. The **Tate pairing** is defined as follows:

Definition 2.15 (the Tate pairing) *Let $P \in E(\mathbb{F}_p)[r]$ from which we know that there is a function f whose divisor is $(f) = r(P) - r(\mathcal{O})$. Let $Q \in E(\mathbb{F}_{p^k})/E(\mathbb{F}_p)[r]$ be an element of the trace zero subgroup and let D_Q be a degree zero divisor that is equivalent to $(Q) - (\mathcal{O})$ but whose support is disjoint to that of (f). We define the Tate pairing as:*

$$t_r : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{F}_{p^k} \tag{2.30}$$

computed as:

$$t_r(P, Q) = f(D_Q) \tag{2.31}$$

This definition of the pairing *almost respects* the desired properties. The only problem with it is that the outputs lie in what we call **equivalence classes**[9]. In order to normalize these elements and map them all to one unique element in that equivalence class one must perform a final exponentiation on the output of the Tate pairing. This method is called the **reduced Tate pairing** and is defined as follows:

Definition 2.16 (the reduced Tate pairing) *Let P, f, Q, D_Q be defined as in Definition 2.15. The reduced Tate pairing is defined as follows:*

$$T_r : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{F}_{p^k} \tag{2.32}$$

and is computed as:

$$T_r(P, Q) = f_{r,P}(D_Q)^{\frac{p^k-1}{r}} \tag{2.33}$$

The reduced Tate pairing theoretically has all the required properties, but in practice computing it is quite difficult, in the sense that one must first compute the function in Eq. 2.26. This is where the simplifications seen in subsection 2.2.3 will come in useful. Miller ([20] Chapter 5.3) provided a basic algorithm for computing this function. The algorithm goes as follows:

[9] for more details on equivalence classes see [20] p. 70-72

Algorithm 1 Miller's algorithm

Input: $P \in E(\mathbb{F}_p)[r]$, $D_Q \sim (Q) - (\mathcal{O})$ with support disjoint from $(f_{r,P})$ and $r = (r_{n-1}r_{n-2}...r_1r_0)$ with $r_{n-1} = 1$

Output: $f_{r,P}(D_Q) \leftarrow f$

$R \leftarrow P$, $f \leftarrow 1$
for $i = n - 2$ down to 0 **do**
    $R \leftarrow [2]R$
    $f \leftarrow f^2 \cdot \frac{l_{R,R}}{v_{[2]R}}(D_Q)$
    if $r_i = 1$ **then**
        $R \leftarrow R + P$
        $f \leftarrow f \cdot \frac{l_{R,P}}{v_{R+P}}(D_Q)$
    end if
end for
return f

After computing the function f one can compute the pairing by performing the final exponentiation as explained above.

2.2.5 Pairing as homomorphic encryption

So far we have talked about elliptic curves and about the operations we can perform on such curves. We also briefly explained the concept of pairings, their properties and the way they can be computed. Note that the properties of the pairing (such as described in Eq. 2.29) hold if we define the multiplication of two elements in the same group by the point addition on the elliptic curve and the exponentiation by the multiplication by a scalar.

Now it is time to explain how these concepts come together to provide a homomorphic encryption mechanism. The problem of the El Gamal homomorphic encryption scheme lies in the fact that the encryption can not be *perfectly hiding* as long as the ciphertext is produced by a public key encryption mechanism. The PPATS-CCE scheme tackles this issue by generating a perfectly hiding commitment consistent to the vote using the Pedersen commitment:

Definition 2.17 (Pedersen Commitment) *Let $g_1, g_2 \in \mathbb{G}_1$ be two distinct generators of the cyclic subgroup $\mathbb{G}_1$ of order q and let $r \xleftarrow{\$} \mathbb{Z}_q$. One can commit to a value $v \in \mathbb{Z}_q$ by computing:*

$$c = g_1^r g_2^v \tag{2.34}$$

In this case c is called the commitment and r the opening, as anyone who knows r can verify that v is the value that was commited to.

This scheme is indeed perfectly hiding since g_1^r is randomly distributed in $\mathbb{G}_1$ and furthermore it is homomorphic since the product of two commitments $\langle c_1, c_2 \rangle$ is the commitment of the sum of two values $\langle v_1, v_2 \rangle$. Their opening is the sum of the two randomnesses $\langle r_1, r_2 \rangle$.

The problem with this construction is that r must be encrypted in a homomorphic manner as well. This is where pairings will come in useful. In order to encrypt r one will choose a second group $\mathbb{G}_2$ whose generator is h_1 and will compute the opening as $a = h_1^r$. The opening would work if the following verification process holds:

$$e(c/g_2^v, h1) = e(g_1, a) \tag{2.35}$$

In the above equation we use the property that the exponent of the element in $\mathbb{G}_1$ can be passed to the element in $\mathbb{G}_2$ and vice-versa. If on the right-hand side of the equation we switch the exponent r from a to g_1 we would obtain $e(g_1^r, h_1)$ which is the equivalent of the left-hand side of the equation.

At this stage a can be encrypted homomorphically with the El Gamal encryption and the three resulting elements represent the ciphertext. These three elements are the commitment and the two components of the El Gamal encryption.

The last thing that has to be accounted for is to make sure that the commitment contains an opening that is really a vote and not a random element in $\mathbb{G}_1$. This is solved by adding a consistency proof to the encryption ballot. The full encryption process is defined below[10]

Definition 2.18 *Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be the three groups used in a type-3 pairing. Let $g_1 \in \mathbb{G}_1$ be a generator of $\mathbb{G}_1$ and let $h_1, h_2 \in \mathbb{G}_2$ be two generators of $\mathbb{G}_2$. $\mathbb{G}_1$ and $\mathbb{G}_2$ are two cyclic subgroups of prime order q.*

- ***Key generation**: Pick $x \xleftarrow{\$} Z_q$ and compute the public key g_2^x.*
- ***Encryption**: Pick $r, s \xleftarrow{\$} Z_q$ and compute $(c_1, c_2, d, \sigma_{cc}) = (g_1^s, g_2^s g_1^r, h_1^r h_2^v, \sigma_{cc})$ with σ_{cc} the consistency proof defined below.*
- ***Decryption**: Extract the discrete logarithm of $e(c_1^x/c_2, h_1)e(g_1, d)$ in basis $e(g_1, h_2)$*

[10]Note that the groups were switched and that doing so doesn't make any difference since the pairing is bilinear

Definition 2.19 (Consistency proof) *For encryption* $c = (c_1, c_2, d) = (g_1^s, g_2^s g_1^r, h_1^r h_2^v)$*:*

- ***Commitment computation****: Pick* $r', s', v' \xleftarrow{\$} Z_q$ *and compute* $c' = (c'_1, c'_2, d') = (g_1^{s'}, g_2^{s'} g_1^{r'}, h_1^{r'} h_2^{v'})$
- ***Challenge computation****: Let* $\mathcal{H}$ *be a hash function. Compute* $e = \mathcal{H}(c, c', label)$ *where the label contains several parameters that are unique for the election and for the encryption*[11]*.*
- ***Response****:* $f_r = r' + e \cdot r$*,* $f_s = s' + e \cdot s$ *and* $f_v = v' + e \cdot v$

Output $\sigma_{cc} = (e, f_r, f_s, f_v)$

One may notice that this proof contains only the challenge and the response, but not the commitment. The latter is not sent because it can be reconstructed during the **verification process**, which works as follows:

- First of all, the verifier reconstructs the commitment $c' = (c'_1, c'_2, d')$ such that the proofs hold. It does so by computing $c'_1 = g_1^{f_s}/c_1^e$, $c'_2 = g_1^{f_s} g_2^{f_r}/c_2^e$, $d' = h_1^{f_s} h_2^{f_v}/d^e$
- Finally the proof holds if the computation of the hash function is equal to the challenge, that is if $e = \mathcal{H}(c, c', label)$

Given that this type of encryption replaces El Gamal, the proofs seen in the previous subsection must be adjusted accordingly. First of all let's start with the key generation process. During this stage each trustee picks $x_i \xleftarrow{\$} \mathbb{Z}_q$ and computes their share of g_2 as $g_{2,i} = g_1^{x_i}$ only this time the exponentiation is a multiplication by a scalar of a point in $\mathbb{G}_1$. The correspondent ZKP works in the same manner as in the previous section.

Then there is the conjunctive[12] ZKP. Definition 2.9 is modified as follows:

- For each message v_S that is not the real message v_R pick $e_i, f_i \xleftarrow{\$} \mathbb{Z}_q$ then compute $a_i = f_i - e_i \cdot r$ where r is the randomness used in the encryption. Finally compute $A_i = h_1^{a_i} h_2^{(v_S - v_R)e_i}$ and store all the transcripts $\langle A_i, e_i, f_i \rangle$.
- For the real message as well as for the output transcript the same procedure is followed

[11] see the last paragraph of section 2.1

[12] and the disjunctive particular case

The verification of the conjunctive proof is applied only on the commitment d. Indeed in order to verify the correctness of the proof, for each vote one must verify first that the challenge is correct and then that $h_1^{f_i} = A_i \beta^{e_i}$ where $\beta = d/h_2^v$ with v the vote that is being verified. In the case of the real vote the completeness was proved after the Definition 2.9. For the simulated proofs we see that $\beta = h_1^r h_2^{v_R - v_S}$ thus $A_i \beta^{e_i} = h_1^{a_i + re_i} = h_1^{f_i}$ and the proof holds.

Finally, for computing the decryption factors the trustees receive the final $c_1 = \prod_{i \in voters} c_{1,i} = \prod g_1^{r_i} = g_1^{\sum r_i}$ and applies its secret key on it as described in the previous section. The ZKP follows the same pattern as in Definition 2.10.

On the voting procedure side, it stays the same, with only one extra step. We're talking of course about the creation and the verification of one more ZKP, which is the consistency proof.

Chapter 3

Implementation aspects

This chapter will focus on the practical aspects of the implementation of browser-based electronic voting. The subsections of this chapter aim at explaining the design choices that were made and at showing in which way they are more suitable.

3.1 Programming languages and libraries

In practice the computations required in order to make an election are performed on one hand on the server and on the other hand on the client's browser. On the client side I decided to use what appears to be the most suitable engine for performing mathematical computations in the browser, *JavaScript*. One of the key points of JavaScript is that it is provided with all major browsers. On the server side I decided to use the *Django Python Framework v2.0.7* [22] since this framework allows the server to work with *Python language v3.5.3* which is an interpreted language. Although it is slower than a compiled language such as *C* the computation time is still orders of magnitude below the execution time in JavaScript, which is enough to achieve the purpose of this thesis.

One of the major issues with cryptographic computations is that they are performed with integers whose sizes can go up to several thousands of bits. This is why special libraries are used in both Python and JavaScript to deal with these special numbers. In Python the C-coded Python extension module called *gmpy2* [25] was used since it is faster than operations with `ints`[1]. However, the choice is not as easy in JavaScript where several libraries are proposed, such as JSBN [29], Leemon [24] or VJSC [28]. The latter was chosen since it has support for

[1]Although slow, Python3 allows to perform operations with built-in integers of any sizes. However, Python2 doesn't have this option.

fixed-based exponentiations, which allows for much faster computations in the El Gamal encryption [19]. In the rest of this document we may refer to VJSC as *Verificatum*.

3.2 El Gamal Optimizations

This section will focus on the optimizations made on the first voting prototype, especially on the El Gamal encryption.

The operation that takes most of the encryption time is the **modular exponentiation**. In order to perform the El Gamal encryption the *Verificatum* library proposes objects and functions that compute the ciphertext. However, these functions being quite *black box* I decided to compare them with my personal re-implementation of the protocol using the basic-operation functions proposed by the library (multiplication, exponentiation etc). Not surprisingly the encryption time is similar between the two methods. Figure 3.1 shows the average encryption time for n plaintexts where n is plotted on the x-axis.

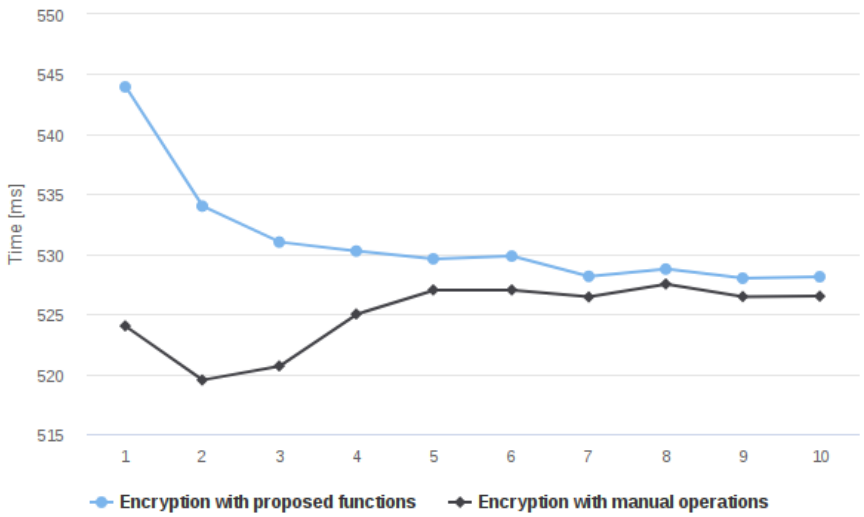


Figure 3.1: Average time per encryption as a function of the number of plaintexts

This result confirms the fact that my implementation is correct and as optimal as the one of the author. However, there is one more optimization that can be done which is specific to this voting procedure. Indeed during the encryption process one must raise several times g and h to different powers. To be more precise, in an election with N candidates where the voter must choose only one candidate,

one computes N ciphertexts (1 exponentiation in g and 1 exponentiation in h per encryption), N disjunctive proofs (1 exponentiation in g and 2 exponentiations in h per proof) and one conjunctive proof (1 exponentiation in h). In total the encryption of ones choice requires $2N$ exponentiations in g and $3N + 1$ exponentiations in h. The power of the *Verificatum* library exploits this actual need to perform **fixed base exponentiations**. There are functions that allow to perform several precomputations in order to make the exponentiations faster. Figure 3.2 shows how the average encryption time in case the fixed-base exponentiation for g and h is used compares to the two (rather equivalent) previous methods. It clearly shows that this method is more advantageous in case we perform more than 2 encryptions in the same base.

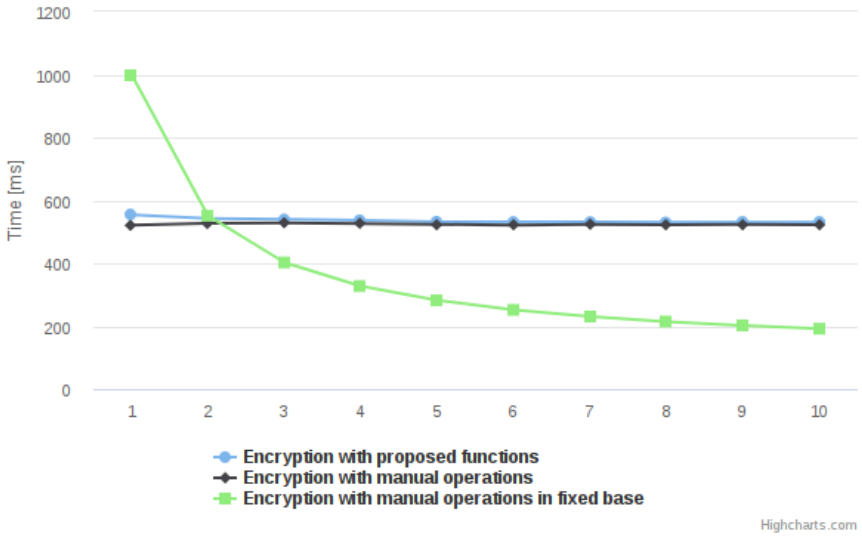


Figure 3.2: Average time per encryption as a function of the number of plaintexts

Another adjustment that was done to the code was performed directly on the *Verificatum* library. In the early stages of the implementation, while understanding how the encryption modules work, several tests were conducted to see how the encryption and decryption times evolve as a function of the number to be encrypted, both for python and JavaScript[2]. In order to obtain these results, a long set of computations needed to be performed. Unfortunately, computation in JavaScript freezes the browser and the user is unable to interact with the page or even switch to other tabs. In order to alleviate this issue I decided to send the computations to **workers**, one for each language. Here is where I realized that workers can not handle random number generators. Indeed, the El Gamal Object performing

[2]the results are at the end of this section

encryption needed to pick a randomness at the beginning of the encryption process, so the fact that the worker wasn't able to compute this step was a problem. This is why I modified the library to allow the encryption function to accept the randomness as an argument, so that it could be computed in the main script and then be simply passed over to the worker who would compute the rest of the encryption.

Another small modification to the library was to add the homomorphic encryption and decryption functions for El Gamal, as they didn't initially exist since the library was intended to be used in mixnet voting.

These modifications are useful because if they were pushed into the library it would allow people to perform homomorphic encryption with the same library, but also because many browser-based implementations of the voting use workers to pre-compute ciphertexts while the person is making their choice.

Concerning the groups in which we're working it is currently advisable to work in groups where the modulus p is 3072-bit long to achieve the desired 128-bit security ([16] p.53). The *Verificatum library* proposes groups with such a modulus. The only problem with this group is that the order q of its generator is almost the same size as the modulus $p = 2q + 1$. This is an issue when performing exponentiations as the randomness is then about 3071-bits long as well so more computation has to be performed. Thus I decided to work in a group whose generator's prime order is only 256-bit long, since this doesn't affect the security level ([16] p.53) and the computations are much faster. The difference between the mean encryption time between groups with an order length of 256 and 3072 is given in Figure 3.3.

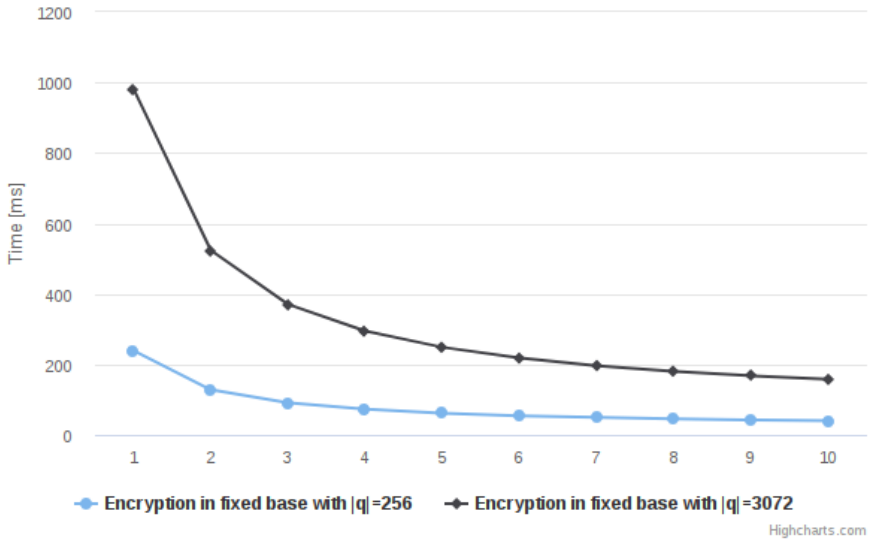


Figure 3.3: Average time per encryption as a function of the number of plaintexts

On the ZKP side some optimizations were done in the amount of data to be sent for each proof. We've seen in the previous chapter that it was interesting to only send the challenge and the response as the commitment can easily be computed from them and the verification can be done on the challenge consistency with the hash function only. This requires more computation on the server side, but this is not an issue since the latter is already very fast. On the positive side less data is sent from the client. In general, for a Schnorr proof the commitment is in $\mathbb{Z}_p$ where p is the group prime modulus while the challenge and the response are in $\mathbb{Z}_q$ where q is the group order. In the case of the group I employed this reduces the size of a Schnorr proof by a factor of $12/14 = 86\%$. In the case of a Chaum-Pedersen ZKP the gain would be even higher $24/26 = 92\%$. This is why all the proofs were modified to account for it.

To give an actual example, in the case of a vote with N candidates there are N 6144-bit ciphertexts (since there are two 3072-bit elements in the El Gamal encryption), N 1024-bit disjunctive proofs (256-bit challenge and 256-bit response for two proofs) and one 512-bit conjunctive proof (in case the voter has to vote for exactly one option there is only a 256-bit challenge and a 256-bit response). In total there are $7168N + 512$ bits per reduced ballot. In comparison, if the commitments were sent as well the disjunctive and conjunctive proofs would add 3072 bits per element in the commitment. This would add up to $10240N + 6656$. The ratio would then be given by the following formula:

$$R = \frac{7168N + 512}{10240N + 6656} \tag{3.1}$$

which in case $N = 5$ for instance gives $R = 62.83\%$.

Some of the hypothesis that have been made so far can also be confirmed experimentally. First of all we stated that Python is faster than JavaScript. Figure 3.4 shows the encryption and decryption execution times for both Python and JavaScript and confirm this hypothesis.

Second of all we supposed that the homomorphic encryption is usually done with messages whose sizes are much smaller than the size of the group order. In such a case the encryption time shouldn't differ from one message to another. We see in Figure 3.5 that this is the case.

Finally the decryption is done by brute forcing all the possible values starting from 0 and moving on. This operation should increase linearly in the size of the message and we see it is correct in Figure 3.6

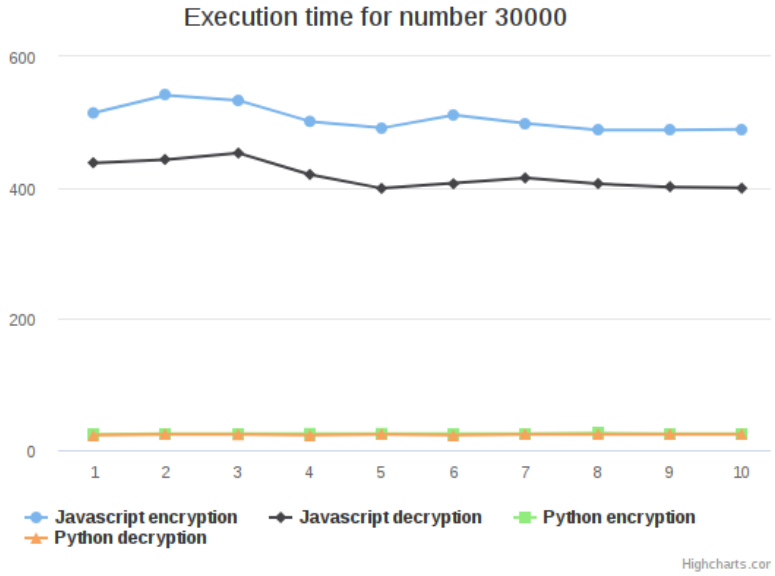


Figure 3.4: Execution time for several samples

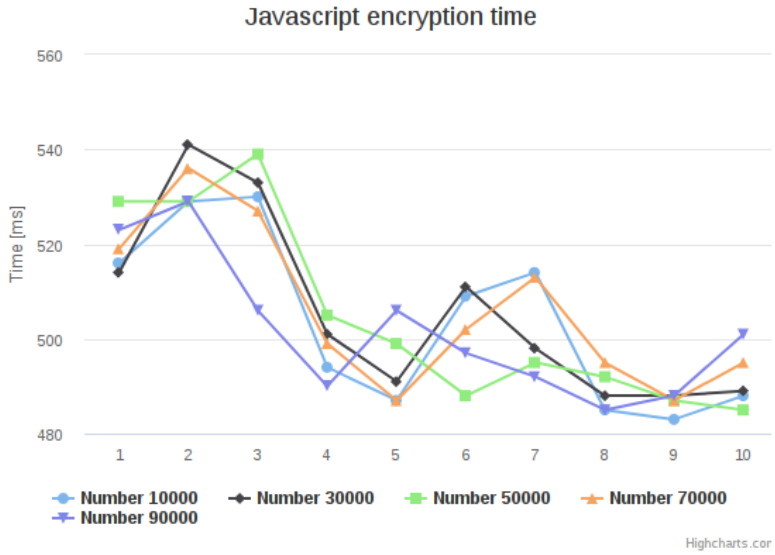


Figure 3.5: JavaScript encryption time for 10 samples

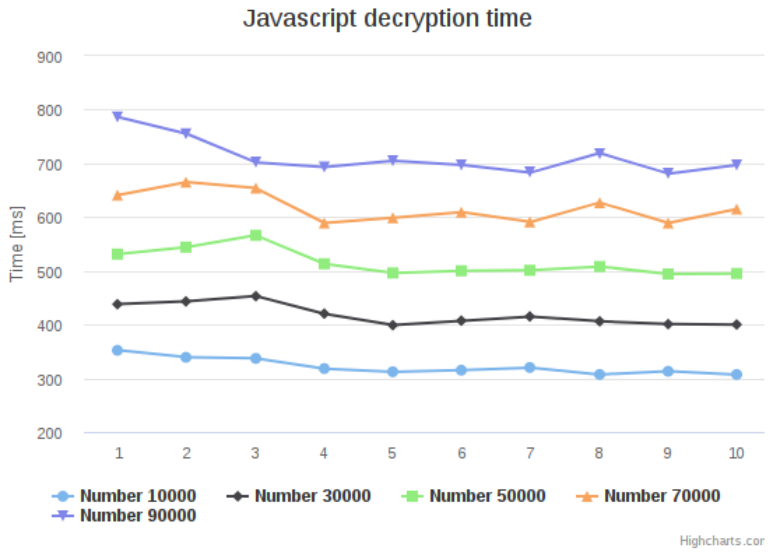


Figure 3.6: JavaScript decryption time for 10 samples

3.3 Curve selection

From this section on we will be focusing on the choices and optimizations regarding the elliptic curves and the pairing implementation. The first thing to be defined is of course the elliptic curve on which we are going to work.

In the literature a family of curves on which a lot of optimization was done is the **Barreto-Naehring (DB)** family of curves ([17] s.39). The curves in this family are of the form:

$$E : y^2 = x^3 + b \tag{3.2}$$

with $b \in \mathbb{F}_p$ with p the prime modulus of the group over which the curve is defined. Depending on a security parameter u, the group modulus as well as the group order r and the Frobenius trace t are defined as follows:

$$\begin{aligned} p(u) &= 36u^4 + 36u^3 + 24u^2 + 6u + 1 \\ r(u) &= 36u^4 + 36u^3 + 18u^2 + 6u + 1 \\ t(u) &= 6u^2 + 1 \end{aligned}$$

which according to equation 2.17 means that $\#E(\mathbb{F}_p) = r$ which means that all the points on this curve are in the r-torsion. Also the embedding degree of the curve is $k = 12$, thus the example in section 2.2.2.

When fixing the security parameter above the equations do not hold for any b. Some of the pairing-friendly curves for which they hold are shown in [12] where we have either that $b = c^4 + d^6$ or $b = c^6 + 4d^4$ for any $c, d \in \mathbb{Z}_p^\times$.

Taking account of these constraints, in this paper I chose the following parameters which are inspired from [15]:

$$\begin{aligned} c &= 1 \\ d &= 1 \\ E &\equiv y^2 = x^3 + 2 \\ x &= -(2^{62} + 2^{55} + 1) \\ p = p(x) &= (2523648240000001ba344d80000000086121000000000013a700000000000013)_{16} \\ r = r(x) &= (2523648240000001ba344d8000000007ff9f800000000010a10000000000000d)_{16} \\ |p| &= |r| = 254 \end{aligned}$$

In comparison to the field over which El Gamal encryption is defined, elliptic curves can achieve the same 128-bit security level over a 256-bit field.

3.4 The sextic twist

Operations in extension fields are one of the bottlenecks of pairing computations. The higher the extension degree the more time it will take for multiplication and exponentiation operations to be performed, since their computation time increases quadratically. We saw in section 2.2.2 that one way to accelerate these computations was to make use of towered extensions which allow via specific optimizations to reduce the computation time to half. But this time is still enormous compared to the time taken by the same operation in $\mathbb{F}_p$.

Fortunately enough, elliptic curves come with the advantage that one may switch to another curve, perform the operations there and then switch back to the original curve. This would yield the same result as if the operations were computed over the original curve. This equivalence between points over the two curves is called an isomorphism. The main advantage of this isomorphism is that the second curve, called the **twist**, can be defined over a different field. In our case this will be an extension field with a lower extension degree. Thus if our curve E is defined over $\mathbb{F}_{p^k}$ and we perform a degree d twist on it, the new curve E' will be defined over $\mathbb{F}_{p^{k/d}}$. One may notice that the degree of the twist must be a factor of k.

In ([20] p.63-64) we learn that there are only a few possible twists: $d \in \{2, 3, 4, 6\}$ and it is thus desired that k has any of this factors in its decomposition. One of

the reasons why the BN-curve is optimal is that it respects all the conditions to perform a **sextic twist** (e.g. twist with $d = 6$) over it. The definition of the sextic twist is the following

Definition 3.1 *One can perform a sextic twist over an elliptic curve* $E/\mathbb{F}_{p^k} : y^2 = x^3 + ax + b$ *if* $6 \mid k$ *and if* $a = 0$. *In that case the twisted curve has the equation* $E'/\mathbb{F}_{p^{k/6}} : y^2 = x^3 + b\omega^6$ *where* $\omega^6 \in \mathbb{F}_{p^{k/6}}$, $\omega^3 \in \mathbb{F}_{p^{k/3}}$, $\omega^2 \in \mathbb{F}_{p^{k/2}}$. *We define* Ψ *as the function mapping elements from* $E'(\mathbb{F}_{p^{k/6}})$ *to* $E(\mathbb{F}_{p^k})$ *and* Ψ^{-1} *as the function mapping elements from* $E(\mathbb{F}_{p^k})$ *to* $E'(\mathbb{F}_{p^{k/6}})$

We understand immediately that for our curve which has embedding degree $k = 12$ the sextic twist maps to a a curve defined over $\mathbb{F}_{p^2}$. To continue the definition of the curve that I used, in example 2.2 a parameter ξ was used. This parameter in the case of our curve (see [12] p.8) is $\xi = c^2 + d^3 i$ or $\xi = c^3 + 2d^2 i$. In this case, the twisted curve is defined as:

$$E'/\mathbb{F}_{p^2} : y^2 = x^3 + \overline{\xi} \tag{3.3}$$

where $\overline{\xi}$ is the conjugate of ξ. In this case the isomorphism function Ψ is defined as:

$$\Psi : (x, y) \to (\mu^2 x, \mu^3 y) \tag{3.4}$$

where μ is a root of $(Y^2 - \xi)$ in $\mathbb{F}_{p^{12}}$.

To complete the definition let us add the formulas for computing the generators of each of the two groups of the pairing:

- The generator of $\mathbb{G}_1$ is given by $g_1 = (-d^2, c^2)$ or $g_1 = (-c^2, 2d^2)$
- The generator of $\mathbb{G}_2$ is given by $g_2 = [h]g_2'$ where $h = 2p - r$ allows to make sure the element is in the r-torsion and where we have wither $g_2' = (-di, c)$ or $g_2' = (-c, d(1 - i))$

In our case, since $c, d = 1$ we will have:

$$\begin{aligned} \xi &= 1 + i \\ E' : y^2 &= x^3 + (1 - i) \\ g_1 &= (-1, 1) \\ g_2 &= [h](-i, 1) \end{aligned}$$

Working on the twisted curves reduces the cost of a multiplication to 4 times the cost of the multiplication in $\mathbb{F}_p$, while if we stayed on the original curve defined

over $\mathbb{F}_{p^{12}}$ we would have needed to perform in the best case scenario the equivalent of 54 multiplications in $\mathbb{F}_p$ (see [20] p.102). Furthermore, the client never needs to work in $\mathbb{F}_{p^{12}}$ since the only moment when we need to switch back to the original curve is when computing the pairing, and this is done on the server, which is much faster than the browser.

3.5 Jacobian coordinates

So far we've talked about points on elliptic curves as (x, y) coordinates. However, in practice there are many other ways of representing such points. In this section we're going to talk about one of these representations, called the **Jacobian coordinates**.

The idea behind this representation is to pass from a 2-coordinate system (x, y), called **Affine representation** to a 3-coordinate system (X, Y, Z). In Jacobian coordinates we do this by fixing Z and computing X and Y from the formulas:

$$x = \frac{X}{Z^2} \qquad y = \frac{Y}{Z^3} \tag{3.5}$$

The point at infinity is defined as (0,1,0) but mainly any point that has its Z-component equal to 0 represents the point at infinity. Since this is a representation of a point that has two constraints (x and y) and three degrees of freedom (X, Y and Z) it is required to fix one of the three parameters and compute the two others afterwards via the formulas above. This degree of freedom means that there is an infinity of Jacobian representations for the same affine point. Thus one of the main problems with this representation is that it is difficult to compare the equality between two points without switching back to affine coordinates.

But this is one small drawback compared to the gain of using such coordinates. The reason why I decided to switch to these coordinates is that there are efficient algorithms that compute the addition and the doubling of points in Jacobian coordinates on elliptic curves. Namely the most important optimization is that **the inversion can be removed**.

Throughout my implementation I used one verification and three algorithms. Their exact use is to be explained in the next section.

The goal of the verification is to make sure that the point is on the curve. Indeed, not any triple of points in $\mathbb{F}_p$ (or $\mathbb{F}_{p^2}$) lies on the curve so a verification has to be made. If we worked with affine coordinates it would suffice to replace

the x and y coordinates in the formula and verify the equality. With Jacobian coordinates a little modification has to be made. If we replace x and y in definition of the curve by the equations in 3.5 and remove the denominators we would obtain:

$$E : Y^2 = X^3 + b \cdot Z^6 \tag{3.6}$$

To perform the verification it suffices to replace X, Y and Z in this equation. Note that no inversion needs to be performed.

The first one of the algorithms that performs the addition between two points in Jacobian coordinates ([22] add-1998-cmo-2) is Algorithm 2:

Algorithm 2 Jacobian-with-Jacobian Addition

Input: $P_1 = (X1, Y1, Z1)$ and $P_2 = (X2, Y2, Z2)$
Output: $P_3 = P_1 + P_2 = (X3, Y3, Z3)$

1: $\mathrm{Z1Z1} = \mathrm{Z1}^2$
2: $\mathrm{Z2Z2} = \mathrm{Z2}^2$
3: $\mathrm{U1} = \mathrm{X1} \cdot \mathrm{Z2Z2} \bmod \mathrm{p}$
4: $\mathrm{U2} = \mathrm{X2} \cdot \mathrm{Z1Z1} \bmod \mathrm{p}$
5: $\mathrm{S1} = \mathrm{Y1} \cdot (\mathrm{Z2} \cdot \mathrm{Z2Z2} \bmod \mathrm{p})$
6: $\mathrm{S2} = \mathrm{Y2} \cdot (\mathrm{Z1} \cdot \mathrm{Z1Z1} \bmod \mathrm{p})$
7: $\mathrm{H} = \mathrm{U2} - \mathrm{U1}$
8: $\mathrm{HH} = \mathrm{H}^2$
9: $\mathrm{HHH} = (\mathrm{H} \cdot \mathrm{HH} \bmod \mathrm{p})$
10: $\mathrm{r} = \mathrm{S2} - \mathrm{S1}$
11: $\mathrm{V} = (\mathrm{U1} \cdot \mathrm{HH} \bmod \mathrm{p})$
12: $\mathrm{X3} = (\mathrm{r}^2 \bmod \mathrm{p}) - \mathrm{HHH} - 2 \cdot V \bmod \mathrm{p}$
13: $\mathrm{Y3} = (\mathrm{r} \cdot (\mathrm{V} - \mathrm{X3}) \bmod \mathrm{p}) - \mathrm{S1} \cdot \mathrm{HHH} \bmod \mathrm{p}$
14: $\mathrm{Z3} = \mathrm{Z1} \cdot \mathrm{Z2} \cdot \mathrm{H} \bmod \mathrm{p}$
15: **return** $(X3, Y3, Z3)$

Although this algorithm is already very fast, there is an even faster algorithm for the point addition. However for this second algorithm the second point must be in its Affine form[3], which can be achieved as we'll see in the next section. This algorithm (Algorithm 3) comes from the same website ([22] madd-2007-bl)

[3] A point in its Jacobian form where $Z = 1$ can be considered as an Affine representation

Algorithm 3 Jacobian-with-Affine Addition

Input: $P_1 = (X1, Y1, Z1)$ and $P_2 = (X2, Y2, 1)$
Output: $P_3 = P_1 + P_2 = (X3, Y3, Z3)$

1: $\mathrm{Z1Z1} = \mathrm{Z1}^2$
2: $\mathrm{U2} = \mathrm{X2} \cdot \mathrm{Z1Z1} \bmod \mathrm{p}$
3: $\mathrm{S2} = \mathrm{Y2} \cdot \mathrm{Z1} \cdot \mathrm{Z1Z1} \bmod \mathrm{p}$
4: $\mathrm{H} = \mathrm{U2} - \mathrm{X1}$
5: $\mathrm{HH} = \mathrm{H}^2$
6: $\mathrm{I} = 4 \cdot \mathrm{HH}$
7: $\mathrm{J} = \mathrm{H} \cdot \mathrm{I} \bmod \mathrm{p}$
8: $\mathrm{r} = 2 \cdot (\mathrm{S2} - \mathrm{Y1})$
9: $\mathrm{V} = \mathrm{X1} \cdot \mathrm{I}$
10: $\mathrm{X3} = \mathrm{r}^2 - \mathrm{J} - 2 \cdot \mathrm{V} \bmod \mathrm{p}$
11: $\mathrm{Y3} = (\mathrm{r} \cdot (\mathrm{V} - \mathrm{X3}) \bmod \mathrm{p}) - 2 \cdot \mathrm{Y1} \cdot \mathrm{V} \bmod \mathrm{p}$
12: $\mathrm{Z3} = (\mathrm{Z1} + \mathrm{H})^2 - \mathrm{Z1Z1} - \mathrm{HH} \bmod \mathrm{p}$
13: **return** $(X3, Y3, Z3)$

Finally the point doubling algorithm (Algorithm 4), which is even faster than both the additions above, is taken from ([22] dbl-2009-l):

Algorithm 4 Jacobian Doubling

Input: $P_1 = (X1, Y1, Z1)$
Output: $P_3 = [2]P_1 = (X3, Y3, Z3)$

1: $\mathrm{A} = \mathrm{X1}^2$
2: $\mathrm{B} = \mathrm{Y1}^2$
3: $\mathrm{C} = \mathrm{B}^2 \bmod \mathrm{p}$
4: $\mathrm{D} = 2 \cdot (((\mathrm{X1} + \mathrm{B})^2 \bmod \mathrm{p}) - \mathrm{A} - \mathrm{C})$
5: $\mathrm{E} = 3 \cdot \mathrm{A}$
6: $\mathrm{F} = \mathrm{E}^2 \bmod \mathrm{p}$
7: $\mathrm{X3} = \mathrm{F} - 2 \cdot \mathrm{D} \bmod \mathrm{p}$
8: $\mathrm{Y3} = (\mathrm{E} \cdot (\mathrm{D} - \mathrm{X3}) \bmod \mathrm{p}) - 8 \cdot \mathrm{C} \bmod \mathrm{p}$
9: $\mathrm{Z3} = 2 \cdot \mathrm{Y1} \cdot \mathrm{Z1} \bmod \mathrm{p}$
10: **return** $(X3, Y3, Z3)$

If we consider that a squaring is equivalent to a multiplication then the *Jacobian-with-Jacobian Adding* algorithm costs 16M[4], the *Jacobian-with-Affine Adding* algorithm costs 11M and the *Doubling* algorithm costs 7M. Considering that one

[4]where M represents one multiplication in the field where the operation is performed

inversion is worth 100M we understand why the Jacobian coordinates are more appealing.

One may have noticed that in the algorithms above the modular reduction is not always applied. This is another optimization brought to these algorithms. Although the reduction operation doesn't cost very much, if we managed to perform it as few times as possible it would accelerate the algorithms above. However we must make sure that the sizes of the intermediate variables in the algorithm do not get very large, since operations over larger integers also take more time. Thus, a compromise must be found. As a finger rule I tried not to exceed[5] the double of the modulus length. As a result, the main execution time of the *Jacobian-with-Affine Addition* passed from 0.0524ms with modular reductions everywhere to 0.0429ms in the optimized form.

Finally, during this stage of the implementation I found that there was a slight issue with the *Verificatum* library. This problem emerged when, during the computation of several consecutive point additions and doublings I observed that the time per operation started to increase until it froze and stopped working. While debugging the code, I observed that one of the variables inside the `LargeInteger` object of the library, namely the `length` variable was increasing very fast, although the actual number it represented was much smaller. The role of this variable was to provide the number of 28-bit pieces that compose the large integer. In my case this should have been 9 or 10 (since I work in a group modulus of size 256), but in some cases it was reaching several millions! This is why I modified the library to update this variable to the actual number after any of its built-in functions was called. After this modifications the code started running as it was supposed to.

3.6 Pre-computed tables and NAF

In section 2.2.1 we briefly talked about the ECDL problem and about the fact that there exists a polynomial-time algorithm that computes the multiplication of a point by a scalar. This **double-and-add** algorithm is described in (Algorithm 5). This algorithm is polynomial-time since increasing the size (in bits) of k makes the execution time increase linearly. Since k is usually a random number, in this algorithm we would have exactly n doubling operations and on average $\frac{n}{2}$ point additions. If we decide to have P in affine coordinates then this operation would take[6] $256 \cdot 7M = 1792$M from the doublings and $128 \cdot 11M = 1408$M from the

[5] in terms of variable size

[6] for a 256-bit scalar

additions, for a total of 3200M on average.

Algorithm 5 Double-and-add algorithm

Input: $P \in E(\mathbb{F}_p)$ and $k = (k_{n-1}k_{n-2}...k_1k_0)$ with $k_{n-1} = 1$
Output: $R \leftarrow [k]P$

$R \leftarrow P$
for $i = n - 2$ down to 0 **do**
    $R \leftarrow [2]R$
    if $r_i = 1$ **then**
        $R \leftarrow R + P$
    end if
end for
return R

There are ways to decrease the number of multiplications. The first optimization is to represent k in its **Non-adjacent form (NAF)**. One can compute the NAF of any number by following Algorithm 6:

Algorithm 6 Non-adjacent form

Input: $k \in \mathbb{Z}_p$
Output: k in NAF

$i \leftarrow 0$
while $k \geq 1$ **do**
    if k is odd **then**
        $k_i \leftarrow 2 - (k \bmod 4)$
        $k \leftarrow k - k_i$
    else
        $k_1 \leftarrow 0$
    end if
    $k \leftarrow k/2$
    $i \leftarrow i + 1$
end while
return $k_{n-1}, ..., k_1, k_0$

Thus we have that $k_i \in \{0, \pm 1\}$. The reason why the NAF is interesting is that there will never be 2 adjacent non-zero elements in the representation of any number. That means that on average there will be two thirds of zeros and one third of non-zeros. The double and add algorithm can be modified to account for this change. The only difference is that there would be an extra condition

on non-zero elements: If $k_i = 1$ then $R \leftarrow R + P$, If $k_i = -1$ then $R \leftarrow R - P$. This is not a problem since point subtraction costs the same as point addition. Indeed we have that $P - Q = P + (-Q)$ and $-Q$ is the same point as Q with a negative Y-component, since as we saw in the previous section, the negative of a point is its symmetrical with respect to the x-axis. Since only one third of the operations are point addition or subtraction, the total cost of the additions becomes: $85 \cdot 11M = 935$M. Since the number of doublings doesn't change we will have a total of 2727M which is less than what we obtained with the binary representation.

We see that most of the computation time is taken by the doubling operation. Fortunately, this operation can actually be avoided, if we make a time-memory trade-off. Since Python is much faster than JavaScript the idea that optimizes the scalar multiplication the most is to precompute all the powers of 2 of a point P in Python and then send a **precomputed table** with all these values to the client. It follows that the client would only have to assemble the elements in this table to achieve to desired $[k]P$. Combining this method with the NAF, the client only needs to perform, on average, 87 point additions, which sums up to only 957M. This is about 70% faster than the initial method in algorithm 5.

Concerning this last method there are several things to be noted:

- The number of multiplications (957M) assumed that all the point additions are performed between a point in Jacobian coordinates and a point in Affine coordinates. This can be the case if the pre-computed values are in Affine coordinates. It means that the server has to normalize each of the 256 points.
- The size of the pre-computed tables can increase very fast. Indeed during the computation of the pairing there are 4 generators which require pre-computations. Two of them are points over $E(\mathbb{F}_p)$ and two over $E(\mathbb{F}_{p^2})$. Each point is in Affine coordinates, which means we have 512 bits per point for the first two generators respectively 1024 bits per point for the last two. Assuming there are 256 points per table the total number of bits is given by $256 \cdot (2 \cdot (512) + 2 \cdot (1024)) = 768$Kb $= 96$KB. Although this number is not enormous it is not negligible either. The time-memory trade-off becomes (96KB, 957M) for (0KB, 3200M).

This aspects can be observed in practice as well. When performing the three operations 1000 times each we observe a mean execution time of 18.1ms for the basic scalar multiplication, 15.7ms when replacing the binary representation with the NAF and 4.1ms when using the pre-computed tables. Once more, this confirms the power of precomputed tables. However we would have expected to have a

smaller ratio between the computation time without the precomputed tables and the one with. This is due to the fact that although there are supposed to be only 7 multiplications in the doubling operation, in this implementation there are 5 multiplications with small integers (less than 10). These multiplications are not optimized, that is the same functions as for the multiplication between two large integers are used. This is why in the case where point doubling is performed (no precomputed tables) the measured time is greater.

Finally, during the implementation I found out that the computation of the NAF of a integer, if not implemented properly may take quite some time. Since the operations required to perform this conversion are quite basic I decided to modify Algorithm 6 and perform bit-wise operations. Thus, the following modifications were performed:

- Instead of comparing two `LargeInteger` objects, that is comparing the updating k in the while loop condition with 1 I verify the sign attribute of the object. if the number is greater than or equal to 1, then the sign must be 1.
- Instead of computing (k mod 2) to see if the number is even or odd, I am only looking at the least significant bit.
- If the number is odd, instead of computing (k mod 4) in step 4 of the algorithm I slice the last two bits and depending of the second least significant bit I push a 1 or a -1 to the NAF representation.
- Again if the number is odd and it is required to subtract 1, I simply don't do it since the next step will take care of it. However if I need to add 1 then I perform a normal addition using the built-in function.
- Once the NAF was updated instead of dividing by two, I shift right the number binary representation of 1 bit. This is one of the most important accelerations since divisions are never easy.

The difference in the actual implementation is that the optimized implementation takes about 0.14ms while the naive implementation takes almost a full millisecond.

3.7 Personalized optimizations

In this section I'll explain how I simplified some functions in order to make them faster.

The first implementation I made of the point addition and point doubling were very general purpose, almost library-like. They would take as parameters the two points in Jacobian coordinates, the value of the parameter b of the curve and eventually an irreducible polynomial in case the points were defined over an extension field. Then inside each functions I would perform several verifications, such as verifying if each point was on the curve and whether the two points were the same, in which case a doubling should have been performed.

First of all I decided to remove the verification of the point belonging to the curve at each addition since this would add a useless overhead in the case of the scalar multiplication. I still keep such verifications in the code but they are only present in sesitive areas, like at the end of the encryption when we obtain a point which represents a ciphertext for instance. If the point is not on the curve then there was a problem somewhere so there is no purpose in continuing with the ZKPs. Furthermore, since most of the operations are scalar multiplication, I know for sure that all the points are on the curve and that the code will never try to add 2 equivalent points. Concerning the other individual additions, which do not occur very often (5 per candidate encryption and ZKP) it is impossible for the two points not to be on the curve since they are the output of the scalar multiplication function, and the probability of them being equivalent is negligible. However, had the case arrived it would be caught by the verification which is done at the end. Finally, strange scenarios could occur if someone tempers with the code and with the data used by it. This is the kind of malicious behaviour that can not be avoided. However, the mathematically theoretical safety behind the implementation guarantees that the adversary has negligible power to cheat in such a scenario.

The second optimization was performed after realizing that the point addition and doubling operations are only performed over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$. Since there are only two possible scenarios and since the extension field is none other than the complex field (the irreducible polynomial being $i^2 + 1 = 0$) it is easy to split the general-purpose function in two optimized specific functions.

The differences between the optimized functions and the first implementation in $\mathbb{F}_p$ are the following:

- For the **point addition** the initial function takes 0.109ms while the opti-

mized function that skips the extra verifications and that is dedicated to computations in $\mathbb{F}_p$ takes 0.045ms

- For the **point doubling** the initial function takes 0.075ms while the optimized function takes 0.047ms

3.8 Pairing function computation

So far we have discussed the optimizations made in JavaScript. This section aims at describing some of the optimizations that were made on the server's side to increase the speed for computing the pairing.

3.8.1 Jacobian coordinates

We've seen in the previous section that the computation of the pairing is performed via Miller's algorithm. In this Algorithm 1 we need to compute the line passing through two points. This line is of the form $y - \lambda x - \nu = 0$. Once this line is computed one needs to evaluate it at a certain divisor $D_Q \sim (Q) - (\mathcal{O})$. This value is then used to update the output of Miller's algorithm f.

As we've seen in the formulas for computing λ and ν they require inversions, which are very time-consuming. In this chapter we've seen that we can avoid inversions if we work with Jacobian coordinates. Thus, the equation of a line tangent to a point $P_1 = (X_1, Y_1, Z_1)$ becomes:

$$2Y_P Z_P^3 y - 3X_P^2 Z_P^2 x - (2Y_P^2 - 3X_P^3) = 0 \tag{3.7}$$

while the equation of a line passing through two points $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$ becomes:

$$(Z_2 Z_1^3 X2 - Z_1 Z_2^3 X1)y - (Z_1^3 Y_2 - Z_2^3 Y_1)x - (Z_2 X_2 Y_1 - Z_1 X_1 Y_2) = 0 \tag{3.8}$$

3.8.2 Irrelevant factors

The first thing to talk about in this section is based on a result by Barreto *et al.* ([3] Lemma 5 Corr.2). They observed that for an embedding degree k let e be any factor of k, then we have that $q^e - 1 | (q^k - 1)/r$. This means that any factor in $\mathbb{F}_{p^e}$ is sent to 1 via the final exponentiation in Miller's algorithm.

Equations 3.7 and 3.8 above allow, after replacing (x, y) by the coordinates of the points composing divisor D_Q to update the evaluation of function f. But there

is a small problem with these formulas. As previously stated, there is an infinity of Jacobian representations for a point P, thus there is an infinity of representations for the line function as well. In conclusion depending on the Jacobian representation of point P the output is going to be different, up to a multiplying factor. However, this issue is tackled by the presence of the final exponentiation which, as seen in the previous paragraph, sends this constant to 1 thus reducing any representation to a unique value.

Furthermore the divisor (D_Q) in the equation $f_{r,P}(D_Q)^{\frac{q^k-1}{r}}$ can be replaced by (Q) since the last exponentiation maps the equation to the same element, no matter the choice of the divisor. Thus it is better to choose a one-point representation since there is less computation to be performed.

The last optimization comes from the observation that if, for $k = 2l$, we take (at the very least) $x_Q \in \mathbb{F}_{p^l}$ then $y_Q \in \mathbb{F}_{p^{\|}}/\mathbb{F}_{p^l}$. In virtue of the property proposed at the beginning of this subsection, the equation of the vertical line (which appears in the denominator of the function update statement in Miller's algorithm) will be reduced to 1 as well via the final exponentiation. This is the case because $v_R(Q) : x - x_R = 0$ where $x_R \in \mathbb{F}_p$ which implies $(x - x_R) \in \mathbb{F}_{p^l}$.

Algorithm 7 represents the updated Miller's algorithm.

Algorithm 7 Miller's updated algorithm

Input: $P \in E(\mathbb{F}_p)[r]$ and $r = (r_{n-1}r_{n-2}...r_1r_0)$ with $r_{n-1} = 1$
Output: $f_{r,P}(Q) \leftarrow f$

$R \leftarrow P$, $f \leftarrow 1$
for $i = n - 2$ down to 0 **do**
    $R \leftarrow [2]R$
    $f \leftarrow f^2 \cdot l_{R,R}(Q)$
    if $r_i = 1$ **then**
        $R \leftarrow R + P$
        $f \leftarrow f \cdot l_{R,P}(Q)$
    end if
end for
return f

3.8.3 Operations in $\mathbb{F}_{p^{12}}$

Although there are not any operations in $\mathbb{F}_{p^{12}}$ performed in JavaScript, the computation of the pairing on the server's side requires operations such as multiplication,

squaring and inverse over this field. I used as an example the algorithms from [10].

However, at the same time I also compared my implementation of these functions with the one here ([21] /mathTools/otosEC.py) and I arrived at two conclusions:

First of all the multiplication function `tmulFp12` and squaring function `tsqrtFp12` are faster than the algorithms in the paper, but they use pre-computed values. The two mean execution times are $42\mu s$ and $27\mu s$ compared to $83\mu s$ and $77\mu s$ for the multiplication and squaring algorithms in the paper.

The second thing concerns the inversion function. In order to implement the inversion over $\mathbb{F}_{p^{12}}$ I needed to pass through the inversion in $\mathbb{F}_{p^6}$ (among other functions). This is where something odd struck me while reading the Algorithm 8 below:

Algorithm 8 Inverse in $\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[v]/(v^3 - \xi)$

Input: $A = a_0 + a_1 v + a_2 v^2 \in \mathbb{F}_{p^6}$
Output: $C = c_0 + c_1 v + c_2 v^2 = A^{-1} \in \mathbb{F}_{p^6}$

1: $t_0 \leftarrow a_0^2$
2: $t_1 \leftarrow a_1^2$
3: $t_2 \leftarrow a_2^2$
4: $t_3 \leftarrow a_0 \cdot a_1$
5: $t_4 \leftarrow a_0 \cdot a_2$
6: $t_5 \leftarrow a_2 \cdot a_3$
7: $c_0 \leftarrow t_0 - \xi \cdot t_5$
8: $c_1 \leftarrow \xi \cdot t_2 - t_3$
9: $c_2 \leftarrow t_1 \cdot t_4$
10: $t_6 \leftarrow a_0 \cdot c_0$
11: $t_6 \leftarrow t_6 + \xi \cdot a_2 \cdot c_1$
12: $t_6 \leftarrow t_6 + \xi \cdot a_1 \cdot c_2$
13: $t_6 \leftarrow t_6^{-1}$
14: $c_0 \leftarrow c_0 \cdot t_6$
15: $c_1 \leftarrow c_1 \cdot t_6$
16: $c_2 \leftarrow c_2 \cdot t_6$
17: **return** $C = c_0 + c_1 v + c_2 v^2$

It was surprising to see that there was a parameter, namely a_3 at line 6, which wasn't defined anywhere. Thus I looked up the sources of this algorithm and found out it came from a source code found at this webpage: `https://cryptojedi.org/crypto/#dclxvi`. I downloaded the code, localized the function and reversed it to find what the actual algorithm should look like. Indeed

there was an error at line 6, but there was another one at line 9 as well. The first difference is that the a_3 should have been a a_1. The second difference is that instead of multiplying t_1 and t_4 they should rather be subtracted, such that the line becomes $c_2 \leftarrow t_1 - t_4$. The changes were implemented, the algorithm was tested and it worked with these modifications.

3.8.4 The final exponentiation

The main bottleneck of the pairing computation becomes the final exponentiation. Indeed, in our case where $k = 12$ the number of bits of the exponent becomes very large (about 2816-bits long). Such an exponentiation is rather long, thus faster methods were proposed in [8]. The main idea behind this optimization is to write the exponent as:

$$(p^{12} - 1)/r = (p^6 - 1) \cdot (p^2 + 1) \cdot [(p^4 - p^2 + 1)/r] \tag{3.9}$$

which simplifies the exponent due to the ease of computation of the Frobenius operator:

Definition 3.2 (Frobenius operator) *For $x \in \mathbb{F}_{p^d}$ One defines the Frobenius operator as:*

$$Frob(x) = x^p \tag{3.10}$$

where p is the group modulus

The first parenthesis can be computed via 6 consecutive Frobenius maps and an inversion while the second parenthesis can be computed via 2 Frobenius maps (which have already been computed for the first parenthesis). The last parenthesis still remains tough to perform though. It would be interesting to re-write it under a different form:

$$(p^4 - p^2 + 1)/r = \lambda_3 p^3 + \lambda_2 p^2 + \lambda_1 p + \lambda_0 \tag{3.11}$$

with:

$$\begin{aligned}
\lambda_3(x) &= 1 \\
\lambda_2(x) &= 6x^2 + 1 \\
\lambda_1(x) &= -36x^3 - 18x^2 - 12x + 1 \\
\lambda_0(x) &= -36x^3 - 30x^2 - 18x - 2
\end{aligned}$$

These coefficients can be computed in a non-redundant manner by following the steps in ([8] p.5-6).

Indeed, these optimizations reduce very much the execution time. If we compute the final exponentiation via the naive way it would take about 76ms while with the optimized method it only takes 4ms.

3.9 Possible improvements

Looking back at the current implementation I realize that if I had to start over with the implementation there would be a series of changes that I would make to it.

The first thing I would change would be the base in which the encrypted data was stored and sent. For the moment everything is in hexadecimal mainly since the Verificatum library has a built-in function that converts large integers to hexadecimal strings. The advantage of working with strings in base64 instead is that they would take 33% less space on memory and also would reduce the data to be sent and received by the client by the same percentage.

The second thing it would be more interesting to spend some time on is the optimization of the ZKPs for elliptic curves. In section 3.2 we've seen that the size of ZKPs can be reduced if we only send the challenge and the response. Unfortunately this is not as easy to do with elliptic curves, because of the Jacobian representation. When re-computing the commitment from the challenge and the response the commitment, although equivalent to the real commitment, will almost certainly have another form, that will depend on the algorithms used to compute it. Thus, if the 2 commitments are not exactly the same, the hash function output will not be the same either, so the proof won't hold. One solution would be to require both commitments to have the same form: the affine form with $Z = 1$ but this would require an inversion on the client's side, which would largely increase the execution time. Another solution would be for the client to ask the server to perform the inversion. Even another solution would be for the client to only send two out of the three coordinates of the commitment (X and Y for instance) and let the server compute the third one. This would slightly decrease the size but it wouldn't solve the problem entirely.

The third aspect that should be looked into is the optimization of the multiplication by small scalars. As seen in section 3.6 for the moment the same functions used to multiply two large integers are used if we want to multiply a large integer

by a small scalar. It would be interesting to look further into optimizations of such particular multiplications.

Finally the last optimization that could be made to the code is on the decryption side. When performing the discrete logarithm it would be faster if the server used a baby-step giant-step approach, rather than iterating over every possible value from 0 and on. This would require some pre-computing but it would only be performed once and on the storage side it wouldn't take up a lot of memory either.

Chapter 4

Results

In this chapter are presented the results of the tests performed throughout this year. These results have several goals:

- Compare the two methods (El Gamal and PPATS)
- Compare the execution time between several browsers (same machine)
- Compare the execution time between several machines (same browser)
- Discuss the practicality of each method
- Discuss the relevance of the obtained results
- Compare the obtained results with already existent results

Section 1 will present the hardware used for these tests. It will also give the execution times of the *main mathematical primitives* that compose the two methods.

Sections 2 to 4 will discuss and compare the execution times of the *key generation*, the *ballot creation* and the *decryption factor computation* for both methods. We will also talk about the *precomputation* times.

Section 5 will give some information about the Python execution times. In this section the *pairing computation* time will be given.

4.1 Hardware and mathematical primitives

The reference computer `Comp1` which performed the comparison between different browsers has an i5-4460@3.2GHz processor and 8GB of RAM and it is a mid-range tower PC. Two other computers were used for comparison: `Comp2` (i7-6700HQ@2.6GHz with 8GB of RAM - mid to high-range laptop) and `Comp3` (PentiumB960@2.2GHz with 6GB of RAM - old low-range laptop). The server was run on a laptop (i7-4700HQ@2.40GHz with 16GB of RAM) running an `apache2` server.

Alternatively, two smart devices were used for comparison since more and more people use smart devices nowadays: `Smart1` - a Samsung Galaxy Tab S3 mid to high-range tablet (Snapdragon860@2.15GHz with 4GB of RAM) and `Smart2` - a Samsung Galaxy J5 2016 low-range smartphone (1.2GHz with 2GB of RAM).

Concerning the *main mathematical primitives* used throughout the implementation the operation that takes the most time in the El Gamal method is the **modular exponentiation**. On the other hand the equivalent operation in the PPATS method is the **elliptic curve point scalar multiplication**. The latter comes in two flavors depending on the field over which it is performed. As seen in the previous chapter, operations in $\mathbb{G}_1$ are performed over $\mathbb{F}_p$ while operations in $\mathbb{G}_2$ are performed over $\mathbb{F}_{p^2}$. The benchmark of my implementation is given in table 4.1.

primitive	modExp	$\mathbb{G}_1$	$\mathbb{G}_2$
time(ms)	2.91	4.08	13.64

Table 4.1: Execution times for main mathematical primitives with precomputed tables

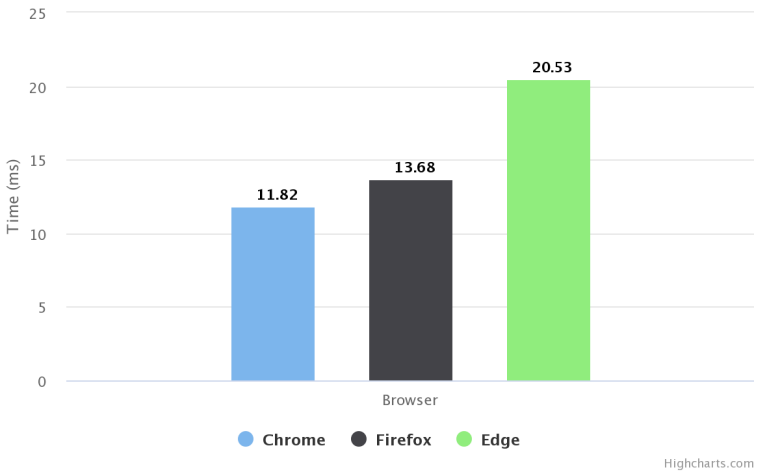
Something very important to be noted here is that these times suppose a precomputation was performed beforehand. In case no precomputation is done table 4.2 is more accurate.

primitive	modExp	$\mathbb{G}_1$	$\mathbb{G}_2$
time(ms)	33.66	15.74	50.55

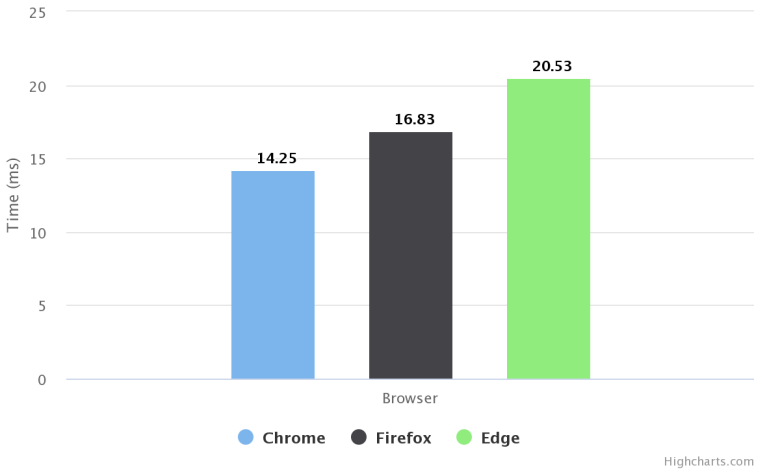
Table 4.2: Execution times for main mathematical primitives without precomputed tables

4.2 Key Generation

The key generation is performed before the beginning of the vote. This procedure depends on the number of trustees. In practice each of them can use their device to perform this action. Alternatively they can all take turns at performing this operation on the same machine. This operation is not required to be very fast (compared to the actual voting) but it doesn't have to be very slow either. In practice, people not understanding the procedure tend to believe something wrong is going on behind the curtain if the computation takes too much time, usually more than one second. The timing results for different browsers are in Figure 4.1



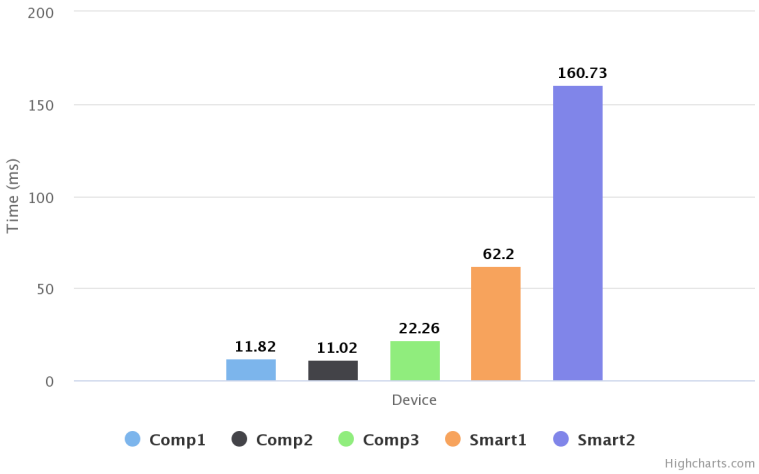
(a) El Gamal key generation time



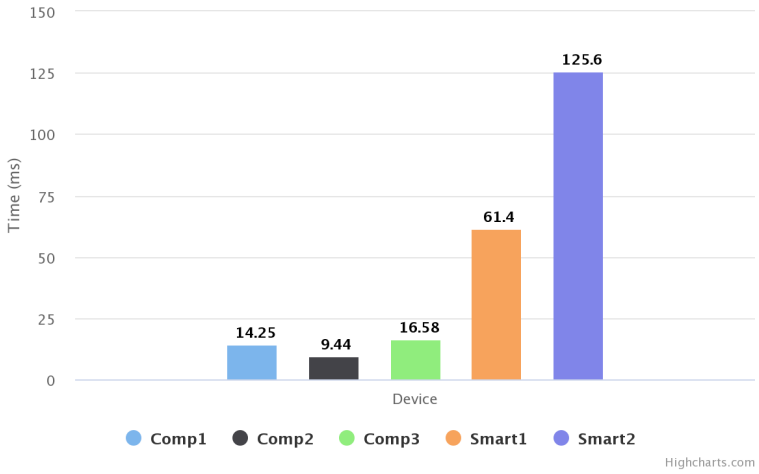
(b) PPATS key generation time

Figure 4.1: Key generation for different browsers

Figure 4.1 shows that the key generation takes little time for both methods. We also see that the `Chrome` browser is faster than its two main competitors `Firefox` and `Edge`. Let us now see how this time changes for different devices in Figure 4.2.



(a) El Gamal key generation time



(b) PPATS key generation time

Figure 4.2: Key generation for different devices

As expected, the smart devices are slower than the computers. However, since the execution time doesn't exceed one second for any of these devices one could regard the smartphone or the tablet as a valid alternative if, for instance, only the WiFi network would be available at the moment of the key generation. In most cases this shouldn't occur, since the key generation process must be prepared in advance and counter measures need to be taken against the possibility of network

failure.

Finally in Figure 4.3 the difference between the two methods is shown:

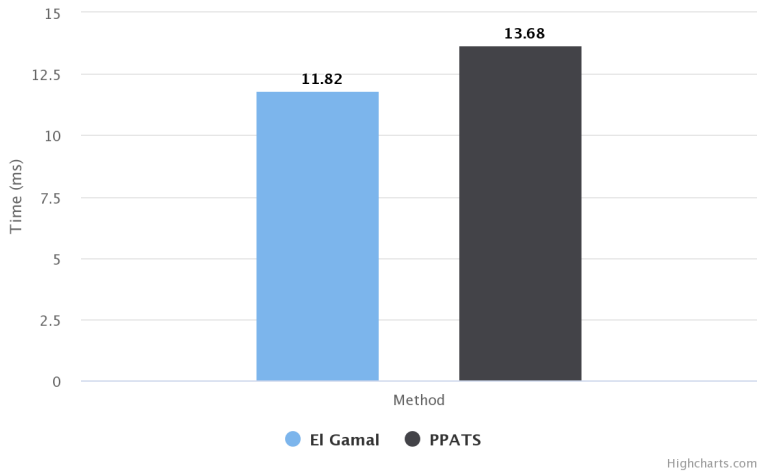


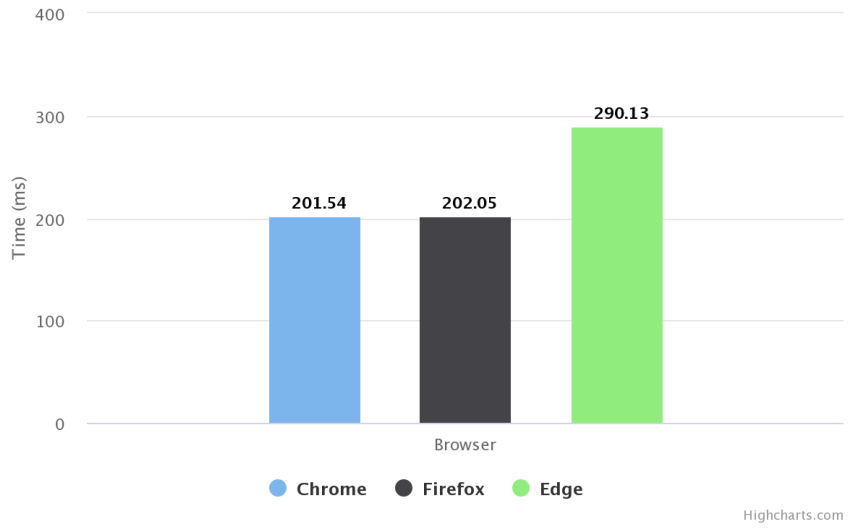
Figure 4.3: Key Generation time (El Gamal vs PPATS)

This figure shows that both methods have comparable execution times. In contrast, Figure 4.2 shows that on other devices PPATS is faster than El Gamal. This difference may be due to a non-optimal configuration of the precomputed table for the modular exponentiation. Indeed, the method proposed by the *Verificatum* library to compute the table requires an estimate number of exponentiations to be performed. The higher the number the longer the precomputation time but the shorter the exponentiation time. In the benchmark, the number of exponentiations was very big, while in an actual key generation process this number was smaller. The problem of precomputed tables is to be addressed in the next section.

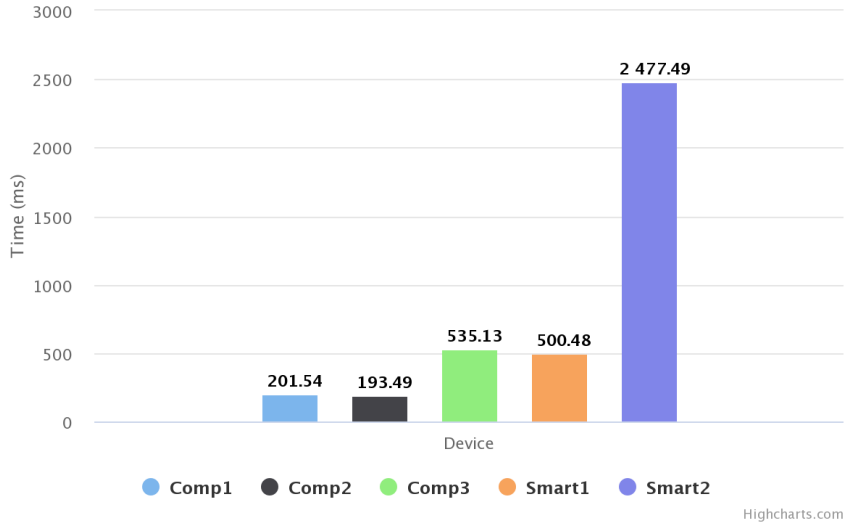
4.3 Ballot creation

Before delving into the execution times of the ballot creation let us discuss the matter of precomputed tables. We've seen so far that the modular exponentiation can be faster or slower than the point scalar multiplication over $\mathbb{G}_1$. This is due to the size of the precomputed table in both cases. However, in our implementation the tables for the modular exponentiation are computed in JavaScript while the ones for elliptic curves are computed in Python. Furthermore, different algorithms are used in both cases. In JavaScript the algorithm used comes from [19] p.9 while in Python the table only contains the first 255 powers of 2 that multiply the desired point. The JavaScript table creation is dependent on the number of exponentiations

to be performed. This is not the case for Python. Thus, the difference in the results in the previous section comes from here. To illustrate this, the precomputation of one table in Python takes 2.52ms for $\mathbb{G}_1$ and 11.31ms for $\mathbb{G}_2$. In JavaScript the precomputing times for a predicted number of 120 exponentiations is shown in Figure 4.4.



(a) Table precomputation times with respect to the browser

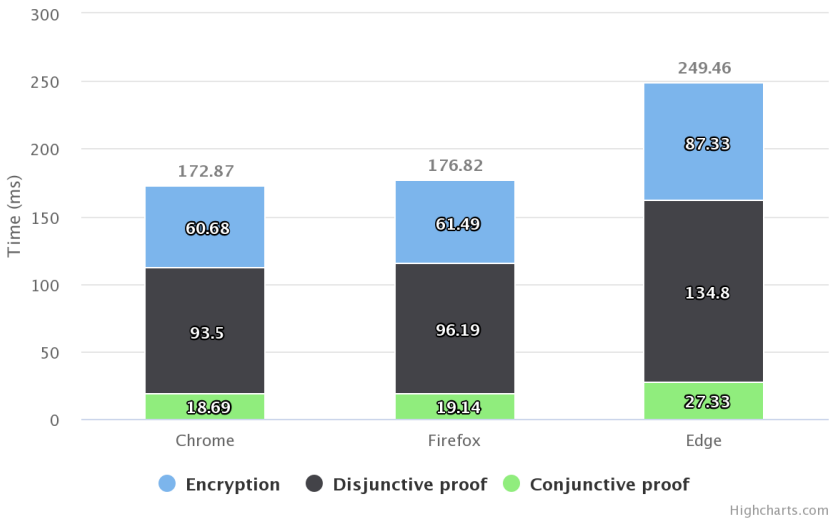


(b) Table precomputation times with respect to the device

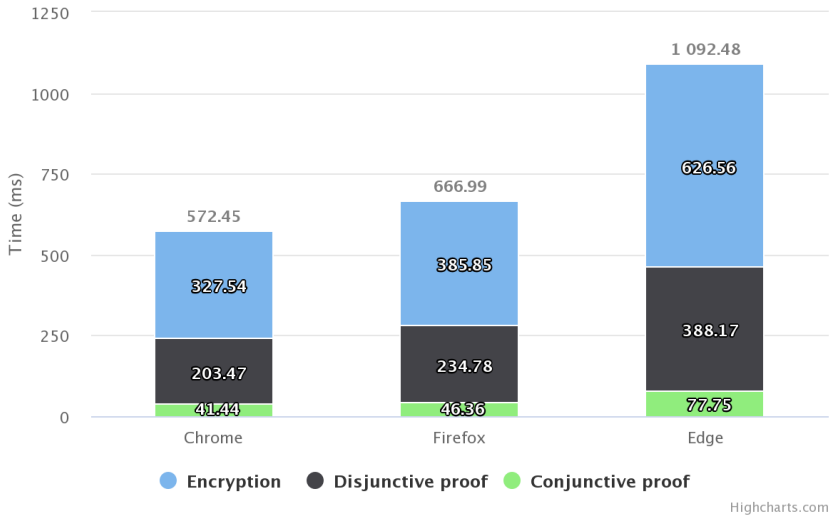
Figure 4.4: Table precomputation time

We see that in JavaScript the precomputing time is much higher. However, in practice one can precompute the JavaScript tables in Python as well, which would virtually allow for as much precomputation as desired. So we can theoretically consider that modular exponentiation is faster, as suggested in section 4.1.

Let us now consider the ballot creation itself. As for the key generation let us see how long it takes for each browser (Figure 4.5) to compute a ballot containing the encryption for **5 candidates** along with their respective **disjunctive proofs** and with the **overall conjunctive proof** (vote for at most one person):



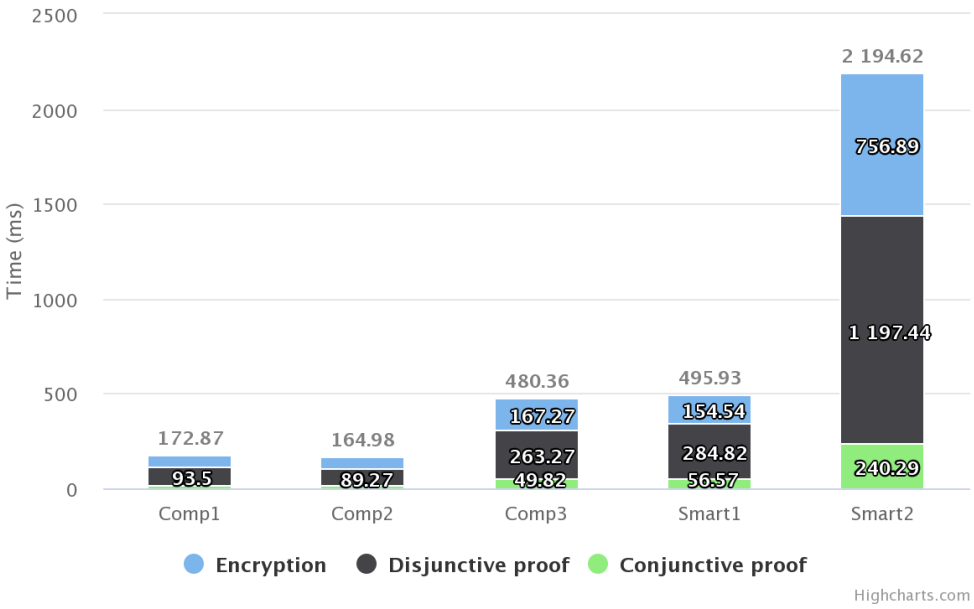
(a) El Gamal ballot creation times with respect to the browser



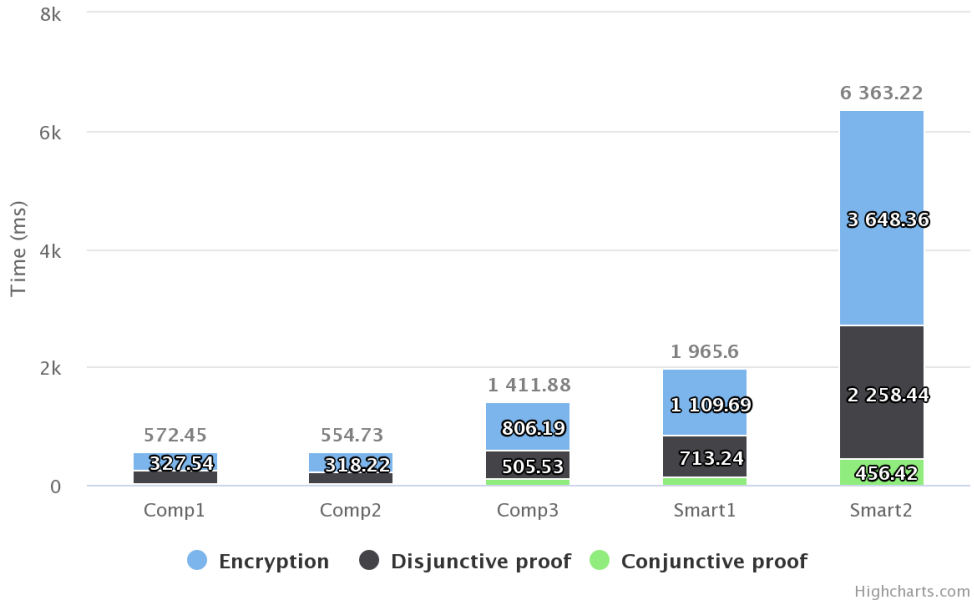
(b) PPATS ballot creation times with respect to the browser

Figure 4.5: Ballot creation time

As well as the evolution of the encryption times with respect to the devices in Figure 4.6.



(a) El Gamal ballot creation times with respect to the device



(b) PPATS ballot creation times with respect to the device

Figure 4.6: Ballot creation time

The are several things we can note from these graphs:

- First of all we confirm that the Chrome browser is faster than the other browsers here again
- Secondly, we see that PPATS is slower than El Gamal. This is mainly due to the extra computations in $\mathbb{G}_2$. Figure 4.7 shows the clear difference between the two methods.
- Finally, if we consider the *less than one second rule* for creating the ballot than the low-range smartphone is too slow for any of the two methods, while PPATS requires to perform the computation on at least a mid-range PC or laptop.

The final observation is quite strong since we suppose that the encryption is performed only after the voter made their choice. However, we've seen in the previous chapter that, using a worker, the voter can already perform some of the encryptions while they are making their choice. In that case, if we suppose it takes twice as much time to compute all the ballots (since both `yes` and `no` choices have to be made) and if we suppose the voter spends about 10 seconds on the voting page, then all these devices would be able to do the work.

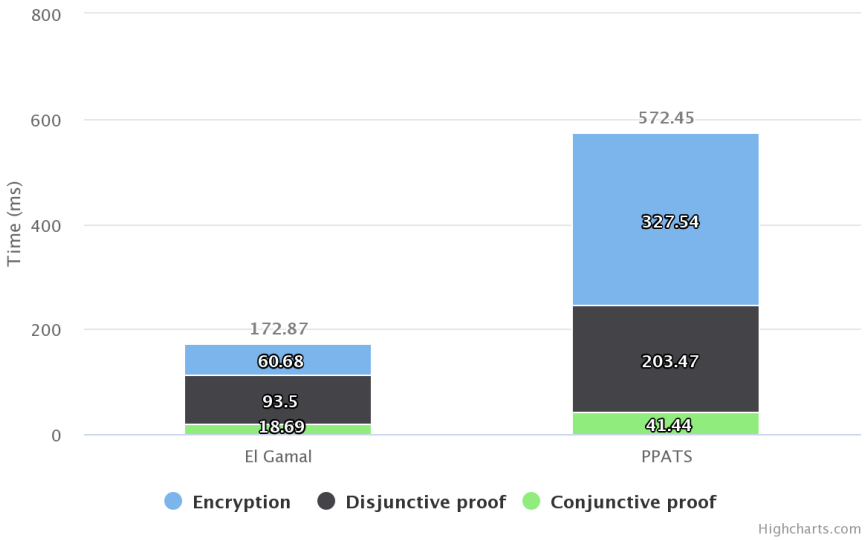
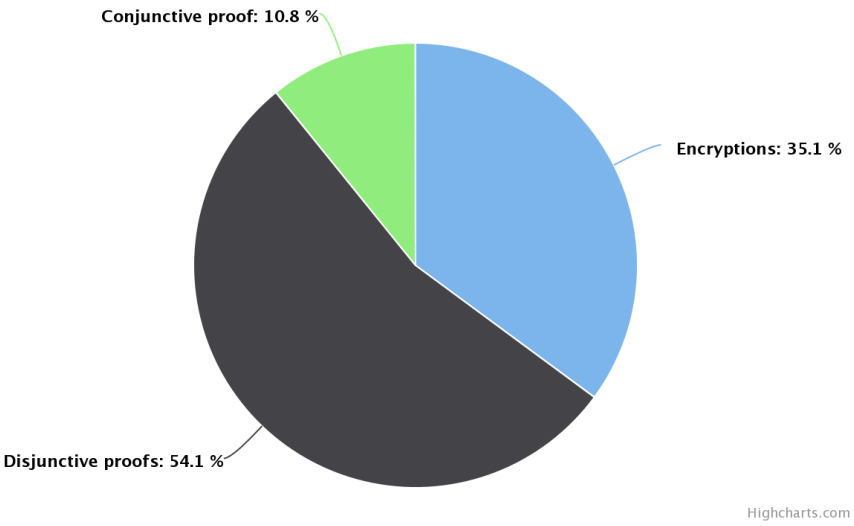


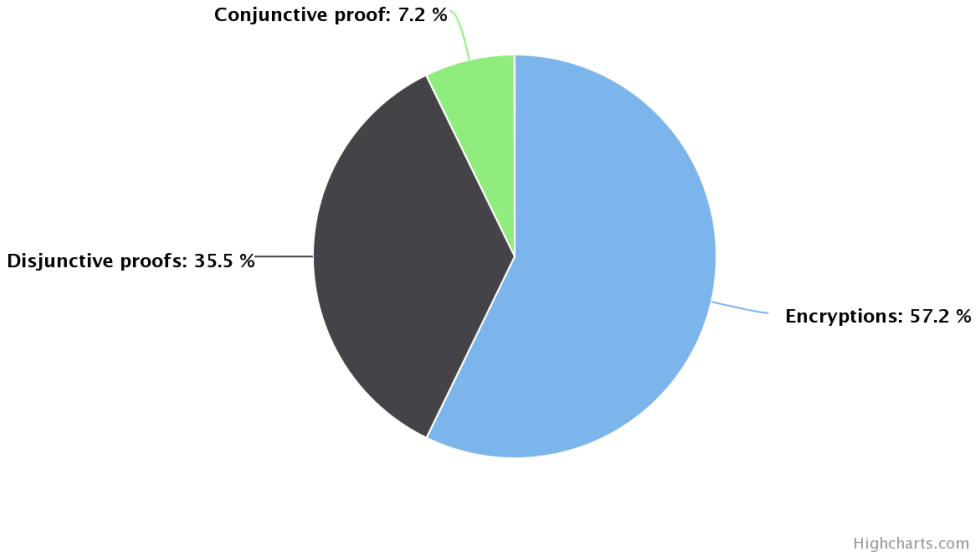
Figure 4.7: Ballot creation time (El Gamal vs PPATS)

One can see that the distribution between the building blocks of the ballot creation is not the same for the two methods. This is better depicted in Figure 4.8. The reason why the encryption takes more time in PPATS is because of the

consistency proof, which takes more than the rest of the encryption itself. Thus, the encryption process takes more than half of the encryption time in PPATS. The conjunctive proof is negligible since there are only two possibilities (0 or 1). We'll see later how the encryption time evolves with the increase in the number of possibilities.



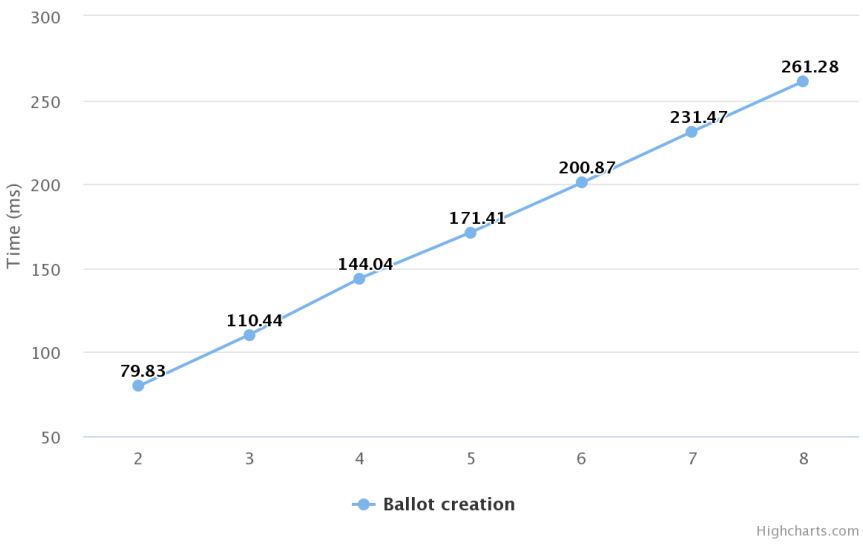
(a) Time distribution of the steps in the El Gamal ballot creation



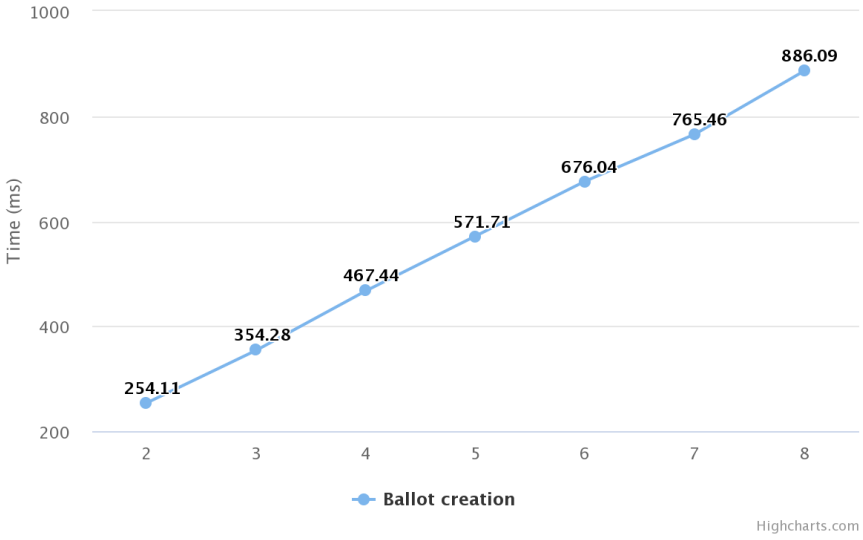
(b) Time distribution of the steps in the PPATS ballot creation

Figure 4.8: Time distribution charts

So far we have considered the case where the number of candidates is fixed (5 candidates). Let's see how the ballot creation time evolves for different numbers of candidates. Figure 4.9 shows the evolution of the ballot creation time for a number of candidates ranging from 2 to 8, for both methods.



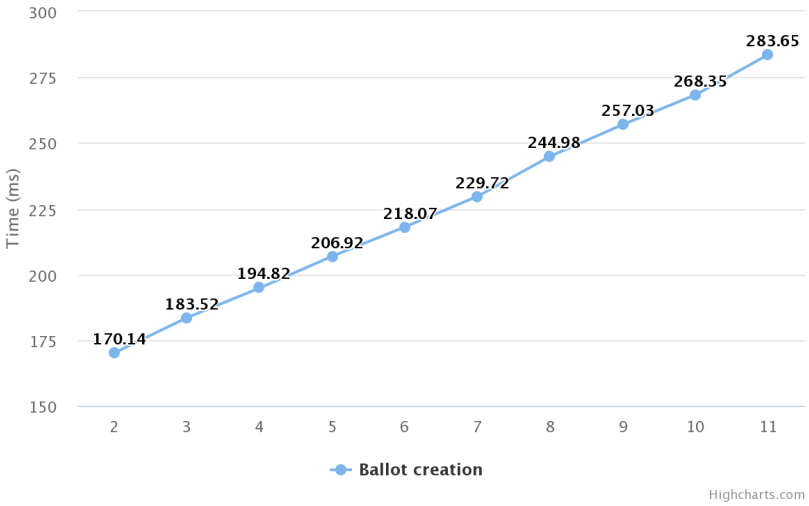
(a) El Gamal Ballot creation time evolution w/r to the number of candidates



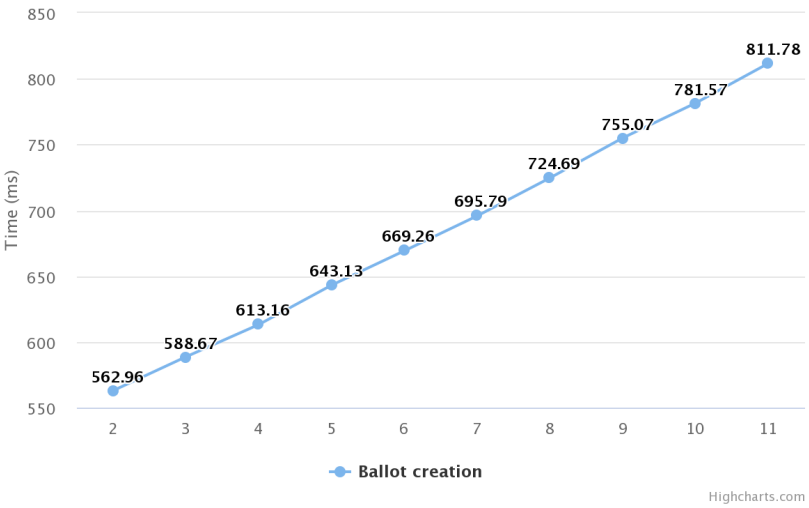
(b) PPATS Ballot creation time evolution w/r to the number of candidates

Figure 4.9: Time evolution with respect to the number of candidates

The main conclusion of Figure 4.9 is that adding more candidates is easier for El Gamal than for PPATS. Indeed, for the former one extra candidate adds about 30ms to the ballot creation time while for the latter it adds about 100ms. This difference is due of course to the presence of the scalar multiplications in $\mathbb{G}_2$ which account for an overhead of 6 scalar multiplications (3 for the encryption and 3 for the disjunctive proof) per extra candidate. The evolution with respect to the number of candidates is linear, as expected.



(a) El Gamal Ballot creation time evolution w/r to the number of choices



(b) PPATS Ballot creation time evolution w/r to the number of choices

Figure 4.10: Time evolution with respect to the number of choices

Finally, the last aspect we will analyze in this section is the one where the conjunctive proof needs to simulate a variable number of choices. So far the conjunctive proof needed to prove that the sum of the ballots was either 0 or 1. Figure 4.10 shows how the ballot creation time evolves when the maximum number of choices changes. Once again we see that El Gamal is less sensitive to this modification (about 13 ms per extra choice) than PPATS (about 25 ms). This is due to the presence of 2 extra scalar multiplications in $\mathbb{G}_2$ per extra candidate.

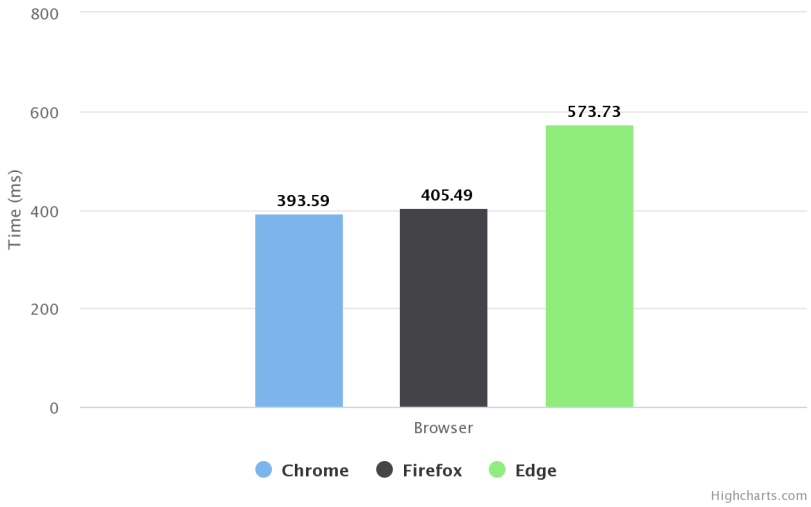
4.4 Decryption factor computation

The last step of the voting process is the decryption. In practice, the computation taking up most of the time in the case of homomorphic encryption is the computation of the decryption factors. This is because each trustee has to decrypt the ballot of each candidate with their share of the key, which in practice may take some time in order to organize each person to perform this operation and to convince them that no cheating is involved. As for the key generation process, a good practice here is to have this decryption factor be computed as fast as possible, to avoid raising any unnecessary suspicion. This computation is just a part of the whole decryption process, but it is the only one performed in JavaScript. Thus an analysis on the browser and device axis can be performed.

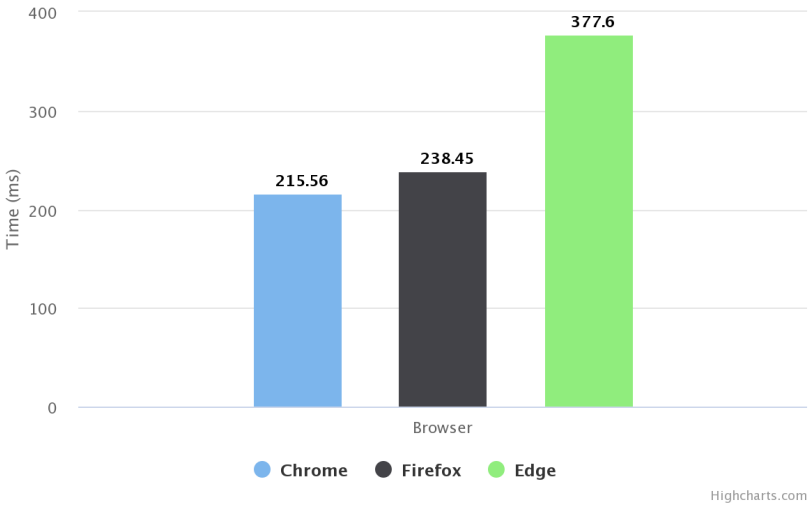
We start off again by showing the difference between several browsers for the two methods, in Figure 4.11, as well as the difference between the different devices, in Figure 4.12. The first conclusion one can make from these graphs is that the *one second rule* is respected for almost any device among the ones cited. The only problem would be with the smartphone *Smart2*, but since computing the decryption factor with a phone is highly improbable this is not a problem. On the other hand we see that PPATS performs better, although Edge seems to be less optimized for these operations. Indeed, in PPATS Edge takes 75% more than in Chrome, while in El Gamal it only takes only about 45% more time.

So far, for the first two steps of the voting process we've seen that both El Gamal and PPATS were faster for finally more computations. The reason behind this decrease in the speed is that neither the modular exponentiations (for El Gamal) nor the scalar multiplication (for PPATS) are performed in a fixed base. There are two such exponentiations (resp. scalar multiplications) and both are in the same base, which is the first element of the encryption ciphertext. One could argue that since the server possesses all the ballots it can as well perform precomputations over them, and it is true. This is one possible optimization. The question is whether it is worth computing a table per candidate for finally 2 exponentiations (resp. scalar

multiplications) per trustee. From a time consumption point of view if there are more than two trustees (which is almost always the case) it becomes interesting. However each table has a non negligible size and the total size of the information to be sent would increase linearly in the number of candidates. This would be less of a drawback for PPATS where the size of the tables would be smaller than for El Gamal since each element is 256-bit long compared to 3072 bits for the latter.

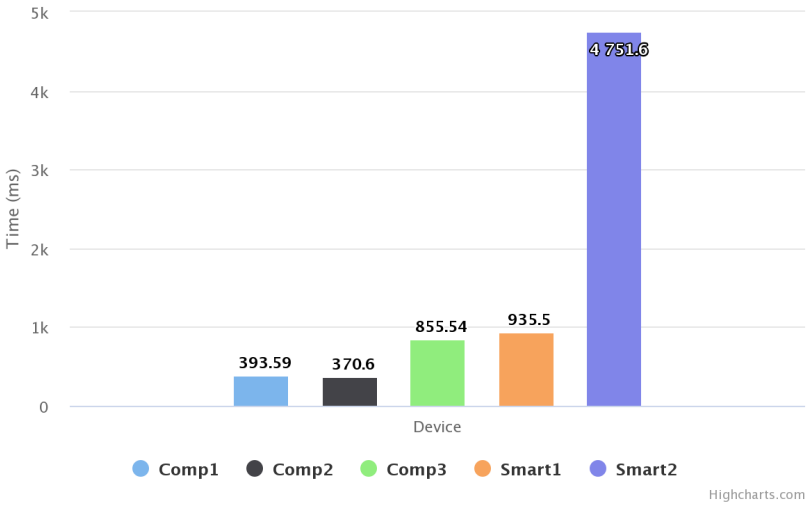


(a) El Gamal decryption factor computation time

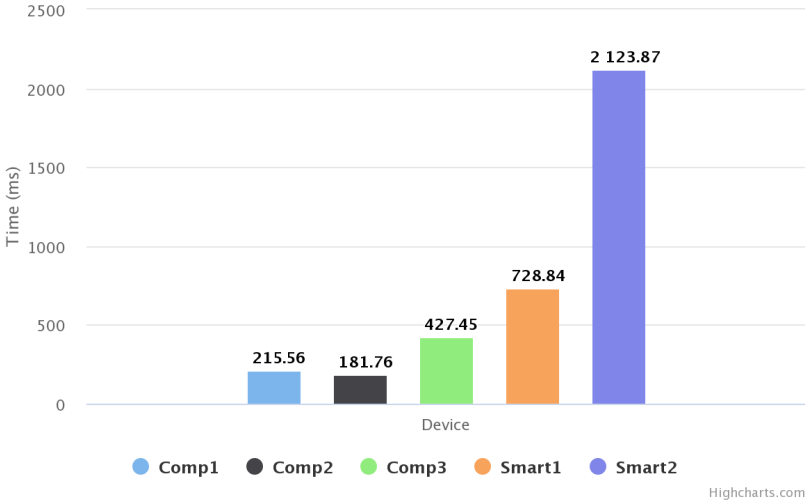


(b) PPATS decryption factor computation time

Figure 4.11: Time evolution for different browsers



(a) El Gamal decryption factor computation time



(b) PPATS decryption factor computation time

Figure 4.12: Time evolution for different devices

In order to underline the difference between the two methods, Figure 4.13 shows that PPATS is almost twice as fast as El Gamal. This difference comes from the fact that, as shown in the first section of this chapter, scalar multiplication in $\mathbb{G}_1$ is faster than modular exponentiation when no tables were precomputed. It also helps to remind that no computations in $\mathbb{G}_2$ are performed at this stage.

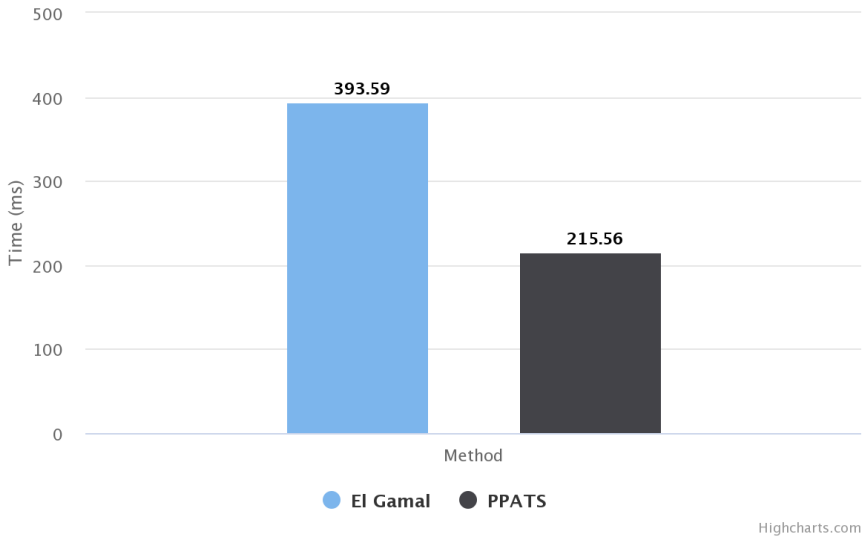


Figure 4.13: Decryption factor computation time (El Gamal vs PPATS)

Finally, let us see how this decryption time evolves with the number of candidates. Figure 4.14 shows that El Gamal adds about 80ms while PPATS only adds about 45ms per extra candidate. These numbers correspond to two extra exponentiations (resp. scalar multiplications) per extra candidate. If we look at the benchmark times of these two primitives in the first section of this chapter we see that they represent roughly twice the numbers in Table 4.2.

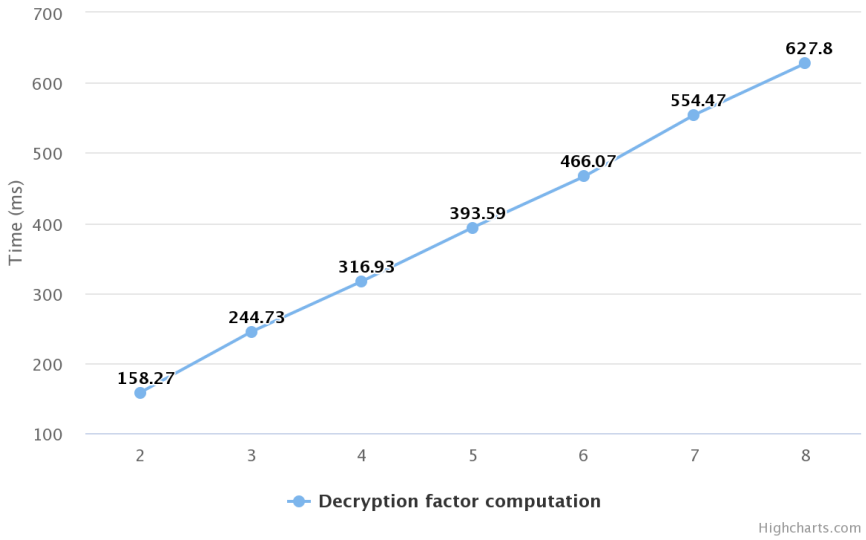
4.5 Python and pairing execution time

So far we've talked about execution times in JavaScript. Now it is time to give the execution times of the equivalent operations in a much faster programming language: Python. Table 4.3 gives the execution times of the main mathematical primitives already described in the first section of this chapter

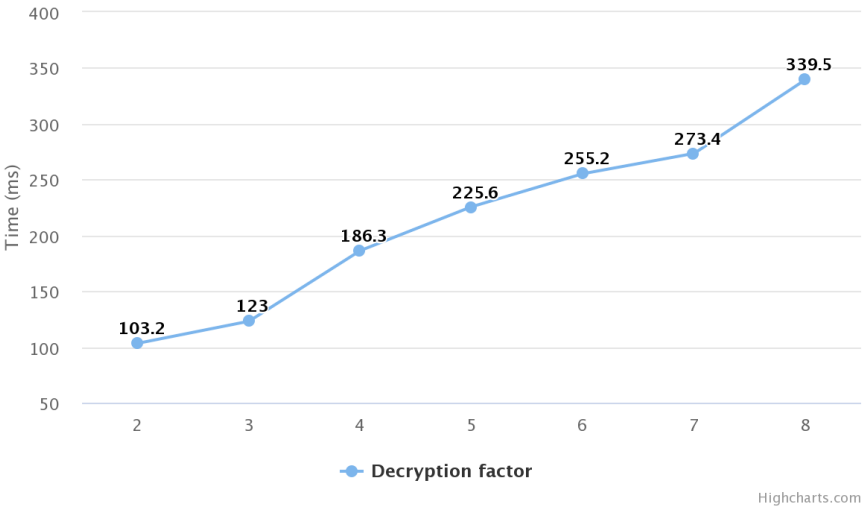
primitive	modExp	$\mathbb{G}_1$	$\mathbb{G}_2$
time(ms)	1.11ms	0.47ms	1.83ms

Table 4.3: Execution times for main mathematical primitives in Python

Note that only the scalar multiplication functions were implemented manually, with precomputed tables. The modular exponentiation used for this benchmark (and throughout the implementation) is the `powmod()` function provided with the `gmpy2` library.



(a) El Gamal decryption factor computation time



(b) PPATS decryption factor computation time

Figure 4.14: Evolution w/r to the number of candidates

We see that the times of the elliptic curve methods are almost 10 times smaller than the ones in JavaScript, thus proving the power of this language. The one of the modular exponentiation is only 3 times faster, which is due to the use of a function wiithout precomputed tables.

If we compare these results with [14] p.9 we see that this implementation is rather good. Scalar multiplication in $\mathbb{G}_1$ takes twice the time from the paper, but this is normal since in the latter MIRACL SDK is used which is a compiled code. It also holds because although compiled codes can be more than twice faster the becnhmark was performed in 2010 on a i5 processor, which is slower than the one used in this benchmark. The difference with the paper comes from the ratio between the execution times in $\mathbb{G}_2$ with respect to $\mathbb{G}_1$. In the paper this ratio is 2 while in my implementation the ratio is 4. This indicates that my quadratic complexity in the extension degree can be reduced to a linear complexity. This is another possible optimization that can be analyzed.

The next important thing to discuss is the computation time of the **precomputed tables**. The results are presented in Table 4.4

primitive	$\mathbb{G}_1$	$\mathbb{G}_2$
time(ms)	2.52ms	11.31ms

Table 4.4: Precomputed tables in Python

These times are reasonable in the sense that they only need to be computed once per fixed base. In the case of the PPATS method 4 tables (2 in $\mathbb{G}_1$ and 2 in $\mathbb{G}_2$) need to be computed before the beginning of the election. This amounts for a total of less than 30ms to be performed only once. In case one required the decryption factors to be computed using fixed base scalar multiplication, then one table in $\mathbb{G}_1$ would need to be computed for each candidate, and since the number of candidates is usually small the total time would again be negligible. The issue with these tables is their size. In Section 3.6 we've seen that the size of one table in $\mathbb{G}_1$ is 16KB while in $\mathbb{G}_2$ it is 32KB. For the election where two of each are used a total of 96KB needs to be sent to the user. On the other hand, for the decryption factors, if for example there are 10 candidates then an extra 160KB have to be sent to the client. While these amounts are not enormous they are not negligible either and a person with a bad internet connection would need to wait a little for the voting page to load.

Let us now see the execution time of the most computational task in the decryption process: **the pairing**. The benchmark time taken to compute one pairing is 19.3ms. If we compare again with [14] p.9 we see that the times are rather equivalent. Since it wasn't the case for the scalar multiplication this may indicate either:

- that my pairing implementation is better: many optimizations were applied to its computation as seen in the previous chapter.
- the elliptic curve scalar multiplication is not optimal: if other more optimized algorithms using precomputed tables could have been used

As discussed in the previous chapter the final exponentiation was one of the bottlenecks of the pairing computation. Figure 4.15 shows that thanks to the optimizations this operation only takes only 3.21ms of the whole process.

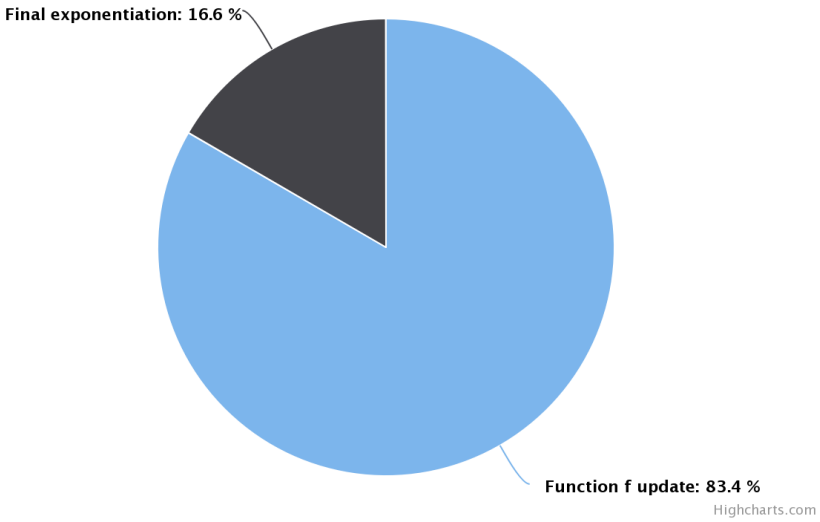


Figure 4.15: Pairing computation

Chapter 5

Conclusions

During the last year I worked on the implementation of browser-dedicated electronic voting based on homomorphic encryption. Throughout this implementation I came up with two operational simulations of the entire voting procedure. The first implementation was based on the El Gamal homomorphic encryption, while the second was based on Perfectly Private Audit Trail (PPAT) Commitment Consistent Encryption (CCE). The main advantages of each method is that the first one is faster, while the second one is more secure.

The first result of this paper is that a full implementation of the whole voting process was coded and is ready to be used in real elections. These implementations use a new library (VJSC) which is the fastest yet to come for manipulating large integers. It is also one of the first implementations of pairing-based cryptography for browser-based electronic voting.

The second result of this paper is that the first method turned out to be indeed faster than the second one. This is due to the extra computation needed to be performed in order to encrypt a ballot. It is also due to the fact that the basic primitive used in the first method, that is the modular exponentiation, is faster than the second method's both primitives: scalar multiplication of a point on an elliptic curve defined over $\mathbb{F}_p$ and over $\mathbb{F}_{p^2}$. For both these methods precomputed tables were used in order to accelerate the computations. These tables have non-negligible sizes and they depend on the size (in bits) of the modulus. In order to obtain the same 128-bit security level, the first method must use a modulus at least 3072-bit long while the second method can use a modulus as small as 256-bit. The downside of the first method is that, although faster, it needs to use more space.

The third result of this paper is to give an idea of the timing of these two methods. There are benchmarks for the main three steps of the election, which

are performed in JavaScript. There are also benchmarks for the counter part that is computed in Python. This information provides the interested reader with the adequate information in case they wish to use the code in a real election. As a bonus to this point, BlueKrypt sprl already decided to implement the PPATS code in their future elections.

Finally, the last outcome of this paper is that a series of inconsistencies were found in both the VJSC library and in one of the scientific articles. However, it is not useful to find an issue without solving it so the authors must be contacted.

From a personal point of view, I found this project both interesting and useful. From the academic point of view there were a lot of new mathematical concepts I discovered during this project. For instance, before starting working on this project I only knew the concept of the operations one is able to compute on elliptic curves. Since then I have broaden my knowledge with concepts such as the Jacobian representation and elliptic curves over extension fields, both of which I found really interesting. Regarding the notion of pairing, I have heard of it, its properties and its use before, but I had never understood how it worked. Now I understand that there are several types of pairings, how they are built, how the groups are chosen, and how to efficiently compute them.

From an implementation point of view, although El Gamal is a basic cryptographic algorithm I have seen many times before, the fact of coding it allowed me to better understand the concepts of group order and cyclic subgroup but also to understand how we choose the modulus, the order and the generator depending on what we wish to achieve. The implementation of ellitpic curve operations showed me how many different algorithms there are for performing something as general as an addition or a doubling. Working with large integers also made me realize the issue of working with non-standard types.

Finally I find it interesting to put my work in a nowadays context and answer the question of whether it is useful or not. From my point of view electronic voting will soon become the norm in terms of voting in Europe. Machines become more and more powerful and algorithms become more and more optimized and secured. This and the increase on the number of people using smart devices will make it easier for electronic voting to come into our lives. The only real problem I see is that people will have a hard time trusting the organizing party, and for good reasons. Someone understanding the voting procedure, if able to access the code can very easily cheat and modify the vote according to their desire. So first of all one must find a way to assure the voters that nobody cheats, but also make sure this is really the case. I believe my work aims at making a step forward towards the spread of electronic voting and I was glad to bring my contribution to it.

Bibliography

[1] N. Koblitz, *A course in number theory and cryptography*, 2nd ed. Springer-Verlag, 1994.

[2] D. Boneh, "The decision diffie-hellman problem", *Lecture Notes in Computer Science*, vol. 1423, 1998.

[3] P. S. L. M. Barreto, B. Lynn, H. Kim, and M. Scott, "On the selection of pairing-friendly groups", *Selected Areas in Cryptography - Lecture Notes in Computer Science*, vol. 3006, 2003.

[4] M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren, "The handbook of elliptic and hyperelliptic curve cryptography", in. CRC, 2005, p. 273, Theorem 13.13.

[5] S. D. Galbraith, "Pairings, volume 317 of london mathematical society lecture notes", in. Cambridge University Press, 2005, pp. 183–213, Theorem IX.2.

[6] R. Gonggrijp and W.-J. Hengeveld, "Studying the nedap/groenendaal es3b voting computer a computer security perspective", *USENIX*, 2007.

[7] B. Adida, O. D. Marneffe, O. Pereira, and J. Quisquater, "Electing a university president using open-audit voting: Analysis of real-world use of helios", *EVT/WOTE*, 2009.

[8] S. M., B. N., C. M., D. P. L.J., and K. E.J., "On the final exponentiation for calculating pairings on ordinary elliptic curves", *Pairing-Based Cryptography – Pairing 2009. Pairing 2009. Lecture Notes in Computer Science,*, vol. 5671, 2009.

[9] J. H. Silverman, "The arithmetic of elliptic curves (2nd edition)", in. Springer, 2009, pp. 137–138.

[10] B. JL., G.-D. J.E., M. S., O. E., R.-H. F., and T. T., "High-speed software implementation of the optimal ate pairing over barreto-naehrig curves", *Pairing-Based Cryptography - Pairing 2010. Pairing 2010. Lecture Notes in Computer Science*, vol. 6487, 2010.

[11] P. Bulens, D. Giry, and O. Pereira, "Running mixnet-based elections with helios", 2011.

[12] G. C. Pereira, M. A. Simplicio Jr, M. Naehrig, and P. S. Barreto, "A family of implementation-friendly bn elliptic curves", *Lecture Notes in Computer Science, Elsevier*, 2011.

[13] S. Bell, J. Benaloh, M. D. Byrne, D. DeBeauvoir, B. Eakin, G. Fisher, P. Kortum, N. McBurnett, J. Montoya, M. Parker, O. Pereira, P. B. Stark, D. S. Wallach, and M. Winn, "Star-vote: A secure, transparent, auditable, and reliable voting system", *USENIX Journal of Election Technology and Systems (JETS)*, vol. 1, no. 1, 2013.

[14] O. Pereira, "Verifiable elections with commitment consistent encryption", 2014.

[15] E. Cuvelier, "Privacy-preserving audit mechanisms for multi-party protocols", PhD thesis, Ecole Polytechnique de Louvain, 2015.

[16] E. Barker, *Recommendation for key management*, NIST Special Publication 800-57 Part 1, Revision 4, Accessed on 2019-06-09, 2016.

[17] D. F. Aranha, *Introduction to pairings*, `https://ecc2017.cs.ru.nl/slides/ecc2017school-aranha.pdf`, Accessed on 2019-05-22, 2017.

[18] BlueKrypt, *Bluekrypt - oadeo - security and audit*, `https://www.bluekrypt.com/en/oadeo-security-and-audit`, Slide 41, Accessed on 2019-05-11, May 2019.

[19] R. Haenni, P. Locher, and N. Gailly, "Improving the performance of cryptographic voting protocols", 2019.

[20] C. Costello, *Pairings for beginners*, Tutorial for pairings.

[21] E. Cuvelier, *Ppat prototype*, `https://github.com/ecuvelier/PPAT`, Accessed on 2019-05-27.

[22] D. S. Foundation, *Django python framework*, `https://www.djangoproject.com/`, Accessed on 2019-05-20.

[23] hyperelliptic.org, *Jacobian coordinates with a4=0 for short weierstrass curves*, `http://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-0.html`, Accessed on 2019-05-23.

[24] B. Leemon, *Big integer library by leemon*, `https://github.com/Evgenus/BigInt`, Accessed on 2019-05-20.

[25] Python, *The gmpy2 python library*, `https://gmpy2.readthedocs.io/en/latest/`, Accessed on 2019-05-20.

[26] Wikipedia, *Elliptic curve*, https://en.wikipedia.org/wiki/Elliptic_curve, Accessed on 2019-05-14.

[27] ——, *Extended euclidean algorithm*, https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm, Accessed on 2019-05-15.

[28] D. Wikström, *Verificatum javascript cryptographic library*, https://www.verificatum.org/, Accessed on 2019-05-19.

[29] T. Wu, *Rsa and ecc in javascript*, http://www-cs-students.stanford.edu/~tjw/jsbn/, Accessed on 2019-05-20.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**