UCLouvain

École polytechnique de Louvain

Fast tallying for UCL student elections

Author: **Cédric Legrand**
Supervisors: **Olivier Pereira, Edouard Cuvelier**
Readers: **Olivier Pereira, Edouard Cuvelier, Thomas Peters**
Academic year 2020–2021
Master [120] in Data Sciences Engineering

Abstract

Elections are organized and structured through complex and fragile processes that allow to govern the democracy in which we live. The evolution of technology appropriates these processes from a paper-based use to an electronic one, which enlarges the window of possibilities. This master thesis aims at optimizing the voter side process of UCL student elections, which can be enlarged to elections with a hundred of candidates and ten thousand voters. The use of the ElGamal homomorphic encryption scheme on web-browsers is further developed here, like the computation of modular exponentiations and Zero-Knowledge Proofs (ZKP) in order to optimize the performances of the electoral process.

Acknowledgements

I wish to express my gratitude to my supervisors, Olivier Pereira and Edouard Cuvelier, for their continuous support during the whole development of this master thesis. They introduced me to a fascinating subject that was initially unknown to me, and they guided me through with availability, altruism and thoughtfulness.

I also would like to thank my family and my friends for their support.

Contents

Chapter 1

Introduction

Today, the majority of government systems is based on democracy in which power is held by elected representatives. Even if elections are complex and fragile processes, they make it possible to firmly maintain the pillars of equality that rule the world we live in. With the constant evolution of technology and the way we use it, more and more paper-based processes can be replaced by computerized tools, and elections are part of them.

The use of an electronic voting system compared to a paper-based one is still often controversial. Indeed, each system has its advantages and drawbacks whose arguments can be categorized as follows :

- **Transparency** : This aspect is fundamental, as it allows voters to trust the electoral system. Anyone can understand the functioning of paper-based election systems. Each step of the process can easily and quickly be explainable to someone which is not part of it. While it is pretty clear for paper-based systems, it more looks like a black-box for electronic voting. Indeed, the procedures and algorithms used to carry out an electronic voting scheme are very complex, as the mathematical knowledge needed to understand them. Even if the tallying of electronic elections is public by displaying the ballots opening on a bulletin board, voters need to trust the structure of the system with eyes closed.

- **Human resources** : Whether printing ballots, preparing the sites or counting ballots, a lot of volunteers have to be mobilized in order to accomplish a paper-based election. They must achieve a huge quantity of manual work and consume their personal time. Moreover, the tallying process of such an election is always a long and grueling stage. Even if an electronic voting system also needs trustees, only some computer scientists and technicians also have to be present in order to check that each step of the process follows its course.

- **Monitoring** : As mentioned above, paper-based elections need a lot of volunteers. They furthermore have to be honest, as it is very difficult to control them. Checking that no one is cheating is a hard task and most of the processes are based on trust, needing a lot of people controlling each other. In electronic voting, every vote is encoded via the internet and stored in a centralized database. Moreover, the reference of each valid vote submitted is publicly displayed on a bulletin board, making sure it is not modified and it will be taken into account in the tallying process. Such a monitoring is impossible in paper-based voting and no one can be sure its vote has indeed been taken into account.

- **Freedom of shaping and modalities** : The use of electronic voting schemes allows to create elections whose scenarios can be different than the usual "one out of n candidates". The format of the election can be easily model, whether by simply changing the number of candidates or for instance by making weighted votes for each candidate. Such types of election schemes are more difficult to set up on paper-based systems.

Concerning the practical aspects, an electronic voting system can be organized either on dedicated machines or on personal devices :

1. **Dedicated machine** : This is a computer whose environment has only been developed for elections. The main advantage of using such a machine is that the security can be increased. Effectively, any connection with an external network can be prevented such that the window of hacking possibilities is reduced. Moreover, a dedicated machine can be built in order to optimize the calculation times. Such a device can then increase the processing speed of the ballots. However, a main drawback of using such a machine in an election is the fact that every voter needs to go on site in order to vote, like on paper-based elections.

2. **Personal devices** : When every voter can vote on its personal device, the previous drawback is clearly resolved. Everyone can vote from everywhere, which greatly facilitates the organisational aspect of the election. But the consequences are that the security is harder to maintain and the calculations are processed slower. In order to give access to every type of devices to the voting platform, the use of web-browsers seems to be the most practical. Moreover, their performances are optimized every year, which will increase the speed of calculations.

While the choice of the material support can be easily done for a paperless election depending on its needs, another type of choice is more difficult to make. There are actually two main electronic voting schemes, each used for different purposes :

1. **Homomorphic voting scheme** : It designates any voting structure in which an *homomorphic encryption* of the votes is used. It is a form of encryption that allows voters to perform computations on their encrypted data without first decrypting it. For instance, the server that recovers and centralizes all the homomorphically encrypted votes is able to check the validity of each encrypted vote and sum all the votes corresponding to a candidate, without decrypting it. The tallying then only consists in decrypting the sum of the ballots for each candidate. Moreover, the feasibility of such a scheme as already been proven. But the main drawback resides in the fact that the computational time of encrypting the ballots increases linearly with the number of candidates.

2. **Mixnet** : In this voting scheme, all the encrypted votes are centralized on the server and *shuffled*. The tallying consists then in decrypting every vote, one by one. The main drawback is that the more votes there are, the more time it takes to tally the results.

In fact, the drawback of one method is each time the advantage of the other. As each voting scheme has its specificities, some types of election can easily choose which one fits best its needs. For example, the US presidential elections would be more likely to use a homomorphic scheme as the number of candidates is small, compared to the number of voters. But this choice can be more difficult to do.

The student elections of the *Université Catholique de Louvain* (UCL) are performed every year in order to represent students within the university. These elections are made of a hundred of candidates distributed on about fifteen lists, while the number of voters never exceeds ten thousands. The voting scheme used for years for these elections is the mixnet one because the number of candidates is big compared to the number of voters. As mentioned above, the use of a mixnet takes more time to tally the votes as they need to be decrypted one by one. Furthermore, as technologies have evolved since the mixnet was set up years ago, one may reconsider this choice. Would it be preferable to use a homomorphic voting scheme for these elections ?

More generally, which voting scheme would best fits the needs of the UCL student elections ?

This thesis project aims at finding an answer to these questions by focusing on the **homomorphic voting scheme**, in order to let the administrators of the UCL student elections compare its performances with the mixnet currently in place. Moreover, the use of dedicated machines would certainly decrease the participation rate because of the need for students to move on site in order to vote, which was not the case for years. The use of the **web-browsers** from students personal devices is then considered here.

This project mainly focuses on the voter side of the ElGamal homomorphic voting scheme. The choice of the language and framework used to develop it draws the first guideline of the project. Even if the use of Rust and WebAssembly seems to be promising, the voting scheme is implemented with the JavaScript programming language. It is also based on the Verificatum library, allowing to work with the *big integers* needed to respect the security requirements. In addition, another part of the work achieved concerns the optimization of the most intensive tasks needed to be realized on this voting scheme, like the computation of modular exponentiations and Zero-Knowledge Proofs (ZKP).

Chapter 2

Theoretical aspects

All the different mathematical notions used to complete this thesis project are introduced in this chapter. The first section starts by explaining some basic concepts needed to understand the first voting scheme that will be described in details : the *ElGamal homomorphic encryption*. The voting procedure will also be detailed as the concept of Zero-Knowledge Proofs (ZKP). In the second section, another voting scheme called *Mixnet* is presented.

2.1 Number theory

Before explaining the ElGamal homomorphic voting scheme and its type of encryption, it is necessary to begin with some number theory. The definitions and theorems used below are taken from [1], Chapter 2.

Definition 2.1 ($\mathbb{Z}$)**.** Let the symbol $\mathbb{Z}$ define the set of integers :

$$\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\} \tag{2.1}$$

Definition 2.2 (Prime integer)**.** An integer $p \geq 2$ is said to be *prime* if its only positive divisors are 1 and p. Otherwise, p is called *composite*.

Definition 2.3 (Division algorithm for integers)**.** If a and b are integers with $b \geq 1$, then ordinary long division of a by b yields integers q (the *quotient*) and r (the *remainder*) such as

$$a = qb + r, \qquad \text{where } 0 \leq r < b \tag{2.2}$$

Moreover, q and r are unique. The remainder of the division is denoted $a \bmod b$, and the quotient is denoted $a \operatorname{div} b$.

Definition 2.4 (Greatest common divisor)**.** A non-negative integer d is the *greatest common divisor* of integers a and b, denoted $d = \gcd(a, b)$, if

(i) d is a common divisor of a and b ;

(ii) Whenever $c|a$ and $c|b$, then $c|d$.

Equivalently, $\gcd(a, b)$ is the largest positive integer that divides both a and b, with the exception that $\gcd(0, 0) = 0$.

Definition 2.5 (Modulus)**.** If a and b are integers, then a is said to be *congruent to b modulo n*, written $a \equiv b \pmod{n}$, if n divides $(a-b)$. The integer n is called the *modulus* of the congruence.

Example. $24 \equiv 9 \pmod{5}$ since $24 - 9 = 3 \cdot 5$.

The congruence respects the following properties $\forall a, a_1, b, b_1, c \in \mathbb{Z}$:

(i) $a \equiv b \pmod{n}$ if and only if a and b leave the same remainder when divided by n.

(ii) (*Reflexivity*) $a \equiv a \pmod{n}$.

(iii) (*Symmetry*) If $a \equiv b \pmod{n}$ then $b \equiv a \pmod{n}$.

(iv) (*Transitivity*) If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$.

(v) If $a \equiv a_1 \pmod{n}$ and $b \equiv b_1 \pmod{n}$, then $a+b \equiv a_1+b_1 \pmod{n}$ and $ab \equiv a_1 b_1 \pmod{n}$.

The *equivalence class* of an integer a is the set of all integers congruent to a modulo n. From properties (ii), (iii), and (iv) above, it can be seen that for a fixed n the relation of congruence modulo n partitions $\mathbb{Z}$ into equivalence classes. Now, if $a = qn + r$, where $0 \leq r < n$, then $a \equiv r \pmod{n}$. Hence each integer a is congruent modulo n to a unique integer between 0 and $n-1$, called the *least residue* of a modulo n. Thus a and r are in the same equivalence class, and so r may simply be used to represent this equivalence class.

These properties bring us to the following definition :

Definition 2.6 ($\mathbb{Z}_q$)**.** The *ring of integers modulo q*, denoted $\mathbb{Z}_q$, is the set of (equivalence classes of) integers $\{0, 1, 2, \ldots, q-1\}$:

$$\mathbb{Z}_q = \{a \in \mathbb{Z} \mid 0 \leq a < q\} \tag{2.3}$$

We can define the following properties over this group :

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

Example. $\mathbb{Z}_{25} = \{0, 1, 2, \ldots, 24\}$. In $\mathbb{Z}_{25}$, $13 + 16 = 4$, since $13 + 16 = 29 \equiv 4 \pmod{25}$. Similarly, $13 \cdot 16 = 8$ in $\mathbb{Z}_{25}$.

It is important to note that, due to the presence of the element 0 in the ring $\mathbb{Z}_q$, we cannot perform division for any two elements. If we want to, we first need to define the multiplicative inverse :

Definition 2.7 (Multiplicative inverse)**.** Let $a \in \mathbb{Z}_q$. The *multiplicative inverse* of a modulo q is an integer $x \in \mathbb{Z}_q$ such that $ax \equiv 1 \pmod{q}$. If such an x exists, then it is unique, and a is said to be *invertible*, or a *unit*. The inverse of a is denoted by a^{-1}.

Definition 2.8 (Division)**.** Let $a, b \in \mathbb{Z}_q$. *Division* of a by b modulo q is the product of a and b^{-1} modulo q, and is only defined if b is invertible modulo q.

Example. The invertible elements in $\mathbb{Z}_9$ are $1, 2, 4, 5, 7$ and 8. For example, $4^{-1} = 7$ because $4 \cdot 7 \equiv 1 \pmod{9}$.

Definition 2.9 (Multiplicative group)**.** The *multiplicative group* of $\mathbb{Z}_q$ is $\mathbb{Z}_q^\times = \{a \in \mathbb{Z}_q \mid \gcd(a, q) = 1\}$. In particular, if q is a prime, then

$$\mathbb{Z}_q^\times = \{a \mid 1 \leq a \leq q-1\}$$

The definition of a multiplicative group is followed by this theorem (proof at [2] p.19 Proposition I.3.1) :

Theorem 2.1. Let q be a prime integer, then

$$\forall a \in \mathbb{Z}_q^\times,\ \exists b \in \mathbb{Z}_q^\times \ s.t.\ a \cdot b \equiv 1 \pmod{q} \tag{2.4}$$

b is then called the *inverse* of $a \pmod{q}$.

It is important to note from this theorem that, if q is prime, we may always find an inverse of any element in $\mathbb{Z}_q^\times$, and this inverse is also in $\mathbb{Z}_q^\times$.

Definition 2.10 (Euler phi function)**.** For $q \geq 1$, let $\phi(q)$ denote the number of integers in the interval $[1, q]$ which are relatively prime to q. The function ϕ is called the *Euler phi function* (or the *Euler totient function*).

It as the following properties :

(i) If q is a prime number, then $\phi(q) = q - 1$.

(ii) The Euler phi function is *multiplicative*. That is, if $\gcd(m, n) = 1$, then $\phi(mn) = \phi(m) \cdot \phi(n)$.

Definition 2.11 (Order of $\mathbb{Z}_q^\times$)**.** The *order* of $\mathbb{Z}_q^\times$ is defined to be the number of elements in $\mathbb{Z}_q^\times$, namely $|\mathbb{Z}_q^\times|$.

It follows from the definition of the Euler phi function (**Definition 2.10**) that $|\mathbb{Z}_q^\times| = \phi(q)$. Note also that if $a \in \mathbb{Z}_q^\times$ and $b \in \mathbb{Z}_q^\times$, then $a \cdot b \in \mathbb{Z}_q^\times$, and so $\mathbb{Z}_q^\times$ is closed under multiplication.

Definition 2.12 (Order of $a \in \mathbb{Z}_q^\times$)**.** Let $a \in \mathbb{Z}_q^\times$. The *order* of a, denoted $\text{ord}(a)$, is the least positive integer t such that $a^t \equiv 1 \pmod{q}$.

Example. Let $q = 21$. Then $\mathbb{Z}_{21}^\times = \{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}$. Note that $\phi(21) = \phi(7) \cdot \phi(3) = (7 - 1) \cdot (3 - 1) = 12 = |\mathbb{Z}_{21}^\times|$. The orders of elements in $\mathbb{Z}_{21}^\times$ are listed in TABLE 2.1.

$a \in \mathbb{Z}_{21}^\times$	1	2	4	5	8	10	11	13	16	17	19	20
order of a	1	6	3	6	2	6	6	2	3	6	6	2

Table 2.1: Orders of elements in $\mathbb{Z}_{21}^\times$

It leads to the *Euler's theorem*, explained below.

Theorem 2.2 (Euler's theorem)**.** Let q be a prime integer. If $a \in \mathbb{Z}_q^\times$, then

$$a^{\phi(q)} = a^{q-1} \equiv 1 \pmod{q} \tag{2.5}$$

The concept of generators can now be introduced.

Definition 2.13 (Generator)**.** Let $g \in \mathbb{Z}_q^\times$. If the order of g is $\phi(q)$, then g is said to be the *generator* of $\mathbb{Z}_q^\times$. If $\mathbb{Z}_q^\times$ has a generator, then $\mathbb{Z}_q^\times$ is said to be *cyclic*.

The generators of $\mathbb{Z}_q^\times$ fulfil the following properties :

(i) $\mathbb{Z}_q^\times$ has a generator if and only if $q = 2, 4, p^k$ or $2p^k$, where p is an odd prime and $k \geq 1$. In particular, if p is a prime, then $\mathbb{Z}_q^\times$ has a generator.

(ii) If g is a generator of $\mathbb{Z}_q^\times$, then

$$\mathbb{Z}_q^\times = \{g^i \pmod q \mid 0 \leq i \leq \phi(q) - 1\} \tag{2.6}$$

Especially, $\mathbb{Z}_q^\times = \{1, g, g^2, g^3, \ldots, g^{q-2}\} \pmod q$ if q is prime.

(iii) Suppose that g is a generator of $\mathbb{Z}_q^\times$. Then $b = g^i \pmod q$ is also a generator of $\mathbb{Z}_q^\times$ if and only if $\gcd(i, \phi(q)) = 1$.
It follows that if $\mathbb{Z}_q^\times$ is cyclic, then the number of generators is $\phi(\phi(q))$.

(iv) $g \in \mathbb{Z}_q^\times$ is a generator of $\mathbb{Z}_q^\times$ if and only if $g^{\phi(q)/p} \not\equiv 1 \pmod q$ for each prime divisor p of $\phi(q)$.

Example. $\mathbb{Z}_{21}^\times$ is not cyclic since it does not contain an element of order $\phi(21) = 12$ (see TABLE 2.1). On the other hand, $\mathbb{Z}_{25}^\times$ is cyclic, and has a generator $g = 2$. The order of $\mathbb{Z}_{25}^\times$ is indeed $\phi(25) = 20$ which is also the order of $2 \in \mathbb{Z}_{25}^\times$ as 20 is the least positive integer such that $2^{20} \equiv 1 \pmod{25}$.

Now that the concept of generator of a cyclic group $\mathbb{Z}_q^\times$ was introduced, it is time to explain what a *cyclic subgroup* of a group $\mathbb{Z}_q^\times$ is :

Definition 2.14 (Cyclic subgroup)**.** If $g \in \mathbb{Z}_q^\times$, then the set of all powers of g forms a *cyclic subgroup* of $\mathbb{Z}_q^\times$, called the subgroup *generated by* g, and denoted by $\langle g \rangle$:

$$\langle g \rangle = \{g, g^2, g^3, \ldots, g^t\} \tag{2.7}$$

where t is the least positive integer such that $g^t \equiv 1 \pmod q$.

It is important to note that :

(i) All the elements inside this subgroup are different.

(ii) From **Theorem 2.2**, the order of this subgroup, in this case t, must divide the order $\phi(q)$ of $\mathbb{Z}_q^\times$, i.e. $r \mid \phi(q)$. It means that the order of any subgroup of $\mathbb{Z}_q^\times$ must be a factor of $\phi(q)$.

(iii) Every element g^i has an inverse g^{t-1} which belongs to the same subgroup.

Definition 2.15 (Size of a subgroup)**.** Let $g \in \mathbb{Z}_q^\times$ be an element of finite order t. Then $|\langle g \rangle|$, the size of the subgroup generated by g, is equal to t.

This section aims at finding a **prime order cyclic subgroup** of $\mathbb{Z}_q^\times$ with q prime. The following example illustrates it :

Example. Consider $q = 13$ as being the modulus of the multiplicative group $\mathbb{Z}_q^\times = \{1, 2, \ldots, 12\}$. Since q is prime, we have $\phi(13) = 12$ which is the order of $\mathbb{Z}_{13}^\times$. The group is cyclic (**Definition 2.13(i)**) and a generator is $g = 2$. From the **Definition 2.14(ii)**, the order of the subgroup generated by any element of $\mathbb{Z}_{13}^\times$ is either $1, 2, 3, 4, 6$ or 12 (i.e. all the factors of $\phi(13)$). The TABLE 2.2 lists the result of $g^i \pmod{13}$, $\forall g, i \in \mathbb{Z}_{13}^\times$, as well as the subgroup associated and its order (which is the first occurrence of 1 in each line).

g \ i	1	2	3	4	5	6	7	8	9	10	11	12	Subgroup	Order
1	1	1	1	1	1	1	1	1	1	1	1	1	$\{1\}$	1
2	2	4	8	3	6	12	11	9	5	10	7	1	$\{1, 2, 3, \ldots, 12\}$	12
3	3	9	1	3	9	1	3	9	1	3	9	1	$\{1, 3, 9\}$	3
4	4	3	12	9	10	1	4	3	12	9	10	1	$\{1, 3, 4, 9, 10, 12\}$	6
5	5	12	8	1	5	12	8	1	5	12	8	1	$\{1, 5, 8, 12\}$	4
6	6	10	8	9	2	12	7	3	5	4	11	1	$\{1, 2, 3, \ldots, 12\}$	12
7	7	10	5	9	11	12	6	3	8	4	2	1	$\{1, 2, 3, \ldots, 12\}$	12
8	8	12	5	1	8	12	5	1	8	12	5	1	$\{1, 5, 8, 12\}$	4
9	9	3	1	9	3	1	9	3	1	9	3	1	$\{1, 3, 9\}$	3
10	10	9	12	3	4	1	10	9	12	3	4	1	$\{1, 3, 4, 9, 10, 12\}$	6
11	11	4	5	3	7	12	2	9	8	10	6	1	$\{1, 2, 3, \ldots, 12\}$	12
12	12	1	12	1	12	1	12	1	12	1	12	1	$\{1, 12\}$	2

Table 2.2: The subgroups of $\mathbb{Z}_{13}^{\times}$

From this TABLE 2.2, we can conclude that 3 and 9 are generators of the same prime order subgroup which is $\{1, 3, 9\}$. This subgroup is interesting because any of its elements (except 1) can generate the entire subgroup. It is then called a **prime order cyclic subgroup**.

An important property of this type of subgroup is the fact that **the mapping** between the exponents i and the elements of the subgroup g^i **is one way**. Effectively, by looking at any element g^i and by knowing g, the only way to extract i is to try every possible value. There is not any more efficient method that allows to get i in another way. This problem is known as the **Discrete logarithm (DL) problem**. The voting schemes we will analyze in this thesis use cryptographic primitives whose safety reside on the difficulty of the DL problem.

Note that calculating the inverse function, so computing g^i by knowing g and i can be done with efficient polynomial-time algorithms. These are algorithms whose execution time is upper bounded by a polynomial expression in the size of the input.

2.2 A first voting scheme

2.2.1 ElGamal homomorphic encryption

Now that all the arithmetical notions were detailed in the previous section, we have the background notions to explain the ElGamal encryption method.

Definition 2.16 (**ElGamal 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 message to encrypt, $m \leftarrow \mathbb{Z}_r$.

(i) $\mathbf{Gen}(\mathbf{1^n})$ runs a polynomial-time algorithm to generate $\mathbb{G}, q, g$ as described above, then chooses a random $x \xleftarrow{\$} \mathbb{Z}_r$ and computes $h = g^x$. The *public key* is $pk = \langle \mathbb{G}, q, g, h \rangle$ and the *private key* is $sk = \langle \mathbb{G}, q, g, x \rangle$.

(ii) $\mathbf{Enc_{pk}(m)}$ chooses a random $y \xleftarrow{\$} \mathbb{Z}_r$ and outputs the ciphertexts $\langle c_1, c_2 \rangle = \langle g^y, h^y \cdot m \rangle$.

(iii) $\mathbf{Dec_{sk}}(\langle \mathbf{c_1}, \mathbf{c_2} \rangle)$ computes $m := (c_2 / c_1^x)$

The privacy of the secret key is guaranteed with the DL problem. Nevertheless, the security of the encryption cannot only reside under this assumption. Effectively, if by knowing $\langle g, h(= g^x), c_1(= g^y) \rangle$, one can leak information about $h^y = g^{xy}$, and the information about the message m will be disclosed. Thus, information can only be secure under the **Decisional Diffie-Hellman (DDH) assumption**.

Definition 2.17 (**Diffie-Hellman problem** [2])**.** Suppose that two users A (Alice) and B (Bob) want to agree upon a key – a random element of $\mathbb{Z}_q^\times$ – which they will use to encrypt their subsequent messages to one another. Alice chooses a random integer a between 1 and $q-1$, which she keeps secret, and computes $g^a \in \mathbb{Z}_q$, which she makes public.
Bob does the same : he chooses a random b and makes public g^b. The secret key they use is then g^{ab}. Both users can compute this key. For example, Alice knows g^b (which is public knowledge) and her own secret a. However, a third party knows only g^a and g^b. If the following assumption (DDH assumption) holds for the multiplicative group $\mathbb{Z}_q^\times$, then an unauthorized third party will be unable to determine the key.

Definition 2.18 (**Decisional Diffie-Hellman (DDH) assumption** [2])**.** It is computationally infeasible to compute g^{ab} knowing only g^a and g^b.

The Diffie-Hellman assumption is *a priori* at least as strong as the assumption that discrete logarithms (DL) cannot be feasibly computed in the group. That is, if discrete logarithms (DL) can be computed, then obviously the Diffie-Hellman assumption fails. Some people would conjecture that the converse implication also holds, but that is still an open question. In other words, no one can imagine a way of passing from g^a and g^b to g^{ab} without first being able to determine a or b; but it is conceivable that such a way might exist.

For know, by making :

- The assumption that the discrete logarithm (DL) problem in not resolvable with a polynomial-time algorithm,
- The Diffie-Hellman (DDH) assumption,

the security of the ElGamal encryption method is guaranteed.

It is now time to describe the ElGamal homomorphic encryption. This encryption method is the one we will use further in this thesis project. Compared to the classical ElGamal encryption method :

- A first variation is that the message m is here encoded as g^m instead of simply m.
- A second variation is the use of the "*homomorphic*" term.

From [4], *homomorphic encryption* is a form of encryption that allows users to perform computations on its encrypted data without first decrypting it. These resulting computations are left in an encrypted form which, when decrypted, result in an identical output to the one that would be obtained by performing the operations on the unencrypted data.

Definition 2.19 (**ElGamal 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 message to encrypt and a relatively small positive integer.

(i) $\mathbf{Gen}(\mathbf{1^n})$ runs a polynomial-time algorithm to generate $\mathbb{G}, q, g$ as described above, then chooses a random $x \xleftarrow{\$} \mathbb{Z}_r$ and computes $h = g^x$. The *public key* is $pk = \langle \mathbb{G}, q, g, h \rangle$ and the *private key* is $sk = \langle \mathbb{G}, q, g, x \rangle$.

(ii) $\mathbf{Enc_{pk}}(\mathbf{m})$ chooses a random $y \xleftarrow{\$} \mathbb{Z}_r$ and outputs the ciphertexts $\langle c_1, c_2 \rangle = \langle g^y, h^y \cdot g^m \rangle$.

(iii) $\mathbf{Dec_{sk}}(\langle \mathbf{c_1}, \mathbf{c_2} \rangle)$ computes $m := \log_g (c_2 / c_1^x)$

This encryption method is indeed homomorphic. Effectively, if we want to encrypt two messages m_1 and m_2, we only need to compute $\mathbf{Enc_{pk}}(\mathbf{m_1})$ and $\mathbf{Enc_{pk}}(\mathbf{m_2})$ such that it gives the following two ciphertexts :

$$\langle c_{1,1} = g^{y_1} \ , \ c_{2,1} = h^{y_1} \cdot g^{m_1} \rangle \tag{2.8}$$

$$\langle c_{1,2} = g^{y_2} \ , \ c_{2,2} = h^{y_2} \cdot g^{m_2} \rangle \tag{2.9}$$

We can combine the ciphertexts by multiplying them element by element such that $C_1 = c_{1,1} \cdot c_{1,2}$ and $C_2 = c_{2,1} \cdot c_{2,2}$ to obtain :

$$\langle C_1 = g^{y_1+y_2} \ , \ C_2 = h^{y_1+y_2} \cdot g^{m_1+m_2} \rangle \tag{2.10}$$

Finally, we can decrypt these ciphertexts by applying $\mathbf{Dec_{sk}}(\langle \mathbf{C_1}, \mathbf{C_2} \rangle)$ to reach :

$$\begin{aligned}
\log_g \{C_2 \ / \ C_1^x\} &= \log_g \left\{ \left(h^{y_1+y_2} \cdot g^{m_1+m_2}\right) \ / \ \left(g^{x(y_1+y_2)}\right) \right\} \\
&= \log_g \left\{ \left(h^{y_1+y_2} \cdot g^{m_1+m_2}\right) \ / \ \left(g^{x\ y_1 + x\ y_2}\right) \right\} \\
&= \log_g \left\{ \left(h^{y_1+y_2} \cdot g^{m_1+m_2}\right) \ / \ \left(g^{x\ y_1} \cdot g^{x\ y_2}\right) \right\} \\
&= \log_g \left\{ \left(h^{y_1+y_2} \cdot g^{m_1+m_2}\right) \ / \ \left(h^{y_1} \cdot h^{y_2}\right) \right\} \\
&= \log_g \left\{ \left(h^{y_1+y_2} \cdot g^{m_1+m_2}\right) \ / \ \left(h^{y_1+y_2}\right) \right\} \\
&= \log_g \left\{ g^{m_1+m_2} \right\} \\
&= m_1 + m_2
\end{aligned} \tag{2.11}$$

Even if the last step of the ElGamal homomorphic encryption method ($\mathbf{Dec_{sk}}(\langle \mathbf{c_1}, \mathbf{c_2} \rangle)$) is the computation of a discrete logarithm, it is not an issue as long as the sum of the messages $m_1 + m_2$ is relatively small ($< 2^{16}$). The size of the messages is a very important parameter when performing such an encryption. In our case, we will understand in the next sections that it won't be a problem for us, as the messages m_i we will encrypt are votes which are only 0 or 1.

2.2.2 The voting procedure

In the following section, the voting procedure for the UCL student elections will be detailed. These elections are made of more or less a hundred candidates distributed over a dozen of lists. Every year, there are more or less ten thousand ($\pm 10\,000$) voters. A voter can vote for as many candidates as he wants as long as they are all in the same list. Only one list can then be voted for. A blank ballot can also be returned. Then, for every voter's ballot, there will be a hundred of messages m to encrypt which are in fact the votes v expressed by 1 if a candidate is chosen and 0 if it is not.

The global voting procedure is divided in 5 main steps :

1. To begin, all the designated trustees of the election must compute their own public and private keys. Afterwards, they publish their public key and keep their private key secret. All the public keys of the trustees are then homomorphically added such that the global public key of the election is computed.

2. The election then begins. Every voter must authenticate on the voting platform. This is essential to know the identity of the voter such that his ballot is linked to him. Several security barriers must be developed : for example the voter cannot vote several times, but this is out of the scope of this project.

   Every voter fulfils its ballot and encrypts it with the public parameters of the election including g and the public key. The vote is sent to the server of the election which centralizes all the election information. A signature of each voter's ballot is generated and published on a public bulletin board in order that every voter can check that its vote has been taken into account, validated and not modified.

3. When the election is over, all the ballots are homomorphically added for each candidate as in EQUATION 2.10 such that only one main ballot remains. It is sent to each trustee.

4. The trustees come together with their individual secret key to decrypt the final ballot of each candidate.

5. At the end, the results are published and the individual secret keys are destroyed.

The FIGURE 2.1 below depicts how the global voting procedure works.

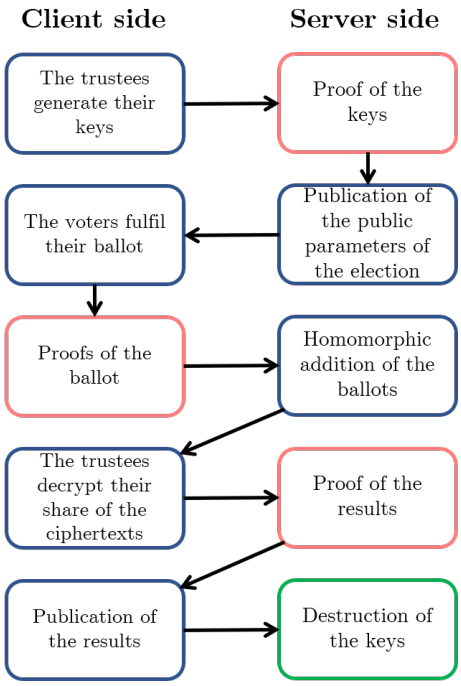


Figure 2.1: Voting procedure describing operations on client and server sides

We will now detail how the different steps are implemented.

In the **first** step, before the computations of the keys by the trustees, the server generates a description of a cyclic group $\mathbb{G} \subset \mathbb{Z}_q^\times$ of prime order r with $||r|| = n$ *bits* and with generator g. The server adds these information in a `.json` file where it also sets global and public information about the election as the group modulus, the detailed lists of candidates, the date of the election, its duration, the name of the election, etc. For now, the only information missing in this file is the public key of the election. The server then stores this `.json` file in an online database such that the trustees can query and retrieve it.

Afterwards, each trustee computes its public key pk_i and its private key sk_i. One might think that it would be enough to have only one trustee and not several. However, if there was only one trustee, he would pick a random private key $x \xleftarrow{\$} \mathbb{Z}_r$, compute $h = g^x$ and send it to the server such that h would be the public key of the election. The main problem with it is that this trustee could use its private key to cast any vote and then know which voter voted for which candidate. And this would break the anonymity of the voters, which is a basic principle in an election.

Moreover, to rebuild the private key x from the shared private keys x_i of the trustees, we need **all** the private keys x_i. So for example, if from n trustees, $n-1$ of them decide to exchange their private key, they won't be able to retrieve the global private key x. It is then mandatory to have more than one trustee in the election, and it is important that they do not know themselves or communicate. In fact, at most there are trustees and at most the security of the election will be guaranteed, because as long as at least one trustee does not cheat the private key will be kept secret.

The protocol to compute the public key pk of the election is the following :

1. Each trustee i picks at random $x_i \xleftarrow{\$} \mathbb{Z}_r$ and keeps secret its private key $sk_i = \langle \mathbb{G}, q, g, x_i \rangle$.
2. Each trustee computes its share of the public key $pk_i = \langle \mathbb{G}, q, g, h_i \rangle$ with $h_i = g^{x_i}$.
3. Each trustee computes a non-interactive Zero-Knowledge Proof (ZKP) [1] (**Definition 2.20**) to prove that he knows its private key.
4. Each trustee sends its share of the public key pk_i to the server with the associated ZKP. The server then stores every share of public key if its corresponding ZKP is correct.
5. The server adds homomorphically every share of public key pk_i to obtain the global public key of the election $pk = \langle \mathbb{G}, q, g, h \rangle$ with

$$h = \prod_i h_i = \prod_i g^{x_i} = g^{\sum_i x_i} = g^x$$

   Here, x is the global private key of the election that **nobody** knows.
6. The server adds the public key in its `.json` file and publishes it on its database in order that anybody can have access to it.

In the **second** step of the voting procedure, the election starts and every voters connects to the voting platform via a web browser on its own device. The device can be a smartphone, a tablet, a computer or any device that has a web browser on it. Each voter authenticates on the platform, and receives from the server the public parameters that allow to complete the voting procedure :

1. The voter fulfils its ballot by voting for some candidates or leaves the ballot blank. Every candidate n in each list m has a vote $v_{m,n}$ assigned to it from the voter. It is either 1 if the voter votes for this candidate, either 0 if he doesn't.

[1] The concept of ZKP will be explained in the next subsection.

2. Every vote $v_{m,n}$ is encrypted using the ElGamal homomorphic $\mathbf{Enc_{pk}(v_{m,n})}$ function. It produces two ciphertexts $\langle c_1, c_2 \rangle = \langle g^y, h^y \cdot g^{v_{m,n}} \rangle$ which makes the ballot for the vote $v_{m,n}$.

3. For every vote $v_{m,n}$, the voter computes a non-interactive Disjunctive ZKP (**Definition 2.22**) that allows to prove that the vote is either a 0 or a 1 and not something else.

4. The voter sums up homomorphically all its votes $v_{m,n}$ for each list of candidates m such that he obtains a list of ciphertexts (a, b) containing m ciphertexts (a_m, b_m). The message in each of these ciphertexts represents the number of candidates the voter voted for in each list of candidates.
On this list (a, b) of encrypted ballots (a_m, b_m), the voter computes a non-interactive ZKP (**Definition 2.23**) that proves that he voted for a unique list of candidates, e.g. that all non-null ciphertexts lie in a unique list of candidates.

5. The encrypted ballots of each vote $v_{m,n}$, as the list (a, b) of encrypted ballots (a_m, b_m), as the non-interactive ZKP proofs are sent to the server to store the ballots and verify each proof.

6. If the ballots of the voter are valid, they are assigned to a signature that is displayed on a board accessible by all the voters. It allows every voter to check that its vote has correctly been taken into account and that it has not been modified. Note that he can access to this board as many times as he wants until the end of the voting period.

A main disadvantage of the ElGamal homomorphic encryption is that the computations, as the computational time, increase linearly with respect to the number of votes to encrypt, e.g. the number of candidates. This is why the ElGamal homomorphic encryption is unpractical when the number of candidates is too big. And this leads us to the heart of this thesis project.

The UCL student elections is made of a hundred of candidates, which is a lot. Up to now, the ElGamal homomorphic encryption was unpractical for this type of election and another voting scheme was used : **the Mixnet**, which will be presented in the next section. This project thesis aims at using the ElGamal homomorphic voting scheme for these elections as it is easier to use in practice. In the next chapter, I explain the different tricks and technologies used to optimize this voting scheme.

At the **third** part of this voting procedure, the voting period is over. The server has all the ballots of all the voters and adds homomorphically every ballot corresponding to the same candidate. It then creates a set of ciphertexts where each one contains the number of votes for a candidate. These ciphertexts are saved in the database with the global parameters of the election.

The **fourth** part is the tallying phase of the voting procedure. All the trustees need to be present to help decrypt the global ballots detailed above for each candidate. The procedure is as follows :

1. For each candidate, the server sends the global ballot $\langle c_1, c_2 \rangle$ to each trustee.

2. Each trustee computes, for each candidate's global ballot, its *decryption factor* $c_1^{x_i}$ with x_i the share of the private key of the trustee.

3. Each trustee computes a non-interactive Chaum-Pedersen ZKP (**Definition 2.24**) to prove that he used the same share of the private key x_i to compute the decryption factor $(c_1^{x_i})$ as he did in the **first step** to compute its share of the public key $(h_i = g^{x_i})$. He then sends the decryption factors and the proof to the server.

4. The server verifies that the proof is correct and, if it is the case, stores the decryption factor for each candidate.

5. When it has received the decryption factors of all the trustees for all the candidates, the server adds them homomorphically for each candidate to compute

$$\prod_i c_1^{x_i} = \prod_i g^{y\, x_i} = g^{y \sum_i x_i} = g^{y\, x} := c_1^x$$

where x is the global private key of the election.

6. The server can finally proceed to the decryption of the ballot of each candidate by applying the $\mathbf{Dec_{sk}}(\langle \mathbf{c_1}, \mathbf{c_2} \rangle)$ function of the ElGamal homomorphic encryption scheme and obtain for each candidate
$$\sum v := \log_g (c_2/c_1^x)$$
where $\sum v$ represents the number of votes for a candidate.

It is crucial to note that the trustees cannot loose their share of the private key. Indeed, if one misses, the decryption cannot be carried out and the results of the election cannot be obtained.

The **fifth** and last step of the voting procedure is the publication of the results by the server. The trustees also have to destroy their share of the private key in order to make sure that the election remains secured and that the results cannot be modified, or that any *a posteriori* attack cannot be performed.

2.2.3 Zero-Knowledge Proofs

In this section, I will explain what a Zero-Knowledge Proof (ZKP) is, by using the examples and definitions from [5], [6], [7], [8] and [9]. I will develop the mathematical theory of ZKPs based on so-called Σ-protocols. I also will describe in details some ZKPs that will be used in the developed voting scheme.

A ZKP is a tool that allows to one party called **the prover** to prove to another party called **the verifier** that he has done a certain operation correctly, without revealing more information than the fact that it was done correctly. This type of proof must respect three different conditions :

1. **Completeness** : if the hypothesis is true, the honest verifier (that is, one verifier following the protocol properly) will always be convinced of this fact by an honest prover.
2. **Soundness** : if the hypothesis is false, no cheating prover can convince the honest verifier that it is true, except with some small (negligible) probability.
3. **Honest verifier zero-knowledge** : the honest verifier doesn't learn anything from the messages exchanged during the proving procedure.

We will base our ZKPs on Sigma protocols (Σ-protocols). A Σ-protocol defines a three-pass interaction between a prover P and a verifier V, as depicted on FIGURE 2.2. In the first message, called **the commitment**, P submits a random value a taken from $\mathbb{G}$ to the verifier V. This commitment will be used to blind the secret value about which P wants to make a statement. The second message, called **the challenge**, contains a random integer e chosen by V. It it crucial for the soundness of the proof to be respected that P does not know e when he commits through a. Eventually, P sends **the response** f to V, which is typically made of elements of $\mathbb{Z}_q$. V makes use of the proof statement a, e and f to decide whether he accepts the proof.

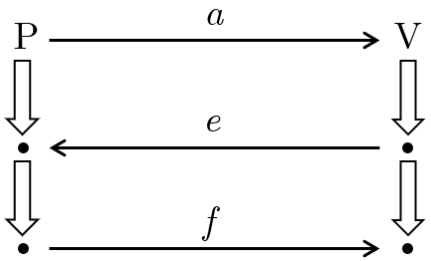


Figure 2.2: A three-pass Σ-protocol

As an example, we describe below the **Schnorr's protocol**, the most basic ZKP known.

Definition 2.20 (**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 P 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, depicted on FIGURE 2.3, is used :

1. The prover P picks $a \xleftarrow{\$} \mathbb{Z}_r$ and computes **the commitment** $A = g^a$. P sends it to the verifier V.
2. V picks **the challenge** $e \xleftarrow{\$} \mathbb{Z}_r$ and sends it back to P.
3. P computes **the response** $f = a + e \cdot x$ and sends it to V. The protocol is over.

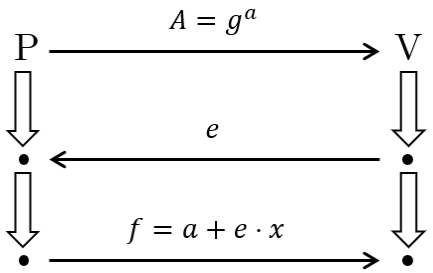


Figure 2.3: Schnorr's protocol

We can check how the Schnorr's protocol respects the 3 properties of the ZKP :

1. **Completeness** : If the prover is honest and knows x, then he will respect the protocol and send a correct transcript $\langle A, e, f \rangle$ to the verifier. This one can verify the correctness of the transcript by comparing g^f and $A \cdot h^e$ $(= g^a \cdot g^{e \cdot x} = g^{a+e \cdot x})$ that should be equal.
2. **Soundness** : If we assume that the prover does not know x but is still able to provide a correct transcript $\langle A, e, f \rangle$. It means that, for two different challenges e_1 and e_2 sent by the verifier, the prover can send the two correct responses f_1 and f_2. But if he can compute these responses, the prover can find x by computing

$$\begin{aligned} (f_1 - f_2) &= (a + e_1 \cdot x) - (a + e_2 \cdot x) \\ &= (a - a) + x \cdot (e_1 - e_2) \\ \Leftrightarrow \qquad x &= \frac{f_1 - f_2}{e_1 - e_2} \end{aligned} \tag{2.12}$$

   However, this contradicts the initial hypothesis that the prover does not know x.
3. **Honest verifier zero-knowledge** : To prove that the honest verifier does not learn anything from the messages exchanged during the proving procedure, we only have to show that he can generate a valid transcript without knowing x. Effectively, by picking $e, f \xleftarrow{\$} \mathbb{Z}_r$ and computing $A = \frac{g^f}{h^e}$ $(= \frac{g^a \cdot g^{e \cdot x}}{h^e} = g^a)$, the resulting $\langle A, e, f \rangle$ is a valid transcript.

In practice, we will use similar protocols in a modified version. Effectively, a disadvantage of this protocol is that it exchanges information between the prover and the verifier. In our case, the prover is the voter and the verifier the server. But in our browser-based elections, we need to execute the proofs locally, without any connection with the server. We will then use **non-interactive** ZKP. Indeed, the only message that is exchange between the voter and the server in a normal ZKP is the challenge. Therefore, we will contrive ourselves to generate locally this challenge. To do this, we generate a random element in $\mathbb{Z}_r$. But to simulate this randomness, we will use a **Hash function** and put in it a maximum of parameters, such as all the global public elements of the elections accessible by the voter. We also set in it the commitment.

Definition 2.21 (Hash function [12])**.** A hash function is any function that can be used to map data of arbitrary size to fixed-size values. The values returned by a hash function are called hash values and are *pseudo-random*.

The non-interactive Schnorr's protocol is used in the **first** step of the voting procedure, in order for each trustee to prove that he knows its share of the private key x_i given their share of the public key pk_i.

The second ZKP proof that will be used in the voting scheme is the non-interactive Disjunctive proof. It is used to prove that the ciphertext of a vote is an encryption of either 0 or 1 and not anything else.

Definition 2.22 (**Non-interactive Disjunctive ZKP**)**.** Let g be the generator of a cyclic subgroup $\mathbb{G} \subset \mathbb{Z}_q^\times$ 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 $\langle c_1, c_2 \rangle$ be an ElGamal encrypted ciphertext of a vote v. The prover wishes to prove that the vote v he encrypted is either 0 or 1. He does so by creating a simulated proof transcript for the value the vote is not, called $\bar{v}$, and by computing a real proof on the real encrypted value v. The following protocol is used :

1. For the value $\bar{v}$, the prover picks $e_{\bar{v}}, f_{\bar{v}} \xleftarrow{\$} \mathbb{Z}_r$, computes the first **commitments**

$$A_{\bar{v}} = \frac{g^f}{c_1^e}$$
$$B_{\bar{v}} = \frac{h^f}{\beta^e} \quad \text{where } \beta = \frac{c_2}{g^{\bar{v}}}$$

   and stores the transcript $\langle A_{\bar{v}}, B_{\bar{v}}, e_{\bar{v}}, f_{\bar{v}} \rangle$.

2. For the real vote v, the prover picks $a \xleftarrow{\$} \mathbb{Z}_q$ and computes the second **commitment**

$$A_v = g^a$$
$$B_v = h^a$$

3. The prover then computes e which is a hash function applied to all the commitments and a series of parameters such as the ciphertext and global parameters of the election (including the public key and the generator g) :

$$e = H(A_{\bar{v}}, B_{\bar{v}}, A_v, B_v, c_1, c_2, g, h, \dots)$$

   Afterwards, the prover subtracts from e the challenge $e_{\bar{v}}$ used in the simulated proof to obtain the **challenge** e_v that will be used in the proof of the real vote v.

4. Finally, the prover computes the **response** $f_v = a + e_v \cdot y$ where y is the randomness used to perform the ElGamal encryption. The prover outputs the proof π containing the commitments, the challenges and the responses :

$$\pi = \langle A_v, A_{\bar{v}}, B_v, B_{\bar{v}}, e_v, e_{\bar{v}}, f_v, f_{\bar{v}} \rangle$$

The verification of this non-interactive ZKP is performed in the same way as the verification of the Schnorr's protocol with some extra steps :

1. For each transcript $i \in \{v, \bar{v}\}$, the verifier has to compute and check that the following equalities hold :

$$g^{f_i} = A_i \cdot c_1^{e_i}$$
$$h^{f_i} = B_i \cdot \beta_i^{e_i} \quad \text{where } \beta_i = \frac{c_2}{g^i}$$

2. The verifier must re-compute the hash function and check that it is equal to the sum of the challenges :
$$e_v + e_{\bar{v}} = H(A_{\bar{v}}, B_{\bar{v}}, A_v, B_v, c_1, c_2, g, h, \dots)$$

The verification equations above prove the **Completeness** character of the proof. The **Soundness** of this non-interactive Disjunctive ZKP can be reduced to the soundness of the real sub-proof, which was demonstrated for Schnorr's protocol (EQUATION 2.12). Concerning the **Honest verifier Zero-Knowledge** character of the proof, the verifier could simulate the real sub-proof in the same way the others were simulated, thus obtaining a correct set of transcripts.

Another non-interactive ZKP to be discussed here is the one to proof that the prover voted for a unique list of candidates, e.g. that all non-null ciphertexts lie in a unique list of candidates.

Definition 2.23 (**Non-interactive ZKP that all non-null ciphertexts lie in a unique list of candidates**)**.** Let g be the generator of a cyclic subgroup $\mathbb{G} \subset \mathbb{Z}_q^\times$ 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 (C_1, C_2) be a list of ElGamal homomorphically encrypted ciphertexts $\langle c_{1,i}, c_{2,i}\rangle$ of the sum of the votes v_i in each of the n lists of candidates. We have then $\forall i \in \{1, \dots n\}$, $C_1 = \{c_{1,1}, \dots, c_{1,i}\}$ and $C_2 = \{c_{2,1}, \dots, c_{2,i}\}$. The prover wishes to prove that at most one tuple $(c_{1,k}, c_{2,k})$ from the list of ciphertexts (C_1, C_2) is an encryption of non-null votes, where k is the index of the list (C_1, C_2) with non-null candidate's votes. He does so by creating a simulated proof transcript for every list of candidates except the k^{th} one, and by computing a real proof on the k^{th} list of candidates. The following protocol is used :

1. For every list of candidates $i \in \{1, \dots n\}$ except the k^{th} one, the prover picks $a \xleftarrow{\$} \mathbb{Z}_r$, computes the first **commitment**
$$A_i = g^a$$
$$B_i = h^a$$
and stores the transcript $\langle A_i, B_i\rangle$.

2. For the list of candidates k, the prover picks $e_k, f_k \xleftarrow{\$} \mathbb{Z}_q$ and computes the second **commitment**
$$A_k = \frac{g^{f_k}}{c_{1,k}^{e_k}}$$
$$B_k = \frac{h^{f_k}}{c_{2,k}^{e_k}}$$

3. The prover then computes e which is a hash function applied to all the commitments and a series of parameters such as the ciphertexts and global parameters of the election (including the public key and the generator g) :
$$e = H(A_i, B_i, A_k, B_k, C_1, C_2, g, h, \dots)$$
Afterwards, the prover computes the **challenges** e_i for every list of candidates i such that they all lie on the same straight line whose y-intercept is the output of the hash function e :
$$e_i := \frac{e_k - e}{k} \cdot i + e$$

4. Finally, for every list of candidates $i \in \{1, \dots n\}$ except the k^{th} one, the prover computes the **responses** $f_i = a + e_i \cdot y_i$ where y_i is the sum of the random values used to perform the ElGamal homomorphic encryption. The prover outputs the proof π containing the commitments, the challenges and the responses :
$$\pi = \langle A_i, B_i, e_i, f_i\rangle \qquad \forall i \in \{1, \dots, n\}$$

The verification of this non-interactive ZKP is performed in the same way as the verification of the Schnorr's protocol with some extra steps :

1. For every list of candidates i, the verifier has to compute and check that the following equalities hold :

$$g^{f_i} = A_i \cdot c_{1,i}^{e_i}$$
$$h^{f_i} = B_i \cdot c_{2,i}^{e_i}$$

2. The verifier must also re-compute the hash function :

$$e = H(A_i, B_i, A_k, B_k, C_1, C_2, g, h, \dots)$$

3. Finally, the verifier checks that the challenges all lie on the same straight line whose y-intercept is the output of the hash function e. He does it by checking that the following equality holds for every list of candidates $i \in \{\mathbf{2}, \dots, n\}$:

$$e_i - e = (e_1 - e) \cdot i$$

The verification equations above prove the **Completeness** character of the proof. As the structure of this non-interactive ZKP is similar to the Disjunctive one, its **Soundness** aspect can also be reduced to the soundness of the real sub-proof, which was demonstrated for Schnorr's protocol (EQUATION 2.12). Concerning the **Honest verifier Zero-Knowledge** character of the proof, the verifier could simulate the real sub-proof of the following **Definition 2.24**, thus obtaining a correct set of transcripts.

The last non-interactive ZKP we need to discuss here is the one made by the trustees during the decryption process in the **fourth** step. This one must proof that the same share of the private key x_i was used to go from g to $pk_i = g_i^x$ as to go from c_1 to the *decryption factor* $c_1^{x_i}$. This proof is called the **non-interactive Chaum-Pedersen ZKP** and is detailed below.

Definition 2.24 (**Non-interactive Chaum-Pedersen ZKP**)**.** Let g_1 and g_3 be two elements of a cyclic subgroup $G \subset \mathbb{Z}_q^\times$ 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 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 of inputs such as the commitments and global parameters of the election including the public key and the generator g. The output of the 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 non-interactive ZKP must respect the following conditions :

1. The output of the hash function must be consistent with the challenge of the proof.
2. The verifier must compute and check the following equalities :

$$g_1^f = A \cdot g_2^e$$
$$g_3^f = B \cdot g_4^e$$

The above equations prove the **Completeness** property of this non-interactive ZKP. The **Soundness** property is proved by the same argument as the one of Schnorr's protocol. Effectively, if we make the hypothesis that a non-honest prover who does not know x can produce a right response

f_1 for a challenge e_1, then he can produce a right response f_2 for a second challenge e_2. But if he does it, he can extract $x = \log_{g_1}(g_2) = \log_{g_3}(g_4) = \frac{f_1 - f_2}{e_1 - e_2}$ and then know the value of x which contradicts the initial hypothesis. Finally, the **Honest verifier Zero-Knowledge** property is respected by the fact that the verifier is able to pick $e, f \xleftarrow{\$} \mathbb{Z}_r$ and compute $A = \frac{g_1^f}{g_2^e}$ and $B = \frac{g_3^f}{g_4^e}$. this produces a correct transcript without needing to know x.

A last but not least important point to discuss about is the choice of the parameters being used as arguments of the hash function.

- The **commitment** of the ZKP is the most important parameter to put in the hash function as it is one of the main parameters of the specific proof. This is the parameter that makes the hash different from the one of another proof.
- The **ciphertext** corresponding to the ZKP is also mandatory in the hash function. It avoids being able to compute the same hash for the same kind of proof made on another ciphertext. Effectively, someone could cheat by using the same randomness to compute the above commitments. It is then an additional security barrier against cheating.
- The **global parameters and details of the election** must also be in the hash function as it makes the difference between an election and another. Effectively, the group modulus and the group generator can be reused for other elections. It is then also important to add the election duration, starting date, name, ID, description, etc. to the arguments of the hash function. It allows to fix the ZKP for this election and avoids to use the same proof in other elections.

2.3 A second voting scheme

I will now describe a second voting scheme in this section, which is the **mixnet-based** variant. I will not present it in detail like the ElGamal homomorphic encryption scheme as it will not be implemented in this thesis project. But as this voting scheme was used in the UCL student elections, it is still interesting to know how it works. The following explanations are based on [13], which explains the workflow of a mixnet use in the *Helios* system, an open-audit web-based voting system used by various institutions in real-stake elections during the last few years.

2.3.1 Mixnet-based elections

Mixnet-based elections offer another approach for the tallying of the election compared to the ElGamal homomorphic encryption voting scheme. They provide **anonymization services** for encrypted ballots, by **shuffling** and **rerandomizing** them in a verifiable way. The anonymized ballots can then be safely decrypted, and the ballot validity verification and counting procedure can be achieved publicly on the decrypted ballots, rather than in the encrypted domain as for the previous voting scheme.

The main advantage of such a scheme is that there is no limitation on the format of the encrypted ballots. The procedure to prepare ballots is therefore much simpler and efficient, which is helpful for voting client that are computationally limited.

However, there is an important limitation from a security point of view. Much more information are revealed about intentions of the voter than in the case of homomorphic tallying techniques. It brings several difficulties :

(i) It is now impossible to implement procedures involving weighting of votes while fully hiding the level of support of candidates as a function of the different weights.

(ii) It increases the risk of coercion compared to approaches that only disclose the election result.

(iii) It requires more care about independent ballot submission. For instance, a voter who submits a (possibly re-randomized) copy of someone else's ballot could then search for two identical decrypted ballots once the tally is completed in order to find out who that voter voted for.

The voting scheme presented below is a variant of the TDH2 encryption scheme of *V. Shoup* and *R. Gennaro* [14]. It preserves submission security and embeds a homomorphic ElGamal ciphertext while preserving the efficiency of the original scheme.
I will detail the voting scheme in three main parts being the election setup, the ballot preparation and submission and finally, the election tallying

2.3.2 Election setup

To start the election, its setup is likely the same as the one of an ElGamal homomorphic voting scheme. The election administrator defines the global public parameters as the election name, description, ID, URL, the questions and lists of candidates, the list of voters, etc. It also includes the description of the cyclic subgroup $\mathbb{G}$ and the generator g. Note that the administrator must also choose to hide or not the names of the voters or to assign them aliases because the ballots will be publicly decrypted at the end of the election during the tallying.

While all these parameters are globally the same than in a homomorphical voting scheme, one setup diverges. The choice of trustees is indeed different, because here we do not only need trustees for the decryption step of the election, but also for the ballots shuffling. In fact, some mixnets offer the possibility to shuffle and decrypt the ballot at the same time, but in this scheme we separate the two concerns :

1. **Shuffling trustees** are responsible of shuffling the ballots. These trustees do not have to *store* any secret, but only to *internally produce* secret data during the shuffling process, as

making random permutations, etc. But the important point is that they must erase these data immediately when their task is completed. These trustees do not need to be determined in advance by the administrator of the election, as they are not required to generate any secret before the election starts.

2. **Decryption trustees** are responsible of decrypting the shuffled ballots. These trustees each hold a share of the private key of the election, like in the ElGamal homomorphic previously explained voting scheme. It implies that they have to be chosen in advance by the administrator of the election, such that they jointly produce the public key of the election which is then added to the global public parameters of the election by the administrator.

However, there are two mandatory requirements :

- The shuffling trustees do not reveal their secret shuffling permutations as it could allow someone to link the submitted ballots to the decrypted ones.
- The decryption trustees do not violate the privacy of the voters by decrypting the ballots before the shuffling step of the election.

In both cases, like in the homomorphical voting scheme, each type of trustees should need to cheat all together, as their all have only a share of the secret. So, if at least one trustee of each group is honest, these two requirements will not be violated.
However, the correctness of the tallying procedure do not relies on the honesty of the trustees, but only on the computational assumptions through the *Soundness* of the ZKPs and the *Collision resistance* of the hash functions that are used to check integrity.

In the following procedure, we keep the freedom to adapt the shuffling procedure on-the-fly as performing a verifiable shuffle is a computationally intensive task. It then lets the possibility to have a set of available servers around the world offering verifiable shuffling services, and let the election organizers query those servers at tallying time, depending on their availability.

2.3.3 Ballot preparation and submission

Once the election is open, the voters can prepare ballots for submission to the voting server. We are going to see that the ballot preparation is quite different that the one used for elections targeting homomorphic tallying. Effectively, the ballots are here simpler and the cryptographic algorithms used for ballot preparation serve different purposes.

Simpler ballots

The ballot format required for ElGamal homomorphic tallying needs to enable everyone to check the validity of all the votes in the encrypted domain, and to compute an encryption of the election outcome. In mixnet based tallying procedure, since all the individual ballots must be publicly decrypted after shuffling, the validity of ballots is checked after decryption, and not at submission time. Moreover, the format of the encrypted votes does not need to allow homomorphic ballot addition.

Even if the decryption of all the individual ballots considerably simplifies the ballot preparation, the ballots need to be decrypted publicly, which even more challenges privacy of the voters. Effectively, their privacy can be compromised through sophisticated attacks. For instance, a dishonest voter could copy ballots submitted by other voters and search for identical ballots after decryption. It should allow him to recognize the copied ballots with high probability when there is a large number of ways to fill in a ballot, like in our UCL student elections (as we have a dozen lists of about ten different candidates each, which makes a hundred of candidates). Additionally, the detection of these copied ballots by some honest observers can be quite difficult. The dishonest voter could, besides simply copying the ballot of someone else, produce different but related ballots

and re-randomize previously submitted ballots.

To avoid this, it is therefore necessary to prevent these types of attacks by making sure the ballots are submitted independently of each other.

Submission Secure Augmented (SSA) cryptosystems

To ensure that this does not happen, we are going to use a Submission Secure Augmented (SSA) cryptosystem. It is an **IND-CCA2 encryption scheme** whose *augmented ciphertexts* contain others *basic ciphertexts* from, generally, a basic homomorphic encryption scheme. A **stripping procedure** must also enable everyone to check the validity of an augmented ciphertext, without compromising the confidentiality of the basic ciphertext.

Definition 2.25 (IND-CCA2 encryption scheme [15]). For a probabilistic asymmetric key encryption algorithm, Indistinguishability under adaptive Chosen Ciphertext Attack (IND-CCA2) is defined by the following game between an adversary $\mathbb{A}$ and a challenger $\mathbb{C}$. $\mathbb{A}$ is modeled by a probabilistic polynomial time *Turing machine*, meaning that it must complete the game and output a guess within a polynomial number of time steps. In this definition :

- $\mathbf{Gen}()$ is a polynomial-time algorithm that generates a *public key* pk and a *private key* sk based on a cyclic subgroup $\mathbb{G} \subset \mathbb{Z}_q^\times$ of prime order r with $||r|| = n$ *bits*.
- $\mathbf{Enc_{pk}(m)}$ is a function that encrypts the *message* m under the public key pk.
- $\mathbf{Dec(c)}$ is an oracle that decrypts an arbitrary *ciphertext* c at the request of $\mathbb{A}$, returning the message m.

The IND-CCA2 encryption scheme is defined by the following game :

1. $\mathbb{C}$ uses $\mathbf{Gen}()$ to generate pk and sk. He publishes pk to $\mathbb{A}$ but keeps sk secret.
2. $\mathbb{A}$ may perform any number of encryptions, or calls to the decryption oracle $\mathbf{Dec}()$ based on arbitrary ciphertexts, or other operations.
3. $\mathbb{A}$ submits two distinct chosen messages m_0 and m_1 to $\mathbb{C}$.
4. $\mathbb{C}$ selects a bit $b \in \{0, 1\}$ uniformly at random, and sends the *challenge* ciphertext $e = \mathbf{Enc_{pk}(m_b)}$ back to $\mathbb{A}$.
5. $\mathbb{A}$ is free to perform any number of additional computations or encryptions, and may make further calls to the decryption oracle $\mathbf{Dec}()$ except submitting the challenge ciphertext e using $\mathbf{Dec(e)}$ (otherwise, the definition would be trivial).
6. Finally, $\mathbb{A}$ outputs a guess for the value of b. If its guess is correct, the *game succeeds*.

A scheme is **IND-CCA2 secure** if any adversary $\mathbb{A}$ has a *negligible advantage* in winning the above game. An adversary is said to have a *negligible advantage* if he wins the above game with probability $\left(\frac{1}{2}\right) + \epsilon(n)$, where $\epsilon(n)$ is a negligible function.

Now, because of the non-malleability of the IND-CCA2 augmented ciphertext, the only possibility to submit a ciphertext that would be related to another one is to submit an exact copy of it, which is easy to detect.
Some trustees might also decide to use the same private key in several contexts (other elections for example). Thereby, we will use an SSA cryptosystem in which the validity verification of a ciphertext includes the verification of a *scope* where that ciphertext can be used. The search of exact copies can then be reduced to ciphertexts that are valid in that specific scope. To this aim, the global public parameters of the election will also be integrated in each ciphertext, to prevent copying a ciphertext from one context to another that uses the same key.

The HTDH2 cryptosystem

The cryptosystem presented in this subsection is a variant of the TDH2 encryption scheme proposed by *V. Shoup* and *R. Gennaro* [14], which is an efficient CCA2 threshold encryption scheme that provides security relying on the hardness of the DDH problem in the random oracle model. A variant is here used because the TDH2 scheme is based on the Hash-ElGamal cryptosystem, which is not homomorphic. Effectively, the ciphertexts have the form :

$$\langle c_1, c_2 \rangle = \langle g^r, m \oplus H\left(h^r\right) \rangle$$

where the hash function H is modeled as a random oracle. This variant cryptosystem is then called HTDH2 because of the "H" emphasizing the presence of a Homomorphic encryption scheme.

Definition 2.26 (**The HTDH2 cryptosystem** [13])**.** Let $\mathbb{G} \subset \mathbb{Z}_q^\times$ be a cyclic subgroup of prime order r with $||r|| = n$ *bits*, and let g and g_0 be two generators of this group. Let m be a message to encrypt, $m \leftarrow \mathbb{Z}_r$.

(i) $\mathbf{Gen}(\mathbf{1^n})$ runs a polynomial-time algorithm to generate $\mathbb{G}, q, g, g_0$ as described above and then chooses a random $z \xleftarrow{\$} \mathbb{Z}_r$ and computes $h = g^z$. It also selects a collision resistant hash function $H : \mathbb{G}^5 \times \{0,1\}^n \to \mathbb{Z}_r$. The *public key* is $pk = \langle H, \mathbb{G}, q, g, g_0, h \rangle$ and the *private key* is $sk = \langle \mathbb{G}, q, g, g_0, z \rangle$.

(ii) $\mathbf{Enc_{pk}}(\mathbf{m}, \mathbf{L})$ with $L \in \{0,1\}^n$ a public label, chooses at random $r, s \xleftarrow{\$} \mathbb{Z}_r$, computes :

1. The *shared secret* $t = h^r$
2. $u = g^r$ and $u_0 = g_0^r$
3. $e = g^m \cdot t$
4. The **commits** $w = g^s$ and $w_0 = g_0^s$
5. The **challenge** $v = H(u, u_0, e, w, w_0, L)$
6. The **response** $f = s + r \cdot v$

and outputs the ciphertext $\langle c_1 \rangle = \langle L, u, u_0, e, v, f \rangle$.

(iii) $\mathbf{Strip_{pk}}(\langle \mathbf{c_1} \rangle)$ computes :

$$w = \frac{g^f}{u^v} \quad \left(= \frac{g^s \cdot g^{rv}}{g^{rv}} = g^s \right)$$
$$w_0 = \frac{g_0^f}{u_0^v} \quad \left(= \frac{g_0^s \cdot g_0^{rv}}{g_0^{rv}} = g_0^s \right)$$

and if $v = H(u, u_0, e, w, w_0, L)$ then outputs $\langle u, e \rangle$.

(iv) $\mathbf{Dec_{sk}}(\langle \mathbf{u}, \mathbf{e} \rangle)$ computes $m := \log_g(e \cdot u^{-z}) \quad \left(= \log_g(e \cdot t^{-1}) = \log_g(g^m) \right)$.

When the ballot of a voter has been correctly encrypted and submitted with this encryption scheme, its hash is displayed on an election board. It allows the voter to check that its vote has correctly been taken into account and that it has not been modified.

2.3.4 Election tallying

Once the ballot submission phase is complete and everyone agrees on the web bulletin board content, the tallying phase starts. First, the validity and uniqueness of all SSA ciphertexts included in the ballots are verified, and the embedded homomorphic ciphertexts are extracted before entering into the shuffling phase.

Shuffling

The shuffling trustees run the shuffle out of their browser, e.g. with a Python script on a global server which allocates the needed computation power. Effectively, the computational load of a mixing server is fairly large, compared to the work required to produce one ballot or to decrypt a single ciphertext.
The mixing server, which is a verifiable re-encryption mixnet, is made of a series of mix servers that each take a sequence of homomorphic ciphertexts as input and compute a shuffled and re-encrypted version of these ciphertexts together with a proof of valid shuffling.

Decryption and ballot counting

Once the decryption trustees agree with the fact that the proper mixing occurred, they compute the decryption of the resulting ciphertexts. This task is more intensive than the decryption phase in an ElGamal homomorphic voting scheme because we need here to decrypt all the shuffled ballots. The decryption trustees also need to compute their share of the decryption of each ballot with the corresponding validity proofs on a Python script on an external server.

Finally, the decryption shares are assembled and combined, allowing everyone to publicly recover the vote of each individual ballot. The validity of each ballot regarding the ballots filling rules can be checked and verified publicly, and the final tally can be computed.

Chapter 3

Implementation aspects

This chapter aims at explaining the design choices made to develop the ElGamal homomorphic encryption voting scheme. The sections of this chapter go through the language and frameworks chosen and also through the optimization tricks developed to improve the performances of the voting scheme. Especially, I will develop how I optimized the computation of modular exponentiations and Zero-Knowledge Proofs.

3.1 Languages and frameworks

An important point to bring up in this thesis is the programming language the voting scheme will be developed with. Two main relevant conditions need to be respected to facilitate the development and the usage of the tool :

- **Ease of access and use** : Whoever wanting to participate to an election must have access to it and use it easily, whatever the personal status of the voter. A website is one of the most common way to use an online tool. The voting scheme can then be developed such that it can run on any browser via a website. It will allow voters to participate to an election with any device connected to the internet and having an integrated web-browser.
- **Computations** : The use of cryptography in a tool results in manipulating *big integers*. These consume more calculation time and memory than normal numbers. The language or the framework used to develop the voting scheme must then be optimized in the use of the memory of a device and the computation time.

JavaScript is a powerful language, very used for years. It has already prove itself, and seemed to be a good choice to develop the voting scheme. But the emergence of new tools got me thinking that the **Rust** language coupled with **WebAssembly** could be a solution that would surpass the JavaScript performances.

3.1.1 JavaScript and Verificatum

To develop a voting scheme in **JavaScript**, I had to choose between several powerful *big integers* libraries :

- **JSBN** [19] : An implementation of large-number math in pure JavaScript, allowing RSA cryptography and Diffie-Hellman key agreement using elliptic curves.
- **bignumber.js** [20] : A JavaScript library that replicates all the basis functions and methods of JavaScript's Number type. It also supports cryptographically-secure pseudo-random number generation and is easy to use.
- **VJSC** or **Verificatum** [21] : A JavaScript library providing the cryptographic routines needed by an electronic voting client. It has all the basic functions and methods needed to manipulate *big integers*, but also has routines to work with elliptic curves, and furthermore a complete Mix-Net (VMN) voting scheme.

By running through the documentation of each of those three libraries, it seemed that **Verificatum** was the most complete and well documented library. It would then be easier to take in charge and use this tool. Moreover, I could have needed to develop a Mix-Net protocol. Effectively, my goal was first to develop the ElGamal homomorphic voting protocol with optimized aspects. And with the results obtained, if these were not interesting or sufficient enough, I could quickly change the focus of my developments to implement a Mix-Net based protocol. In this case, this library could allow me to pivot easily and not restart my developments from the beginning with new tools and libraries.

3.1.2 Rust, WebAssembly and Crates

The **Rust** [22] programming language is designed and developed by *Mozilla Research* [1] since 2010. Rust is more and more used to develop websites since its first stable release in 2015, as it is fast and memory efficient. Effectively, it does not have a runtime environment or garbage collector, it can run in embedded systems, and easily integrate with other languages. Moreover, it is well documented. The main difference with JavaScript is that the memory can be better managed in Rust as variables are type annotated with primitives like signed and unsigned integers with different sizes, going from 8 to 128 bits.

WebAssembly is an open standard of the World Wide Web to develop applications and is used to complete JavaScript with superior performances. As WebAssembly only specifies a low-level programming language, the byte-code is usually produced by compiling a higher level programming language, as Rust. The WebAssembly stack machine is designed to be encoded in a size- and load-time-efficient binary format. I will then use WebAssembly to compile my Rust code in a navigator.

Now that I have chosen the language and the framework I will use to develop my voting scheme, I need to check which library is the best suited to make it. With the help of *Cargo* [2], the Rust package manager, I can browse all public packages and libraries used in Rust. These are called **Crates** [23]. I found different Crates that I could use to manipulate *big integers* :

- **num-bigint** [24] : This crate is developed in pure Rust and allows to manipulate big signed and unsigned integers. It also supports the generation of random big integers when the **rand** crate is enabled. A good point is that it is very well documented.
- **ramp** [26] : It is a high-performance multiple-precision library for working with big numbers. It provides operators overloads for by-value, which consumes the operand(s) and by-reference, which does not. The by-value overloads will attempt to re-use the space for the result. It then allows easy operations for some Rust types such as *i32* and *usize*.

[1] https://research.mozilla.org/
[2] https://doc.rust-lang.org/cargo/index.html

This crate has been designed with high-level code (Rust) and low-level routines. These are predominantly unsafe functions that work with raw pointers, and some of the routines are implemented using inline assembly to gain access to processor-specific functionality.

Unfortunately, due to the use of unstable features such as inline assembly, this crate can only be compiled with a *nightly* (i.e. unstable) *build* of the Rust compiler, which cannot be used in production.

- **rug** [28] : It provides integers and floating-point numbers with arbitrary precision and correct rounding, and is a high-level interface of the GMP [29], MPFR [30] and MPC [31] libraries. The operators are overloaded to work on Rug types alone or on a combination of Rug types and Rust primitives. This ease of use is similar to the one of the **ramp** crate.

As those three crates seemed to fit my needs to manipulate *big integers*, I decided to implement some functions with all those three crates, and then compare their performances in time and memory consumption. The best one should then be the one I would use in the voting scheme. The results of those computations will be detailed in Chapter 4 of this document.

Nevertheless, I already can explain that I finally chose to use the **num-bigint** crate, as my choice was done by removing the two others.

Effectively, the **ramp** crate can only be compiled with a *nightly build* of the Rust compiler, which cannot be used in production. Moreover, it also means that I cannot use this crate with WebAssembly, as it is unstable. This crate will then definitely not be useful in the conception of the voting scheme.

Concerning the **rug** crate, it bundles GMP [29] via the *gmp-mpfr-sys* [32] crate which needs an installation of MSYS2 [33]. Even if the use of the **rug** crate is convoluted, I understood later that it is not compatible with WebAssembly.

3.2 A first optimization – Modular exponentiation

The operation taking the biggest amount of time in the ElGamal homomorphic encryption is the **modular exponentiation**. Effectively, several ones need to be calculated to encrypt a vote, but much more must be calculated to carry out the Zero-Knowledge proofs (ZKP). There are therefore two ways to optimize the computational time compared to the modular exponentiations. On the one hand, we can find the best algorithm to compute one. On the other hand, we can model and rewrite the Zero-Knowledge proofs such that the least possible modular exponentiations are computed in the protocol, and that they run as fast as possible.

This section and the next one aim at explaining these two ways to optimize the ElGamal homomorphic encryption.

3.2.1 How to make a modular exponentiation ?

A **modular exponentiation** is an exponentiation performed over a modulus. This operation calculates the remainder when an integer g (the **base**) raised to a positive integer power e (the **exponent**), g^e, is divided by a positive integer m (the **modulus**). The modular exponentiation a is written :

$$a := g^e \pmod{m}$$

To enhance the reading comfort, we won't write the modulus operator anymore for a modular exponentiation in the following sections, such that

$$g^e \pmod{m} := g^e$$

The most naive way to compute g^e is to do $e-1$ multiplications of g. But when e is large enough, typically 256 or 3072-bit long, it is infeasible for a computer to make $e-1$ multiplications. There is then two ways to reduce the computation time :

- Decrease the time to multiply two elements ;
- Reduce the number of multiplications.

A proper manner would be to do both.

Actually, we consider here three types of exponentiation algorithms, all taken from the *Handbook of Applied Cryptography* [1] :

1. **Basic techniques for exponentiation** : Useful when you have arbitrary choices of the base g and the exponent e.
2. **Fixed-exponent exponentiation algorithms** : The exponent e is fixed and the base g can be chosen arbitrary.
3. **Fixed-base exponentiation algorithms** : Contrarily, the base g is fixed and the exponent e can be chosen arbitrary.

The last type of algorithms will be the one we should normally use in our voting scheme, as we need to make modular exponentiations with a fixed base g being the generator of a cyclic group G. Thereby, we will not go through the second type of algorithms in this thesis as the goal is opposite to the one we expect here. We will then compare basic techniques with fixed-base exponentiation algorithms to retain the one whose results show the best compromise between memory usage and computational time.

3.2.2 Basic techniques for exponentiation

The most basic technique to compute a modular exponentiation is the *Left-to-right binary exponentiation*, detailed in Algorithm 1. The principle is to loop through the decomposition of the exponent e in binary format and to multiply the result A by the i times squared basis g when the bit e_i of e is set to 1. We denote here t as being the bitlength of the binary representation of e.

Algorithm 1 HAC - 14.79 : Left-to-right binary exponentiation

Input: g and $e = (e_{t-1}, \ldots, e_1, e_0)_2$
Output: g^e

```
1: function COMPUTE(g, e)
2:     A ← 1
3:     for i ← 0 to t do
4:         if e_i = 1 then
5:             A ← A · g
6:         end if
7:         g ← g · g
8:     end for
9:     return A
10: end function
```

As an example, if we want to compute g^{141}, as $141 = (10001101)_2$ with $t = 8$, we have :

i	0	1	2	3	4	5	6	7
e_i	1	0	1	1	0	0	0	1
A	g	g	g^5	g^{13}	g^{13}	g^{13}	g^{13}	g^{141}

We can improve this first algorithm by decomposing the exponent e in a base b greater than 2. The *Left-to-right k-ary exponentiation* detailed in ALGORITHM 2 applies it such that it processes more than one bit of the exponent per iteration.

An important observation to note is that the algorithm is here divided in two functions, the **pre-computation** and the **computation**. The goal is to precompute some values only once and store them, such that they can be re-used to make as much modular exponentiations as desired.

In this algorithm, the parameter k can be tuned to change the parameter $b := 2^k$. And, by running through the bit representation of e in base b, we multiply the result A by the precomputations B only when the bit e_i is non-null.

Algorithm 2 HAC - 14.82 : Left-to-right k-ary exponentiation

Input: g and $e = (e_{n-1}, \ldots, e_1, e_0)_b$, where $n = \frac{t}{k}$ and $b = 2^k$ for some $k \geq 1$
Output: g^e

```
1: function PRECOMPUTE(g, k)
2:     B_0 ← 1
3:     B_1 ← g
4:     for i ← 2 to 2^k do
5:         B_i ← B_{i−1} · g
6:     end for
7:     return B
8: end function

9: function COMPUTE(e, k, B)
10:    A ← 1
11:    for i ← n to 0 do
12:        for j ← 0 to k do
13:            A ← A · A
14:        end for
15:        if e_i ≠ 0 then
16:            A ← A · B_{e_i}
17:        end if
18:    end for
19:    return A
20: end function
```

A *Modified left-to-right k-ary exponentiation* presented in ALGORITHM 3 improves the last algorithm by changing the condition of multiplying the result A by the precomputations B in terms of the bit e_i. The condition becomes :

$$\forall i \in \{0, \ldots, n\} \quad \text{if} \quad \begin{cases} e_i \neq 0 \quad \text{then} \quad \begin{cases} e_i = 2^{h_i} u_i \\ u_i \pmod 2 = 1 \end{cases} \\ e_i = 0 \quad \text{then} \quad h_i = u_i = 0 \end{cases}$$

The consequence of changing this condition is the reduction of the amount of precomputations.

Algorithm 3 HAC - 14.83 : Modified left-to-right k-ary exponentiation

Input: g and $e = (e_{n-1}, \ldots, e_1, e_0)_b$, where $n = \frac{t}{k}$ and $b = 2^k$ for some $k \geq 1$
Output: g^e

```
1:  function PRECOMPUTE(g, k)
2:      B_0 ← 1
3:      B_1 ← g
4:      B_2 ← g · g
5:      for i ← 1 to 2^{k−1} do
6:          B_{2i+1} ← B_{2i−1} · B_2
7:      end for
8:      return B
9:  end function

10: function GETUH(u, h, x, i, wordsize)
11:     bitIndex ← i * wordsize
12:     u ← 0
13:     for j ← 0 to wordsize do
14:         b ← x_{bitIndex}
15:         u ← u | b << j
16:         bitIndex ← bitIndex + 1
17:     end for
18:     h ← 0
19:     if u ≠ 0 then
20:         while u & 1 = 0 do
21:             u ← u >> 1
22:             h ← h + 1
23:         end while
24:     end if
25:     return (u, h)
26: end function

27: function COMPUTE(e, t, k, B)
28:     A ← 1
29:     n ← (t+k−1)/k
30:     for i ← n to 0 do
31:         (u, h) ← GETUH(u, h, e, i, k)
32:         for j ← 0 to k − h do
33:             A ← A · A
34:         end for
35:         if u ≠ 0 then
36:             A ← A · B_u
37:         end if
38:         for j ← 0 to h do
39:             A ← A · A
40:         end for
41:     end for
42:     return A
43: end function
```

Another approach would be to use the **Sliding-window exponentiation** explained in ALGORITHM 4. Here, the exponent e is used in binary representation as in ALGORITHM 1. The parameter k, called the *window size*, can be tuned to reduce the amount of precomputations and the average number of multiplications performed.

Algorithm 4 HAC - 14.85 : Sliding-window exponentiation

Input: g and $e = (e_{t-1}, \ldots, e_1, e_0)_2$, with $e_{t-1} = 1$ and some $k \geq 1$
Output: g^e

```
1: function PRECOMPUTE(g, k)
2:     B_0 ← 1
3:     B_1 ← g
4:     B_2 ← g · g
5:     for i ← 1 to 2^{k-1} do
6:         B_{2i+1} ← B_{2i-1} · B_2
7:     end for
8:     return B
9: end function

10: function COMPUTE(e, t, k, B)
11:     A ← 1
12:     i ← t − 1
13:     while i ≥ 0 do
14:         if e_i = 0 then
15:             A ← A · A
16:             i ← i − 1
17:         else
18:             l ← max(i − k + 1, 0)
19:             while e_l = 0 do
20:                 l ← l + 1
21:             end while
22:             for j ← 0 to i − l + 1 do
23:                 A ← A · A
24:             end for
25:             u ← 0
26:             for j ← i to l do
27:                 u ← (u << 1) | e_j
28:             end for
29:             A ← A · B_u
30:             i ← l − 1
31:         end if
32:     end while
33:     return A
34: end function
```

As an example, if we take $e = 17637 = (100010011100101)_2$ and $k = 3$, we have $t = 15$. The precomputations table B looks like :

i	0	1	2	3	4	5	6	7
B	1	g	g^2	g^3	0	g^5	0	g^7

The steps of the `COMPUTE` function are detailed in the following table :

i	A	Longest bitstring s.t. $i - l + k \leq k$	l
14	1	**1** 00010011100101	14
13	$(1)^2 \cdot B_1 = g$	–	–
12	g^2	–	–
11	$(g^2)^2 = g^4$	–	–
10	$(g^4)^2 = g^8$	1000 **1** 0011100101	10
9	$(g^8)^2 \cdot B_1 = g^{17}$	–	–
8	$(g^{17})^2 = g^{34}$	–	–
7	$(g^{34})^2 = g^{68}$	1000100 **111** 00101	5
4	$(g^{68})^8 \cdot B_7 = g^{551}$	–	–
3	$(g^{551})^2 = g^{1102}$	–	–
2	$(g^{1102})^2 = g^{2204}$	100010011100 **101**	0
-1	$(g^{2204})^8 \cdot B_5 = g^{17637}$	–	–

3.2.3 Fixed-base exponentiation algorithms

Similarly to ALGORITHM 1, the most basic technique to compute a modular exponentiation is to loop through the binary decomposition of the exponent e and multiply the result A by the i times squared base g when the bit e_i of e is set to 1. As we only consider in this section the exponentiations with a fixed base g, we can precompute the i times squared base g and store those values in a vector B. This improvement is detailed in ALGORITHM 5.

Algorithm 5 Fixed-base exponentiation

Input: g and $e = (e_{t-1}, \ldots, e_1, e_0)_2$
Output: g^e

```
1: function PRECOMPUTE(g, t)
2:     B_0 ← g
3:     for i ← 1 to t do
4:         B_i ← B_{i−1} · B_{i−1}
5:     end for
6:     return B
7: end function

8: function COMPUTE(e, t, B)
9:     A ← 1
10:    for i ← 0 to t do
11:        if e_i = 1 then
12:            A ← A · B_i
13:        end if
14:    end for
15:    return A
16: end function
```

As an example to better visualize the above algorithm, the following tables detail the steps for $e = 5349 = (1010011100101)_2$, such that $t = 13$:

PRECOMPUTE													
i	0	1	2	3	4	5	6	7	8	9	10	11	12
B	g	g^2	g^4	g^8	g^{16}	g^{32}	g^{64}	g^{128}	g^{256}	g^{512}	g^{1024}	g^{2048}	g^{4096}

COMPUTE		
i	A	$(e_{t-1}, \ldots, e_1, e_0)_2$
0	$1 \cdot B_0 = g$	101001110010 **1**
2	$g \cdot B_2 = g^5$	1010011100 **1** 01
5	$g^5 \cdot B_5 = g^{37}$	1010011 **1** 00101
6	$g^{37} \cdot B_6 = g^{101}$	101001 **1** 100101
7	$g^{101} \cdot B_7 = g^{229}$	10100 **1** 1100101
10	$g^{229} \cdot B_{10} = g^{1253}$	10 **1** 0011100101
12	$g^{1253} \cdot B_{12} = g^{5349}$	**1** 010011100101

Similarly to Algorithm 4, fixed-base exponentiations can be improved by integrating a *window size* k, such that e is decomposed in a base $b := 2^k$. We then obtain the Algorithm 6. We can observe here that the precomputations are not a vector anymore but a matrix. We then increase the number of precomputations to decrease the average number of multiplications in the computational function.

Algorithm 6 HAC - 14.109 : Fixed-base windowing method for exponentiation

Input: g and $e = (e_{n-1}, \ldots, e_1, e_0)_b$ with $b = 2^k$ and $n = \frac{t}{k}$ for some $k \geq 1$
Output: g^e

```
1:  function PRECOMPUTE(g, n, k)
2:      B_{0,0} ← g
3:      for j ← 1 to 2^k − 1 do
4:          B_{0,j} ← B_{0,j−1} · g
5:      end for
6:      for i ← 1 to n do
7:          for j ← 0 to 2^k − 1 do
8:              for k ← 0 to 2^k do
9:                  B_{i,j} ← B_{i,j} · B_{i−1,j}
10:             end for
11:         end for
12:     end for
13:     return B
14: end function

15: function COMPUTE(e, n, B, k)
16:     A ← 1
17:     for i ← 0 to n do
18:         if e_i ≠ 0 then
19:             A ← A · B_{i,e_i−1}
20:         end if
21:     end for
22:     return A
23: end function
```

We can illustrate this windowing algorithm with the same previous example by taking $k = 3$ and $e = 5349 = (1010011100101)_2 = (12345)_8$, such that $b = 8$, $t = 13$ and $n = 5$. The following is the precomputation matrix of this Algorithm 6. It is constructed such that each line is a 2^k exponentiation of the previous one :

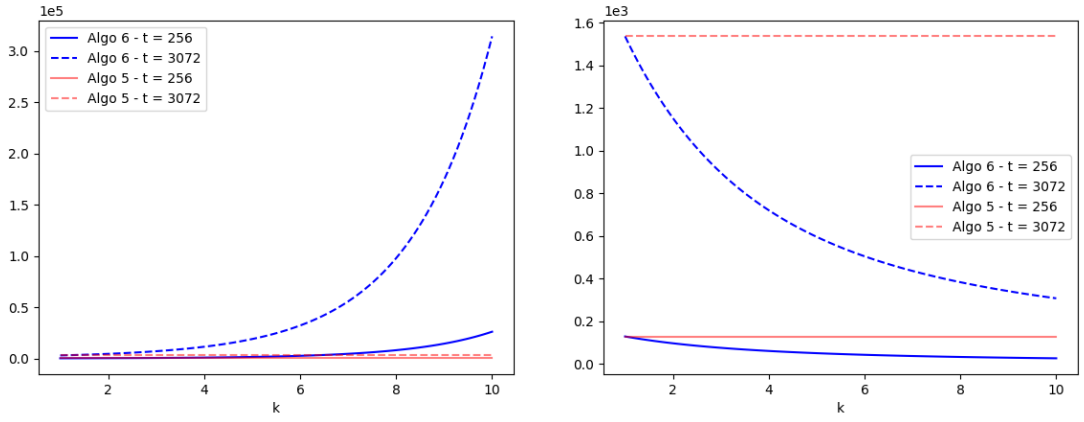
$$B_{i,j} = \begin{bmatrix} g & g^2 & g^3 & g^4 & g^5 & g^6 & g^7 \\ g^8 & (g^2)^8 = g^{16} & (g^3)^8 = g^{24} & (g^4)^8 = g^{32} & (g^5)^8 = g^{40} & (g^6)^8 = g^{48} & (g^7)^8 = g^{56} \\ g^{64} & g^{128} & g^{192} & g^{256} & g^{320} & g^{384} & g^{448} \\ g^{512} & g^{1024} & g^{1536} & g^{2048} & g^{2560} & g^{3072} & g^{3584} \\ g^{4096} & g^{8192} & g^{12288} & g^{16384} & g^{20480} & g^{24576} & g^{28672} \end{bmatrix}$$

The steps of the `COMPUTE` function are detailed in the following table :

`COMPUTE`		
i	A	$(e_{n-1}, \ldots, e_1, e_0)_b$
0	$1 \cdot B_{0,4} = g^5$	1234 **5**
1	$g^5 \cdot B_{1,3} = g^{37}$	123 **4** 5
2	$g^{37} \cdot B_{2,2} = g^{229}$	12 **3** 45
3	$g^{229} \cdot B_{3,1} = g^{1253}$	1 **2** 345
4	$g^{1253} \cdot B_{4,0} = g^{5349}$	**1** 2345

We can observe that from the Algorithm 5 to the Algorithm 6, the number of multiplications in the `COMPUTE` functions decreased from 7 to 5. In fact, the parameter k can be tuned to optimize the trade-off between memory usage and computational time. Effectively, when k increases, the number of precomputed values (– the memory usage) increases whereas the average number of multiplications in the `COMPUTE` function (– the computational time) decreases.

The Figure 3.1 displays a comparison between the Algorithms 5 and 6 in terms of, respectively, their number of precomputed values in the `PRECOMPUTE` function and their number of multiplications computed in the `COMPUTE` function. We can observe that the memory consumption increases exponentially with k and the number of bits t for Algorithm 6, while it is linear with t for Algorithm 5. Moreover, the average number of multiplications decreases exponentially with k and the number of bits t for Algorithm 6, while it is also linear with t for Algorithm 5.



(a) `# precomputed values` (b) `# multiplications computed`

Figure 3.1: Comparison of Algorithms 5 and 6 in terms of k and t

We will now focus on the ALGORITHM 7 which makes use of the Euclidean method [18]. Effectively, the goal is to recursively use the following equality to calculate x^e :

$$x_0^{e_0} \cdot x_1^{e_1} = (x_0 \cdot x_1^q)^{e_0} \cdot x_1^{e_1} \pmod{e_0}$$

where $q = \lfloor \frac{e_1}{e_0} \rfloor$, which is a Euclidean division of the exponent e_1 by e_0 used to return a quotient q and a rest $e_1 \pmod{e_0}$.

The steps of the algorithm are :

1. Precompute $B_i = g^{(2^k)^i}$
2. Find x_M and x_N, the two largest bit values in the copy x of the decomposition of the exponent e in a base $b := 2^k$. The algorithm details it in several steps :
   - Initialize x as a copy of e
   - Find x_M being the maximum value with index M in $(x_{n-1}, \ldots, x_1, x_0)_b$
   - Initialize y as a copy of x
   - Set y_M to 0
   - Find y_N being the maximum value with index N in $(y_{n-1}, \ldots, y_1, y_0)_b$
3. While $x_N \neq 0$:
   3.1. Set $q = \lfloor \frac{x_M}{x_N} \rfloor$
   3.2. Set $B_N = (B_M)^q \cdot B_N$
   3.3. Set $x_M = x_M \pmod{x_N}$
   3.4. Find x_M and x_N, the two largest bit values in x
4. Return $B_M^{x_M}$

The following tables illustrate this algorithm with the same previous example $k = 3$ and $e = 5349 = (1010011100101)_2 = (12345)_8$, such that $b = 8$, $t = 13$ and $n = 5$.

i	0	1	2	3	4
B	g	g^8	g^{64}	g^{512}	g^{4096}

x	x_M	x_N	M	N	q	B_0	B_1	B_2	B_3	B_4
1 2 3 4 5	5	4	0	1	1	g	g^9	g^{64}	g^{512}	g^{4096}
1 2 3 4 1	4	3	1	2	1	g	g^9	g^{73}	g^{512}	g^{4096}
1 2 3 1 1	3	2	2	3	1	g	g^9	g^{73}	g^{585}	g^{4096}
1 2 1 1 1	2	1	3	0	2	g^{1171}	g^9	g^{73}	g^{585}	g^{4096}
1 0 1 1 1	1	1	0	1	1	g^{1171}	g^{1180}	g^{73}	g^{585}	g^{4096}
1 0 1 1 0	1	1	1	2	1	g^{1171}	g^{1180}	g^{1253}	g^{585}	g^{4096}
1 0 1 0 0	1	1	2	4	1	g^{1171}	g^{1180}	g^{1253}	g^{585}	g^{5349}
1 0 0 0 0	1	0	4	0	–	g^{1171}	g^{1180}	g^{1253}	g^{585}	g^{5349}

Algorithm 7 HAC - 14.113 : Fixed-base Euclidean method for exponentiation

Input: g and $e = (e_{n-1}, \ldots, e_1, e_0)_b$ with $b = 2^k$ and $n = \frac{t}{k}$ for some $k \geq 1$
Output: g^e

1: **function** PRECOMPUTE(g, n, k)
2: $B_0 \leftarrow g$
3: **for** $i \leftarrow 1$ to n **do**
4: $B_i \leftarrow B_{i-1}$
5: **for** $j \leftarrow 1$ to 2^k **do**
6: $B_i \leftarrow B_i \cdot B_{i-1}$
7: **end for**
8: **end for**
9: **return** B
10: **end function**

11: **function** COMPUTE(e, n, B, k)
12: $x \leftarrow e$
13: $M \leftarrow i \mid x_i = \max\{(x_{n-1}, \ldots, x_1, x_0)_{2^k}\}$
14: $y \leftarrow x$
15: $y_M \leftarrow 0$
16: $N \leftarrow i \mid y_i = \max\{(y_{n-1}, \ldots, y_1, y_0)_{2^k}\}$
17: **while** $x_N \neq 0$ **do**
18: $q \leftarrow \lfloor \frac{x_M}{x_N} \rfloor$
19: $bn \leftarrow B_N$
20: $B_N \leftarrow B_M$
21: **for** $i \leftarrow 0$ to $q - 1$ **do**
22: $B_N \leftarrow B_N \cdot B_M$
23: **end for**
24: $B_N \leftarrow B_N \cdot bn$
25: $x_M \leftarrow x_M \mod x_N$
26: $M \leftarrow i \mid x_i = \max\{(x_{n-1}, \ldots, x_1, x_0)_{2^k}\}$
27: $y \leftarrow x$
28: $y_M \leftarrow 0$
29: $N \leftarrow i \mid y_i = \max\{(y_{n-1}, \ldots, y_1, y_0)_{2^k}\}$
30: **end while**
31: $A \leftarrow B_M$
32: **for** $i \leftarrow 0$ to $x_M - 1$ **do**
33: $A \leftarrow A \cdot B_M$
34: **end for**
35: **return** A
36: **end function**

Algorithm 8 HAC - 14.117 : Fixed-base comb method for exponentiation

Input: g and $e = (e_{t-1}, \ldots, e_1, e_0)_2$
Output: g^e

1: **function** PRECOMPUTE(g, t)
2: $h \leftarrow h \in [1, t], h \in \mathbb{N}$
3: $a \leftarrow \lceil \frac{t}{h} \rceil$
4: $v \leftarrow v \in [1, a], v \in \mathbb{N}$
5: $b \leftarrow \lceil \frac{a}{v} \rceil$
6: $B_0 \leftarrow g$
7: $B_1 \leftarrow g$
8: **for** $j \leftarrow 0$ to a **do**
9: $B_1 \leftarrow B_1 \cdot B_1$
10: **end for**
11: **for** $i \leftarrow 2$ to h **do**
12: $B_i \leftarrow B_{i-1}$
13: **for** $j \leftarrow 0$ to a **do**
14: $B_i \leftarrow B_i \cdot B_i$
15: **end for**
16: **end for**
17: **for** $i \leftarrow 1$ to 2^h **do**
18: $ibytes \leftarrow (i_{h-1}, \ldots, i_1, i_0)_2$
19: **for** $j \leftarrow 0$ to h **do**
20: **if** $j < |ibytes|$ and $ibytes_j = 1$ **then**
21: $w \leftarrow B_j$
22: **else**
23: $w \leftarrow 1$
24: **end if**
25: $G_{0,i-1} \leftarrow G_{0,i-1} \cdot w$
26: **end for**
27: **if** $v > 1$ **then**
28: $G_{1,i-1} \leftarrow G_{0,i-1}$
29: **for** $j \leftarrow 0$ to b **do**
30: $G_{1,i-1} \leftarrow G_{1,i-1} \cdot G_{1,i-1}$
31: **end for**
32: **end if**
33: **for** $j \leftarrow 2$ to v **do**
34: $G_{j,i-1} \leftarrow G_{j-1,i-1}$
35: **for** $k \leftarrow 0$ to b **do**
36: $G_{j,i-1} \leftarrow G_{j,i-1} \cdot G_{j,i-1}$
37: **end for**
38: **end for**
39: **end for**
40: **return** (G, a, b)
41: **end function**

```
42: function COMPUTE(e, t, h, a, b, v, G)
43:     while |e| < a * h do
44:         e_{|e|} ← 0
45:     end while
46:     for i ← 0 to h do
47:         for j ← 0 to a do
48:             I_{i,j} ← e_{a*i+j}
49:         end for
50:     end for
51:     A ← 1
52:     for k ← b − 1 to 0 do
53:         A ← A · A
54:         for j ← v − 1 to 0 do
55:             x ← 0
56:             for i ← h − 1 to 0 do
57:                 if j * b + k ≥ a then
58:                     x_{|x|} ← 0
59:                 else
60:                     x_{|x|} ← I_{i,j*b+k}
61:                 end if
62:             end for
63:             if x ≠ 0 then
64:                 A ← G_{j,x−1} · A
65:             end if
66:         end for
67:     end for
68:     return A
69: end function
```

The last but not least algorithm we will run through in this paper is the **Fixed-base comb method** detailed in ALGORITHM 8. The precomputation part of this method computes a matrix G with much more values than the previous algorithms, but it allows to compute any modular exponentiation with base g faster than any of them.

Here, the exponent e is used in its binary representation $e = (e_{t-1}, \ldots, e_1, e_0)_2$. The algorithm can be tuned in terms of the two parameters h and v. The integer h must be chosen between 1 and t while the integer v must be between 1 and $a := \lceil \frac{t}{h} \rceil$.

The `PRECOMPUTE` function first builds a vector B of size h which entries are $B_i = g^{2^{ia}}$ in order to build and output a matrix $G_{i,j}$ of size $(v \times 2^h - 1)$. The entries of this matrix are built such that :

1. The entries of the first row are the product of each entry B_j exponent the number i_j, which is the bit at index j in the binary representation of number i where i is the index of the column entry being calculated. We have then :

$$G_{0,i} = \prod_{j=0}^{h-1} B_j^{i_j}$$

   where i_j is the element at index j in $i = (i_{h-1}, \ldots, i_1, i_0)_2$.

2. The j next rows are the first row exponent 2^{jb} such that :

$$G_{j,i} = G_{0,i}^{2^{jb}}$$

Afterwards, the `COMPUTE` function first adds, if necessary, 0's to the left of e such that the bitlength of e is $a * h$. e has then the form $e = (0_{ah-1}, \ldots, 0_t, e_{t-1}, \ldots, e_1, e_0)_2$. The goal is to form an array

$I_{i,j}$ of size $h \times a$ called the *exponent array*. It is constructed such that each row i contains the i^{th} bitstring of length a of the exponent e :

$$I_{i,j} = \begin{bmatrix} e_0 & e_1 & \cdots & e_{a-1} \\ e_a & e_{a+1} & \cdots & e_{2a-1} \\ \vdots & \vdots & \ddots & \vdots \\ e_{a*(h-1)} & e_{a*(h-1)+1} & \cdots & e_{ah-1} \end{bmatrix}$$

Then, the result A is calculated by applying the following steps :

For every k from $b-1$ down to 0:

1. $A = A^2$
2. For j from $v-1$ down to 0:
   2.1. Set x as being the number formed by the bits of e in the column k of I :
   $$x = (I_{h-1,k}\ , \ldots,\ I_{0,k})_2$$
   2.2. $A = A \cdot G_{j,x-1}$

The following tables illustrate the ALGORITHM 8 with the same previous example $e = 5349 = (1010011100101)_2$, such that $t = 13$. I picked here $h = 3$ and $v = 2$, giving $a = \lceil \frac{13}{3} \rceil = 5$ and $b = \lceil \frac{5}{2} \rceil = 3$:

$$B_i = \begin{bmatrix} g & g^{32} & g^{1024} \end{bmatrix} \quad ; \quad I_{i,j} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$\mathbf{G_{i,j}}$							
$i \setminus j$	0	1	2	3	4	5	6
0	B_0	B_1	$B_0\ B_1$	B_2	$B_0\ B_2$	$B_1\ B_2$	$B_0\ B_1\ B_2$
1	B_0^8	B_1^8	$B_0^8\ B_1^8$	B_2^8	$B_0^8\ B_2^8$	$B_1^8\ B_2^8$	$B_0^8\ B_1^8\ B_2^8$

COMPUTE			
k	j	x	A
2	–	–	1
2	1	$(0,0,0)_2 = 0$	1
2	0	$(1,1,1)_2 = 7$	$1 \cdot \mathbf{G_{0,6}} = B_0\ B_1\ B_2$
2	–	–	$B_0^2\ B_1^2\ B_2^2$
1	1	$(0,0,0)_2 = 0$	$(B_0^2\ B_1^2\ B_2^2)$
1	0	$(0,1,0)_2 = 2$	$(B_0^2\ B_1^2\ B_2^2) \cdot \mathbf{G_{0,1}} = B_0^2\ B_1^3\ B_2^2$
1	–	–	$B_0^4\ B_1^6\ B_2^4$
0	1	$(0,0,0)_2 = 0$	$(B_0^4\ B_1^6\ B_2^4)$
0	0	$(1,1,1)_2 = 7$	$(B_0^4\ B_1^6\ B_2^4) \cdot \mathbf{G_{0,6}} = B_0^5\ B_1^7\ B_2^5$

$$\begin{aligned} B_0^5\ B_1^7\ B_2^5 &= g^5\ (g^{32})^7\ (g^{1024})^5 \\ &= g^{5349} \end{aligned}$$

3.2.4 Theoretical analysis of the algorithms

The TABLE 3.1 refers, for ALGORITHMS 1 TO 8, the number of multiplications in each of the `Precomputation` and `Computation` phase and the number of values stored in memory in the `Precomputation` phase.

	Precomputation		**Computation**
	# multiplication	# values in memory	# multiplication
`Algorithm 1` (HAC - 14.79)	–	–	$\frac{3t}{2}$
`Algorithm 2` (HAC - 14.82)	$2^k - 2$	2^k	$\frac{t}{k}\left(1 - \frac{1}{2^k}\right) + t$
`Algorithm 3` (HAC - 14.83)	$\frac{2^k}{2} - 1$	2^k	$\frac{t}{k}\left(1 - \frac{1}{2^k}\right) + t$
`Algorithm 4` (HAC - 14.85)	$\frac{2^k}{2} - 1$	2^k	Difficult to analyze
`Algorithm 5` (Fixed-base exp)	$t - 1$	t	$\frac{t}{2}$
`Algorithm 6` (HAC - 14.109)	$(2^k - 1) \cdot \left[\frac{2^k - 2}{2} + 2^k\left(\frac{t}{k} - 1\right)\right]$	$\frac{t}{k}(2^k - 1)$	$\frac{t}{k}\left(1 - \frac{1}{2^k}\right)$
`Algorithm 7` (HAC - 14.113)	$(2^k - 1) \cdot \left(\frac{t}{k} - 1\right)$	$\lceil \frac{t}{k} \rceil$	$\leq 2\ \frac{t}{k} + (2^k - 1)$
`Algorithm 8` (HAC - 14.117)	$(h-1)\ a + (2^h - 1) \cdot [h + (v-1)\ b]$	$v(2^h - 1)$	$b\ (v+1)$

Table 3.1: Theoretical analysis of the ALGORITHMS 1 TO 8

An important point to explain concerns the choice of the adaptable parameters k for ALGORITHMS 2 TO 7 and parameters (h, v) for the ALGORITHM 8. If we refer to the expressions given for those algorithms in TABLE 3.1, we can compute the best k for the 6 first algorithms by analytically compute the derivative under k of the expressions and equal it to 0.

To write that down, I need to define 2 more parameters, namely :

- n, the number of modular exponentiations to compute with the same precomputation values ;
- z, the maximal number of values that can be stored in memory ;

The following equations compute the derivative under k of the sum of the precomputations and n times the computations. For each one, I give the best rounded k for $t = 256$ and $n = 2000$:

`Algorithm 2`

$$\begin{aligned} &\frac{\partial}{\partial k}\left\{2^k - 2 + n * \left[\frac{t}{k}\left(1 - \frac{1}{2^k}\right) + t\right]\right\} && = 0 \\ \Leftrightarrow\ & n\left[\frac{2^{-k}\ t\ \log(2)}{k} - \frac{(1 - 2^{-k})\ t}{k^2}\right] + 2^k\ \log(2) && = 0 \qquad \underset{t,n}{\Rightarrow}\ k = \lfloor 12.26 \rceil = 12 \end{aligned}$$

`Algorithm 3`

$$\begin{aligned} &\frac{\partial}{\partial k}\left\{\frac{2^k}{2} - 1 + n * \left[\frac{t}{k}\left(1 - \frac{1}{2^k}\right) + t\right]\right\} && = 0 \\ \Leftrightarrow\ & n\left[\frac{2^{-k}\ t\ \log(2)}{k} - \frac{(1 - 2^{-k})\ t}{k^2}\right] + 2^{k-1}\ \log(2) && = 0 \qquad \underset{t,n}{\Rightarrow}\ k = \lfloor 13.07 \rceil = 13 \end{aligned}$$

`Algorithm 4` is difficult to analyze and `Algorithm 5` does not contain any k parameter.
`Algorithm 6`

$$\begin{aligned}
&\frac{\partial}{\partial k}\left\{(2^k-1)\cdot\left[\frac{2^k-2}{2}+2^k\left(\frac{t}{k}-1\right)\right]+n*\left[\frac{t}{k}\left(1-\frac{1}{2^k}\right)\right]\right\} && = 0\\
\Leftrightarrow\ & n\left[\frac{2^{-k}\ t\ \log(2)}{k}-\frac{(1-2^{-k})\ t}{k^2}\right]\\
&\quad+\left(2^k-1\right)\left[-\frac{2^k t}{k^2}+2^k\ \log(2)\left(\frac{t}{k}-1\right)+2^{k-1}\ \log(2)\right]\\
&\quad+2^k\ \log(2)\left[2^k\left(\frac{t}{k}-1\right)+\frac{2^k-2}{2}\right] && = 0 \quad \underset{t,n}{\Rightarrow}\ k=\lfloor 4.20\rceil = 4
\end{aligned}$$

`Algorithm 7`

$$\begin{aligned}
&\frac{\partial}{\partial k}\left\{(2^k-1)\cdot\left(\frac{t}{k}-1\right)+n*\left[2\ \frac{t}{k}+(2^k-1)\right]\right\} && = 0\\
\Leftrightarrow\ & n\left(2^k\ \log(2)-\frac{2t}{k^2}\right)-\frac{(2^k-1)t}{k^2}+2^k\ \log(2)\left(\frac{t}{k}-1\right) && = 0 \quad \underset{t,n}{\Rightarrow}\ k=\lfloor 4.91\rceil = 5
\end{aligned}$$

Concerning Algorithm 8, we can try to analyze which choice could be the best for both parameters h and v. This is very important, as the algorithm can be really inefficient if they are badly picked. For example, the Table 3.2 gives 2 choices of the parameters h and v that each minimize one of the expression of Table 3.1. For the first example, there are huge values that are unfeasible for a normal computer to manage in a reasonable time frame. The second example seems feasible, but the total amount of multiplications for 2000 computation of modular exponentiations seems big. We definitely can do better.

h	v	a	b	**Precomputation**		**Computation**	Total # multiplications for 2000 computations
				# multiplication	# values in memory	# multiplication	
256	1	1	1	$2,964\cdot 10^{79}$	$1,158\cdot 10^{77}$	2	$2,964\cdot 10^{79}$
1	1	256	256	1	1	512	$1,024\cdot 10^{6}$

Table 3.2: Examples of bad choices of parameters h and v for Algorithm 8

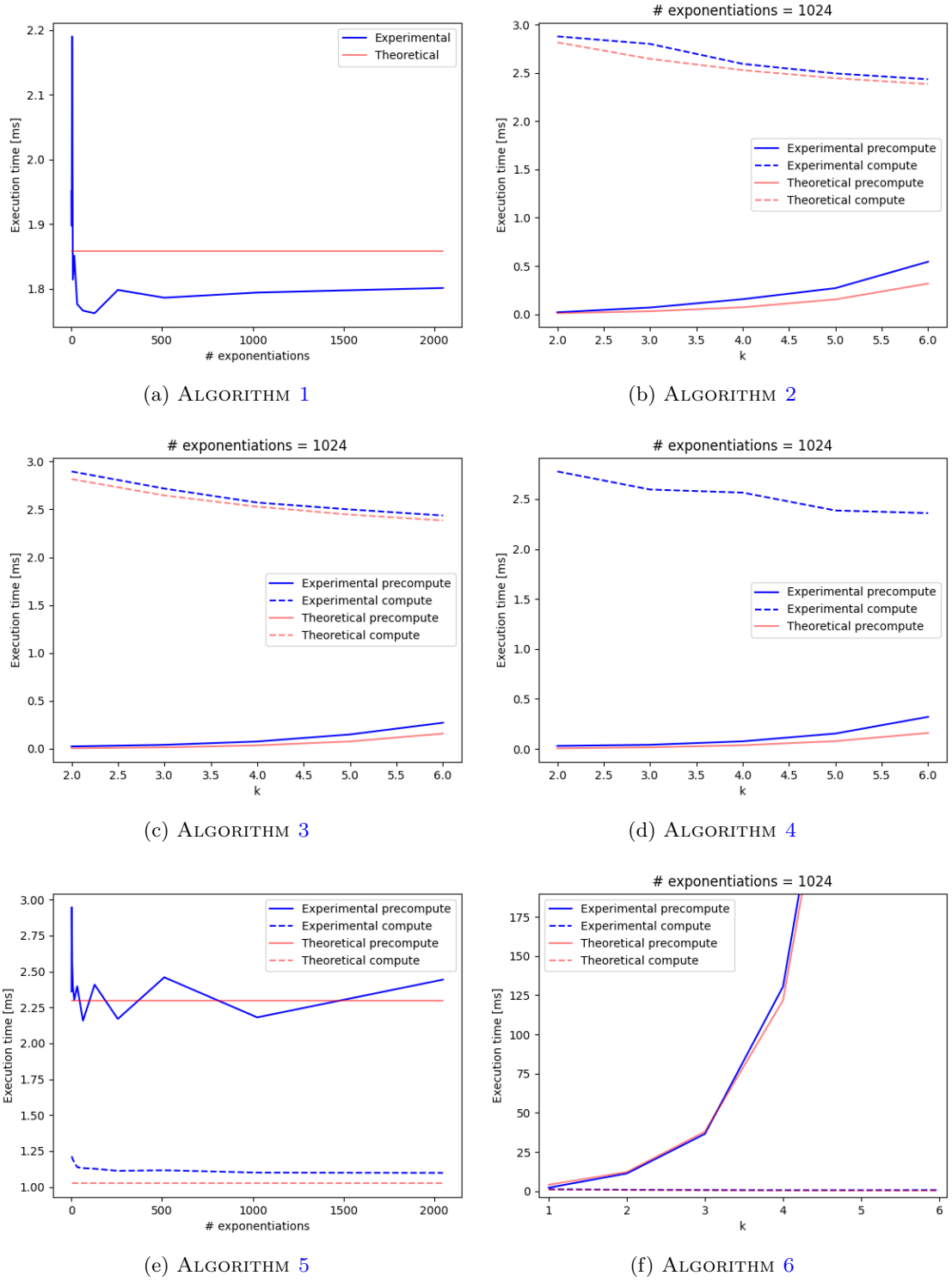
We have to express an optimization problem. By using n and z defined above, the non-linear optimization problem can then be expressed :

$$\begin{aligned}
\underset{h,v}{\text{minimize}}\quad & (h-1)\ a+(2^h-1)\cdot[h+(v-1)\ b]+n\ b\ (v+1)\\
& h\le t && \forall h\in\mathbb{N}^+\\
& a=\left\lceil\frac{t}{h}\right\rceil\\
& v\le a && \forall v\in\mathbb{N}^+\\
& b=\left\lceil\frac{a}{v}\right\rceil\\
& v(2^h-1)\le z && \forall z\in\mathbb{N}^+
\end{aligned}$$

I have not been through the resolution of this problem as this is out of the scope of this thesis subject. As we will work in our voting scheme with 256-bits exponents, I instead made a little Excel program that displays all the 1713 possible values of h and v for $t = 256$. Afterwards, I sorted them in terms of the value of the objective function of the above problem for $n = 2000$. Finally, I took the 100 first best values of h and v and run them on my computer. I then retained the 10 best computational times and checked the amount of memory usage of each. As a result, the best trade-off between computational time and memory usage was $h = 6$ and $v = 9$.

3.2.5 Experimental analysis of the algorithms

For each of the algorithms described above, I get their execution time on my computer. The FIGURE 3.2 below shows that my experimental results follow the theoretical ones, which were drawn with the expressions in TABLE 3.1. I need to note that the divergences in the beginning of the first graph are due to instabilities on the computer when launching the benchmarks for only a few (1 to 4) exponentiations.



(a) ALGORITHM 1

(b) ALGORITHM 2

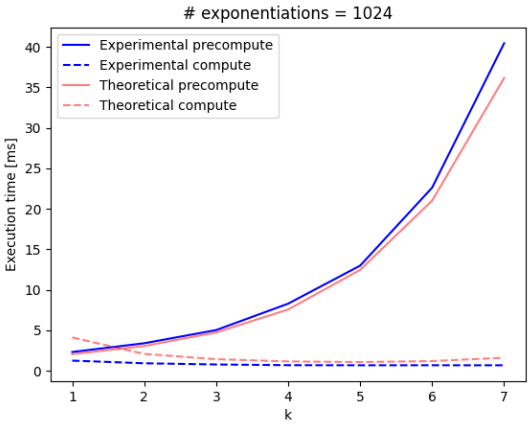
(c) ALGORITHM 3

(d) ALGORITHM 4

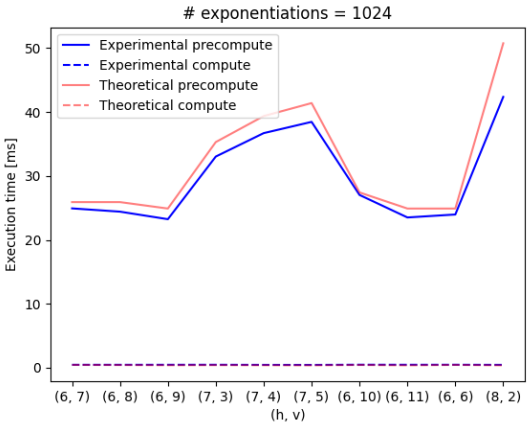
(e) ALGORITHM 5

(f) ALGORITHM 6

Figure 3.2: Comparison of experimental and theoretical execution times for ALGORITHMS 1 TO 8



(g) Algorithm 7



(h) Algorithm 8

Figure 3.2: Comparison of experimental and theoretical execution times for Algorithms 1 to 8

3.3 A second optimization – Zero-Knowledge proofs

This section develops the different Zero-Knowledge proofs used on the voter device (client side) of the ElGamal homomorphic encryption voting scheme. The ones that are detailed here are :

- The non-interactive Disjunctive ZKP (**Definition 2.22**) ;
- The non-interactive ZKP that all non-null ciphertexts lie in a unique list of candidates (**Definition 2.23**) ;
- A protocol to encrypt n votes $0-1$ with its non-interactive ZKP to prove their validity ;

Each of these ZKP is first algorithmically detailed, and secondly improved and rewrote such that the least possible modular exponentiations are computed in the voter side of the protocol, and that they run as fast as possible.

3.3.1 Non-interactive Disjunctive ZKP

This ZKP proves that the ciphertext of a vote is an encryption of either 0 or 1 and not anything else. The first protocol is the basic one, as described in the (**Definition 2.22**). It is taken from the **Definition 26** (pages 34-35) of [5]. The second protocol is a rewrite such that the precomputation of the modular exponentiations can be reused for each computation, and the there is the least possible number of modular exponentiations computed. The goal is to obtain a faster computational time than with basic modular exponentiation algorithms.

Protocol 1

The following is the voter's protocol for proving that a ciphertext is an encryption of 0 *or* 1*. The voter's inputs are the parameters* (p, q, g)*, the election public key* pk*, the voter's ciphertext* $(a, b) = (g^r, g^v \cdot pk^r)$*, her vote* $v \in \{0, 1\}$ *and the random value* r *that she used to encrypt her vote.*

If your vote is $v = 0$:

1. Simulate the protocol for proving $v = 1$. Pick c_1 randomly from $\{0, 1, \ldots, n\}$ and r_1'' from $\mathbb{Z}_q$ at random. Set
$$b' := b/g^1 \pmod p$$
$$a_1' := g^{r_1''}/a^{c_1} \pmod p$$
$$b_1' := pk^{r_1''}/(b')^{c_1} \pmod p$$
2. Set up the proof that $v = 0$. Create a value r_0' from $\mathbb{Z}_q$ at random and set
$$a_0' := g^{r_0'} \pmod p$$
$$b_0' := pk^{r_0'} \pmod p$$
3. Get the challenge for the $v = 0$ proof. Compute
$$c := H(pk, a, b, a_0', b_0', a_1', b_1')$$
$$c_0 := c - c_1 \pmod n$$
4. Complete the $v = 0$ proof. Compute
$$r_0'' := r_0' + c_0 \cdot r \pmod q$$
5. Your proof π is the tuple
$$(a_0', a_1', b_0', b_1', c_0, c_1, r_0'', r_1'')$$

If your vote is $v = 1$:

1. Simulate the protocol for proving $v = 0$. Pick c_0 randomly from $\{0, 1, \ldots, n\}$ and r_0'' from $\mathbb{Z}_q$ at random. Set
$$a_0' := g^{r_0''}/a^{c_0} \pmod p$$
$$b_0' := pk^{r_0''}/b^{c_0} \pmod p$$
2. Set up the proof that $v = 1$. Create a value r_1' from $\mathbb{Z}_q$ at random and set
$$a_1' := g^{r_1'} \pmod p$$
$$b_1' := pk^{r_1'} \pmod p$$
3. Get the challenge for the $v = 1$ proof. Compute
$$c := H(pk, a, b, a_0', b_0', a_1', b_1')$$
$$c_1 := c - c_0 \pmod n$$
4. Complete the $v = 1$ proof. Compute
$$r_1'' := r_1' + c_1 \cdot r \pmod q$$
5. Your proof π is the tuple
$$(a_0', a_1', b_0', b_1', c_0, c_1, r_0'', r_1'')$$

To verify such a proof, the following equations need to be checked :

$$\begin{aligned} g^{r_0''} &= a_0' \cdot a^{c_0} \pmod p \\ g^{r_1''} &= a_1' \cdot a^{c_1} \pmod p \\ pk^{r_0''} &= b_0' \cdot b^{c_0} \pmod p \\ pk^{r_1''} &= b_1' \cdot (b/g^1)^{c_1} \pmod p \\ c_0 + c_1 &= H(pk, a, b, a_0', b_0', a_1', b_1') \pmod n \end{aligned}$$

Protocol 2

The following protocol is the voter's protocol for proving that a ciphertext is an encryption of 0 *or* 1*. The voter's inputs are the description of the group* $\mathbb{G}$*, the parameters* (q, g)*, the election public key* pk*, the voter's ciphertext* $(a, b) = (g^r, g^v \cdot pk^r)$*, her vote* $v \in \{0, 1\}$ *and the random value* r *that she used to encrypt her vote.*

If your vote is $v = 0$:

1. Simulate the protocol for proving $v = 1$. Pick c_1 randomly from $\{0, 1, \ldots, q\}$ and r_1'' from $\mathbb{Z}_q$ at random. Set
$$\begin{aligned} a_1' &:= g^{r_1'' - r \cdot c_1 \pmod q} \pmod p \\ b_1' &:= g^{c_1} \cdot pk^{r_1'' - r \cdot c_1 \pmod q} \pmod p \end{aligned}$$
2. Set up the proof that $v = 0$. Create a value r_0' from $\mathbb{Z}_q$ at random and set
$$\begin{aligned} a_0' &:= g^{r_0'} \pmod p \\ b_0' &:= pk^{r_0'} \pmod p \end{aligned}$$
3. Get the challenge for the $v = 0$ proof. Compute
$$\begin{aligned} c &:= H(g, pk, a, b, a_0', b_0', a_1', b_1') \\ c_0 &:= c - c_1 \pmod q \end{aligned}$$
4. Complete the $v = 0$ proof. Compute
$$r_0'' := r_0' + c_0 \cdot r \pmod q$$
5. Your proof π is the tuple
$$(a_0', a_1', b_0', b_1', c_0, c_1, r_0'', r_1'')$$

If your vote is $v = 1$:

1. Simulate the protocol for proving $v = 0$. Pick c_0 randomly from $\{0, 1, \ldots, q\}$ and r_0'' from $\mathbb{Z}_q$ at random. Set
$$\begin{aligned} a_0' &:= g^{r_0'' - r \cdot c_0 \pmod q} \pmod p \\ b_0' &:= g^{-c_0} \cdot pk^{r_0'' - r \cdot c_0 \pmod q} \pmod p \end{aligned}$$
2. Set up the proof that $v = 1$. Create a value r_1' from $\mathbb{Z}_q$ at random and set
$$\begin{aligned} a_1' &:= g^{r_1'} \pmod p \\ b_1' &:= pk^{r_1'} \pmod p \end{aligned}$$
3. Get the challenge for the $v = 1$ proof. Compute
$$\begin{aligned} c &:= H(pk, a, b, a_0', b_0', a_1', b_1') \\ c_1 &:= c - c_0 \pmod q \end{aligned}$$
4. Complete the $v = 1$ proof. Compute
$$r_1'' := r_1' + c_1 \cdot r \pmod q$$
5. Your proof π is the tuple
$$(a_0', a_1', b_0', b_1', c_0, c_1, r_0'', r_1'')$$

To verify such a proof, the following equations need to be checked :

$$\begin{aligned} g^{r_0''} &= a_0' \cdot g^{r \cdot c_0} \pmod p \\ g^{r_1''} &= a_1' \cdot g^{r \cdot c_1} \pmod p \\ pk^{r_0''} &= b_0' \cdot b^{c_0} \pmod p \\ pk^{r_1''} &= b_1' \cdot (b/g^1)^{c_1} \pmod p \\ c_0 + c_1 &= H(g, pk, a, b, a_0', b_0', a_1', b_1') \pmod q \end{aligned}$$

The computation of the **Protocol 2** needs to precompute the 2 bases of modular exponentiations g and pk, and to compute 5 modular exponentiations distributed over the points 1 and 2, instead of 6 modular exponentiations computed in the **Protocol 1**.

Modular exponentiation of a negative exponent

Note that the modular exponentiation of a negative exponent is calculated as

$$\begin{aligned} g^{-e} \pmod m &= g^{q-e} \pmod{q} \pmod m \\ &= g^q \cdot g^{-e} \pmod m \\ &= 1 \cdot g^{-e} \pmod m \\ &= g^{-e} \pmod m \end{aligned}$$

with g, p, q such that :

$$\begin{cases} p = a\,q + 1 & \forall a \in \mathbb{N}_0 \text{ and } q \text{ prime} \\ g^q = 1 \pmod p \end{cases}$$

3.3.2 Non-interactive ZKP to prove that all the non-null ciphertexts lie in a unique list of candidates

The following ZKP is the voter's protocol for proving that all the candidates he voted for lie in a unique list, such that he cannot vote for candidates in different lists. In this protocol, I wrote in red the improvements about modular exponentiations, such that some computational time to run the protocol can be spared.

The protocol

The voter's inputs are :

- The description of the group $\mathbb{G}$ and the parameters (q, g)
- The election public key pk
- The tuple (a, b) representing the homomorphic encryption of the votes in each of the m lists of candidates with
  - v_i the sum of the votes in each list of candidates
  - r_i the sum of the random values used to encrypt the votes in each list of candidates
  - k the index of the list with non-null candidate's vote(s)

  The tuple (a, b) has the following form :

$$a = \{g^{r_1}, \ldots, g^{r_m}\} \qquad b = \{pk^{r_1}, \ldots, g^{v_k} \cdot pk^{r_k}, \ldots, pk^{r_m}\}$$

1. For every list of candidates except the k^{th} one, set up the proof that $v_i = 0$. Create a value r'_i from $\mathbb{Z}_q$ at random and set

$$a'_i := g^{r'_i} \pmod p$$
$$b'_i := pk^{r'_i} \pmod p$$

2. For the list of candidates k, simulate the protocol for proving $v_k = 0$. Pick y_k and r''_k randomly from $\mathbb{Z}_q$ and set

$$a'_k := g^{r''_k}/a_k^{y_k} \pmod p \quad \Leftrightarrow \quad a'_k := g^{r''_k - r_k \cdot y_k \pmod q} \pmod p$$
$$b'_k := pk^{r''_k}/b_k^{y_k} \pmod p \quad \Leftrightarrow \quad b'_k := g^{-v_k \cdot y_k \pmod q} \cdot pk^{r''_k - r_k \cdot y_k \pmod q} \pmod p$$

3. Get the challenge for every $v_i = 0$ proof. The challenges $(x_i, y_i) = (i, y_i)$ are computed such that they all lie on the same straight line whose y-intercept is the global challenge c

$$c := H(g, pk, a, b, a', b')$$
$$y_i := \frac{y_k - c}{k} \cdot i + c \pmod q$$

4. Complete every $(v_i = 0 \mid i \neq k)$ proof by computing the responses

$$r''_i := r'_i + y_i \cdot r_i \pmod q$$

5. Your proof π is the tuple (a', b', c, y, r'')

To verify such a proof, the following equations need to be checked :

For every list of candidates, verify the commitments :

(1) $g^{r''_i} = a'_i \cdot a_i^{y_i} \pmod p$

(2) $pk^{r''_i} = b'_i \cdot b_i^{y_i} \pmod p$

Verify that that the output of the hash function corresponds to the global challenge c :

(3) $c = H(g, pk, a, b, a', b')$

For every list of candidates except the 1^{st} one, verify the challenges :

(4) $y_i - c = (y_1 - c) \cdot i \pmod{q}$

Details on the verification equations

We can check the calculations of the verification equations. All these lines are $\pmod{p}$, but it has not been written to soften the reading comfort :

(1) 1. For every list of candidates except the k^{th} one

$$\begin{aligned} g^{r_i''} &= a_i' \cdot g^{r_i \cdot y_i} \\ &= g^{r_i'} \cdot g^{r_i \cdot y_i} \\ &= g^{r_i' + y_i \cdot r_i} \\ &= g^{r_i''} \end{aligned}$$

2. For the list of candidates k

$$\begin{aligned} g^{r_k''} &= a_k' \cdot g^{r_k \cdot y_k} \\ &= \frac{g^{r_k''}}{a_k^{y_k}} \cdot g^{r_k \cdot y_k} \\ &= \frac{g^{r_k''}}{(g^{r_k})^{y_k}} \cdot g^{r_k \cdot y_k} \\ &= g^{r_k''} \end{aligned}$$

(2) 1. For every list of candidates except the k^{th} one

$$\begin{aligned} pk^{r_i''} &= b_i' \cdot g^{v_i \cdot y_i} \cdot pk^{r_i \cdot y_i} \\ &= pk^{r_i'} \cdot g^{0 \cdot y_i} \cdot pk^{r_i \cdot y_i} \\ &= pk^{r_i'} \cdot 1 \cdot pk^{r_i \cdot y_i} \\ &= pk^{r_i' + y_i \cdot r_i} \\ &= pk^{r_i''} \end{aligned}$$

2. For the list of candidates k

$$\begin{aligned} pk^{r_k''} &= b_k' \cdot g^{v_k \cdot y_k} \cdot pk^{r_k \cdot y_k} \\ &= \frac{pk^{r_k''}}{b_k^{y_k}} \cdot g^{v_k \cdot y_k} \cdot pk^{r_k \cdot y_k} \\ &= \frac{pk^{r_k''}}{(g^{v_k} \cdot pk^{r_k})^{y_k}} \cdot g^{v_k \cdot y_k} \cdot pk^{r_k \cdot y_k} \\ &= pk^{r_k''} \end{aligned}$$

(4) For every list of candidates except the 1^{st} one, we verify that each challenge y_i lies on the same straight line whose y-intercept is the global challenge $c = H(pk, a, b, a', b')$. We then compute that the slope between every point (i, y_i) except the first one and $(x_0, y_0) = (0, c)$ equals the slope between the first point $(1, y_1)$ and $(x_0, y_0) = (0, c)$.

$$m_i = m_1$$

$$\begin{aligned}
\frac{\Delta y_i}{\Delta x_i} &= \frac{\Delta y_1}{\Delta x_1} \pmod q \\
\frac{y_i - y_0}{x_i - x_0} &= \frac{y_1 - y_0}{x_1 - x_0} \pmod q \\
\frac{y_i - c}{i} &= \frac{y_1 - c}{1} \pmod q \\
y_i - c &= (y_1 - c) \cdot i \pmod q
\end{aligned}$$

Prove that the ZKP is correct and safe

The 3 properties of the ZKP are respected :

- **Completeness** : The verification equations above prove this property. Effectively, if a prover is honest and knows r_i for some $i \in \{1, \ldots, m\}$, then he will respect the protocol and send correct transcripts $\langle a', b', y, r'' \rangle$ to the verifier. This one can verify that the above equalities hold.

- **Soundness** : If we assume that the prover does not know r_i for some $i \in \{1, \ldots, m\}$ but is still able to provide correct transcripts $\langle a', b', y, r'' \rangle$. It means that, for two different sets of challenges $y_{1,i}$ and $y_{2,i}$ sent by the verifier, the prover can send the two sets of correct responses $r''_{1,i}$ and $r''_{2,i}$. But if he can compute these responses, the prover can find r_i ($\forall i \in \{1, \ldots, m\}$) by computing

$$\begin{aligned}
(r''_{1,i} - r''_{2,i}) &= (r'_i + y_{1,i} \cdot r_i) - (r'_i + y_{2,i} \cdot r_i) \\
&= (r'_i - r'_i) + r_i \cdot (y_{1,i} - y_{2,i}) \\
\Leftrightarrow \qquad r_i &= \frac{r''_{1,i} - r''_{2,i}}{y_{1,i} - y_{2,i}}
\end{aligned}$$

  However, this contradicts the initial hypothesis that the prover does not know r_i.

- **Honest verifier zero-knowledge** : To prove that the honest verifier doesn't learn anything from the messages exchanged during the proving procedure, we only have to show that he can generate a valid transcript without knowing r_i for some $i \in \{1, \ldots, m\}$. Effectively, by picking $y_i, r''_i \xleftarrow{\$} \mathbb{Z}_r$ and computing

$$a'_i = \frac{g^{r''_i}}{a_i^{y_i}} \quad \left(= \frac{g^{r'_i} \cdot g^{y_i \cdot r_i}}{g^{r_i \cdot y_i}} = g^{r'_i} \right)$$

$$b'_i = \frac{pk^{r''_i}}{b_i^{y_i}} \quad \left(= \frac{pk^{r'_i} \cdot pk^{y_i \cdot r_i}}{pk^{r_i \cdot y_i}} = pk^{r'_i} \right)$$

  The resulting $\langle a'_i, b'_i, y_i, r''_i \rangle$ is a valid transcript.

The computation of this protocol needs to precompute the 2 bases of modular exponentiations g and pk, and to compute $2(m-1)+3 = 2m+1$ modular exponentiations distributed over the points 1 and 2, with m the total number of lists of candidates in the election. The basic protocol needs 1 more computation of modular exponentiation at the point 2 ($= 2m + 2$).

3.3.3 Protocol to encrypt n votes $0-1$ and prove their validity

The following protocol is an optimization of the first ZKP of this section (the non-interactive Disjunctive ZKP). Effectively, by running this protocol once, one can encrypt n votes $0-1$ and prove their validity instead of running n times the non-interactive Disjunctive ZKP. We are going to see at the end of this subsection the amount of operations spared demonstrating the improvement.

The first protocol below details how to generate the private and public keys, and how to encrypt the n votes of the voter. It also shows how the decryption phase occurs, but we do not implement it in this project as it is computed on the server side, and not on the voter device (client side). From there, the second protocol is the non-interactive ZKP that proves the validity of the n votes encryption, such that they are an encryption of either 0 or 1 and not anything else.

Note that the two protocols partially use the same precomputations of modular exponentiations. They can then, for instance, be sent by the server to the client side once the keys are generated. The sending of these informations will depend on the size of these precomputations and the internet connection of the client side. We will discuss about it in the next section.

Protocol 1 – Encryption of the n votes

Key generation

1. Generate the description of a cyclic group G of order q with generators g and h ;
2. Pick $x_1, \ldots, x_n$ randomly from $\mathbb{Z}_q$;
3. For every vote, compute :

$$y_i := g^{x_i}$$

Encryption

1. Pick r randomly from $\mathbb{Z}_q$;
2. Compute $a := g^r$;
3. For every vote, compute the **shared secrets** :

$$s_i := y_i^r$$

4. For every vote, compute :

$$b_i := h^{v_i} \cdot s_i$$

5. Send the ciphertext $\{a, b\} = \{a, (b_1, \ldots, b_n)\} = \{g^r, (h^{v_1} \cdot y_1^r, \ldots, h^{v_n} \cdot y_n^r)\}$

Decryption *We develop here the decryption phase, but it is not part of the global protocol. It will be performed during the tallying phase, at the end of the election.*

1. For every vote, compute :

$$s_i = a^{-x_i} \pmod p$$
$$h^{v_i} = b_i \cdot s_i^{-1} \pmod p$$

The computation of this protocol needs to precompute the $n+1$ bases of modular exponentiations, being g and y_i $\forall i \in \{1, \ldots, n\}$. Effectively, the base h does not need to be precomputed as its exponent is v_i being always 0 or 1. The protocol also needs to compute n modular exponentiations to generate the public keys y_i, and $n+1$ modular exponentiations to encrypt the ciphertext (a, b).

Protocol 2 – Non-interactive ZKP to prove the validity of the n votes encryption

The protocol — The following is the voter's protocol for proving that the ciphertexts of the n encrypted votes are an encryption of 0 or 1. The voter's inputs are :

- The description of the group $\mathbb{G}$ and the parameters (q, g, h)
- The election public keys $(y_1, \ldots, y_n)$
- The voter's ciphertext $\{a, b\} = \{a, (b_1, \ldots, b_n)\} = \{g^r, (h^{v_1} \cdot y_1^r, \ldots, h^{v_n} \cdot y_n^r)\}$
- The votes $v_i \in \{0, 1\}$
- The random value r the voter used to encrypt its votes

1. Prove that we can open the ciphertext (a, b). Create the values r'_i $(\forall i \in \{1, \ldots, n\})$ and s' from $\mathbb{Z}_q$ at random and set

   $$a' := g^{s'}$$
   $$b'_i := h^{r'_i} \cdot y_i^{s'}$$

2. Prove that we can open each of the above commitments b'_i to 0 or 1. Pick t_i from $\mathbb{Z}_q$ at random and set

   $$d'_i := h^{r'_i \cdot v_i} \cdot y_i^{t_i}$$

3. Compute the challenge c

   $$c := H(g, h, y, a, b, a', b', d')$$

4. Complete the proofs that we can open the ciphertext (a, b) by computing the responses

   $$a'' := s' + c \cdot r \pmod{q}$$
   $$b''_i := r'_i + c \cdot v_i \pmod{q}$$

5. Complete the proofs that we can open the commitments b'_i to 0 or 1 by computing the responses

   $$d''_i := t_i + r \cdot (c - b''_i) \pmod{q}$$

6. Your proof π is the tuple $(a', b', d', c, a'', b'', d'')$

To verify such a proof, the following equations need to be checked :

(1) $g^{a''} = a' \cdot a^c \pmod{p}$

(2) For every vote :

   1. $h^{b''_i} \cdot y_i^{a''} = b'_i \cdot b_i^c \pmod{p}$
   2. $y_i^{d''_i} = d'_i \cdot b_i^{(c - b''_i)} \pmod{p}$

(3) $c = H(g, h, y, a, b, a', b', d')$

Details on the verification equations — We can check the calculations of the verification equations. All these lines are $\pmod p$, but it has not been written to soften the reading comfort :

(1)

$$\begin{aligned} g^{a''} &= a' \cdot a^c \\ &= g^{s'} \cdot (g^r)^c \\ &= g^{s'} \cdot g^{r \cdot c} \\ &= g^{s' + r \cdot c} \\ &= g^{a''} \end{aligned}$$

(2) 1.

$$\begin{aligned} h^{b_i''} \cdot y_i^{a''} &= b_i' \cdot b_i^c \\ &= h^{r_i'} \cdot y_i^{s'} \cdot (h^{v_i} \cdot y_i^r)^c \\ &= h^{r_i'} \cdot y_i^{s'} \cdot h^{v_i \cdot c} \cdot y_i^{r \cdot c} \\ &= h^{r_i' + v_i \cdot c} \cdot y_i^{s' + r \cdot c} \\ &= h^{b_i''} \cdot y_i^{a''} \end{aligned}$$

2.

$$\begin{aligned} y_i^{d_i''} &= d_i' \cdot b_i^{(c - b_i'')} \\ &= h^{r_i' \cdot v_i} \cdot y_i^{t_i} \cdot (h^{v_i} \cdot y_i^r)^{(c - b_i'')} \\ &= h^{r_i' \cdot v_i + v_i \cdot (c - b_i'')} \cdot y_i^{t_i + r \cdot (c - b_i'')} \\ &= h^{r_i' \cdot v_i + v_i \cdot (c - r_i' - c \cdot v_i)} \cdot y_i^{d_i''} \\ &= h^{r_i' \cdot v_i + v_i \cdot c - v_i \cdot r_i' - c \cdot v_i^2)} \cdot y_i^{d_i''} \\ &= h^{c \cdot v_i \cdot (1 - v_i)} \cdot y_i^{d_i''} \\ &= h^{c \cdot 0} \cdot y_i^{d_i''} \\ &= y_i^{d_i''} \end{aligned}$$

Prove that the ZKP is correct and safe — The 3 properties of the ZKP are respected :

- **Completeness** : The verification equations above prove this property. Effectively, if a prover is honest and knows r and v_i for some $i \in \{1, \ldots, n\}$, then he will respect the protocol and send correct transcripts $\langle a', b', d', c, a'', b'', d'' \rangle$ to the verifier. This one can verify that the above equalities hold.
- **Soundness** : If we assume that the prover does not know r and v_i for some $i \in \{1, \ldots, n\}$ but is still able to provide correct transcripts $\langle a', b', d', c, a'', b'', d'' \rangle$. It means that, for two different challenges c_1 and c_2 sent by the verifier, the prover can send the two sets of correct responses $\langle a_1'', b_{1,i}'', d_{1,i}'' \rangle$ and $\langle a_2'', b_{2,i}'', d_{2,i}'' \rangle$. But if he can compute these responses, the prover can find r and v_i by computing

$$\begin{aligned} (a_1'' - a_2'') &= (s' + c_1 \cdot r) - (s' + c_2 \cdot r) \\ &= (s' - s') + r \cdot (c_1 - c_2) \\ \Leftrightarrow \qquad r &= \frac{a_1'' - a_2''}{c_1 - c_2} \end{aligned}$$

$$\begin{aligned}(b''_{1,i} - b''_{2,i}) &= (r'_i + c_1 \cdot v_i) - (r'_i + c_2 \cdot v_i) \\ &= (r'_i - r'_i) + v_i \cdot (c_1 - c_2) \\ \Leftrightarrow \qquad v_i &= \frac{b''_{1,i} - b''_{2,i}}{c_1 - c_2}\end{aligned}$$

However, this contradicts the initial hypothesis that the prover does not know r and v_i.

- **Honest verifier zero-knowledge** : To prove that the honest verifier doesn't learn anything from the messages exchanged during the proving procedure, we only have to show that he can generate a valid transcript without knowing r and v_i for some $i \in \{1, \ldots, n\}$. Effectively, by picking $c, a'', b''_i, d''_i \xleftarrow{\$} \mathbb{Z}_r$ and computing

$$a' = \frac{g^{a''}}{a^c} \quad \left(= \frac{g^{s'+c\cdot r}}{g^{r\cdot c}} = g^{s'} \right)$$

$$b'_i = \frac{h^{b''_i} \cdot y_i^{a''}}{b_i^c} \quad \left(= \frac{h^{r'_i + c\cdot v_i} \cdot y_i^{s'+c\cdot r}}{h^{v_i \cdot c} \cdot y_i^{r\cdot c}} = h^{r'_i} \cdot y_i^{s'} \right)$$

$$d'_i = \frac{y_i^{d''_i}}{b_i^{(c-b''_i)}} \quad \left(= \frac{y_i^{t_i + r\cdot(c-b''_i)}}{h^{v_i\cdot(c-b''_i)} \cdot y_i^{r\cdot(c-b''_i)}} = \frac{y_i^{t_i}}{h^{v_i\cdot(c-b''_i)}} = \frac{y_i^{t_i}}{h^{c\cdot v_i\cdot(1-v_i) - r'_i\cdot v_i}} = h^{r'_i\cdot v_i} \cdot y_i^{t_i} \right)$$

The resulting $\langle a', b', d', c, a'', b'', d'' \rangle$ is a valid transcript.

The computation of this protocol needs to precompute the $n+2$ bases of modular exponentiations being g, h and y_i $\forall i \in \{1, \ldots, n\}$. The bases y_i can be sent by the server to the voter, as they were computed during the key generation phase of the first protocol. The feasibility of this sending will depend on the size of this set and the internet connection quality of the voter. I will discuss about it in the next section.

The protocol also needs to compute $3n + 1$ modular exponentiations, with n the number of votes, distributed over the following points of the protocol :

1. 1 for a' and $2n$ for b'_i as $i \in \{1, \ldots, n\}$
2. n for the $y_i^{t_i}$ part of d'_i as the exponent of h is partly already computed in the expression of b'_i at point 1. The v_i part of the exponent can be ignored as it is always 0 or 1.

Comparison with the non-interactive Disjunctive ZKP

Compared to the fact that this protocol is the same as using n times the non-interactive Disjunctive ZKP, which needs the precomputation of 2 bases and the computation of 5 modular exponentiations, we can compute the minimal n at which using the above protocol would be more advantageous because, despite the bigger amount of precomputations, there would be less computations to make.

We can fix the followings variables :

- x_1 is the precomputation time needed for 1 base
- x_2 is the computation time needed for 1 modular exponentiation with precomputations on the base
- n is the number of votes

We need to analyze the non-interactive Disjunctive ZKP, called P_1, and the protocol above, called P_3. We are also going to analyze two cases with the protocol P_3. The first that we will call P_{3a} is when the precomputations to compute modular exponentiations on the bases $(y_1, \ldots, y_n)$, which are the public keys, are made by the voter (on the client side). The second one, called P_{3b} is when those public keys are computed on the server side, and sent by it to the voter.

Let's write the equations for each of the 3 protocols that compute the total computational time to prove the validity of encryption of n votes :

$$t[P_1] \equiv 2x_1 + 5nx_2 \tag{3.1}$$
$$t[P_{3a}] \equiv (n+2)x_1 + (3n+1)x_2 \tag{3.2}$$
$$t[P_{3b}] \equiv 2x_1 + (3n+1)x_2 \tag{3.3}$$

It is easy to compare $t[P_1]$ with $t[P_{3b}]$ as they have the same factor in front of x_1. We can then pose the following inequality to obtain the minimal n such that the P_1 will be faster in computational time than P_{3b} :

$$\begin{aligned} 2x_1 + 5nx_2 &\leq 2x_1 + (3n+1)x_2 \\ 5n &\leq 3n + 1 \\ n &\leq \frac{1}{2} \end{aligned}$$

As the number of votes n is a positive integer, we can conclude that **the computational time of P_{3b} will always be smaller than the one of P_1**, whatever the number of votes n to compute. But this is **without taking into account the fact that we have a limited amount of memory usage for precomputations**, as the server needs to send them to the voter. We will then check the feasibility of it in the next section, when we will have the exact amount of memory used by one precomputation, depending on the modular exponentiation algorithm used among the ones explained in the previous section.

Concerning the comparison of P_1 with P_{3a}, it is more difficult. Effectively, before making such an inequality to compare them and obtain a value for n, we would need to substitute x_1 by a factor k of x_2. Indeed, we need the ratio between the precomputation time needed for 1 base and the computation time needed for 1 modular exponentiation with precomputations on that base, that is $\frac{x_1}{x_2}$.

If we make the assumption that an efficient algorithm takes at least the same time for precomputing than for computing a modular exponentiation, e.g. that $k = 1$, we obtain :

$$\begin{aligned} 2x_2 + 5nx_2 &\leq (n+2)x_2 + (3n+1)x_2 \\ 2 + 5n &\leq 3 + 4n \\ n &\leq 1 \end{aligned}$$

indicating that the computational time of P_{3a} will always be smaller than the one of P_1, whatever the number of votes n to compute.

Now if we make another assumption that $k = 5$, which is more likely as at most the precomputational time increases at most the computational time decreases (but not linearly...), we obtain now :

$$\begin{aligned} 10x_2 + 5nx_2 &\leq (n+2)10x_2 + (3n+1)x_2 \\ 10 + 5n &\leq 21 + 13n \\ n &\geq -\frac{11}{8} \end{aligned}$$

indicating that the computational time of P_{3a} will always be bigger than the one of P_1, whatever the number of votes n to compute.

In fact, instead of making assumptions to compute this ratio, we can calculate the factor k for which the equality between $t[P_1]$ and $t[P_{3a}]$ holds :

$$\begin{aligned} 2kx_2 + 5nx_2 &= (n+2)kx_2 + (3n+1)x_2 \\ 2n - 1 &= nk \\ k &= \frac{2n-1}{n} \end{aligned}$$

By calculating its limit for the number of votes n tending to infinity, we finally obtain :

$$\lim_{n \to +\infty} \frac{2n-1}{n} = 2$$

We can conclude with this value that, **if the precomputational time needed for 1 base is 2 times superior to the one needed for 1 modular exponentiation with given precomputations on the base, then the computational time of P_{3a} will always be bigger than the one of P_1, whatever the number of votes n to compute**.

Chapter 4

Results

This chapter is dedicated to the different results obtained during this thesis project. Globally, they aim to :

- Compare the performances of Rust which is coupled with WebAssembly and JavaScript ;
- Compare the execution times of
  - the different algorithms described in Section 3.2 used to compute modular exponentiations,
  - the Zero-Knowledge proofs and protocols detailed in Section 3.3,

  both on different browsers and devices ;
- Compare the results obtained with already existing ones ;
- Discuss about the relevance of the results obtained and their practical feasibility ;

The first section lists the hardware and the software used to obtain the results in the three next sections. Those ones respectively detail the comparison between the use of Rust and JavaScript (Section 4.2), the comparison benchmarks of the modular exponentiation algorithms (Section 4.3) and the benchmarks of the protocols and ZKP (Section 4.4).

The last section of this chapter exposes the benchmarks for a complete ballot creation from a voter side.

4.1 Hardware and software used

To obtain sufficient benchmarks on the algorithms and program's performances, I used 4 different devices detailed here below. 2 of them are computers and the 2 others are smartphones :

- **Device 1** : A *Lenovo ThinkPad*, working on *Windows 10 64 bits*, with a processor *Intel(R) Core(TM) i7-6600U CPU @ 2.60 GHz 2.81 GHz* and *16.0 Go* of Random Access Memory.
- **Device 2** : A *Hewlett-Packard*, working on *Windows 10 64 bits*, with a processor *Intel(R) Core(TM) i5-4210M CPU @ 2.60 GHz 2.60 GHz* and *4.0 Go* of Random Access Memory.
- **Device 3** : An *Apple iPhone 7 32 Go*, working on *iOS 14.2*, with a processor *A10 Fusion Quad-core* and *2.0 Go* of Random Access Memory.
- **Device 4** : A *Samsung Galaxy A50 128 Go*, working on *Android 11*, with a processor *Exynos 9610 Octa-core (4x2.3 GHz Cortex-A73 & 4x1.7 GHz Cortex-A53)*, a GPU *ARM Mali G72 MP3* and *4.0 Go* of Random Access Memory.

Concerning the software part, I used the following 3 browsers :

- **Google Chrome** with versions 91.0.4472.77 on Windows, 91.0.4472.80 on iOS and 91.0.4472.88 on Android ;
- **Mozilla Firefox** with versions 89.0 on Windows, 34.0 on iOS and 89.1.1 on Android ;
- **Microsoft Edge** with versions 91.0.864.41 on Windows, 46.3.20 on iOS and 46.04.2.5157 on Android ;

To develop the Rust coupled with WebAssembly code, I used the integrated development environment "*IntelliJ IDEA 2021.1.2 (Community Edition)*". To implement the JavaScript parts of this thesis project, I used the extensible code editor "*Visual Studio Code 1.56.2*" and the open source software platform "*Node.js 14.16.0*".

The elements of the group $\mathbb{G} \subset \mathbb{Z}_q^\times$ with the prime q of 256 bits that I used to compute the benchmarks in this chapter are the following. They respect the equations $p = aq + 1 \quad \forall a \in \mathbb{N}_0$ and $g^q = 1 \pmod p$

- $p = 52037852794058762148655187488328620183326733544328775764764726813330309066464722455$
  $129469473758159661925779626712106667443488731999500560262286161777505853766261262784921$
  $283265022090155016854297466985046361883589842454584588819272764860156588709982552839375$
  $728328787346228844620854792616300333447092194322006985712574550605651278241993398881912$
  $657653861653098907300037041320115150350761511753494137236987361883774708032934840226695$
  $973669693657552726450908667765156901403714270434884784904508094867277691220800867201585$
  $873914419586359564941468227240508531133032684484470588691155699888231208832456813198305$
  $305000835070263482981979823669169038411575748272964651492948132102270270280817727085780$
  $868435780363154359893975427432413413430199088208101053692286675158756381468665821430905$
  $952070578515579267576420911272743757150892883452512908802083203649159413872675103058425$
  $840050085539451345800835848138588538615906391091384600150 11$
- $q = 79958624757606997148768673019482141441495304812299620824369680833929235034161$
- $g = 63293798476866785418015327916015677678514428595209760657703747937856970332535625332$
  $570865401581859495073788335198905306717031616610811655817633675511692477798663950231284$
  $038162310185391690941142120523151479016844508590031397434386793344975170037614898055442$
  $127016501244194592657330114129193099661951173566966710143575558727906095078130280309360$
  $424744365142608769916042835823990632296497832649102366771492147195314168684642987751884$
  $473147674370076619938462411132572018554323585921277852534678697185192078313697097815081$
  $933940401289627273955405961136001120989256311126820509929629847269569839150347476626781$
  $869324207864590001900980287017387738128235067632650572897316427139461632953747148041900$
  $205213760624531705493264264451490564528391916982394354610813525307081109999762199651646$
  $410879021884136149185123433385365479379863779525228857454611422704948190038178969634660$
  $268259887384182944584533371622295899929024651802300367644 8$

4.2 Rust with WebAssembly or JavaScript ?

The first choices I needed to make in this thesis are the language and framework I would use to implement the voting schemes. As mentioned in SECTION 3.1, my choice would be either JavaScript or Rust coupled with WebAssembly. I decided to start my tests with the latter, as it is less known and more promising.

4.2.1 Rust with WebAssembly

As explained in SECTION 3.1.2, the possibilities of libraries to use with Rust are **num-bigint** [24], **ramp** [26] and **rug** [28]. To obtain benchmarks on them, I computed modular exponentiations with the function inner to each of them.

- The **num-bigint** library uses the function `modpow` [25]. This modular exponentiation function is an implementation of the *Montgomery Multiplication* algorithm presented in [34], which is a fast and very efficient method to compute modular exponentiations with non-fixed bases.
- The **ramp** library uses the function `pow_mod` [27]. This modular exponentiation function is an implementation of the `Algorithm 1`.

Concerning the **rug** crate, I was not able to provide benchmarks for the computation of modular exponentiations. Effectively, the use of this crate is convoluted and is not compatible with WebAssembly, which is not useful for this project as we need to make computations on browsers.

Moreover, the **num-bigint** library allows to use *big integers* that are *signed* or *unsigned*, respectively by using the structures `BigInt` and `BigUint`. I will test both of them in the next benchmarks.

In order to compare the **num-bigint** crate (with `BigInt` and `BigUint` structures) and the **ramp** crate, I performed modular exponentiations with `Algorithm 1`, which is the inner function `pow_mod` for **ramp**.

The FIGURE 4.1 compares the **num-bigint** crate (with `BigInt` and `BigUint` structures) and the **ramp** crate. It shows the execution times needed to compute modular exponentiations with exponents of 256-bits and 3072-bits of size, by using the `Algorithm 1`. These execution times are an average on 2000 samples. They were performed on the **Device 1** with compiled Rust, so without the use of WebAssembly. The computations were then not made on browsers, but only on *Cargo*, the Rust's package manager.

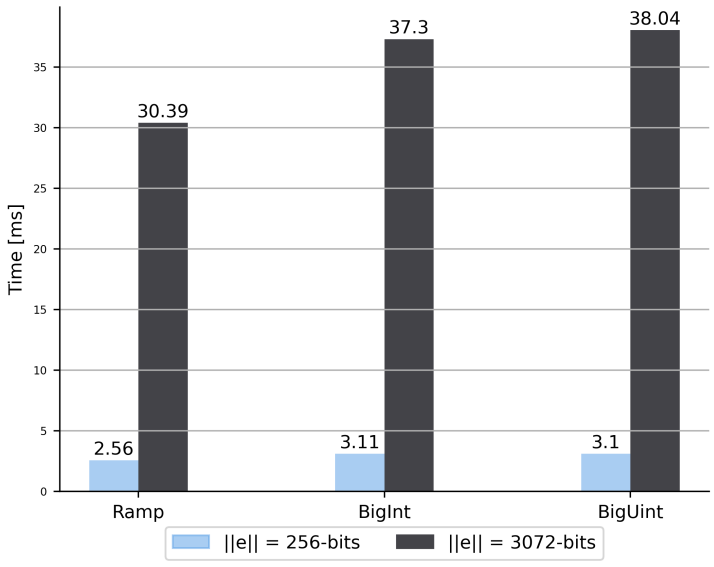


Figure 4.1: Execution times of modular exponentiation computations with `Algorithm 1` on *compiled Rust*

From this FIGURE 4.1, one can conclude that the **ramp** library is the most efficient. Unfortunately, as mentioned in SECTION 3.1.2, due to the use of unstable features such as inline assembly, this library can only be compiled with a *nightly* (i.e. unstable) *build* of the Rust compiler, which cannot be used in production. We then cannot use it with WebAssembly.

An observation that seems coherent is the fact that the execution times for `BigInt` and `BigUint` are globally the same. To choose between these two structures of the library **num-bigint**, I processed modular exponentiations using the inner function `modpow` which is an implementation of the *Montgomery Multiplication* algorithm. I executed it on *compiled Rust* and on *Rust coupled with WebAssembly* on the **Google Chrome** web-browser on **Device 1** to obtain the following FIGURE 4.2 :

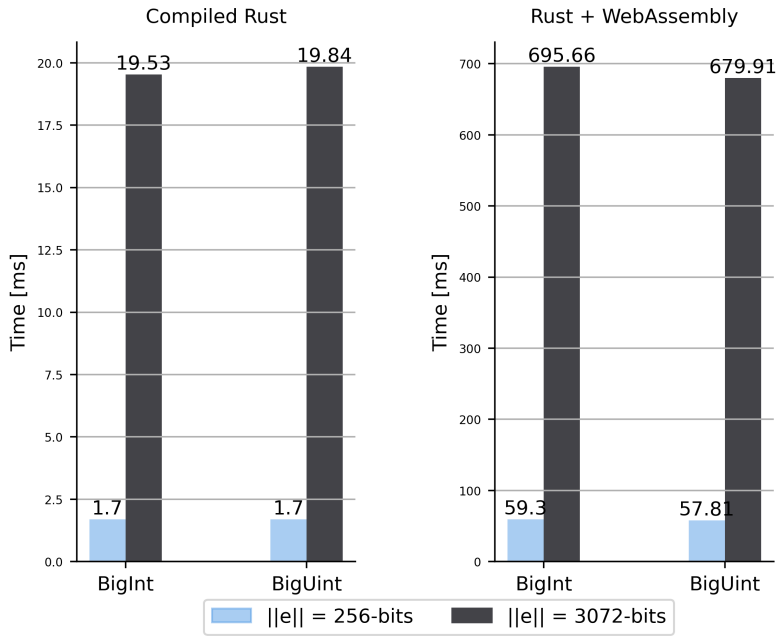


Figure 4.2: Execution times of modular exponentiation computations with *Montgomery Multiplication* algorithm on *compiled Rust* (**left**) and *Rust coupled with WebAssembly* (**right**)

We can see on the above FIGURE that `BigInt` and `BigUint` have more or less the same execution times. As `BigUint` is used for *unsigned big integers*, I will use it to compare Rust and JavaScript as I only use positive exponentiations for the benchmarks, and it will facilitate the implementation. Two interesting observations can also be made on those graphs :

- The *Montgomery Multiplication* algorithm is more or less twice as efficient than the `Algorithm 1` on the **num-bigint** library ;
- Rust coupled with WebAssembly on *Google Chrome* is more or less **35 times slower** than compiled Rust ;

The second observation is a pretty bad result, as in our voting schemes, each voter need to compute more or less a thousand of modular exponentiations on its device, which leads us to 57.81 seconds (note that the graphs are expressed in milliseconds). Even if the algorithm of modular exponentiation can be optimized and use precomputations, I do not think that any voter would agree to wait such an amount of time to process its ballot.

4.2.2 JavaScript

As explained in SECTION 3.1.1, the **Verificatum** (or **VJSC** [21] library will be used with JavaScript.

In order to compare the performances of JavaScript with Rust & WebAssembly, I get with both languages the average execution time of 2000 modular exponentiation (with exponents of 256-bits) computations with `Algorithm 1` and `Algorithm 8`, both on the *Google Chrome* web-browser on **Device 1**. The results obtained are displayed on FIGURE 4.3 :

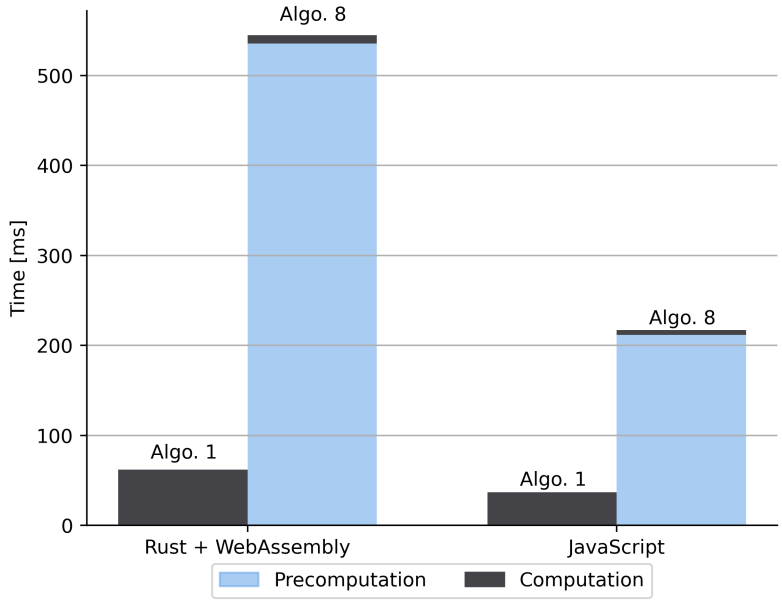


Figure 4.3: Comparison of JavaScript and Rust & WebAssembly with `Algorithm 1` and `8` on the *Google Chrome* web-browser on **Device 1**

The main observation is that JavaScript is more or less twice as efficient than Rust coupled with WebAssembly. From FIGURES 4.2 and 4.3, we can also make the shortcut that JavaScript executed on *Google Chrome* is more or less 17 times slower than compiled Rust.

JavaScript is then the language I will use to obtain the benchmarks of the next sections.

Before testing all the modular exponentiation algorithms, it would be interesting to test the inner `modPow` function of **Verificatum**. This function is in fact an implementation of the `Algorithm 3`. But when making fixed-base exponentiations, the library uses the algorithm differently, by making some precomputations that are calculated with the `modPow` function (that also contains precomputations). More information can be found here [1]. Furthermore, the library allows to output some benchmarks automatically for fixed-base modular exponentiations, which are used here.

The FIGURE 4.4 displays a comparison of the average execution times of the **Verificatum** `modPow` function and the `Algorithm 3` for a fixed number of modular exponentiations.

[1] https://www.verificatum.org/api-vjsc/vjsc-1.1.1.js.html#line5114

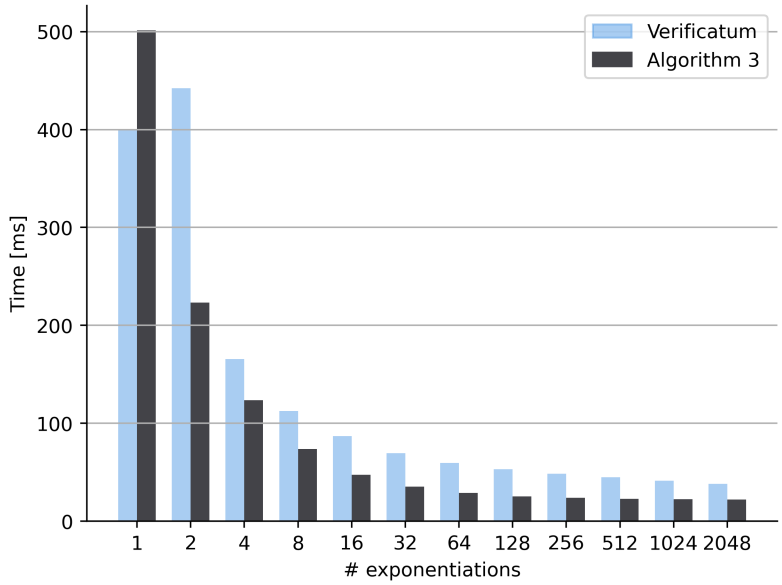


Figure 4.4: Comparison of the **Verificatum** `modPow` function and the `Algorithm 3` for a fixed number of modular exponentiations

We can see that the `Algorithm 3` processes faster than the fixed-base modular exponentiations of **Verificatum** while the number of exponentiations is bigger than 1.
Effectively, it seems that the precomputations of the `modPow` function are optimized compared to the ones of `Algorithm 3`, which makes the difference for only 1 fixed-base exponentiation.

However, the additional precomputations made for several fixed-base exponentiations slow down the performances of **Verificatum**. Moreover, they increase the average execution time going from 1 to 2 fixed-base exponentiations. I don't really know why the creator(s) of **Verificatum** designed it like this, but I presume that there must be some specific reasons, perhaps that serve other methods of the library.

Globally, both algorithms decrease exponentially with respect to the exponential increasing of the number of exponentiations, which seems to be a logical behaviour.

4.3 Modular exponentiation algorithms

This section aims at comparing the 8 modular exponentiations algorithms detailed in Sections 3.2.2 and 3.2.3. I also compare them with the `modPow` function of the **Verificatum** library (without the precomputations for fixed-base exponentiations).

The execution times were carried out on three different browsers, each on the four different devices presented at the beginning of this chapter (Section 4.1).

Note that to implement these algorithms, I needed to perform a correction in a function of the library. It is in the function `LargeInteger.prototype.mod(modulus)` at line 4747 of the JS file of the library [2]. The function is actually :

```
LargeInteger.prototype.mod = function (modulus) {
    return this.divQR(modulus)[1];
};
```

but should be :

```
LargeInteger.prototype.mod = function (modulus) {
    var result = this.divQR(modulus)[1];
    result.length = result.value.length;
    return result;
};
```

Effectively, this lack in the library created useless work for the computer as the `length` attribute of the function's result of type `LargeInteger` was not updated, and began to be a huge number at some point. This led to an overflow of the memory.

In the following graphs, the average execution times of 2000 modular exponentiations with exponent of 256-bits of size are presented for the 9 algorithms. They are displayed for each browser, device by device on Figure 4.5 and for each device, browser by browser on Figure 4.6. The 9 algorithms are each time numbered from 0 to 8, which respectively represent :

(**0**) The `modPow` function of the **Verificatum** library ;

(**1**) The `Algorithm 1` ;

(**2**) The `Algorithm 2` with parameter $k = 12$ (chosen from Section 3.2.4) ;

(**3**) The `Algorithm 3` with parameter $k = 13$ (chosen from Section 3.2.4) ;

(**4**) The `Algorithm 4` with parameter $k = 7$ (chosen as the best from 1 to 20) ;

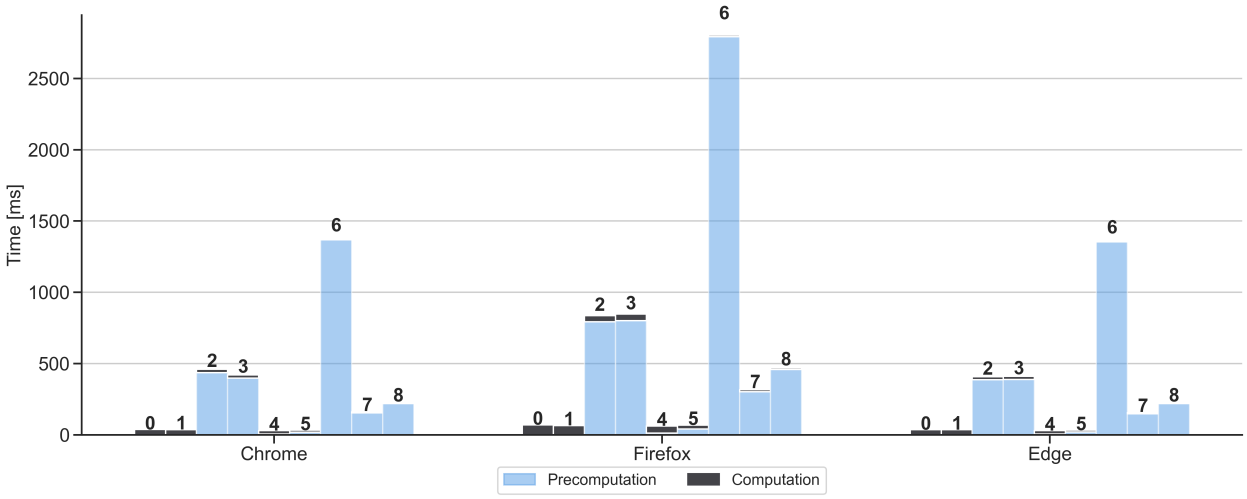
(**5**) The `Algorithm 5` ;

(**6**) The `Algorithm 6` with parameter $k = 4$ (chosen from Section 3.2.4) ;

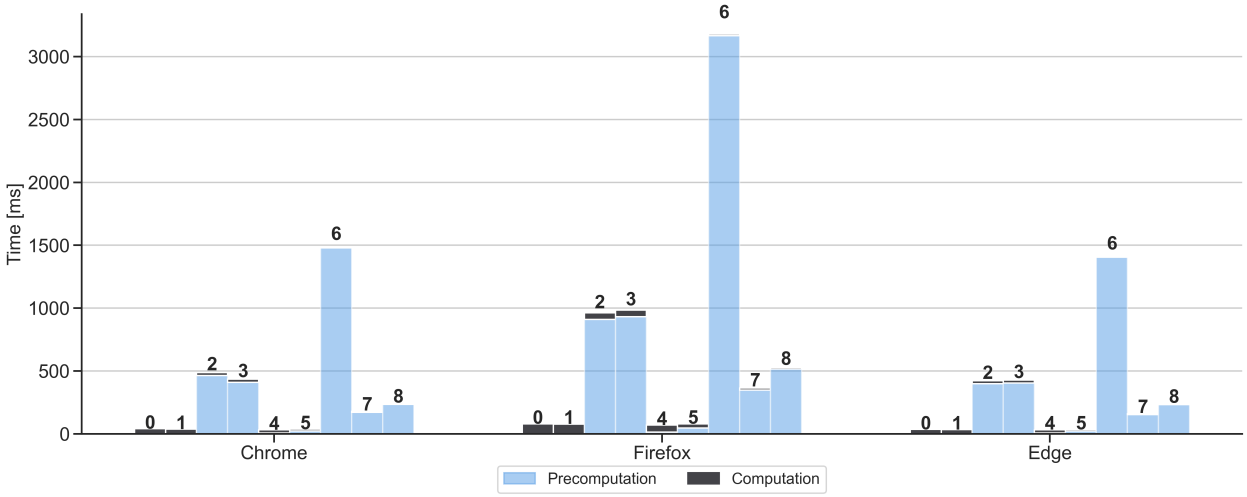
(**7**) The `Algorithm 7` with parameter $k = 5$ (chosen from Section 3.2.4) ;

(**8**) The `Algorithm 8` with parameters $h = 6$ and $v = 9$ (chosen from Section 3.2.4) ;

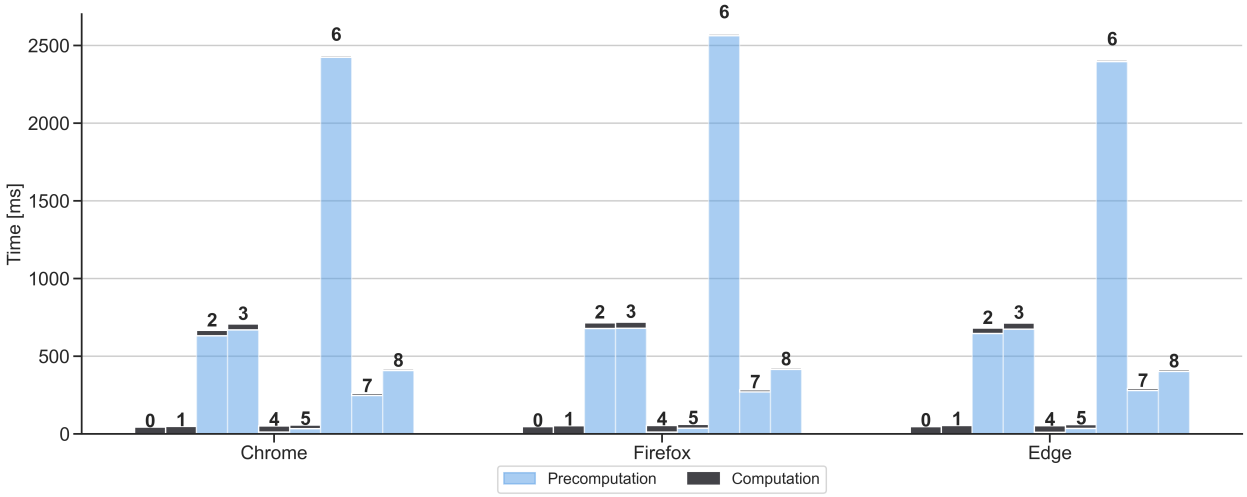
[2] https://www.verificatum.org/files/vjsc-1.1.1.js



(a) **Device 1**

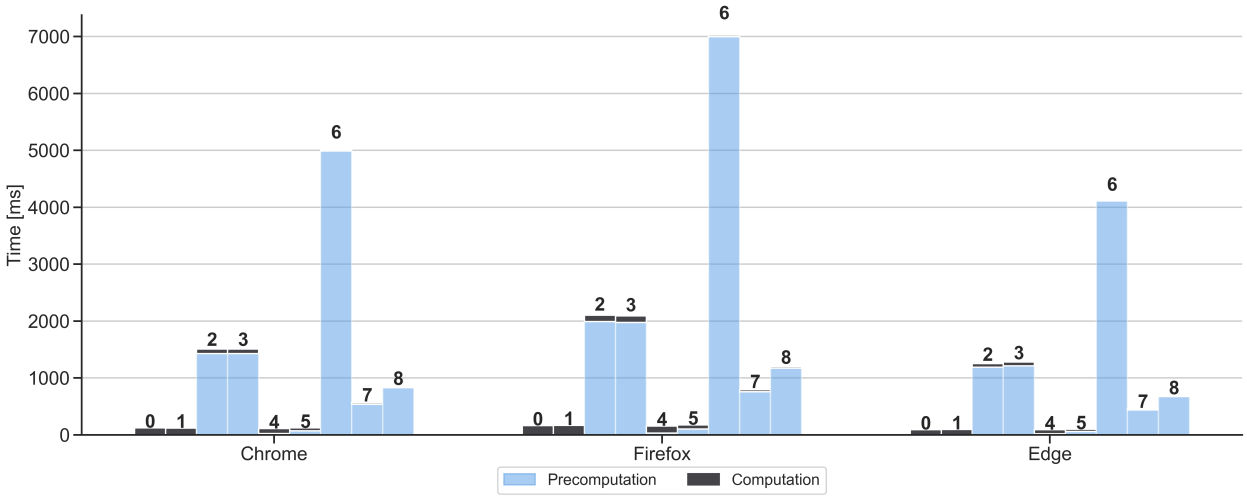


(b) **Device 2**



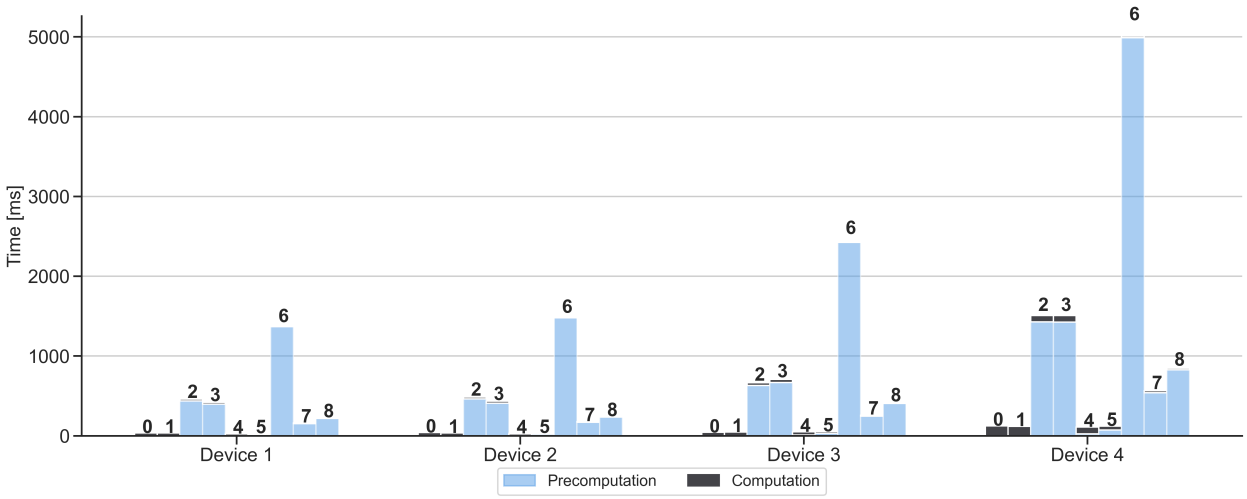
(c) **Device 3**

Figure 4.5: Average execution times of 2000 modular exponentiations with exponent of 256-bits

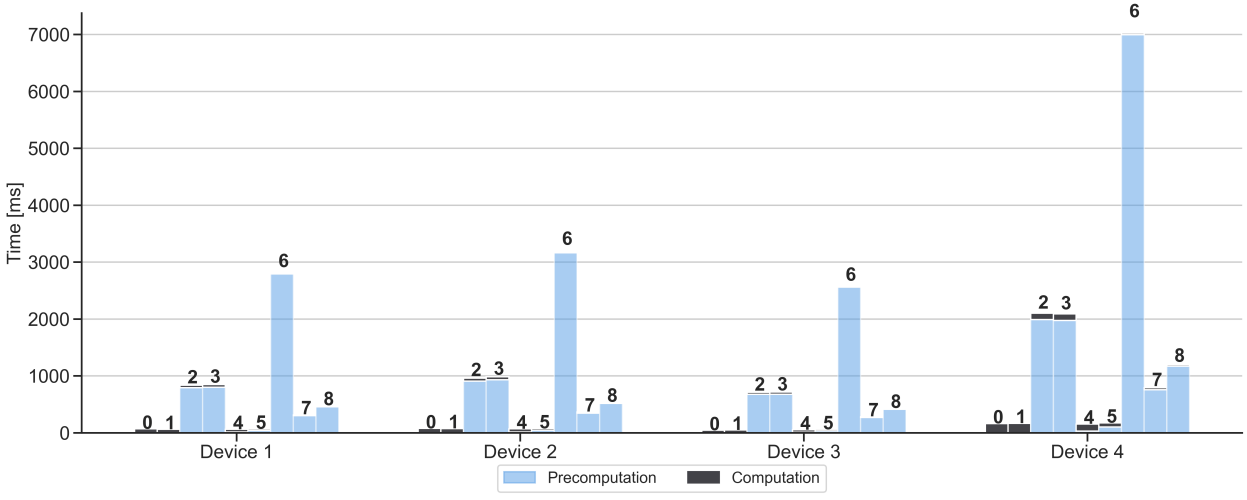


(d) **Device 4**

Figure 4.5: Average execution times of 2000 modular exponentiations with exponent of 256-bits

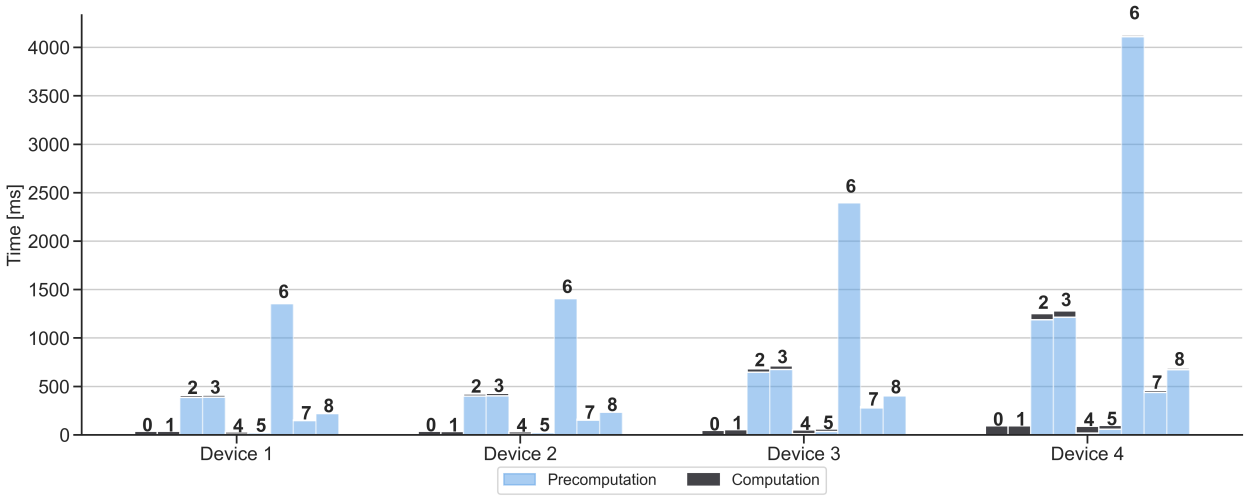


(a) **Chrome**



(b) **Firefox**

Figure 4.6: Average execution times of 2000 modular exponentiations with exponent of 256-bits



(c) **Edge**

Figure 4.6: Average execution times of 2000 modular exponentiations with exponent of 256-bits

Note that for each of the above graph, the precomputation time is the effective one, used to compute the 2000 modular exponentiations. However, the computation time is an average of the 2000 ones. So the sum of the two times is theoretically the same as computing only one modular exponentiation, without considering practical calculation errors of the computer.

Different observations can be established from these graphs.

- Firstly, concerning the hardware, the *Devices 1* and *2* are globally equally efficient. Contrariwise, the 2 smartphones have huge differences in performance, while the *Device 3*, is sometimes faster than the 2 computers. It is a surprising result as the technical specifications of the Android smartphone (*Device 4*) seem better than the ones of the iOS device (*Device 3*). Moreover, the *Device 4* is more recent (compared to the release dates of the models) than *Device 3*.
- Secondly, the *Firefox* web-browser is much slower than the 2 others, while *Chrome* is a little bit slower than *Edge*.
- Finally, the precomputation times of some algorithms, like the 2, 3 and 6 are huge compared to their computation time. For instance, the `Algorithm 6` on *Device 1* on *Chrome* has precomputation and computation times of respectively 1367.633 [ms] and 5.786 [ms] which is a ratio of 236.

As we need to compute a thousand of modular exponentiations in our voting scheme, the FIGURE 4.7 will help us to visualize which are the fastest algorithms for our needs. This graph represents the total average execution time of a thousand of modular exponentiation with exponent of 256-bits of size, where the precomputation time has been averaged with the computation time. These computations were done on *Device 1* on *Google Chrome*.

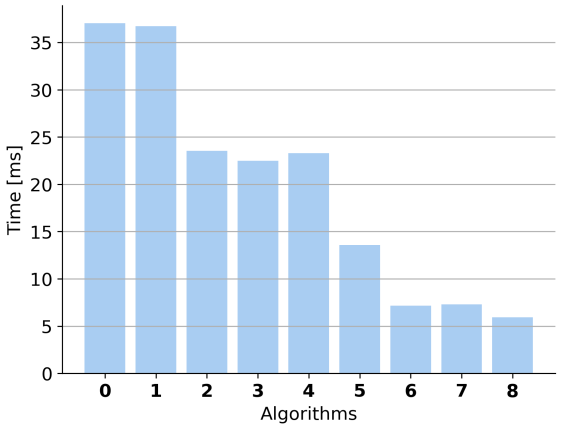


Figure 4.7: Total average execution times of a thousand modular exponentiations for the 9 analyzed algorithms

These results follow my expectations, as Algorithms 5 to 8 are specific to fixed-base modular exponentiations. In order to give precise results, the times of these 4 algorithms are respectively 13.578, 7.154, 7.305 and 5.943 milliseconds. The use of the 5 first algorithms can already be rejected, as a voter will not wait more than 20 seconds (20 milliseconds times 1000 modular exponentiations) to have its ballot processed.

Before analyzing the results of the protocols and ZKP, it could be interesting to know the amount of values each algorithm needs to store, in terms of number and memory size. The following TABLE 4.1 displays these values with the chosen parameters :

# `Algorithm`	Parameters	# values stored	Memory size $[Ko]$
1	–	0	0
2	$k = 12$	4096	3670.016
3	$k = 13$	8192	7340.032
4	$k = 7$	128	114.688
5	–	256	229.376
6	$k = 4$	960	860.16
7	$k = 5$	52	46.592
8	$h = 6$ $v = 9$	567	508.032
	$h = 7$ $v = 3$	381	341.376

Table 4.1: Amount of values each algorithm needs to store to compute a modular exponentiation of one base

Note that the last line (Algorithm 8 with $h = 7$ and $v = 3$) was added because it represents the minimum memory size for some h and v giving a tiny bit more time to execute.

An important observation to make is that Algorithm 7 gives an execution time only 1.23 times bigger than Algorithm 8, while it only needs to store between 7 to 11 less values than Algorithm 8.

4.4 Protocols and Zero-Knowledge proofs

The objective of this section is to compare the execution times of the 3 protocols detailed in Section 3.3. In order to ease the task of the reader, the protocols will be named for the next sections :

- P_1 for the non-interactive Disjunctive ZKP (Section 3.3.1) ;
- P_2 for the non-interactive ZKP to prove that all the non-null ciphertexts lie in a unique list of candidates (Section 3.3.2) ;
- P_{3a} for the non-interactive ZKP to prove the validity of encryption of n votes 0−1 (Section 3.3.3). Here, the precomputations to compute modular exponentiations on the bases $(y_1, \ldots, y_n)$, which are the public keys, are made by the **voter** (on the client side) ;
- P_{3b} for the non-interactive ZKP to prove the validity of encryption of n votes 0−1 (Section 3.3.3). Here, the precomputations to compute modular exponentiations on the bases $(y_1, \ldots, y_n)$, which are the public keys, are made by the **server** and sent by it to the voter ;

The execution times of these protocols were, like in previous section, carried out on three different browsers, each on the four different devices presented at the beginning of this chapter (Section 4.1).

In the following graphs, the precomputation (in blue), computation (in black) and verification (in green) execution times are displayed. But note that the verification is not part of the voter side calculations. Effectively, the proofs are sent by the voter to the server, which verify them before approving it. The verification step is shown to only have an idea of the amount of work this task represents, compared to the precomputational and computational execution times.

The 3 protocols are displayed for each browser, device by device on Figure 4.8 and for each device, browser by browser on Figure 4.9. The 3 protocols are each time numbered from 1 to 8, as they were computed with several different parameters. These numbers represent :

(**1**) P_1 with the encryption of 100 votes $0-1$, computed with Algorithm 7 ;

(**2**) P_1 with the encryption of 100 votes $0-1$, computed with Algorithm 8 ;

(**3**) P_2 with 10 list of candidates, containing 10 candidates each, computed with Algorithm 7 ;

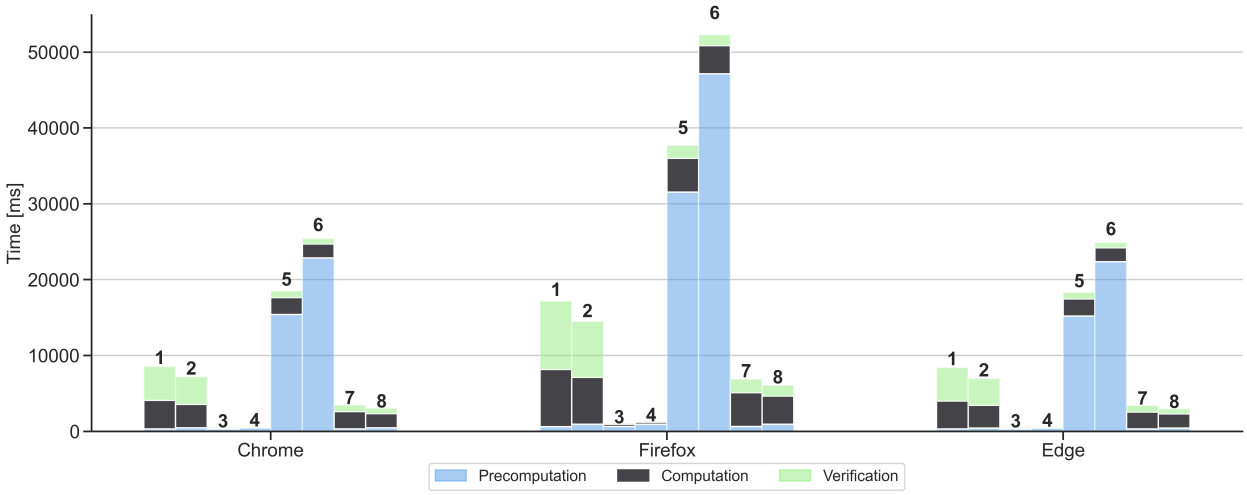
(**4**) P_2 with 10 list of candidates, containing 10 candidates each, computed with Algorithm 8 ;

(**5**) P_{3a} with the encryption of 100 votes $0-1$, computed with Algorithm 7 ;

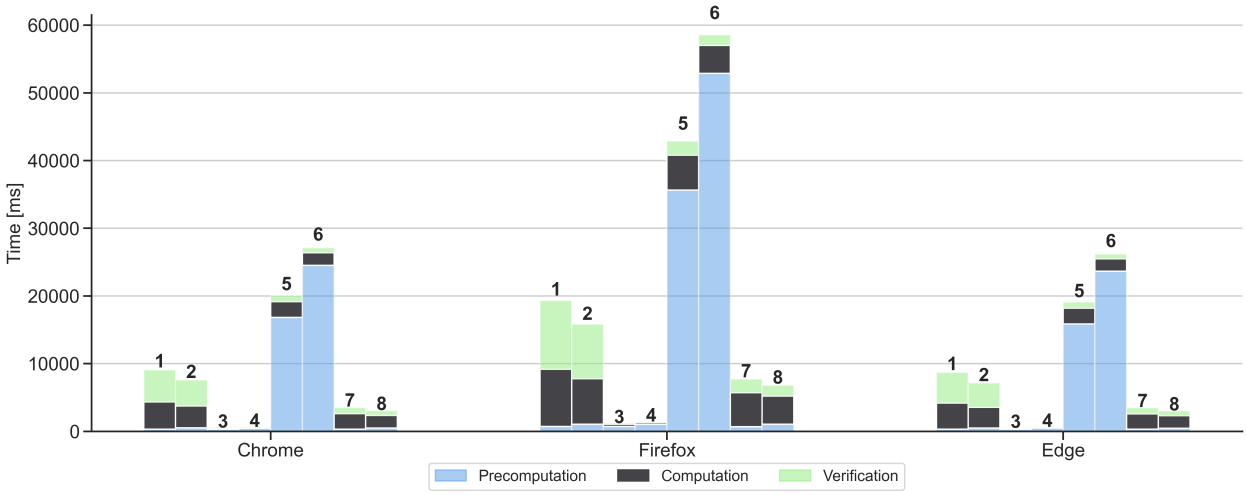
(**6**) P_{3a} with the encryption of 100 votes $0-1$, computed with Algorithm 8 ;

(**7**) P_{3b} with the encryption of 100 votes $0-1$, computed with Algorithm 7 ;

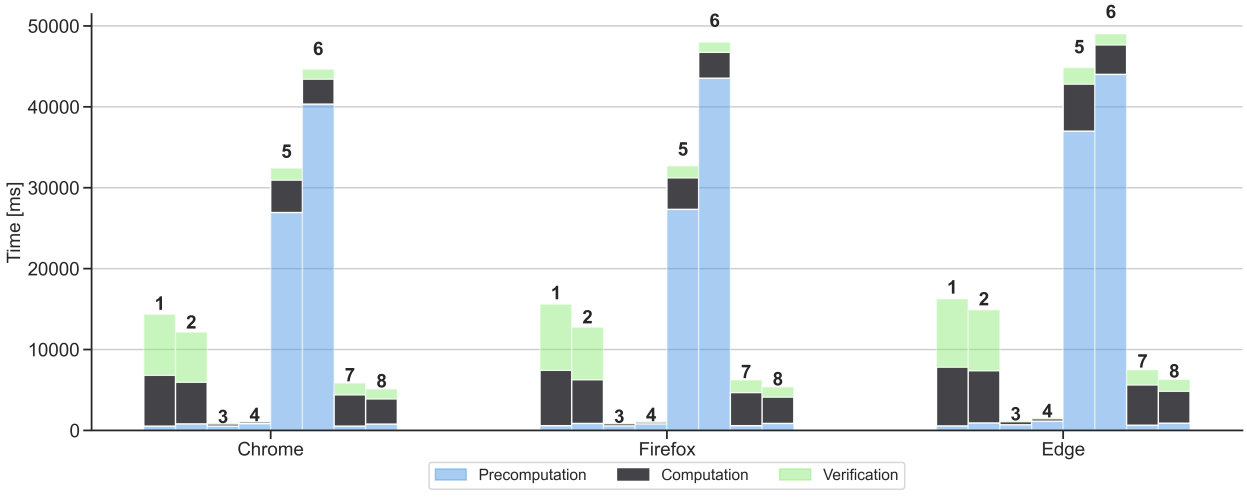
(**8**) P_{3b} with the encryption of 100 votes $0-1$, computed with Algorithm 8 ;



(a) **Device 1**

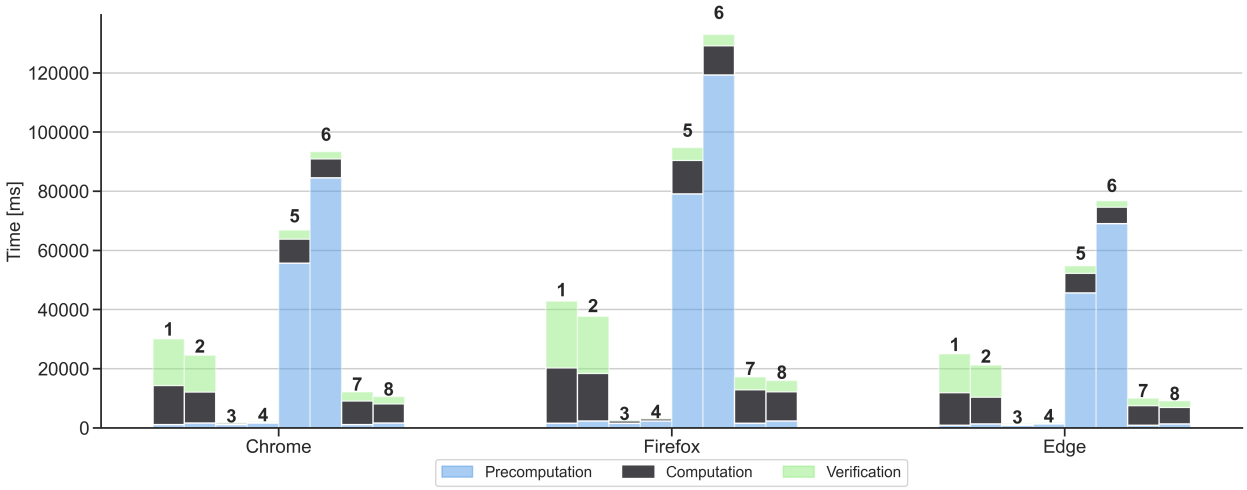


(b) **Device 2**



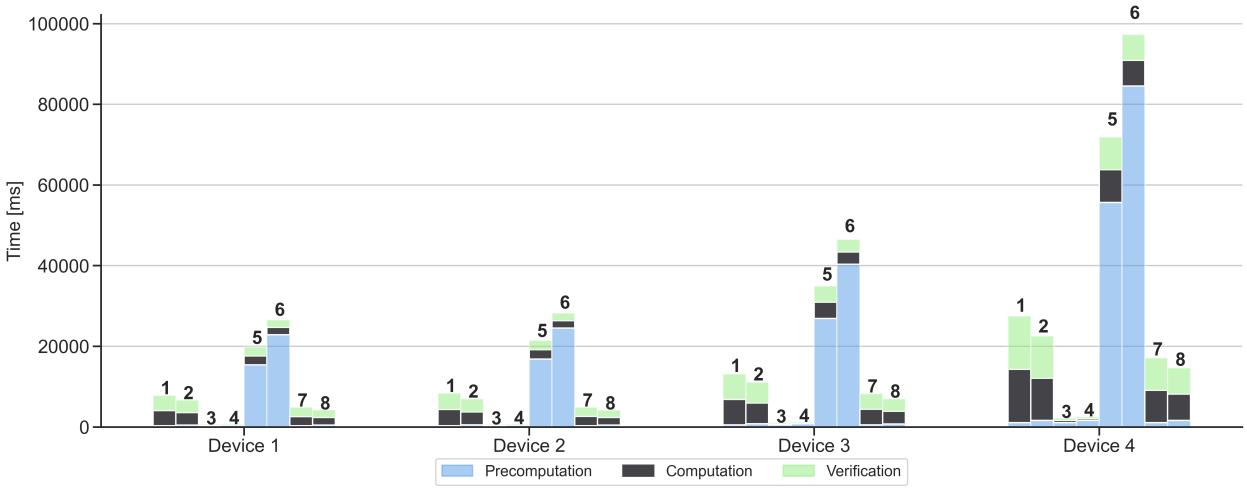
(c) **Device 3**

Figure 4.8: Total execution times of the 3 protocols on 4 different devices

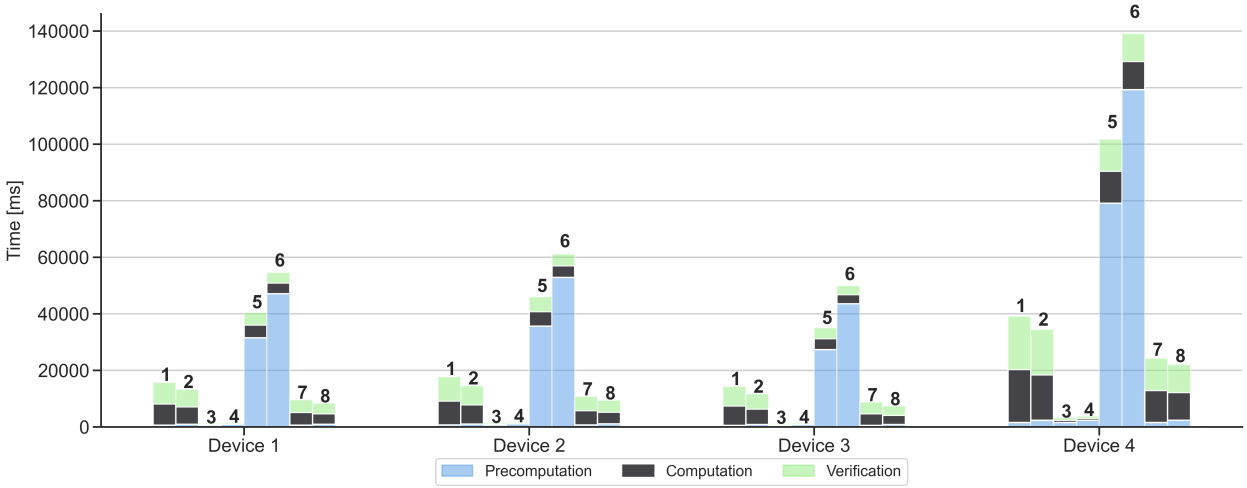


(d) **Device 4**

Figure 4.8: Total execution times of the 3 protocols on 4 different devices

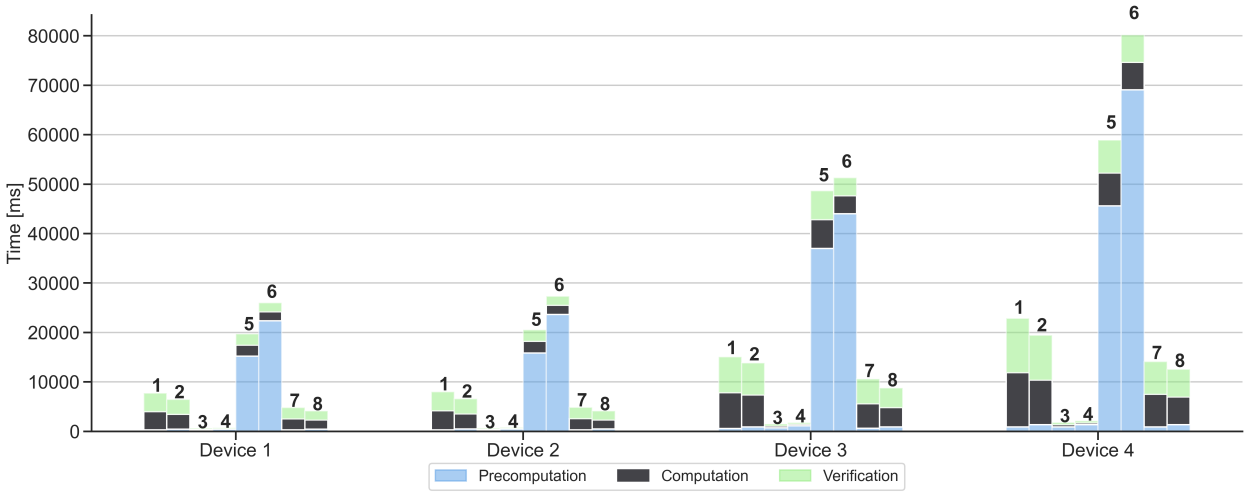


(a) **Chrome**



(b) **Firefox**

Figure 4.9: Total execution times of the 3 protocols on 3 different browsers



(c) **Edge**

Figure 4.9: Total execution times of the 3 protocols on 3 different browsers

The hardware conclusions that can be drawn are globally the same than in previous section.

We can notice that the execution time of P_2 is very small compared to others. To give precise results, the following TABLE 4.2 gives the exact times in milliseconds for the best performances on *Device 1* with the *Edge browser* and for the worst ones on *Device 4* with the *Firefox browser* :

# Device	Browser	# `Algorithm`	Precomputation time [ms]	Computation time [ms]	Verification time [ms]
1	Edge	7	306.8	162.6	50.0
		8	437.5	130.5	40.3
4	Firefox	7	1542.0	823.2	253.4
		8	2328.0	710.3	220.7

Table 4.2: Best and worst performances of P_2

Concerning the execution times of P_{3a}, we can see that `Algorithm 7` always performs better than `Algorithm 8`. It is justified by the fact that precomputations for `Algorithm 8` are longer than the ones of `Algorithm 7`. Unfortunately, we can already conclude that these execution times are practically unfeasible. Effectively, the best performances are about 20 seconds for *Device 1* on *Edge*, while the worst ones are about 90 seconds for *Device 4* on *Firefox*.

Another important point to discuss about is the comparison of P_1 with P_{3a} and P_{3b}. Some theoretical conclusions were already made about this comparison in the last paragraph of SECTION 3.3.3. Those were that :

1. The total computational execution time of P_{3b} will always be smaller than the one of P_1, but without taking into account the fact that we have a limited amount of memory usage for precomputations.

2. The total computational time of P_{3a} will always be bigger than the one of P_1, if the precomputational time needed for 1 base is 2 times superior to the one needed for 1 modular exponentiation with given precomputations on the base.

We can see on those graphs that the second point is always verified. Effectively, from Figures 4.5 and 4.6, the execution time to precompute one base for both `Algorithms 8` and `8` is at least 20 times bigger than the execution time to compute 1 modular exponentiation.

The first point is also verified by the experimental results as the bars 7 and 8 are always, respectively to the algorithm used, smaller than the bars 1 and 2. As mentioned, it is without taking into account the fact that we have a limited amount of memory usage for precomputations. Effectively, in P_{3b}, the server sends the precomputations of the n bases to the voter, where n is the number of votes.

However, if we take the measures of TABLE 4.1, `Algorithm 7` needs to send $52 \cdot n$ values to the voter. While it is $567 \cdot n$ (or can be reduced to $381 \cdot n$) values for `Algorithm 8`. For 100 votes, it represents $4,659$ $[Mo]$ for `Algorithm 7` and 50.803 $[Mo]$ (or can be reduced to 34.138 $[Mo]$) for `Algorithm 8`.

If we make the assumption that a voter with an average internet connection has a bandwidth of 1 $[Mo/s]$, it would take a little less than 5 seconds to download the precomputations for `Algorithm 7`, while this number increases up to 50 seconds (or can be reduced to 34 seconds) for `Algorithm 8`. With such an assumption, it is then practically not feasible to use `Algorithm 8`.

4.5 Creation of a ballot from a voter side

As the use of protocol P_{3a} seems to be not feasible, like the use of `Algorithm 8` for protocol P_{3b}, we can check which algorithm(s) coupled with which protocol(s) would be the best(s) choice(s) to obtain a complete ballot creation from the voter in a minimal execution time.

The FIGURE 4.10 shows the execution times to encrypt a ballot of 100 votes, so that the candidates are distributed by 10 in 10 different lists. We compare here 3 different ways to encrypt the ballot, which are numbered on the graph from 1 to 3 :

(**1**) Use P_1 and P_2 with `Algorithm 7`. Note that the precomputations of the 2 bases g and pk in P_1 are reused in P_2.

(**2**) Same than (**1**), but using `Algorithm 8`.

(**3**) Use P_{3b} and P_2 with `Algorithm 7`. Note that the precomputations of the base g in P_1 is reused in P_{3b}.

The benchmarks were performed on the 4 devices with the **Chrome** web-browser, which gives performances a little weaker than **Edge** (the best one).

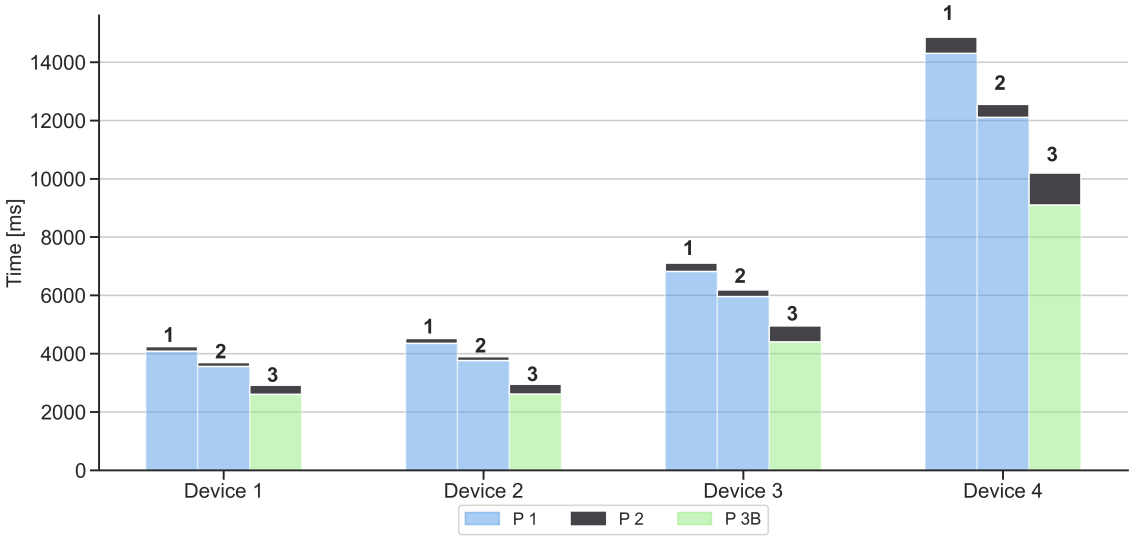


Figure 4.10: Total execution times to create a ballot of 100 votes in 3 different use of P_1, P_2 and P_{3b} on the **Chrome** web-browser

The conclusion to make from this graph is that the use of P_{3b} and P_2 with `Algorithm 7` is the best choice with a total computational time of $2,923$ seconds on *Device 1* and $10,201$ seconds on *Device 4*. But note that it requires the server to send the precomputations of the bases $(y_1, \ldots, y_n)$ to the voter, which costs here $4,659$ seconds by making the assumption that the voter has a bandwidth of 1 $[Mo/s]$. We can imagine to not sum up this time with the total computational time of the ballot's encryption by assuming that those precomputations are sent to the voter while he fills in the voting form.

If the sending of precomputations is not feasible in the election, the use of P_1 and P_2 with `Algorithm 8` seems to be the best choice with a total computational time of $3,697$ seconds on *Device 1* and $12,559$ seconds on *Device 4*.

Finally, a last analyze would be to check whether the previous uses of the protocols evolve linearly with respect to the number of votes, as the theoretical expectations would be. The following FIGURE 4.11 shows the effective linear evolution of the 3 combinations of protocols while processed between 20 to 200 votes. These benchmarks were performed with *Device 1* on the *Chrome* web-browser.

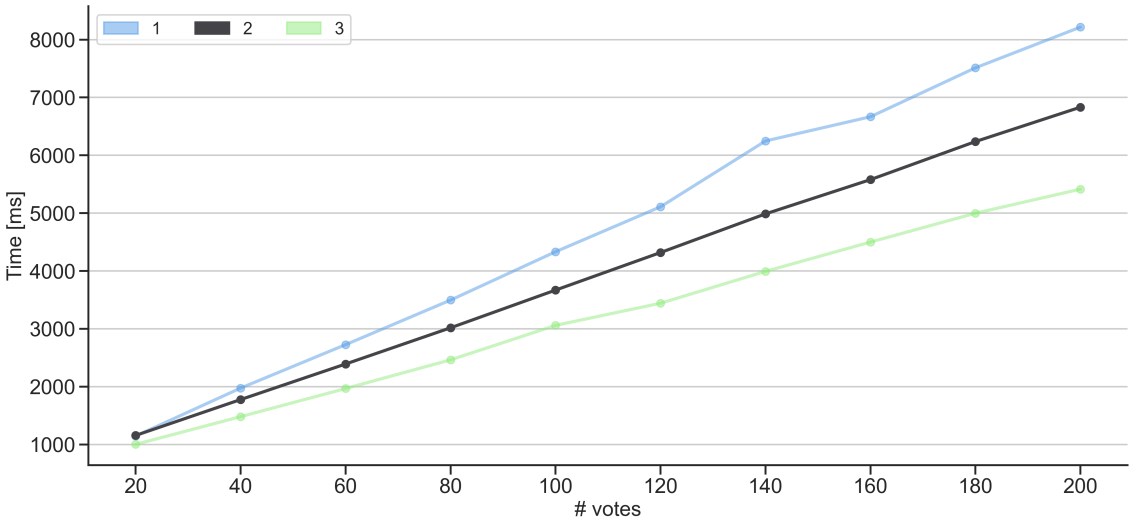


Figure 4.11: Linear evolution of the 3 combinations of protocols

Chapter 5

Conclusions

In this master thesis, I implemented the voter side process on web-browsers of an ElGamal homomorphic encryption scheme. Through this development, I optimized the computation of modular exponentiations and Zero-Knowledge Proofs (ZKP) allowing to create an encrypted ballot in several seconds. As the use of a homomorphic encryption scheme increases the speed of the tallying phase of an election, the UCL student elections that currently use a mixnet-based voting scheme will perhaps change their way of proceeding.

The first result of this paper is that the use of JavaScript with the Verificatum library to manipulate *big integers* is twice as efficient as using the Rust programming language coupled with WebAssembly with other cryptographic libraries. Indeed, the Verificatum library is the fastest yet to come for manipulating *big integers*. Moreover, the compiled Rust is very powerful when executed in command line. But when it is carried out on web-browsers with WebAssembly, its performances are greatly reduced.

The second result of this paper shows that the use of fixed-based exponentiation algorithms is an ideal choice in order to perform modular exponentiations in an ElGamal homomorphic encryption scheme. Moreover, the use of fixed-base comb method is the fastest, while it consumes more storage than other methods, even if a trade-off between execution time and storage capacity can be achieved by choosing specific parameters. If the memory size of precomputations is limited, the use of fixed-base Euclidean method is then preferred in order to perform modular exponentiations.

The third result of this paper highlights the fact that if the voter can receive precomputations from the server with a relatively good bandwidth, he would better compute one ZKP to prove the validity of its complete ballot instead of computing n different ZKP for each of the votes its ballot contains. But if the voter cannot receive such precomputations, it is guaranteed that doing the reverse will be faster. In addition, the non-interactive ZKP proving that all the non-null ciphertexts lie in a unique list of candidates has a very small execution time compared to the two other ones. In fact, it is more than 10 times faster.
In order to have an idea of these execution times, I give here some benchmarks for the computation of 100 votes encryptions in a group of 256-bits prime order and a modulus 3072-bits long. If the voter can receive precomputations from the server with a bandwidth of 1 $[Mo/s]$, the encryption of the ballot can be achieved in less than 3 seconds on a computer, while some smartphones can accomplish it in 5 seconds. If the voter has to make these precomputations himself, these benchmarks respectively grow up to 4 and 6 seconds.

From my personal point of view, I am very grateful to have had the chance to realize such a project. I found it creative, useful and the knowledge of cryptography I could acquire complete my master in data sciences engineering. Effectively, this part of science was unknown to me and I think that a lot of applications reside on those both sectors of science. In fact, I did not even know that most of the cryptographic protocols are based on prime numbers and the modulus operation. I also did not know that creating an electronic voting scheme was so important, difficult and computationally intensive.

Moreover, I hope that these results and this voter side implementation of an ElGamal homomorphic encryption scheme in JavaScript will be useful for UCL student elections, but also for other types of election with the same format.
In my opinion, these results open the possibility to organize elections of one or two hundred of candidates with a homomorphic voting scheme, as the encryption times of the ballots are reasonable.

With the intention of further developing this part of electronic voting schemes, it could be interesting to develop a mixnet and use in it the optimizations achieved in this project. Indeed, using precomputations to perform modular exponentiations with fixed-base comb method can sometimes increase twice the computational performances. Moreover, the use of these ZKP could be replicated in the mixnet in order to faster tally the results of an election. It would be interesting to compare such a mixnet with this ElGamal homomorphic encryption scheme and its results.

However, it could also be interesting to obtain benchmarks of these voting schemes with other methods to perform modular exponentiations. For instance, S. Gueron has developed optimizations on the Montgomery modular multiplication algorithm [34].

Furthermore, technology is changing fast. It is interesting to put forward the fact that the *Microsoft Edge* web-browser is a little more efficient than the *Google Chrome* one, which was a leader for years. Also, the *Mozilla Firefox* web-browser is up to twice as slow as the two others. But these web-browsers will continue to evolve with time and the computational work will be executed faster and faster.

In the current era of technology in which almost everybody has an electronic device connected to internet at hand, I am convinced that we will soon be able to organize government electronic elections. It would open the window of possibilities to more transparency of the results, more monitoring on the tallying and more joint decisions taken to enhance our society.

Bibliography

[1] Menezes A. J., Oorschot P. C. Van, Vanstone Scott A. (1996) *Handbook of Applied Cryptography*. Boca Raton: CRC Press

[2] Koblitz N. (1994), *A course in number theory and cryptography*, 2^{nd} edition. Springer-Verlag

[3] Boneh D. (1998), *The Decision Diffie-Hellman Problem*, Lecture Notes in Computer Science, vol. 1423.

[4] Wikipedia, *Homomorphic encryption*, https://en.wikipedia.org/wiki/Homomorphic_encryption, Accessed on 2021-06-13.

[5] Bernhard D., Warinschi B. (2014) *Cryptographic Voting — A Gentle Introduction*. In: Aldini A., Lopez J., Martinelli F. (eds) *Foundations of Security Analysis and Design VII*. FOSAD 2013, FOSAD 2012. Lecture Notes in Computer Science, vol 8604

[6] Ursu, Razvan Alexandru. *Efficient Implementation of Cryptographic Protocols for Electronic Voting*. Ecole polytechnique de Louvain, Université catholique de Louvain, 2019. Prom. : Pereira, Olivier. http://hdl.handle.net/2078.1/thesis:19580, Accessed on 2021-06-13.

[7] Camenish J., Stadler M. (1997) *Proof Systems for General Statements about Discrete Logarithms*. Technical Report No. 260, Dept. of Computer Science, ETH Zurich. https://crypto.ethz.ch/publications/files/CamSta97b.pdf, Accessed on 2021-06-13.

[8] Pereira O. (2017) *Internet voting with Helios*. In: Feng Hao, Peter Ryan, (eds) *Real-world Electronic Voting: Design, Analysis and Deployment*. https://www.realworldevoting.com/, Accessed on 2021-06-13.

[9] Wikipedia, *Zero-knowledge proof*, https://en.wikipedia.org/wiki/Zero-knowledge_proof, Accessed on 2021-06-13.

[10] Cramer R., Damgard I., Schoenmakers B. (1994) *Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols*. In : *Advances in Cryptology – CRYPTO '94*, Vol. 839 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 174-187

[11] Lueks W., Kulynych B., Fasquelle J., Le Bail-Collet S., Troncoso C. (2019) *zksk: A Library for Composable Zero-Knowledge Proofs*

[12] Wikipedia, *Hash function*, https://en.wikipedia.org/wiki/Hash_function, Accessed on 2021-06-13.

[13] Bulens P., Giry D., Pereira O. (2011) *Running mixnet-based elections with Helios*. Bluekrypt and Université catholique de Louvain, https://www.usenix.org/legacy/events/evtwote11/tech/, Accessed on 2021-06-13.

[14] Shoup V., Gennaro R. (2001) *Securing Threshold Cryptosystems against Chosen Ciphertext Attack*, https://www.shoup.net/papers/thresh1.pdf, Accessed on 2021-06-13.

[15] Wikipedia, *Ciphertext indistinguishability*, https://en.wikipedia.org/wiki/Ciphertext_indistinguishability, Accessed on 2021-06-13.

[16] Carmit H., Yehuda L. (2010) *Efficient Secure Two-Party Protocols*. In : *Efficient Secure Two-Party Protocols: Techniques and Constructions, Information Security and Cryptography*, Springer-Verlag Berlin Heidelberg

[17] Haustenne L., De Neyer Q., Pereira O., *Elliptic Curve Cryptography in JavaScript*, Université catholique de Louvain, ICTEAM – Crypto Group

[18] de Rooij P. (1995) *Efficient exponentiation using precomputation and vector addition chains*. In: De Santis A. (eds) Advances in Cryptology — EUROCRYPT'94. EUROCRYPT 1994. Lecture Notes in Computer Science, vol 950

[19] Wu T. (2009) *RSA and ECC in JavaScript*. http://www-cs-students.stanford.edu/~tjw/jsbn/, Accessed on 2021-06-13.

[20] Mclaughlin M. (2020) *A JavaScript library for arbitrary-precision decimal and non-decimal arithmetic*. https://mikemcl.github.io/bignumber.js/, Accessed on 2021-06-13.

[21] Wikström D. (2020) *The Verificatum JavaScript Cryptographic library (VJSC) is a state of the art cryptographic library written from scratch in pure JavaScript with zero dependencies and made for use with group-based cryptographic protocols, or other protocols that need fast arithmetic*. https://www.verificatum.org/api-vjsc/verificatum.html, Accessed on 2021-06-13.

[22] *The Rust language*. https://www.rust-lang.org/fr, Accessed on 2021-06-13.

[23] *The Rust community's crate registry*. https://crates.io/, Accessed on 2021-06-13.

[24] *Big integer implementation for Rust*. https://crates.io/crates/num-bigint, Accessed on 2021-06-13.

[25] *Modular exponentiation function from the Rust crate* ***num-bigint***. https://docs.rs/num-bigint/0.4.0/num_bigint/struct.BigInt.html#method.modpow, Accessed on 2021-06-13.

[26] *A high-performance multiple-precision arithmetic library*. https://crates.io/crates/ramp, Accessed on 2021-06-13.

[27] *Modular exponentiation function from the Rust crate* ***ramp***. https://docs.rs/ramp/0.6.0/ramp/int/struct.Int.html#method.pow_mod, Accessed on 2021-06-13.

[28] *Arbitrary-precision integers, rational, floating-point and complex numbers based on GMP, MPFR and MPC*. https://crates.io/crates/rug, Accessed on 2021-06-13.

[29] *The GNU Multiple Precision Arithmetic Library*. https://gmplib.org, Accessed on 2021-06-13.

[30] *The MPFR library is a C library for multiple-precision floating-point computations with correct rounding*. https://www.mpfr.org/, Accessed on 2021-06-13.

[31] *MPC is a C library for the arithmetic of complex numbers with arbitrarily high precision and correct rounding of the result*. http://www.multiprecision.org/mpc/, Accessed on 2021-06-13.

[32] *Rust FFI bindings for GMP, MPFR and MPC*. https://crates.io/crates/gmp-mpfr-sys, Accessed on 2021-06-13.

[33] *Software Distribution and Building Platform for Windows*. https://www.msys2.org/, Accessed on 2021-06-13.

[34] Shay Gueron, *Efficient Software Implementations of Modular Exponentiation*. Department of Mathematics, University of Haifa, Israel. https://eprint.iacr.org/2011/239.pdf, Accessed on 2021-06-13.

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