[illegible]

Homomorphic encryption is a special type of encryption that can perform computations on and between ciphertexts and reflect the results in cleartext messages. At the current state of the art, this technology is still under development and has undergone drastic improvements and will probably be used in important applications that will revolutionize Cloud Computing and remote services. This text provides an adequate mathematical foundation, which will then be used to understand the principles of homomorphic cryptography and in particular of the Paillier cryptographic system until complete understanding. The acquired knowledge was then used to build a demonstration application with the Paillier scheme on electronic elections with multiple candidates. The platform is technically described and the results obtained are presented.

[illegible]

- $\gcd(a, b)$: Greatest Common Divisor between a and b.
- $\operatorname{lcm}(a, b)$: Least Common Multiple between a and b.
- $\mathbf{M}$: Clear message space.
- m: Random variable containing a cleartext message. It follows a uniform distribution.
- $\mathbf{C}$: encrypted message space
- c: Random variable containing an encrypted message.
- $\mathbf{K}$: Space containing the keys.
- k^+: Random variable containing a public key.
- k^-: Random variable containing a private key.
- π Indicates the cryptographic scheme in use by the context.
- G: key generation function.
- $E_{k^+}: \mathbf{M} \to \mathbf{C}$ public key encryption function.
- $D_{k^-}: \mathbf{C} \to \mathbf{M}$ private key decryption function.
- Θ: Generic oracle. It can be consulted and has a context-dependent function.

#3. Modular arithmetic and number and group theory requirements

. .. .

- Let $A = \{a_1, a_2, .., a_s\}$ and $B = \{b_1, b_2, .., b_s\}$ be two sets with $|A| = |B| = s$ and $f : A \to B$

> f is injective only if it is surjective and vice versa. Consequently, if it were shown that f possesses at least one of these two characteristics then it would be shown that it also possesses the other one and therefore bijectivity.

. .. . :

- Let's assume f is an injective function: $f(a_1), f(a_2), .., f(a_s)$ are s distinct elements of B Since B contains s elements these must be all the elements of B in some order. So every element of B is reached by at least one element of A and therefore f is also surjective.
- Let's assume f is a surjective function: Each element of A has one and only one image. Since f is surjective, every element of B is obtained from at least one element of A. Then there will be at least s elements of A different through which to obtain all the elements of B. Since A has exactly cardinality s then these are exactly all the different elements of A and no element of B will be obtainable from two different elements of A . So f is also injective.

. .. .

$$\phi(n) = n \prod_{p|n} (1 - \frac{1}{p})$$

Euler's totient function tells us the amount of numbers less than n and cover me with this: $|\{\forall\, m < n : \gcd(n, m) = 1\}|$ . .. . :

- It is assumed that the numbers coprime to n are n
- From n we divide (subtract) all the multiples of the prime numbers up to n

. .. . ϕ

- Let a and b be two mutually coprime natural numbers

$$\phi(ab) = \phi(a) * \phi(b)$$

. .. . :

$$\phi(a) = a\prod_{p|a}(1-\frac{1}{p})$$

$$\phi(b) = b\prod_{p|b}(1-\frac{1}{p})$$

At that time:

$$\begin{aligned}\phi(a) * \phi(b) &= a\prod_{p|a}(1-\frac{1}{p}) * b\prod_{p|b}(1-\frac{1}{p}) \\ &= ab * \prod_{p|ab}(1-\frac{1}{p}) \\ &= \phi(ab)\end{aligned}$$

. .. . ϕ .. ..

$$\phi(p^k) = p^k(1-\frac{1}{p})$$

. .. . : Since p is prime, the only possible values for $\gcd(p^k, m)$ are $1, p, 2p, ..., p^k$ and the only way to get $\gcd(p^k, m) > 1$ is for m to be a multiple of p and there are p^{k-1} smaller multiples of p. So the remaining $p^k - p^{k-1}$ numbers are all relatively prime to p.

. .. . ϕ .. ..

- Let $n = p * q$ be the product of two prime numbers p and q

$$\phi(n^2) = n\phi(n)$$

. .. . : $\phi(n^2) = \phi((pq)^2) = \phi(p^2q^2) = \phi(p^2) * \phi(q^2) = p^2(1-\frac{1}{p}) * q^2(1-\frac{1}{q})$ (for 3.2.3. Value of ϕ for the square of a compound) $= p(p-1) * q(q-1) = p\phi(p) * q\phi(q)$ $= pq * \phi(p)\phi(q) = n\phi(n)$

. .. .ϕ . .. .

- Let a be an integer coprime to n

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

. . .. : Let S_1 be the reduced system of remainder classes modulo n (it will be better defined later).

$$S_1 = \{m_1, m_2, ..m_{\phi(n)}\}$$

Let S_2 be the reduced system of remainder classes modulo n multiplied by a.

$$S_2 = \{am_1, am_2, ..am_{\phi(n)}\}$$

(Note that the various am_k are still coprime to n as they are products of numbers coprime to n, consequently S_2 is a subgroup of S_1 with the same cardinality $\rightarrow$ S_1 and S_2 coincide) Let the product of the elements of S_1 be equal to the product of the elements of S_2.

$$m_1 * m_2 * .. * m_{\phi(n)} \equiv am_1 * am_2 * .. * am_{\phi(n)} \pmod{n}$$

$$\prod_{k=1}^{\phi(n)} m_k \equiv \prod_{k=1}^{\phi(n)} am_k \pmod{n}$$

$$\prod_{k=1}^{\phi(n)} m_k \equiv a^{\phi(n)} \prod_{k=1}^{\phi(n)} m_k$$

Since $\prod_{k=1}^{\phi(n)} m_k$ is also a coprime number with n because it is the product of numbers coprime to n, we can multiply by its multiplicative inverse and obtain:

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

. .. .

The binomial theorem describes the algebraic expansion of the different powers of a binomial:

$$(1+n)^x = \sum_{k=0}^{x} \binom{x}{k} n^k = 1 + nx + \binom{x}{2} n^2 + \ higher\ powers\ of\ n$$

The iterations of the binomial expansion in modulus n are equivalent to the exponent of the modulus (if the binomial is of the form $(1+n)$):

$$(1+n)^x \equiv 1 + nx \mod n^2 \qquad (2 \text{ iterations})$$

This is because the product of the subsequent iterations would always be a multiple of the modulus.

[illegible]

Let a and n be two integers, the order of a, denoted by $|a|$, is defined as its smallest exponent such that:

$$a^{|a|} \equiv 1 \pmod{n}$$

[illegible]: Euler's theorem (see 3.2.4. Value of ϕ as an exponent), although satisfying the congruence, is not *always* the *smallest* integer that satisfies it. In other words, the value of $\phi(n)$ is not *always* equal to $|a|$. Furthermore, by Lagrange's theorem, $|a|$ divides $\phi(n)$.

[illegible]

Let a be a natural number less than n, then all multiples of n added to a form a set called the *congruence class* or *remainder class modulo n*, denoted by $\overline{a_n}$, in which each element is congruent to a modulo n:

$$\overline{a_n} = \{\forall\ a, k \in \mathbb{Z}\ /\ a + kn \equiv a \pmod{n}\}$$

Every integer belongs to one and only one remainder class modulo n. Two integers belonging to different remainder classes are inconsistent in modulo n. For every integer n there exist n remainder classes. The set of all remainder classes modulo n is called the *complete system of remainder classes modulo n* and is denoted by $\mathbb{Z}/n$:

$$\mathbb{Z}/n = \{\overline{a_n} | a \in \mathbb{Z}\} = \{\overline{0_n}, \overline{1_n}, ..., \overline{n-1_n}\}$$

[illegible] n

The complete system of remainder classes modulo n together with the addition and multiplication operations is called the *ring of integers of* n. We define addition and multiplication on this set as:

- $\overline{a_n} + \overline{b_n} = \overline{a_n + b_n}$
- $\overline{a_n} * \overline{b_n} = \overline{(ab)_n}$

[illegible]

Any set with cardinality $\phi(n)$ in which each element is coprime to n and mutually incongruent modulo n, is called a *reduced system of remainder classes* or *multiplicative group of integers modulo n* and is denoted by $(\mathbb{Z}/n\mathbb{Z})^X$. Its elements are called the *tozients* of n.

[illegible]

The natural number x that satisfies the following linear congruence is called the multiplicative inverse:

$$ax \equiv 1 \pmod{n}$$

$\gcd(a, n) = 1$ is a necessary requirement for the existence of x. In this case x is the one and only solution (intended as the number of remainder classes that satisfy the congruence). If there exists x then any element of $\overline{x_n}$ satisfies congruence for any element of $\overline{a_n}$. Each element of the reduced set of remainder classes has a *multiplicative inverse*. In particular, if *n* is prime, then its *complete system of remainder classes* coincides with the reduced system of remainder classes from which we deduce that any element of the set has a *multiplicative inverse*. The value of *x* can be easily found through Euclid's extended algorithm.

[illegible]

Each group has a *generator set*, which is a subgroup in which combinations of its elements generate all the elements of the group. In other words, if the generating set is formed by the set $\{a, b\}$ then each element of the group can be expressed through the expression (in multiplicative notation):

$$a^i * b^j$$

for certain values of i and j. When the *generator set* has cardinality 1 then the only element of the set is said to be a primitive root modulo *n*. In this case the elements of the group are generated by the different combinations of the primitive root g with a certain exponent: $g * g = g^2, g * g^2 = g^3 ...$ (in multiplicative notation).

[illegible]

A group is said to be cyclic if it has a primitive root. If n is a prime power then the group is cyclic and has a primitive root. If n is composite then the elements of the group are the direct product of the elements of the cyclic groups that correspond to the factorization of n:

$$(\mathbb{Z}/n\mathbb{Z})^X \equiv (\mathbb{Z}/p_1^{e_1}\mathbb{Z})^X \ X \ (\mathbb{Z}/p_2^{e_2}\mathbb{Z})^X ..$$

The analogous reasoning applies to the group order:

$$\phi((\mathbb{Z}/n\mathbb{Z})^X) \equiv \phi((\mathbb{Z}/p_1^{e_1}\mathbb{Z})^X) \ X \ \phi((\mathbb{Z}/p_2^{e_2}\mathbb{Z})^X) ..$$

In the case of composite n, there is not necessarily a primitive root.

[illegible]

The Carmichael function associates to each integer n a number $\lambda(n)$ that satisfies the following congruence for each integer $a < n$ with $\gcd(a, n) = 1$:

$$a^{\lambda(n)} \equiv 1 \pmod{n}$$

[illegible]

- If p equals $2, 4$ or an odd prime power:

$$\lambda(p) = \phi(p^e)$$

- If p is equal to powers of 2 greater than 4:

$$\lambda(p) = \frac{\phi(p^e)}{2}$$

[illegible]

- Let $n = p_1^{e_1} p_2^{e_2} ..$ be the prime factorization of n

$$\lambda(n) = \operatorname{lcm}(\lambda(p_1^{e_1}), \lambda(p_2^{e_2}), ..)$$

[illegible]

- Let $n = pq$ with p and q two prime numbers:

$$\lambda(n^2) = n\lambda(n)$$

[illegible]:

$$\begin{aligned}
\lambda(n^2) &= \lambda(p^2 q^2) \\
&= \operatorname{lcm}(\phi(p^2), \phi(q^2))^* \\
&= \operatorname{lcm}(p^2(1 - \frac{1}{p}), q^2(1 - \frac{1}{q})) \\
&= \operatorname{lcm}(p(p-1), q(q-1)) \\
&= p(p-1)q(q-1) \\
&= pq(p-1)(q-1) \\
&= n\lambda(n)
\end{aligned}$$

TODO: to be reviewed I don't think it's right

$*$: (see 3.2.2. Value of ϕ for a power of a prime number)

[illegible]

An integer q modulo n is said to be a quadratic remainder if there exists another integer x modulo n such that:

$$q \equiv x^2 \pmod{n}$$

If q does not satisfy the congruence, it is called a quadratic non-remainder. Since $a^2 \equiv (-a)^2 \equiv (na)^2 \pmod{n}$, to understand which elements of the complete system of remainder classes modulo n are quadratic remainders, it is sufficient to square only the first half of the set. The number of quadratic remainders cannot therefore exceed:

- $\frac{n}{2} + 1$ (for even modulus n)
- $\frac{n+1}{2}$ (for odd n modulus)

[illegible] p

If the modulus is a prime number p then there exist exactly $\frac{p+1}{2}$ quadratic remainders and $\frac{p-1}{2}$ non-quadratic remainders. Since the modulus is prime then the complete system of remainder classes modulo p coincides with the reduced system of remainder classes modulo p, consequently each element of the set has a multiplicative inverse:

- The multiplicative inverse of a quadratic remainder is a quadratic remainder
- The multiplicative inverse of a quadratic non-remainder is a quadratic non-remainder
- The product of two quadratic non-remainders is a quadratic remainder.
- The product of a quadratic remainder and a quadratic non-remainder is a quadratic non-remainder.

[illegible]

The Legendre symbol is a multiplicative function that takes as argument an integer a and a prime number p as modulus and returns the values $1, -1, 0$ respectively if a is a quadratic residue, is a quadratic non-residue or is 0. The Legendre symbol can be calculated efficiently using the formula:

$$\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \pmod{p}$$

[illegible] p^k

A number q coprime to p is a quadratic remainder modulo all powers of p if and only if it is a quadratic remainder modulo p. More formally (with the Legendre symbol):

$$\left(\frac{a}{p^k}\right) = 1 \iff \left(\frac{a}{p}\right) = 1$$

3.12.4. Quadratic remainder of composite modulus n

- If q is a quadratic remainder modulo n then it is also a quadratic remainder modulo all prime powers $p_i^{k_j}$ that factor n.
- If q is a quadratic non-remainder then it is a quadratic non-remainder for at least one of the prime powers $p_i^{k_j}$ that factor n. One can efficiently compute whether an integer a is a quadratic remainder modulo a composite n via the Jacobi symbol.

Jacobi symbol

The Jacobi symbol is a generalization of the Legendre symbol. It takes as input an integer a and a composite integer n as its modulus and returns the values $1, -1, 0$ respectively if a is a quadratic remainder, a non-quadratic remainder, or zero. Let $n = p_1^{k_1} p_2^{k_2}$, the prime factorization of n, then the Jacobi symbol is defined as:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1^{k_1}}\right)\left(\frac{a}{p_2^{k_2}}\right) = \left(\frac{a}{p_1}\right)\left(\frac{a}{p_2}\right)$$

Where $\left(\frac{a}{p_1}\right)\left(\frac{a}{p_2}\right)$ is the Legendre symbol. Unlike the Legendre symbol, $\left(\frac{a}{n}\right) = 1$ *does not* guarantee that a is indeed a quadratic remainder. In particular, according to the properties of 3.12.1. Quadratic remainder of prime modulus p and 3.12.4. Quadratic remainder of composite modulus n, in the case where the Legendre symbol returns -1 for both prime numbers in the decomposition of n and therefore a is a non-quadratic remainder, the multiplication of the two -1 would return 1, thus providing a result that is difficult to interpret.

Integer factorization problem

> Given two integers a and $n = p_1 p_2$ where p_1 and p_2 are two unknown prime numbers, determine whether a is a quadratic remainder modulo n.

The problem could be solved efficiently by using the Jacobi symbol formula, and evaluating the results of the Legendre symbols individually, but this would require knowing the decomposition of n. Consequently, the square remainder problem is as difficult as the integer factorization problem. If we could find an algorithm that could efficiently solve the factorization of integers, then we would also be able to solve the square remainder problem.

d-th remainder

An integer a modulo n is said to be a d-th remainder if there exists another integer x modulo n such that:

$$a \equiv x^d \pmod{n}$$

The d-th remainder is a generalization of the quadratic remainder.

. .. .

> Given the integers a, d and $n = p_1 p_2$ where p_1 and p_2 are two unknown prime numbers, determine whether a is a d-th remainder modulo n.

Or, more formally:

$$a^d \equiv z \pmod{n}$$

The failure to solve this problem efficiently gives rise to what Paillier calls the "decisional composite restistance assumption" (DCRA), which is the basis of the cryptographic security of his scheme.

#4. Homomorphic encryption Homomorphic encryption allows cryptographic schemes that support it to perform computations on encrypted data. The logical mathematical structure of these schemes causes a *homomorphism* (hence the name) to exist between the set of plaintext messages and encrypted ones, where a computation on one is mirrored on the other. More formally:

- Let f be the computation to be performed
- Let π be a homomorphic cryptographic scheme that supports f

$$D_{k^-}(f(E_{k^+}(m_1), E_{k^+}(m_2))) = f(m_1, m_2)$$

[illegible]

This type of encryption was born in 1978 after the birth of RSA (when its homomorphic properties were discovered). Since then, it was not clear how to exploit them until *Craig Gentry* in 2006 theorized the first fully homomorphic scheme. Since then, this technology has made great strides, going from simple conjectures to the next new breakthrough in the world of cryptography.

[illegible]

It has received so much attention lately because of its many applications, especially cloud computing. Many companies need to perform mass computations on their data, which are protected by privacy rights and are protected by common cryptographic schemes such as AES. Consequently, it is necessary to decrypt and re-encrypt the data every time a computation is needed. This requires computational honor and data exposure, which, in addition to being a vulnerability because it puts their secrecy at risk, makes it impossible to have the computations performed by a third-party company already equipped with suitable hardware. With homomorphic encryption it is possible to share ciphertexts and perform operations on them, maintaining the secrecy of both the data and the results of the computations, which will be decipherable only by the client, the sole owner of the key, once these have returned to his hands. Currently, homomorphic encryption is still under development and its still high computational cost leaves room only for purely demonstration applications.

[illegible]

It is easily demonstrable that the behavior of any Boolean function can be reproduced through appropriate combinations of AND and XOR logic gates in a logic circuit. Since these logical operations correspond respectively to binary addition and multiplication, it will be sufficient for a cryptographic scheme to homomorphically support an unlimited number of addition and multiplication operations to produce any type of computation on the data.

A	B	$-$	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_9	Y_{10}	Y_{11}	Y_{12}	Y_{13}	Y_{14}	Y_{15}
0	0		0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1		0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0		0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

$$
\begin{aligned}
Y_0 &= A \cdot \bar{A} \\
Y_1 &= (\bar{A} \cdot \bar{B}) \\
Y_2 &= (\bar{A} \cdot B) \\
Y_3 &= (\bar{A} \cdot B) + (\bar{A} \cdot \bar{B}) \\
Y_4 &= (A \cdot \bar{B}) \\
Y_5 &= (A \cdot \bar{B}) + (\bar{A} \cdot \bar{B}) \\
Y_6 &= (A \cdot \bar{B}) + (\bar{A} \cdot B) \\
Y_7 &= (A \cdot \bar{B}) + (\bar{A} \cdot B) + (\bar{A} \cdot \bar{B})
\end{aligned}
\qquad
\begin{aligned}
Y_8 &= (A \cdot B) \\
Y_9 &= (A \cdot B) + (\bar{A} \cdot \bar{B}) \\
Y_{10} &= (A \cdot B) + (\bar{A} \cdot B) \\
Y_{11} &= (A \cdot B) + (\bar{A} \cdot B) + (\bar{A} \cdot \bar{B}) \\
Y_{12} &= (A \cdot B) + (A \cdot \bar{B}) \\
Y_{13} &= (A \cdot B) + (A \cdot \bar{B}) + (\bar{A} \cdot \bar{B}) \\
Y_{14} &= (A \cdot B) + (A \cdot \bar{B}) + (\bar{A} \cdot B) \\
Y_{15} &= (A \cdot B) + (A \cdot \bar{B}) + (\bar{A} \cdot B) + (\bar{A} \cdot \bar{B})
\end{aligned}
$$

Fig 1: Using an XOR gate we can easily generate a NOT and with XOR, AND and NOT we can create any of the 16 possible functions that combine 2 binary variables.

[illegible]

[illegible]

They support the evaluation of circuits consisting of only one type of operation (e.g. addition or multiplication).

Examples:

- RSA cryptographic scheme:
- Supports unlimited number of modulo multiplications.
- Entering entropy (which is not provided by default in the scheme) makes it unusable in this field.
- The homomorphic property is therefore:

$$
\begin{aligned}
E(m_1) * E(m_2) &= m_1^e * m_2^e \pmod n \\
&= (m_1 m_2)^e \pmod n \\
&= E(m_1 m_2)
\end{aligned}
$$

- El Gamal cryptographic scheme:
- Supports unlimited number of modulo multiplications.
- $k^+ = (G, g, q, h)$ where G is a cyclic group of order q with generator g and $h = g^x$
- $k^- = x$
- $E(m) = (g^r, m * h^r)$
- r is a random number in $\{0, 1, .., q-1\}$
- The homomorphic property is therefore:

$$\begin{aligned} E(m_1) * E(m_2) &= (g^{r_1}, m_1 * h^{r_1})(g^{r_2}, m_1 * h^{r_2}) \\ &= (g^{r_1+r_2}, (m_1 * m_2)h^{r_1+r_2}) \\ &= E(m_1 m_2) \end{aligned}$$

- Goldwasser-Micali cryptographic scheme:
- Supports unlimited number of XOR logical operations.
- $k^+ = (n, x)$ where n is the modulus and x is a quadratic non-remainder
- The one-bit encryption b is $E = x^b r^2 \pmod n$
- r is a random number in $\{0, 1, .., n-1\}$
- The homomorphic property is therefore:

$$\begin{aligned} E(m_1) * E(m_2) &= x^{b_1} r_1^2 x^{b_2} r_2^2 \pmod n \\ &= x^{b_1+b_2}(r_1 r_2)^2 \pmod n \\ &= E(m_1 \oplus m_2) \end{aligned}$$

- Paillier cryptographic scheme:
- Supports unlimited number of modulo additions.
- $k^+ = (n, g)$ where n is the module and g a generator
- $E = g^m r^n \pmod n$
- r is a random number in $\{0, 1, .., n-1\}$
- The homomorphic property is therefore:

$$\begin{aligned} E(m_1) * E(m_2) &= g^{m_1} r_1^n g^{m_2} r_2^n \pmod n \\ &= g^{m_1+m_2}(r_1 + r_2)^n \pmod n \\ &= E(m_1 + m_2) \end{aligned}$$

[illegible]

They support the evaluation of circuits consisting of any computation but of a predetermined depth, that is, after a certain number of operations the cipher becomes indecipherable. This is because, even though homomorphic schemes are malleable by nature, it is still necessary for the ciphertexts to be secret and safe from *known* and *chosen chiphertext attacks*. This means injecting entropy into each ciphertext so that a plaintext message has a different ciphertext each time it is encrypted. In the most common cryptographic schemes, this operation is done at the end of the encryption through a random piece of information that is appended to the ciphertext. However, by introducing entropy at the end of the encryption, the homomorphic link of the scheme is obviously lost. It is therefore necessary to modify the scheme and add the entropy phase to the whole of the encryption and decryption functions. However, after several computations of the intermediate results that are encrypted from time to time, the entropy percentage grows more and more, causing the cipher to deviate too much and making it no longer decipherable.

[illegible]

They support any type of circuit and any type of computation. Fully homomorphic schemes achieve this goal through a phase called "bootstrapping", theorized by *Craig Gentry* in 2006, which consists in reducing the imbalance that generates entropy. Because of this operation, they are the most computationally expensive. Examples:

- CKKS
- GSW

[illegible]

The Paillier cryptosystem, invented by *Pascal Paillier* in 1999, is a probabilistic, asymmetric, and partially homomorphic cryptographic scheme. The security of the system is based on the decision-making assumption of the complex remainder. Formally we will denote the Paillier scheme as $\pi_P = (G, E_{k^+}, D_{k^-})$. The different algorithms that make up the scheme will be presented with the relative formulas and subsequently an in-depth analysis with the mathematical-logical explanations of these. In the presentation of the algorithms there are also tables that build an example. The numbers have been chosen deliberately small to facilitate the reader's understanding.

[illegible]

Let S_n be a multiplicative subgroup of $\mathbb{Z}^*_{n^2}$ of the form:

$$S_n = \{u < n^2 | u \equiv 1 (\mathrm{mod}\ n)\}$$

S_n in particular contains all numbers coprime to n and raised to the Carmichael function (see: 3.11. Carmichael function).

Then we define the function $L : S_n \to Z_n$ as:

$$L(x) = \frac{x-1}{n}$$

Decrementing the numbers in S_n by 1 yields a number divisible by n. Dividing by n yields a non-zero number less than n.

[illegible] G [illegible]

1. Choose two large prime numbers p and q with $\gcd(pq, \phi(pq)) = 1$.
2. Calculate $n = pq$.
3. Calculate $\lambda(n)$
4. Choose $g \in \mathbb{Z}^*_{n^2},\ |g| = kn\ (\mathrm{mod}\ n^2),\ \gcd(pq, L(g^{\lambda(n)} \bmod n^2) = 1$

[illegible] : $k^+ = (n, g)$: public key $k^- = \lambda(n)$: private key

[illegible] : g is a non-zero, random number in the reduced system of remainder classes modulo n^2, such that the order of g is a multiple of n in $\mathbb{Z}_{n^2}$ and $L(g^{\lambda(n)} \bmod n^2)$ is coprime to n. If g does not satisfy these conditions, it will be necessary to randomly choose a different g. [illegible] : The constraints applied to the value of g are motivated later (see: 5.5. Mathematics behind the scenes).

....	....
$p = 7, q = 11, n = pq = 77, n^2 = 5929$	I choose two prime numbers and construct n
$\gcd(77, \phi(77)) = 1$	I verify that n is coprime to $\phi(n)$
$\lambda(77) = \text{lcm}(\phi(7), \phi(11)) = \text{lcm}(6, 10) = 30$	I calculate the value of lambda
$g = 5652$	I randomly choose a value of $g \in \mathbb{Z}^*_{n^2}$
$\lvert 5652 \rvert_{5929} = 2310 = 30 * 77$	I verify that the order of g divides n into $\mathbb{Z}_{n^2}$
$L(5652^{30} \bmod 5929) = 51, \gcd(77, 51) = 1$	I verify that $L(g^{\lambda(n)} \bmod n^2)$ is coprime to n
$k^+ = (77, 5652)$	The public key is formed by the pair n and g
$k^- = 30$	The private key is the value of $\lambda(n)$

.. .. . E_{k^+} . . ..

▌.. .. .. : k^+: public key $m \in \mathbb{Z}_n$: clear message

1. Choose a random integer $r \in \mathbb{Z}^*_n$
2. Calculate:

$$c = E_{k^+}(m) = g^m r^n \mod n^2$$

▌. . .. .. : $c \in \mathbb{Z}^*_{n^2}$: encrypted message

. : The owner of the private key k_{k^-} is able to decrypt c even without knowing the value of r
.

....	....
$m = 42$	The clear message transformed into a sequence of bits
$r = 23$	r is a non-zero random integer in $\mathbb{Z}^*_n$
$c = 5652^{42} 23^{77} \pmod{5929} \equiv 4019 * 606 \pmod{5929} \equiv 4624 \pmod{5929}$	Compute $E_{k^+}(m) = g^m r^n \pmod{n^2}$

. .. ..D_{k^-}. . ...

.. .. .. : k^-: private key c: encrypted message

1. Calculate:

$$m = D_{k^-}(c) = \frac{L(c^{\lambda(n) \bmod n^2})}{L(g^{\lambda(n)} \bmod n^2)} \bmod n$$

. . .. : $m \in \mathbb{Z}_n$: clear message

. : $g \in \mathbb{Z}^*_{n^2}$ is coprime to n^2 and consequently also to n. This makes g suitable for raising to the Carmichael function (see: 3.11. Carmichael function) and we will have that $g^{\lambda(n)} \equiv 1 \pmod n$. $g^{\lambda(n)} \bmod n^2$ is part of S_n and the function L can be applied to it and the result will belong to Z_n. $L(g^{\lambda(n)} \bmod n^2)$ by the hypotheses of 5.2. G: key generation is coprime to both p and q and will then belong to $\mathbb{Z}^*_n$ and by 3.8. Multiplicative inverse it is invertible. Calculating $L(g^{\lambda(n)} \bmod n^2)^{-1}$ is necessary to decrypt c but only once since it depends only on the values of the public key k^+ (in this case $k^+ = (77, 5652)$) and will be the same for each subsequent decryption.

. ...	. ...
$L(5652^{30} \bmod 5929) = 51$	Compute $L(g^{\lambda(n)} \bmod n^2)$ from the public key k^+
$L(5652^{30} \bmod 5929)^{-1} \equiv 74 \bmod 77$	Calculate the inverse of $L(g^{\lambda(n)} \bmod n^2)$ in modulus n
$L(4624^{30} \bmod 5929) = 63$	Calculation $L(c^{\lambda(n) \bmod n^2})$
$m = D_{k^-}(c) = 63 * 74 \pmod{77} \equiv 42 \pmod{77}$	I multiply $L(c^{\lambda(n)} \bmod n^2)$ by $L(g^{\lambda(n)} \bmod n^2)^{-1}$ in modulus n and I get m

. .. .

This chapter will provide a logical-mathematical proof to the algorithms of the scheme. In particular, several hypotheses, theorems and formulas, assumed as fact until now, will be demonstrated until arriving at the decryption formula:

$$m = D_{k^-}(c) = \frac{L(c^{\lambda(n) \bmod n^2})}{L(g^{\lambda(n)} \bmod n^2)} \bmod n$$

[illegible] E_{k^+}

The encryption formula is defined as:

$$E_{k^+} : \mathbb{Z}_n \times \mathbb{Z}_n^* \rightarrow \mathbb{Z}_{n^2}^*$$

$$E_{k^+}(m) = g^m r^n \mod n^2$$

In chapter 5.3. E_{k^+}: encryption one of the conditions imposed for a suitable value of g is that its order be a multiple of n. This is necessary to make the encryption formula bijective as we will demonstrate shortly. In 3.1. Bijectivity it is shown that a function f is bijective if the domain and the codomain have the same cardinality and if f is injective or surjective. Proving that a function is bijective is important because it guarantees the existence of an inverse, or deciphering, formula. With these principles in mind, the following theorem is proved:

[illegible] : $|g| = kn$: the order of g is a non-zero multiple of n . [illegible] : E_{k^+} is bijective.

[illegible] :

- $\|\mathbb{Z}_{n^2}^*\| = \phi(n^2) = n\phi(n) = \|\mathbb{Z}_n \times \mathbb{Z}_n^*\|$ (see: 3.2.3. Value of ϕ for the square of a composite). The cardinality of the codomain is therefore equal to that of the domain.
- Let us assume that $E_{k^+}(m_1) = E_{k^+}(m_2)$: $g^{m_1} r_1^n \equiv g^{m_2} r_2^n \mod n^2$ $g^{m_1 - m_2} (\frac{r_1}{r_2})^n \equiv 1 \mod n^2$ Let's raise both sides of $\lambda(n)$: $g^{\lambda(n)(m_1 - m_2)} (\frac{r_1}{r_2})^{\lambda(n)n} \equiv 1 \mod n^2$ r_1 and r_2 belong to $\mathbb{Z}_n^*$, consequently their quotient $\frac{r_1}{r_2}$ also belongs to $\mathbb{Z}_n^*$ and I can apply 3.11.3. Carmichael function of the square of a composite: $g^{\lambda(n)(m_1 - m_2)} \equiv 1 \mod n^2$ This result implies that $\lambda(n)(m_1 - m_2)$ is a multiple of the order of g and hence, by hypothesis, also a multiple of n. Consequently n divides $\lambda(n)(m_1 - m_2)$ and since $\gcd(\lambda(n), n) = 1$ then n must necessarily divide $(m_1 - m_2)$. More formally: $m_1 - m_2 \equiv 0 \bmod n$ $m_1 \equiv m_2 \bmod n$. Since m_1 and m_2 are elements of $\mathbb{Z}_n$ their congruence in modulo n determines their equality in $\mathbb{Z}$. Going back to the equation $g^{m_1 - m_2} (\frac{r_1}{r_2})^n \equiv 1 \mod n^2$, setting $m_1 = m_2$ we get: $(\frac{r_1}{r_2})^n \equiv 1 \mod n^2$ $r_1^n \equiv r_2^n \mod n^2$ Congruence is satisfied if $r_1 \equiv r_2 \mod n$. Let's consider $r_1 \equiv r_2 + \alpha * n \mod n^2$: $r_1^n \equiv (r_2 + \alpha * n)^n \mod n^2$ $r_1^n \equiv (r_2 + \alpha * n)^n \equiv r_2^n + n * (\alpha * n) * r_2^{n-1} +$ [powers higher than n] $\equiv r_2^n + \alpha * n^2 * r_2^{n-1} \equiv r_2^n \mod n^2$ (see: 3.3. Binomial theorem in module) Since r_1 and r_2 are elements of $\mathbb{Z}_n^*$ their congruence in modulo n determines their equality in $\mathbb{Z}$. So $m_1 = m_2$ and $r_1 = r_2$.

The bijectivity of E_{k^+} allows us to say with certainty that for every $w \in \mathbb{Z}_{n^2}^*$, with n fixed and g a nonzero multiple of n, the pair (m, r) such that $E_{k^+}(m, r) = w \mod n^2$ is unique. To simplify the notation, let us indicate with $[\![w]\!]_g$ that particular and unique value of x which, applied to E_{k^+}, produces the unique and particular w.

[illegible] D_{k^-}

[illegible]

Note that the codomain of E_{k^+} includes all elements of $\mathbb{Z}_{n^2}^*$ and that g itself is an element of this set. It is therefore possible that, for another element $x \in \mathbb{Z}_{n^2}^*$ one can compute $[\![g]\!]_x$ for the exact g chosen.

$$(1 + n)^n \equiv 1 + n * n \equiv 1 \mod n^2$$

The order of $(1+n)$ is n which is a non-zero multiple of n and $(1+n)^{n-1}$ is its multiplicative inverse (this shows that $(1+n)$ clearly belongs to $\mathbb{Z}_{n^2}^*$, see: 3.8. Multiplicative inverse). $(1+n)$ then satisfies the properties as a generator and is suitable as an "other element" and we can compute g:

$$g = E_{(n,(1+n))}(t,z) = (1+n)^t z^n \mod n^2$$

for the unique pair (t,z) where $t = [\![g]\!]_{(1+n)}$ and is called the g-th remainder class. Recall that when encrypting a message m, we compute $c = g^m r^n \mod n^2$. Replacing the value g with the previous expression:

$$\begin{aligned} c = g^m r^n &\equiv \left[(1+n)^{[\![g]\!]_{(1+n)}} z^n\right]^m r^n \mod n^2 \\ &\equiv (1+n)^{m[\![g]\!]_{(1+n)}} z^{nm} r^n \mod n^2 \\ &\equiv (1+n)^{m[\![g]\!]_{(1+n)}} (z^m r)^n \mod n^2 \\ &\Rightarrow c = E_{(n,(1+n))}(m[\![g]\!]_{(1+n)}, z^m r) \end{aligned}$$

(Note that $z \in \mathbb{Z}_n^* \rightarrow z^m \in \mathbb{Z}_n^*$ and that $r \in \mathbb{Z}_n^*$, so $z^m r \in \mathbb{Z}_n^*$) So, by definition:

$$[\![c]\!]_{(1+n)} = m[\![g]\!]_{(1+n)}$$

$$m = [\![c]\!]_{(1+n)} * [\![g]\!]_{(1+n)}^{-1} \mod n$$

Where $[\![g]\!]_{(1+n)}^{-1}$ is the multiplicative inverse. (The product is computed in modulo n since both factors belong to $\mathbb{Z}_n$.) So if we can calculate the value of $[\![g]\!]_{(1+n)}^{-1}$, which will always be the same for every c, then the decryption process will consist of determining $[\![c]\!]_{(1+n)}$ and multiplying it by the value of $[\![g]\!]_{(1+n)}^{-1}$ modulo n.

In this chapter we will see how the factors that make up the decryption formula D_{k^-}, that is $L(c^{\lambda(n) \bmod n^2})$ and $L(g^{\lambda(n)} \bmod n^2)^{-1}$ are actually expressible in the form of remainder classes and that their product results in the decryption of the message m.

$L(g^{\lambda(n)} \bmod n^2)^{-1}$

Let's consider:

$$\begin{aligned} g^{\lambda(n)} &\equiv \left[(1+n)^{[\![g]\!]_{(1+n)}} z^n\right]^{\lambda(n)} \\ &\equiv (1+n)^{\lambda(n)[\![g]\!]_{(1+n)}} z^{\lambda(n)n} \mod n^2 \end{aligned}$$

We apply Carmichael's theorem ($z^{\lambda(n)n} \equiv 1 \mod n^2$) and the binomial theorem in modulo:

$$\begin{aligned} g^{\lambda(n)} &\equiv (1+n)^{\lambda(n)[\![g]\!]_{(1+n)}} \mod n^2 \qquad \equiv 1 + \lambda(n)[\![g]\!]_{(1+n)} n + [powers\ highest\ of\ n] \mod n^2 \\ &\equiv 1 + \lambda(n)[\![g]\!]_{(1+n)} n \mod n^2 \end{aligned}$$

Now applying the L function to the result obtained:

$$\begin{aligned} L(g^{\lambda(n)} \mod n^2) &\equiv L(1 + \lambda(n)[\![g]\!]_{(1+n)} n) \mod n \\ &\equiv \frac{(1 + \lambda(n)[\![g]\!]_{(1+n)} n) - 1}{n} \mod n \\ &\equiv \frac{\lambda(n)[\![g]\!]_{(1+n)} n}{n} \mod n \\ &\equiv \lambda(n)[\![g]\!]_{(1+n)} \mod n \end{aligned}$$

So $L(g^{\lambda(n)} \mod n^2) \equiv \lambda(n)[\![g]\!]_{(1+n)} \mod n$ and in chapter 5.4. D_{k^-}: decryption we proved that $L(g^{\lambda(n)} \mod n^2)$ belongs to $\mathbb{Z}_n^*$ and has a multiplicative inverse. Consequently it is certain that $(\lambda(n)[\![g]\!]_{(1+n)})^{-1} \mod n$ exists.

. .. . $L(c^{\lambda(n) \bmod n^2})$

Let's consider:

$$\begin{aligned} c^{\lambda(n)} &\equiv \left[(1+n)^{[\![c]\!]_{(1+n)}} r^n\right]^{\lambda(n)} \mod n^2 \\ &\equiv (1+n)^{[\![c]\!]_{(1+n)}\lambda(n)} r^{n\lambda(n)} \mod n^2 \end{aligned}$$

We apply Carmichael's theorem ($r^{\lambda(n)n} \equiv 1 \mod n^2$) and the binomial theorem in modulo:

$$\begin{aligned} c^{\lambda(n)} &\equiv (1+n)^{[\![c]\!]_{(1+n)}\lambda(n)} \mod n^2 \qquad \equiv 1 + \lambda(n)[\![c]\!]_{(1+n)} n + [powers\ higher\ than\ n] \mod n^2 \\ &\equiv 1 + \lambda(n)[\![c]\!]_{(1+n)} n \mod n^2 \end{aligned}$$

Now applying the L function to the result obtained:

$$\begin{aligned} L(c^{\lambda(n)} \mod n^2) &\equiv L(1 + \lambda(n)[\![c]\!]_{(1+n)} n) \mod n \\ &\equiv \frac{(1 + \lambda(n)[\![c]\!]_{(1+n)} n) - 1}{n} \mod n \\ &\equiv \frac{\lambda(n)[\![c]\!]_{(1+n)} n}{n} \mod n \\ &\equiv \lambda(n)[\![c]\!]_{(1+n)} \mod n \end{aligned}$$

. .. . ……..……..…… …. ….

$$L(g^{\lambda(n)} \mod n^2)^{-1} \equiv (\lambda(n)[\![g]\!]_{(1+n)})^{-1} \mod n$$

$$L(c^{\lambda(n)} \mod n^2) \equiv \lambda(n)[\![c]\!]_{(1+n)} \mod n$$

So:

$$\begin{aligned} L(c^{\lambda(n)} \mod n^2) * L(g^{\lambda(n)} \mod n^2)^{-1} &\equiv \lambda(n)[\![c]\!]_{(1+n)} * (\lambda(n)[\![g]\!]_{(1+n)})^{-1} \mod n \\ &\equiv \lambda(n)[\![c]\!]_{(1+n)} * \lambda(n)^{-1}[\![g]\!]^{-1}_{(1+n)} \mod n \\ &\equiv [\![c]\!]_{(1+n)} * [\![g]\!]^{-1}_{(1+n)} \mod n \\ &\equiv m \bmod n \end{aligned}$$

Note that for a given choice of public key $k^+ = (n, g)$, the value of $L(g^{\lambda(n)} \mod n^2)^{-1}$ is always the same and needs to be computed only once to decrypt an infinite number of messages. This means that the decryption process consists of an exponentiation modulo n^2, the simple computation of the function $L()$, and a simple multiplication modulo n, making the decryption a computationally economical algorithm where the most expensive operation is a modular exponentiation.

. .. .

In this chapter the various homomorphic properties of the Paillier scheme are explained. Each property is formally described and followed by a practical example. For all examples assume:

- $p = 7, q = 11$
- $n = 77, n^2 = 5929$
- $k^+ = (n, g) = (77, 5652)$
- $k^- = \lambda(n) = 30$
- $L(g^{\lambda(n)} \bmod n^2)^{-1} \equiv 74 \mod n$

. .. .

. ..... : Multiplying ciphertexts results in adding plaintext messages modulo n. . ... :

$$D_{k^-}(E_{k^+}(m_1) * E_{k^+}(m_2) \mod n^2) \equiv m_1 + m_2 \mod n$$

. .. . : Let us be:

- $c_1 = E_{k^+}(m_1) = g^{m_1} r_1^n \mod n^2$
- $c_2 = E_{k^+}(m_2) = g^{m_2} r_2^n \mod n^2$

At that time:

$$\begin{aligned} c_1 * c_2 &\equiv g^{m_1} r_1^n * g^{m_2} r_2^n \mod n^2 \\ &\equiv g^{m_1} g^{m_2} r_1^n r_2^n \mod n^2 \\ &\equiv g^{m_1 + m_2} (r_1 r_2)^n \mod n^2 \end{aligned}$$

Note that $r_1, r_2 \in \mathbb{Z}_n^*$ and that $r_1 * r_2 \in \mathbb{Z}_n^*$ then the multiplication operation generates a new "random" ciphertext from the product of the two random numbers r_1 and r_2 but containing the correct plaintext information ($m_1 + m_2$).

....	....
$(m_1, r_1) = (42, 23)$	I transform the first message into a sequence of bits and choose a random number to obfuscate
$(m_2, r_2) = (15, 61)$	I transform the second message into a sequence of bits and choose a random number to obfuscate
$E_{k^+}(m_1, r_1) = 4624 \mod 5929$	I encrypt the first pair
$E_{k^+}(m_2, r_2) = 1306 \mod 5929$	I encrypt the second pair
$4624 * 1306 = 6038944 \equiv 3222 \mod 5929$	$E_{k^=}(m_1, r_1) * E_{k^=}(m_2, r_2) \mod n^2$
$D_{k^-}(3222) = 57 = 42 + 15$	Decrypting the new ciphertext gives a clear message that is the sum of the two previous ones

. .. .

. ... : Exponentiating a cipher with a plaintext message results in the multiplication of the two plaintext messages. . ... :

$$D_{k^-}(E_{k^+}(m_1)^{m_2} \mod n^2) \equiv m_1 * m_2 \mod n$$

. .. . : Is:

- $c = E_{k^+}(m_1) = g^{m_1} r_1^n \mod n^2$

At that time:

$$c^{m_2} = (E_{k^+}(m_1))^{m_2} \equiv g^{m_1 m_2} r^{n m_2} \mod n^2$$

Note that, during the decryption process, the operation $c^{\lambda(n)}$ is performed and $r^{n\lambda(n)m_2} \equiv 1^{m_2} \equiv 1 \mod n^2$.

. ...	. ...
$(m_1, r) = (42, 23)$	I transform the first message into a sequence of bits and choose a random number to obfuscate
$m_2 = 15$	I transform the second message into a sequence of bits
$E_{k^+}(m_1, r) = 4624 \mod n^2$	I encrypt the first pair
$4624^{15} = 5391 \mod 5929$	Exponentiation of the ciphertext for the second message
$D_{k^-}(5391) \equiv 14 \mod n \equiv 630 \mod 5929 = 14 + 8 * 77 \equiv 15 * 42 \mod 77$	Decrypting the new ciphertext gives a cleartext message that is the product of the two previous ones.

. : In the example there is a very small module and the true value of the product is lost. In reality a very large value of n is used so that most of the products fall within a remainder class of the module.

. .. .

. ..... : You can change the ciphertext value without changing the cleartext value.
. ..... :

$$D_{k^-}(E_{k^+}(m) * g^{nx} \mod n^2) \equiv nx + m \mod n \equiv m \mod n$$

. .. . : Let us be:

- $c = E_{k^+}(m) = g^m r^n \mod n^2$

At that time:

$$\begin{aligned} c * g^{nx} &\equiv g^m r^n * g^{nx} \mod n^2 \\ &\equiv g^{nx+m} r^n \mod n^2 \end{aligned}$$

At the end of the decryption process the message we will obtain will actually be $nx + m$ but in modulo n this will be congruent to m. Although with this multiplication we have not altered the clear message, we have completely changed the encrypted message. An immediate consequence of self-obfuscation is error detection: if the two ciphertexts, c and c' where c' is simply self-obfuscated c (they contain the same cleartext information), are transmitted over a noisy channel, then a transmission error can be identified if c and c' do not decrypt to the same cleartext message. Although it requires a little extra computation, self-obfuscation prevents the scenario where c is sent twice and suffers the same transmission error, resulting in two identical tampered copies being received, misleading the recipient into believing they have received an error-free message. If c and c' suffer the same transmission error they will still not decrypt to the same cleartext message, preserving the integrity of the communication.

....	....
$5652^{(77*15)} = 5652^{1155} \equiv 4115 \mod 5929$	I compute $g^{nx} \mod n^2$
$4624 * 4115 = 19027760 \equiv 1599 \mod 5929$	Compute $E_{k^+}(m) * g^{nx} \mod n^2$
$D_{k^-}(1599) = 42 \mod 77$	Decrypting the new ciphertext gives a cleartext message that is the same as the previous ciphertext

The safety of the Paillier scheme is based on the decision assumption of the composite remainder (see: 3.15. D-th remainder problem). There is currently no algorithm that can efficiently determine whether a given number is a composite n-th remainder. Assuming we could consult an oracle that could efficiently solve *DCRA*, then we could break the cryptosystem.

Is

- Θ an oracle that can solve *DCRA* efficiently
- $x \in \mathbb{Z}$

At that time,

$$\Theta(x,n) = \begin{cases} 1, & \text{if } x \text{ is an nth modulus remainder } n \\ 0, & \text{if } x \text{ is not an n-th remainder modulo } n \end{cases}$$

A ciphertext containing the value 0 as the plaintext message is always an n-th remainder:

$$c = g^0 r^n \equiv 1 * r^n \equiv r^n \mod n^2$$

In this case we could know if $m = 0$ simply by asking the oracle:

$$\Theta(c,n) = 1$$

In the case $m \neq 0$, we could homomorphically subtract any other number k and check whether the result is an n-th remainder up to n or when the oracle answers yes. Then we would know that $m = k$:

$m = 42,\ k = 41$	I decide on a value of k
$c = E_{k^+}(42) * E_{k^+}(-41)$	Subtract the constant k from m (see: 5.6.1. Multiplication of ciphertexts)
$\Theta(c,n) = 0$	c is not an n-th remainder
$k = 42$	I decide another value of k
$c' = E_{k^+}(42) * E_{k^+}(-42)$	I subtract the new constant k from m and obtain a new ciphertext c'
$\Theta(c',n) = 0$	c' is an n-th remainder and $m = k$

[illegible]

[illegible]

Elections and voting are the fundamental instrument of expression of the people as well as a fundamental building block for the construction of any democratic state (voting has existed since the times of the ancient Roman Empire). The most widespread electoral system, in most countries, is currently the physical one, with seats and ballots. Although this modus operandi is highly tested and guarantees adequate security, it also has disadvantages including:

- Costs: the economic amount to finance an election is not at all insignificant, just think of the last election for the presidency of the USA which cost citizens a good *14 billion dollars*.
- Mobilization: To vote, every citizen is forced to go to the nearest polling station and, unfortunately, a significant percentage of them did not vote for this reason too.
- Human error: Human error, as in any system where physical participation is required, must be taken into account and cannot be eliminated. Migrating to an electronic system where voting can be done remotely would obviously eliminate these disadvantages and would also ensure:
- Transparency: since it would actually be possible to see the list of votes and who participated
- Immediacy: the system would be updated in real time and votes are counted immediately
- Non-repudiation: the system keeps track of who actually voted and the information is public so that any statistical analysis on the voting population can be performed.

[illegible]

However, implementing remote voting hides several problems:

- Secrecy: the vote must *always* remain secret at all stages.
- Decryption: Management and confidentiality of the decryption key.
- Counting: It is necessary to develop a system that, given a set of votes, can count them and determine the result.
- Validity: The secret ballot must also be a valid ballot. A mechanism for verifying validity is required. Even the elections we all know have vulnerabilities, the difference is that those of the physical system *do not scale well* with the number of voters: it requires the involvement of a lot of people and requires high costs even for medium-sized elections. To date, there is still no real safe and reliable standard of electronic voting for large-scale elections. It is also true, however, that many countries are investing time and money in research and development in this field and with high probability, in a not so distant future, electronic voting will become the norm and part of everyone's life.

[illegible]

[illegible]

Minosse (name inspired by the famous character from Dante's work) is a web application, client-server, for electronic voting, real-time data collection and counting and the election of multiple candidates. It is currently available online at: `193.70.2.109:3000`. Minosse is a completely open source project under the *MIT* license. The repository with the source code can be found at: https://bitbucket.org/Giulio_Golinelli/tesi-repo/.

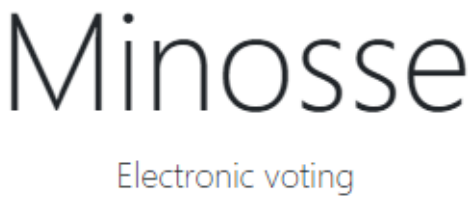

Fig 2: Minos, logo.

[illegible]

[illegible]

One of the major flaws of many existing e-voting systems is the significant component of distrust that is often created in the system-user relationship. In fact, many of these platforms, to ensure security and integrity, close themselves off by becoming closed-source, even at the hardware level (Since 2002, the USA has adopted the so-called DRE, or Direct Recording Electoric voting, which has produced 28.9% of the votes, while in 2004 India has entirely entrusted the elections to its EVM or Electronic Voting Machine). This encapsulation of the platform, whether software or hardware or both, even if justified with an increase in security, casts doubt on the reliability of the system that could have been boycotted by those who have the possibility and no one could discover it. Even the paper electoral system we are used to is a closed system, but the multitude of people with whom it is carried out makes any attempt at boycotting it not advantageous, both economically and in terms of time and difficulty. *Minosse* on the contrary, is built with the idea of obtaining *reliability* through the *transparency* of the electoral system, that is, being completely open-source, interrogable and verifiable. The user must be able to consult the source code, verify that it is actually the one in use and interrogate the updated system to verify the correct performance of the collection and counting of votes.

Minosse has, among its objectives, that of completely eliminating any barrier, both architectural and of actual distance, to allow anyone to vote with ease and comfort. This is said consciously of the quantity and consequent variety of people who access every day to vote and of the different identities and limits of each. For this reason *Minosse* was built from the very beginning with two fundamental principles:

- . …… …… : Minosse is completely responsive and adapts to any type of screen. You can vote from your home computer but also outside the home via tablet or smartphone.
- [illegible] : Minosse meets the limitations of anyone by being fully accessible and it is possible to consult and vote on the site by blind people.

Minosse was designed to adapt to any type of voting. For this reason, it generalizes two fundamental principles: the number of options to vote for and the weight of the votes.

It will be possible to vote for two candidates or options (similar to a referendum or an American election) or propose different candidates (similar to an Italian election). This is possible by assigning an incremental number to each option, which is used as an exponent on the *base* number from the previous chapter (voting is better described in 6.3.2.5. Voting).

With *Minosse* it is also possible to construct an unfair election where the votes of different voters *do not* have the same importance. Obviously this is the generalization of the scenario we are all used to where all votes have the same weight, that is 1. However, there are particular cases in which an unfair cumulative decision is useful and necessary, for example in the case where a particular business strategy should be chosen and employees with more important roles should have more power.

Voters arriving on the site will be immediately presented with a form with the list of candidates. Only one candidate can be chosen. After clicking the submit vote button, a vote receipt is created and this is the only thing that is sent to the *Minosse* servers.

Each vote is expressed through a number that represents it and through this the sum of the votes of each option is also calculated. The value of the vote is built through *base* and *exponent*.

The electoral system of *Minos* requires as a requirement to know the total sum of the weights of the voters, or in other words, the number of votes. The *base* is an *upper bound* of the latter. More formally:

$$b = c + \sum_{i=0}^{n} p_i$$

Where i indicates the voter, n the sum of the voters, p the weight and c a polynomial constant (usually $c = 1$). For a one-option election, to construct the value of the vote, only the base would be enough! In fact, taking each vote, multiplied by its respective weight, and adding them all together would be enough to determine the winner (obviously the one and only in this case). More formally:

$$\text{votes for the one and only option} = \sum_{i=0}^{n} p_i < b$$

This reasoning may seem trivial but it acquires more meaning with the introduction of the *exponent*.

In order to integrate more options into the electoral system, each of these is represented by an incremental number e. This number is then used as an exponent (hence the name) for the base b. More formally:

$$\text{vote}_i = b^e$$

The architecture of the value of the vote is the direct result of the question: *"what would happen if everyone voted for the same option?"*, in fact:

$$\text{number of votes for the same option j} = b^{e_j} \sum_{i=1}^{n} p_i =< b^{e_{j+1}} \iff \sum_{i=1}^{n} p_i < b$$

This causes the bases with their respective exponents to create "containers" that collect the votes and where even if everyone voted for the same option, the base guarantees an *upper bound* of the sum of the votes which, together with the exponents, prevents conflict with the counting of the votes of the other options.

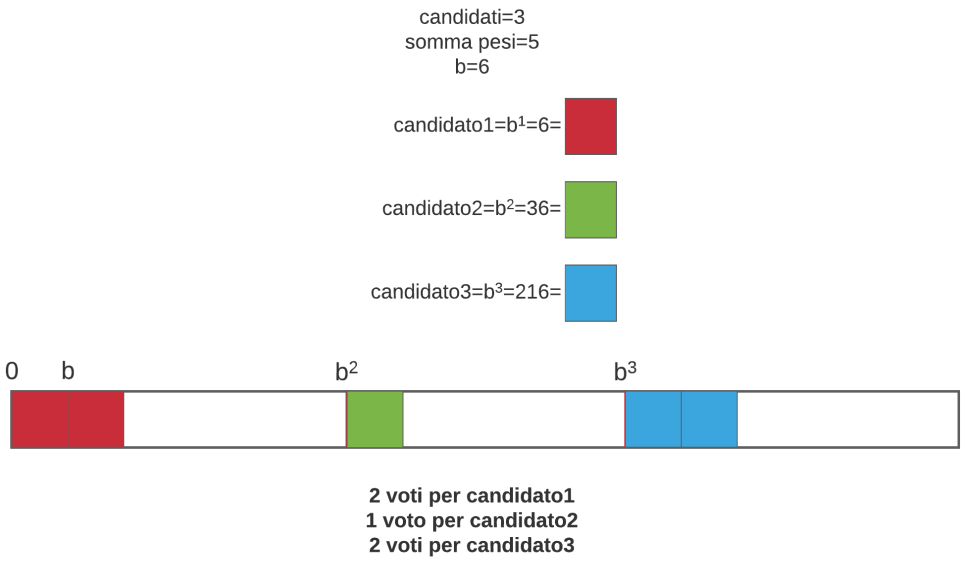

Fig 3: Graphical representation of the grade values in logarithmic scale.

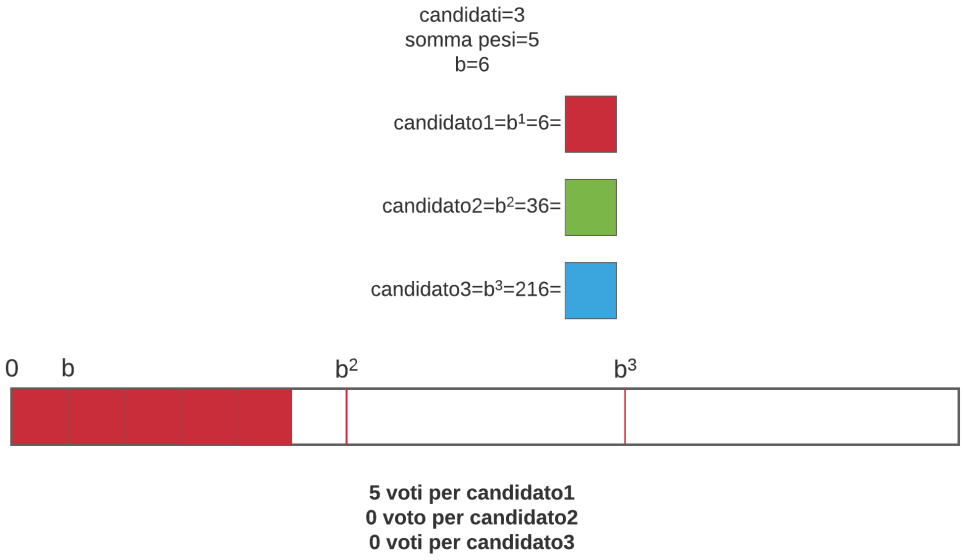

Fig 4: Even if everyone voted for candidate 1, the sum of the votes would still not exceed b^2.

The votes are then added together to form the current sum of votes:

$$\text{current sum of votes} = s = \sum_{j=1}^{t} b^{e_j} \sum_{i=1}^{n} p_i$$

Where t is the number of candidates (or options).

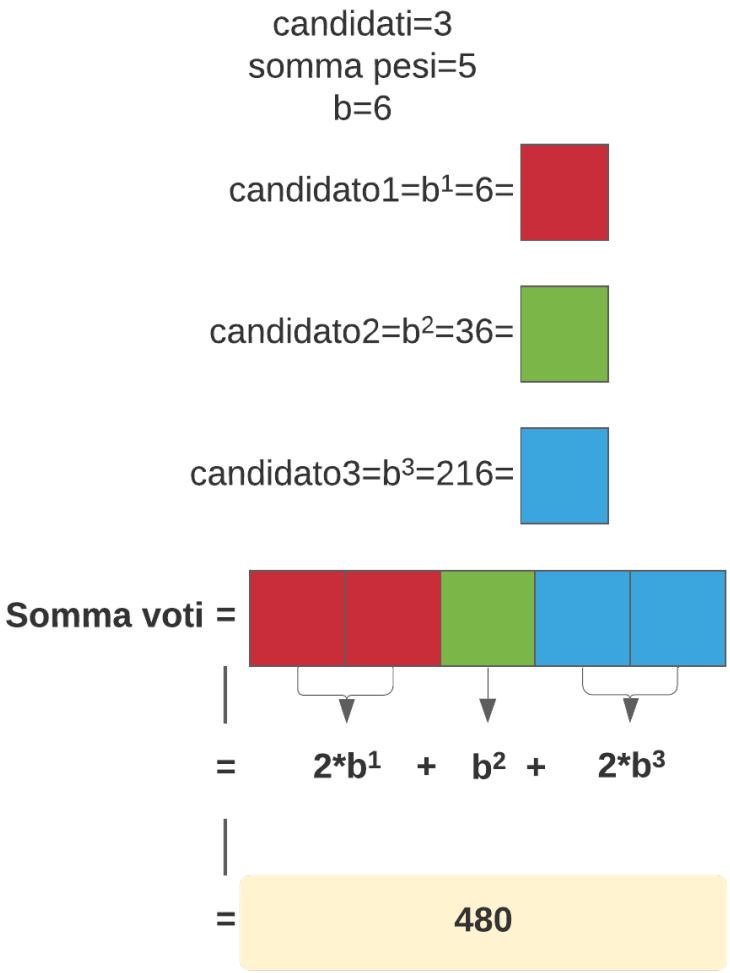

Fig 5: Sum of votes: from graphical representation to numerical value.

Since the value of the sum of the votes is a polynomial of degree t whose coefficients are the respective votes received by each candidate, it is possible to obtain the latter starting from the value of the sum through an algorithm with t polynomial divisions. More formally:

```
for(i=t; i>0; i++){
candidate_vote[i] = sum_votes / b[i]
sum_votes = sum_votes % b[i]
}
```

Where `b[i]` is the power `i` of the base.

Votes must be secret, for this reason it is necessary to use cryptographic schemes to obfuscate them. The votes once received must also be added together without obviously being deciphered. These two requirements are perfectly satisfied by a partially homomorphic cryptographic scheme such as Paillier's. As dictated in 6.3.2.1. Reliability and transparency it will be necessary for this phase to occur completely transparently for the user. To ensure greater security, votes must be shared already obfuscated so that no one who is spying on the conversation between the user and the platform can know the value of the vote.

The vote receipt is a set of data whose structure is as follows:

```
{
"vote": "31235556216316356156356...", //The secret vote
"weight": 1, //The weight with which you vote
"id": "axbr54bhqw" //the voter's unique id
}
```

The voting receipt is essential to ensure:

- Authorization: The authorization of a particular account to participate in the election. It is verified via the `id` field.
- Weight: The weight with which you vote is an important piece of data and closely related to the voter's account. You can check that the weight is correct through the `id` field.
- Vote: The integrity of the vote, while secret, must be maintained. For this reason and to count them, it is necessary to track all votes.

> The voting receipt as it is, is the only thing that is sent to the *Minosse* servers.

Minosse is a completely *real-time* system that collects and updates the vote count as they are submitted by voters. The status of the non-definitive and current result is always available through a graph, updated in real time, in the second section of the site. Immediately below the graph there is also a button to download the list of all secret votes related to the identifying information, collected up to that point. The possibility of downloading and consulting this list is of fundamental importance to guarantee the transparency (see: 6.3.2.1. Reliability and transparency) of a fair election. It will be possible to view the identification codes of the voters and their secret ballot and independently reconstruct the result of *Minosse* by anyone and guarantee its reliability.

- We thought of using a single programming language, namely `Javascript`, for both client and server so as not to have to distinguish the syntax between the two sources and to optimize the realization of the project.
- The server uses the `NodeJS` platform to execute `Javascript` code.
- The server and the client communicate asynchronously with `AJAX` requests. Once the homepage is reached, the server shares all the information needed to allow the user to vote (such as the base, candidates, public key, etc.)

Every library or module used to create *Minosse* is open-source and freely usable within the constraints and limits of the `MIT` license.

`paillier-BigInt`	https://github.com/juanelas/paillier-bigint	Client and Server	Library that implements the Paillier cryptographic scheme, developed in javascript and used by both the server and the client, it uses the new data type `BigInt`*. The client uses a minimized version of this library.
`jsonBingInt`	Implemented independently	Client and Server	implemented module for serialization of `JSON` whose format can also include `BigInt` type data. Used by both client and server.
`Bootstrap`	https://getbootstrap.com/	Client	Library for `Javascript` and `css` to make accessibility, responsiveness and UI/UX of *Minosse*. Used by the client.

JQuery	https://jquery.com/	Client	Javascript code optimization library
Style	Self-implemented	Client	Javascript module for scripting and graphics rendering.
Main	Self-implemented	Client	Module that implements the client's business logic in Javascript.

**Introduced in the ECMA-262 standard (https://tc39.es/ecma262/), BigInt allows to overcome the maximum value of $2^{53} - 1$ that Number have and create arbitrarily large integers, a requirement necessary for most cryptographic applications.*

[illegible]

In order for the client and server logics to be able to interpret the shared information, a common interface is needed for both. The JSON form was therefore used with appropriate data formatting, dependent on the context (request endpoint, dataType and contentType headers of HTTP), for any serialization. Furthermore, apart from the request needed to access the *Minosse* homepage, every request and response between client and server is carried out via AJAX asynchronously, so that the site is real-time and always up-to-date (the *Minosse* client, for example, queries the server autonomously, automatically and periodically, always requesting updated data which it then displays on the page). The server API is endpoint-based following the HTTP REST API standard.

[illegible]

BigInt is a data type defined in the ECMA-262 standard (https://tc39.es/ecma262/) and implemented in javascript. JSON, on the other hand, is a standard for serializing and sharing data regardless of the programming language or application used to read or write them. A standard for serializing BigInt data in JSON format has not yet been defined and probably will never be defined since they are not common to all languages. To overcome this problem, which is actually a well-known problem of all non-common data, the textual representation of the data value is serialized. However, this creates a new problem, namely *"How do you distinguish whether a string representing a number should be interpreted as a BigInt or as a normal number or as something else?"*. To solve it, the jsonBigInt module was implemented independently. The module interface exposes two functions, respectively to serialize an object or deserialize a received JSON. The module recognizes if one or more properties of the object are of type BigInt and represents them in JSON with the textual representation of the value, concatenated with the character 'n'. The deserialization function instead does the mirror work. It would have been possible to coin any other convention for representing BigInt as long as it was consistent for both functions.

[illegible]

Every election requires candidates that voters must be able to consult and visualize at the crucial moment. The candidates along with other necessary information are used by *Minosse* to create the election. This happens in the initial configuration process where a file in json format is filled with all the essential data that cannot be created automatically by *Minosse*. The file in question is called `election.json` and this is its current structure:

```
{
"weights_sum": 6, //The maximum sum of all weights
"candidates": ["John","Marvin","Mark"], //The names of the candidates
"candidates_colors": ["#e82c2c", "#e0da31", "#38d2e0"] //Candidates' co
}
```

From this information, other data, vital for the electoral system (the base, the exponents of the options, the public and private keys, etc.), are derived and constructed by *Minosse* in an autonomous and automatic way.

[illegible]

The entire *Minosse* project is protected by the open-source *MIT* license. The repository with the source codes of the implementation and documentation are available at: https://bitbucket.org/Giulio_Golinelli/tesi-repo/.

[illegible]

[illegible]

[illegible]

Nowadays, all applications that operate on data that should remain secret, and in particular cryptographic implementations, should compute in constant time. This means that regardless of the input to the operation, the time that this operation requires to be executed does not change. More generally, each operation that involves private information should be programmed in such a way that the only information that a malicious attacker can glean from its execution does not depend on the secret information. This type of attack that targets the implementation of algorithms, rather than the security of the algorithms themselves, is called "Side channel attacks" and in particular a sub-genre of this are timing side channel attacks which Javascript's `BigInt` suffers from. In fact, all the operations that can be performed between `BigInt` are not *constant time* and depend on the numbers themselves. Furthermore, the operations between `BigInt` are particularly inefficient (they are about 99% less efficient than the same operations with the `Number` types). For these reasons, `BigInt` are not suitable for cryptographic implementations.

[illegible]

Minosse was designed to be as transparent as possible, so the `Javascript` code is open-source and can be consulted by the user at any time. However, the same code can also be modified by the user, which creates two main problems.

[illegible]

The weight of the vote is sent together with the vote itself and the account `id` in the vote receipt via an asynchronous AJAX request. The code that makes this request is both consultable and freely modifiable by the user who, in a few words, becomes free to assign himself the weight he prefers (a single vote could weigh as much as the maximum sum of the weights and end the election!). The solution to this problem is simple and effective. Since the weight of the vote depends on the account that submits it, once the vote receipt is shared, the *Minosse* servers check the weight with the `id` and if these do not match, the vote is rejected.

[illegible]

The validity problem is a generalization of the one dictated in the previous chapter and is deeply related to the transparency of *Minosse*. The vote is submitted already encrypted in the voting receipt and the `Javascript` code that performs the obfuscation resides in the client to increase security. Since the code is always modifiable by the user, he can encrypt and insert in the voting receipt, not only his vote but any value. He could encrypt the value of his preference multiplied by a high weight and insert the result in the voting receipt with the correct weight and pass the check of *Minosse*. Or he could obfuscate any value, even random, modifying the outcome of the counting that *Minosse* performs. Currently, *Minosse* technology is not able to overcome this problem and for this reason it remains an application for purely demonstration purposes. However, there are solutions to this problem that consist in finding a mechanism to check the validity of the secret ballot submitted. A possible implementation could be that of Ku Peng and Feng Bao in their scientific paper[1].

[illegible]

Minosse has been tested on several elections with different candidates and all of them ended successfully. The election data along with the submitted votes are updated in real time on several devices at the same time. You can test *Minosse* and participate in the ongoing election at the address: `193.70.2.109:3000`.

[illegible]

[illegible]

Currently *Minosse* does not provide a login and account `id` check system. This improvement can be made by integrating a database that can be consulted by *Minosse* at the time of login and submitting the vote of each account.

[illegible]

Minosse only manages one election at a time but the system, with a simple addition, could manage several elections with different public keys at the same time. To implement this feature it will be necessary to assign to each election a unique identifier that will be used in the URL to precisely locate the endpoint of the election and where the voting receipts will also be administered. It will also be possible to perform an `id` check to verify if a particular

account is authorized to participate in a particular election.

[illegible]

Although *Minosse* does not have the technology to validate the value of a secret vote, it is possible to carefully modify the Paillier cryptographic scheme to make this possible. The method and operations to do so are described in detail in the paper by *Kun Peng and Feng Bao*[6].

[illegible]

A major improvement could be to abandon the convenient but insecure `BigInt` technology and implement the Paillier cryptographic scheme ourselves with secure constructs, functions and data structures for cryptographic operations.

- - - - .. - - - - - - - - - - - - - -

I was very happy to have studied and deepened homomorphic cryptography being very passionate about cryptography, it was a very interesting topic and I will deepen it further in my studies. I'm also happy to have developed *Minosse* and made some improvements, I'm sure it can become very interesting also in a professional context. Homomorphic encryption, although still an experimental technology, is very promising and could be used in many different fields.

> *We must invest in scientific research because life is a race against ourselves and we must strive every day to surpass ourselves and be better than we were yesterday.*

- ... ...- - - - - - - -

---

1. *Paillier P. (1999) "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes." In: Stern J. (eds) Advances in Cryptology — EUROCRYPT '99. EUROCRYPT 1999. Lecture Notes in Computer Science, vol 1592. Springer, Berlin, Heidelberg.*
2. *The Paillier Cryptosystem A Look Into The Cryptosystem And Its Potential Application By Michael O Keeffe The College of New Jersey Mathematics Department April 18, 2008*
3. *Koshy, Thomas. "Properties of Function." Discrete Mathematics with Applications.*
4. *No. Pettersen (2016) "Applications of Paillier's Cryptosystem."*
5. https://www.chosenplaintext.ca/articles/beginners-guide-constant-time-cryptography.html
6. *Kun Peng; Feng Bao (2010) "Efficient Proof of Validity of Votes in Homomorphic E-Voting."*