Università
Ca'Foscari
Venezia

Master's Degree
in Computer Science

Final Thesis

Kairos

A P2P End-To-End verifiable voting system framework

Supervisor
Prof. Riccardo Focardi

Graduand
Filippo Bisconcin
852144

Academic Year
2020 / 2021

Contents

Acknowledgments

I would like to thank my supervisor, professor Riccardo Focardi for the patience and support in this massive project which started out as a bunch of confused ideas an turned into a promising solution.

Many thanks to professor Antonino Salibra for helping me confirming the correctness of some formulas needed to build some important blocks of this project.

On a more personal level, I want to thank my coursemates, for sharing highs and lows of this rewarding academic and personal journey. It was worth it.

I wish to share my appreciation to my dear friends, for motivating and supporting me each in your own special way.

Finally, I deeply indebted to my family, your support throughout these years has been invaluable. I owe you everything.

Abstract

End-to-end verifiable voting systems are commonly based on the security offered by cryptographic operations. Popular voting systems make use of cryptosystems like ElGamal which allows to anonymize votes both through Mix Networks (MixNets) and homomorphic encryption. We propose Kairos, a new voting framework partially based on Helios, that groups features of popular voting systems. Remote voting systems like Helios are based on the assumption of a trusted server, in which decryption keys are only used in an intended, secure way. Kairos, instead, implements a peer-to-peer approach that relaxes this assumption and prevents votes to be decrypted and associated to voters even when a subset of the servers are corrupted. Kairos is fully configurable in terms of cryptosystems and protocols. Additionally, it implements a new form of ElGamal re-encryption-decryption MixNets that, to the best of our knowledge, have not appeared previously in the literature.

1 Introduction

Voting systems have to provide several guarantees to the voter and to the public including verifiability, ballot secrecy, incoercibility, usability and accessibility.

Electronic voting machines have been used in elections since the 1960s both as simple tallying devices for paper ballots or later on as ballotless direct-recording systems (DREs) taking votes expressed as digital inputs. In the first case the collection of ballots is conserved and kept secure in case of need of a manual recount while in the latter the absence of a paper proof of votes constitutes one of the main concerns.

The class of end-to-end verifiable voting systems (E2E) provides strong verifiability guarantees, often based on the security offered by cryptographic operations and by proofs of correctness which have replaced the typical chain of trust.

There exist several end-to-end verifiable voting systems, many of which operate by taking a ballot encrypted by the voter, by anonymizing the ballot set and then by tallying to obtain the outcome of the election. The literature describes two main ways of removing the association between the decrypted vote and the voter identity, Mix networks (MixNets) and homomorphic encryption, which are used in several voting systems like Prêt à Voter, Helios, Helios-C and Zeus.

Helios [4] by Adida is one of the most popular open-source voting systems and has been used in several medium-size elections. Although the implementation of Helios currently makes use of homomorphic encryption, in the original paper Adida proposed a voting system based on mix networks, where votes are shuffled and re-encrypted by mix nodes.

Helios is built upon the ElGamal asymmetric encryption scheme, an elastic encryption system that supports both homomorphic encryption and mix networks, based on the Diffie-Hellman key exchange while Prêt à Voter uses RSA .

One limitation of homomorphic encryption is the lack of control over the single decrypted ballot, as the tally is performed in an aggregated manner. This prevents constraints from being defined on the set of candidates (e.g., the chosen candidates must be of the opposite gender) which is a common practice nowadays.

While the possible dishonesty of the bulletin board of Helios has inspired improvements like Helios-C [6], the honesty of the server itself and of the technicians with access to it, have been implicitly assumed. When using re-encryption MixNets on a single server the anonymization

step can be rendered useless if the decryption key, derived by the shares of the trustees, is used to unveil the original un-anonymized ballot set.

Goals The main goal of this project is to develop a voting framework, partially based on Helios, which combines techniques from multiple existing voting systems in order to define a more robust, secure and elastic alternative. By making use of modern frameworks, popular libraries and clean code, we aim at increasing the number of possible reviews and improvements and thus making the system more maintainable.

By providing a Peer-to-peer mechanism also, the trust usually placed in the main server can be reduced and by designing a robust distributed system in which multiple servers have to cooperate, we can achieve fault tolerance and stronger security.

The literature on voting systems and Peer to Peer protocols is very vast and evolving at a fast pace, with the latter benefiting from the amount of solutions based on the blockchain. Analyzing the state of the art technology for each aspect would be unfeasible, so by focusing on the scale of a framework instead of the single module, we can develop a future-proof platform where experts can contribute in a more granular way.

Paper structure In section §2 we will cover the generic voting process, electronic voting and the techniques adopted to make it secure and trustworthy: section §2.1.1 will cover Public Key Cryptography with a focus on ElGamal's cryptosystem, on which Helios, its derivatives and the current project are based. Section 2.3 will cover mix networks, one of the techniques used to anonymize votes in a verifiable manner. In section 2.4 we will describe existing electronic voting systems and their properties.

Section §3 will present Kairos, a new framework partially based on Helios with several improvements and new features.

In section §4 we will cover some experimental results and measures on Kairos.

2 Background and related work

Many different voting systems have been adopted in elections, starting from simple urns in Ancient Greece up to modern solutions based on the blockchain architecture. Electronic devices have been used in traditional voting booths since the 1960s, either as a support devices in paper based voting systems or as the main recording devices.

There exist many different voting approaches, characterized by the voting process and by the guarantees they offer.

In this chapter we will cover important properties required by a voting system in order to be considered secure. We will then move to studying which techniques are adopted to provide anonymity and then in 2.4 we will list some existing voting systems worth mentioning.

2.1 Voting systems

A voting system is a protocol and a set of rules that define multiple tasks, which can be executed in different manners depending on the nature of the system itself. The voting systems described in the next sections follow the same sequence of operations as the list below. Even if it's harder to notice, big country-level elections follow the same procedure, just at a bigger scale and involving many organizations.

The first step of an election is to *define the casting modalities and the rules* used to declare a ballot valid. In traditional country-level elections with paper ballots this set of rules is provided to the people responsible for the tally. It is then required to *freeze* the candidate list and the voter list, preventing any subsequent change.

At this point the list of voters has to be generated, either starting from an existing dataset (e.g., citizens of a Country) or by allowing people to register themself as voters through a registrar entity.

Once both the candidate list and the voter list are frozen, election officials can proceed to the printing of ballots with the name of each candidate, making sure that each voter has at least one ballot to fill. This task is not needed for elections that do not make use of paper ballots.

At voting time voters need to authenticate, proving their voting right to the system by proving they belong to the voter list generated by the registrar.

Once authenticated, the voter is provided a ballot he can fill by choosing from the candidate list printed on it. According to the rules mentioned above, the election can define constraints on the set of candidates that can be chosen. The simplest example of voting constraint is a maximum number of choices of candidates. Big country-level elections can define more complex constraints, such as requiring that if two candidates are picked they must be of the opposite gender or belong to the same party. Depending on the rules mentioned above, the (possibly empty) set of choices of candidates will be considered either valid or invalid.

When using paper ballots the choice is either recorded by a pen mark (e.g., a cross on top of the name of the candidate) or by a perforation of the paper itself by means of perforation devices. In the traditional voting experience, this latter operation is usually performed inside of a voting booth which provides privacy to the voter, preventing any coercion. During this phase also, the voter is usually forbidden to use recording devices such as smartphones that would provide a proof of how the ballot was filled, allowing for the selling of votes or coercion.

The ballot is then cast, usually by inserting it in a closed box that prevents it from being

traced back to a voter which remains closed until the voting phase ends. In digital voting systems this step is usually performed by posting encrypted votes on a *public bulletin board* and by running the anonymization step later on.

At tallying time the anonymized ballots are extracted and the votes are filtered and counted according to the rules mentioned above. Depending on said rules and the election modalities, the outcome of an election can be as straightforward as a single winner or a more complex result.

Properties of voting systems Some voting systems offer more guarantees than others. Depending on its characteristics, a voting system can provide several properties and depending on the guarantees it provides, it can be classified.

Even tough there isn't a unique definition of each property with many agencies providing their own, we can refer to popular guidelines such as The Voluntary Voting System Guidelines [21] paper and the NIST [1] paper which list desirable properties for a voting system.

Auditability A system is *voter-auditable* when it provides a voter enough information to verify the correctness of the process. From the global perspective, a system is *universally-auditable* when an observer is given enough information about the correctness of the process. [1, 4]

Verifiability Verifiability is a property of a voting system which is similar to the previous one but stronger: while a system has to provide some generic information which can help verifying a system, a verifiable system provides all the necessary information needed to prove its correctness. Similarly to Auditability, Verifiability can be seen both from the point of view of a voter and of an observer. [1]

Another important property of a voting system, partially included in *Universal Verifiability,* is *Eligibility Verifiability*, achieved if an observer can verify that ballots come from legitimate voters [9].

[6] defines a voting system as *strongly* verifiable if it is both universally and individually verifiable and if the voter registrar, responsible for voter registration, and the bulletin board are two separate entities not simultaneously malicious.

Incoercibility and Ballot secrecy / Privacy While privacy is usually a condition which can be declined by a person, in the context of a voting booth it has to be enforced, regardless of the voter's will.

Both Privacy and Incoercibility describe the same quality of the communication channel from the two points of view of the voter and of an observer: *ballot secrecy* requires a voting system not to leak the association between the voter and the vote while *incoercibility* requires a voting system not to allow a voter to prove its vote to some observer. [1]

In the traditional voting procedure, ballot secrecy and incoercibility are guaranteed by forbidding the use of any recording device, by preventing an observer to join the voter in the booth, by casting votes in a closed boxed where ballots get shuffled and by discarding ballots presenting any recognizable sign.

When it comes to voting system with unattended booths, as in the case of voting via personal Internet-connected devices, incoercibility is hard to enforce. Several existing voting systems deal

with this issue by either letting a voter cast more than one vote, and by only considering the last one, or by creating fake credentials for voters to use under coercion without the coercer noticing. Both techniques allow a voter to prove a vote which will not be tallied. During the current pandemic, US voters were encouraged to cast their 2020 presidential ballot by mail, an example of a procedure that doesn't offers incoercibility and potentially neither ballot secrecy.

Correctness A voting system is correct if the set of votes in the bulletin board can be validated, preventing a malicious bulletin board from altering the ballot set without any observer noticing. [1, 6]

Usability and Accessibility The Usability of a system is a measure of how intuitive, learnable and efficient a user interface is. There exist techniques to assign an objective usability representation to a particular voting system, which compute scores based on the percentage of users who successfully cast votes, percentage of voters who made mistake and several similar measures. A voting system can be considered voter-usable if these measures combined are higher than a set threshold, indicating the user experience is good enough.

A voting system implementation is *accessible* if it provides an interface which is usable by individuals with disabilities, by providing adequate input interfaces and modalities. A blind voter, for example, should be provided with either a tactile input device (e.g., braille reader) or with an audio interface similar to those implemented in smartphones capable of describing what is displayed on a screen. [1]

The Voluntary Voting System Guidelines [21] paper defines parameters chosen for an optimal usability as accessibility of the voting system such as screen resolution, font sizes and behaviors such as flashing of the screen.

This paper will focus neither on usability nor on accessibility, as these properties involve minor changes to the main infrastructure, with the latter being the main subject of this thesis.

Paper based and Direct-recording electronic voting systems In *Paper based voting systems* the voter marks its vote on a paper ballot, usually with either a pen or a punching device. Electronic devices can be used to perform the tallying process while a paper version of each ballot is still available in case of need of a recount.

In *Direct-recording electronic voting systems (DREs)* the voter utilizes an electronic system, which records the user input through a touch screen and thus doesn't produce any paper ballot. DREs may provide a personal receipt of the vote to prove the correct recording of the voter's intention.

The two families of voting systems just mentioned belong to the E-voting family, where the privacy of the voter is enforced by placing the voting machine in a regular voting booth. The transmission of the data recorded on a E-voting voting machine can either happen on a private network or on a public one (e.g., Internet) which requires many precautions to be taken, as the risk of an attack increases.

An unsupervised approach exists called I-voting, where ballots are cast through personal devices with Internet access capabilities.

While the idea of voting on the Internet has been applied to low stake elections successfully, its application on big Country-level elections resulted in many criticisms, with many of these adopters opting out.

2.1.1 End to End verifiable voting systems

Traditional voting systems with paper ballots rely on a chain of trust put on election officials in order to provide the desired properties mentioned above. The same applies to electronic voting machines and, since even a trusted software relies on the guarantees provided by the underlying hardware, many electronic voting systems use proprietary software running on specialized hardware.

End to End (E2E) verifiable voting systems partially replace this chain of trust with cryptographic operations, which allow an observer to confirm the correctness of the entire process.

These voting systems require neither trusted software nor trusted hardware.

In the next sections we will focus on two End to End verifiable voting systems, one making use of paper ballots (Prêt à Voter, 2.4.1) and one that does not make use of paper at all (Helios, 2.4.2).

Cryptosystems The literature proposes several voting systems mainly based on asymmetric encryption schemes such as RSA, ElGamal and Paillier. Despite Paillier offering nice cryptographic properties, the literature favorites voting systems based on ElGamal, as the former is a patented technology. In section 2.2 we will focus on ElGamal, an elastic cryptosystem heavily adopted in several voting systems.

Anonymization and tallying When the voter casts its ballot, the voting system has to store it until the tally procedure takes place, making sure that the connection between the unencrypted vote and the identity of the voter can not be recovered. The literature offers three main techniques for said task: mix networks (MixNets), homomorphic encryption and blind signature. Table 1 summarizes the techniques employed by several popular voting systems, some of which will be covered more in-depth in the next sections.

Mix networks are formed by a series of mix nodes, each of which takes the list of votes, shuffles them and performs either [re-]encryption or decryption, preventing each vote of the final list from being traced back to the original set. Each mix has to be followed by a correctness proof. 2.3 will cover MixNets more in-depth.

The second technique is *homomorphic encryption*, where cryptographic properties are used to aggregate votes without the need of decrypting all of them first, thus this technique does not require an anonymization step. While homomorphic encryption is much easier to implement than MixNets, the nature of this solution prevents any operation on the single ballot. An example of homomorphic encryption with ElGamal cryptosystem will be given in 2.2.1 while 2.3 will cover mix networks.

The third method is the *blind signature* technique, proposed in [20] by Chaum. Once the voter has filled its ballot, the latter is provided to a trustee in encrypted form, preventing its content from being read. The trustee proceeds to sign the encrypted form, which is then returned to the voter, who submits the unencrypted ballot with the signature. The original paper presents this technique through a simple example: the encrypted ballot can be thought of as a ballot inserted in a opaque carbon-copy envelope capable of transferring the trustee's signature to the ballot. Once the envelope and its content are signed and sent back to the voter, the inner ballot can be transferred to a new envelope and cast. At tallying time, the signed ballots are trusted while

Voting System	Cryptosystem	Technique
Helios (Paper)	ElGamal	Encryption MixNets
Helios	ElGamal	Homomorphism
⟶Helios-C	ElGamal	Homomorphism
⟶Belenios	ElGamal	Homomorphism
⟶Zeus	ElGamal	Encryption MixNets
Pret a Voter (2005)	(Shift cipher)	Decryption MixNets
⟶Pret a Voter (2006)	(Shift cipher)	Encryption MixNets
Fujioka, Okamoto and Ohta[19]	-	Blind signature
⟶Sensus	RSA	Blind signature
⟶E-VOX [18]	RSA	Blind signature
⟶E-VOX-MA	RSA	Blind signature
⟶REVS	RSA	Blind signature

Table 1: Summary of several voting systems. Voting systems derivation is indicated with nesting (⟶).

those lacking the trustee's signature are discarded. In the literature this third option is very often ignored with only Homomorphic Encryption and MixNets being presented.

2.2 ElGamal

By applying a slightly change to the message sequence of the Diffie-Hellman key exchange algorithm, we obtain a encryption scheme called ElGamal, proposed by Taher Elgamal in 1985.

The security of ElGamal is based on the decisional Diffie Hellman assumption (DDH), a stronger assumption than the Discrete Logarithm one that requires a careful choice of the underlying group. The triplet (g, q, p) is made public with safe prime $p = 2q + 1$ where q is a Sophie Germain prime and where g is a generator of the q-order subgroup of $\mathbb{Z}_p^*$. This parameter choice respects the Diffie Hellman assumption.

The encryption of a plaintext m produces a ciphertext composed by two numbers α, β:

$$(\alpha, \beta) = (g^r \mod p, \quad my^r \mod p) \tag{1}$$

A ciphertext $c = (\alpha, \beta)$ can then be decrypted by taking

$$m = \frac{\beta}{\alpha^x} \mod p \tag{2}$$

Since $m \in \mathbb{Z}_q$ may not belong to the q-order subgroup of $\mathbb{Z}_p^*$, [4] shows a mapping of m from $\mathbb{Z}_q$ to m' in the latter subgroup and vice versa:

$$m' = \begin{cases} m+1 & (m+1)^q \equiv_p 1 \\ -(m+1) & \text{otherwise} \end{cases} \mod p \tag{3}$$

$$m = \begin{cases} m' & m' \leq q \\ -(m+1) & \text{otherwise} \end{cases} \mod p \tag{4}$$

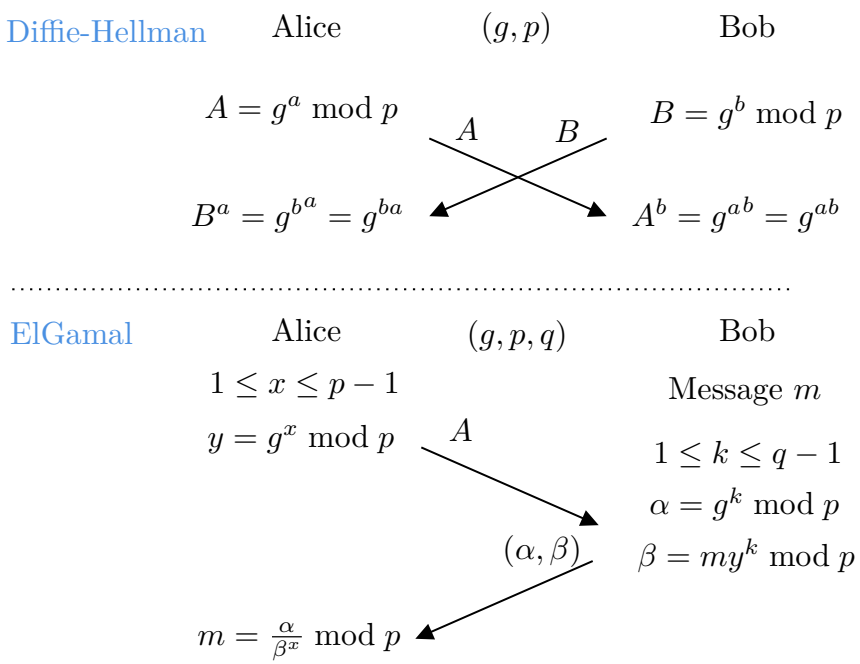

Figure 1: Comparison between the sequence of operations of the Diffie-Hellman scheme and ElGamal's ones.

Proof of decryption Once a ciphertext (α, β) has been decrypted with the secret key x into the plaintext m, the decrypter can prove to a verifier that the decryption is correct without revealing the private key x by making use of the Chaum-Pedersen protocol [4], a Zero-knowledge proof method. As shown in figure 2, the decrypter can prove the decryption by proving to the verifier the equality

$$\log_g y = \log_\alpha \frac{\beta}{m}$$

The interactive verification process requires the verifier to provide an unpredictable value $c \in \mathbb{Z}_q$ but it can be transformed into a non-interactive process by means of the Fiat-Shamir heuristic, thus generating c from a cryptographic secure hash function which can be considered unpredictable because of the pre-image resistance property of the hash function. We will use an identical usage of the Fiat-Shamir heuristic for proving the output of a MixNet in section 2.3.1.

In case the mapping of equation (3), equation (4) is used in encryption and decryption, the last step involving α^t will require the equation (3) mapping to m.

2.2.1 Homomorphic property

A variation of this cryptosystem called *exponential* ElGamal allows to perform homomorphic operations between ciphertexts, obtaining the sum of their plaintexts through their product without the need of decrypting them first. This variation consists in a different formula for β in which $\beta = g^m y^r \mod p$ instead of $\beta = my^r \mod p$. Given two ciphertexts c_1, c_2 of two plaintexts p_1, p_2 obtained *with exponential* ElGamal, by multiplying the equivalent terms we

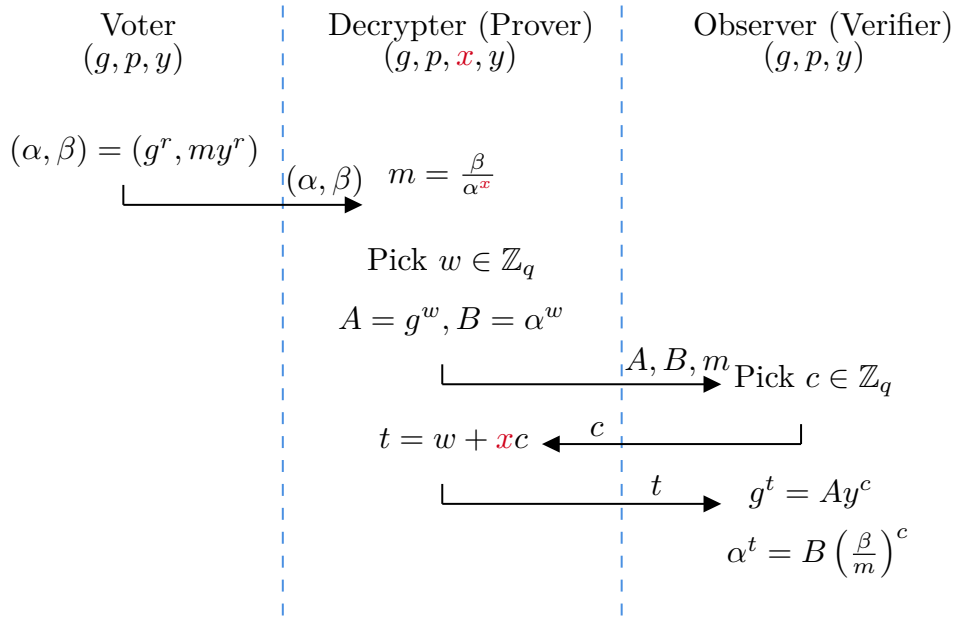

Figure 2: Proof of decryption. The decrypter proves the correct decryption by proving $\log_g y = \log_\alpha \frac{\beta}{m}$ without revealing x instead.

obtain a new ciphertext which is the encryption of $p_1 + p_2$:

$$(\alpha', \beta') = (\alpha_1\alpha_2, \beta_1\beta_2)$$

More in general, the sum of n plaintexts p_i is equivalent to the product of the equivalent terms of their ciphertexts $c_i = (\alpha_i, \beta_i)$:

$$\sum_{i=1}^{n} p_i = \prod_{i=1}^{n} c_i = \left(\prod_{i=1}^{n} \alpha_i, \prod_{i=1}^{n} \beta_i\right) \tag{5}$$

Clearly, at decryption time, the traditional ElGamal decryption formula would return plaintext g^m thus extracting m requires solving the discrete logarithm. Unfortunately this techniques requires to brute force g^m enumerating all the valid values of m, operation that can be sped up by the generation of a lookup table [9]. This technique also requires to skip the mapping into the q-order subgroup of $\mathbb{Z}_p^*$ previously mentioned.

The plaintext has to encode a single number for the sum operation to make sense. Because of this, the tally process with homomorphic encryption requires a set of n preferences of candidates to be converted into n ciphertexts, whereas traditional ElGamal could encode a binary representation of a set of answers in a single ciphertext. Figure 3 shows the two different approaches for a question that allows to specify up to 5 preferences without considering the order. These n ciphertexts can be either 0 or 1 for binary preferences. This technique allows to assign different decisional power to different voters by multiplying their ciphertexts by their voting weights, without decrypting them first. During the tally phase, all nv ciphertexts submitted by v voters

$$\text{ElGamal} \qquad c_1 = E_{P_k}(10010) \qquad \text{(1 ciphertext)}$$

$$\text{Exp ElGamal} \quad \begin{aligned} c_1 &= E_{P_k}(1) \\ c_2 &= E_{P_k}(0) \\ c_3 &= E_{P_k}(0) \\ c_4 &= E_{P_k}(1) \\ c_5 &= E_{P_k}(0) \end{aligned} \qquad \text{(5 ciphertexts)}$$

Figure 3: Difference between ElGamal ciphertexts and Exponential ElGamal ciphertexts for a question that allows to specify up to 5 preferences.

have to be grouped by the corresponding answer and then aggregated with 5 to obtain an integer number that represents how many times the answer was chosen.

Because of the aggregation procedure, the ballots can't be individually validated before the tally phase, and thus the voter has to prove to the bulletin board the validity of its ciphertexts. Not performing this step would allow a malicious voter to submit the encryption of a very big number giving the vote more decisional power. This proof is performed by means of a Zero-knowledge technique.

2.2.2 Re-encryption property

The Elgamal cryptosystem also offers a re-encryption property that allows to alter a ciphertext while maintaining the same decryption.

Re-encrypting a ciphertext (α, β) with randomness r amounts to computing

$$(\alpha', \beta') = (\alpha g^r, \beta y^r) \mod p$$

The encryption process of equation (1) with randomness r amounts to a re-encryption of a ciphertext $(1, m)$ with the same value of r.

2.2.3 $\ell - \ell$ Elgamal Threshold encryption

In order to prevent a single trustee to compromise either the secrecy of ballots or the result of the election, it is possible to pick multiple trustees who must share their public key before the voting phase starts and the secret key at decryption time. In a $\ell - \ell$ threshold encryption scheme, all ℓ parties have to share their private key in order to recreate the secret key needed to decrypt the ciphertext.

Before the voting phase starts, the public keys of the trustees are combined into a unique election public key y:

$$y = \sum_{t=1}^{\ell} y_t \mod p$$

At tallying time, the trustees are supposed to share their private key shares x_t in order to compute the election private key x:

$$x = \sum_{t=1}^{\ell} x_t \mod p \tag{6}$$

The decryption phase of a ciphertext (α, β) is thus

$$\begin{aligned} m =& \beta \left[\alpha^{x}\right]^{-1} \\ =& \beta \left[\alpha^{\sum_{t=1}^{\ell} x_t}\right]^{-1} \\ =& \beta \left[\prod_{t=1}^{\ell} \alpha^{x_t}\right]^{-1} \end{aligned} \tag{7}$$

This encryption scheme does not require any interactive task as a trustee holding a key pair is only required to share it without having to deal with other trustees, making this choice suitable for human trustees. This will not be the case for $t-\ell$ threshold encryption, where trustees have to send each other shares, validate them and reconstruct private keys.

2.2.4 True $t-\ell$ Elgamal threshold encryption

Basic threshold encryption requires all ℓ trustees to cooperate in order to decrypt a ciphertext as a single corrupted or missing key would prevent the decryption. This issue can be addressed by implementing a $t-\ell$ threshold encryption in which only $t+1$ parties out of ℓ are needed to compute the secret key required to decrypt the ciphertext.

Helios-C's paper [6] proposes the use of Pedersen's distributed key generation algorithm (DKG) adapted to ElGamal. Pedersen's DKG is based on the Shamir's secret sharing (SSS) scheme [16, 6, 17] which exploits the need of k points to define a polynomial of degree $k-1$. It's in fact well known how k points can define an infinite number of polynomials of degree k through them, while only one is defined by $k+1$ points. Each peer uses Shamir's secret sharing scheme to distribute its secret value s_i among all other peers and, at reconstruction time, each qualified peer will use all the valid shares he received to recover a part of the secret key. These secret parts will be combined as in equation (6).

Each peer P_i generates a random polynomial of degree t

$$f_i(x) = \overbrace{a_{i0}}^{s_i} + a_{i1}x + \ldots + a_{t1}x^t$$

Each peer P_i computes a secret value to share with each peer P_j by plugging the index j into the polynomial

$$s_{ij} = f_i(j)$$

Each peer P_j can make sure the received value is valid by checking $g^{s_{ij}}$ against the product of the coefficients of the polynomial publicly broadcasted by P_i

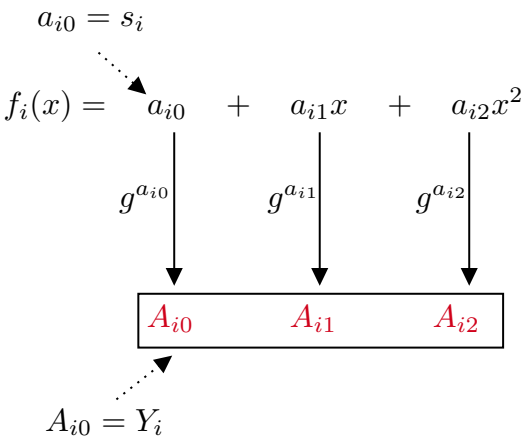

Figure 4: Generation of the values to broadcast of peer i in a 3-4 threshold scheme.

$$g^{s_{ij}} = \prod_{k=0}^{t} (A_{ik})^{[j^k]}$$

If t peers broadcast complaints against peer P_i, it will be excluded. The set of qualified peers Q will be composed by the actors P_i whose shares were proven correct.

At this point all peers can compute the value of the public key from the public values Y_i of the qualified peers $P_i \in Q$:

$$y = \prod_{i \in Q} Y_i$$

Each peer P_j will have a share x_j of the secret key x:

$$x_j = \sum_{i \in Q} s_{ij} \mod p \tag{8}$$

Even if not computed, the private key x would be the product of the secrets $s_i = f_i(0)$:

$$x = \sum_{i \in Q} s_i \mod p$$

Unlike the previous threshold scheme, we now have to select t trustees from the set Q of qualified peers to form the new set I. Once I has been fixed, each peer $j \in I$ sets its secret key as x_j from (8) to the power of $\lambda_{j,\Lambda}$;

$$\widehat{x_j} = x_j^{[\lambda_{j,I}]} \tag{9}$$

where $\lambda_{j,\Lambda}$ is the Lagrange coefficient used for interpolation adapted to modular arithmetic:

$$\lambda_{j,\Lambda} = \prod_{l \in \Lambda,\, l \neq j} \frac{l}{l - j} \mod p$$

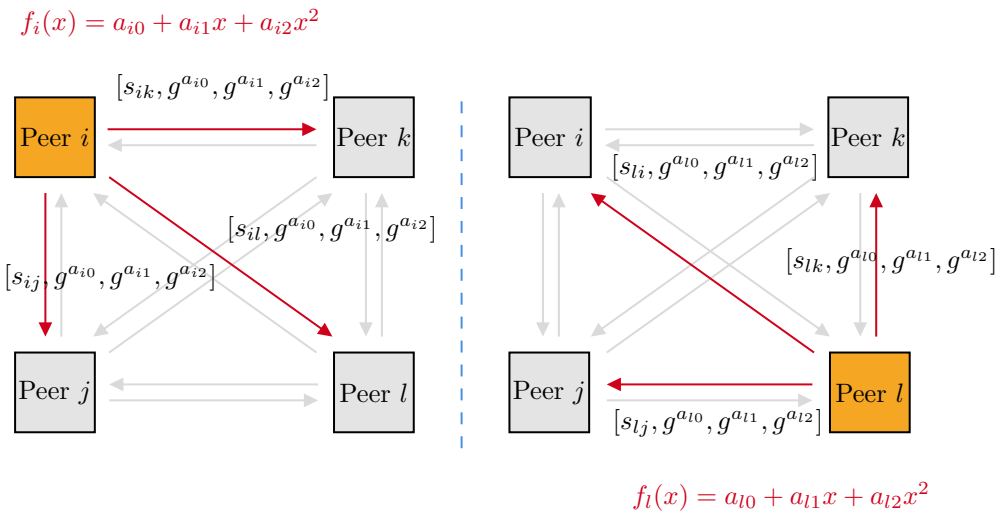

Figure 5: Example of a 3-4 threshold encryption scheme. Peer i broadcasts the values $g^{a_{ik}}, k = 1 \dots t$ to all peers $P_j, j = 1 \dots l$ and sends each share s_{ij} privately.

The final decryption of $c = (\alpha, \beta)$ is (7) with the secret keys obtained from 9 :

$$\begin{aligned} m =& \beta \left[\alpha^{\sum_j \widehat{x_j}}\right]^{-1} \\ =& \beta \prod_{j \in I} \left[\alpha^{\widehat{x_j}}\right]^{-1} \\ =& \beta \prod_{j \in I} \left[\alpha^{x_j^{[\lambda_{j,I}]}}\right]^{-1} \mod p \end{aligned}$$

For any polynomial f_i of degree at most t, we can recover $f_i(0)$ starting from $t+1$ values:

$$\sum_{j \in \Lambda} \overbrace{f_i(j)}^{s_{ij}} \lambda_{j,\Lambda} = \overbrace{f_i(0)}^{s_i}$$

2.3 Mix Networks

As mentioned before, a Mix Network (or MixNet) is a technique used to anonymize a set of ballots where several mix nodes work in a serial manner by taking the output of the previous mix node as input, applying a cryptographic operation to each ballot and by shuffling the resulting ballot set.

The first definition of Mix Network comes from [2], where this technique was proposed in the context of mail systems as a mean to achieve untraceable communication. This first version only offered individual verifiability.

There exist two main ways a MixNet can operate: a node can either add an encryption level (Encryption MixNets) or remove one (Decryption MixNets). Figure 6 shows the two steps that

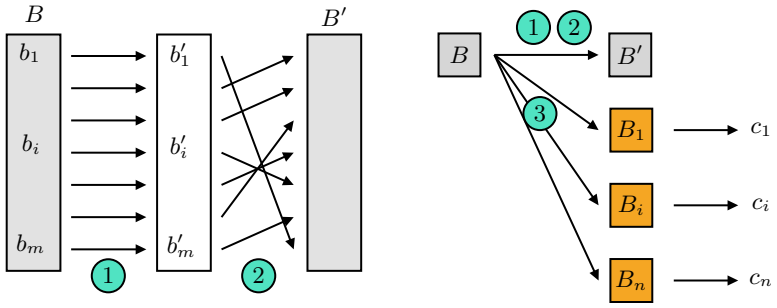

Figure 6: Steps 1 and 2: all the ballots b_i in the ballot set B get either re-encrypted or decrypted to form b'_i and then get shuffled (or sorted) to form the new set B'. When proving the mix, these two steps are repeated n times in step 3 to form n shadow mixes B_i, each with a bit c_i.

transform the original ballot set B into the primary mix B'.

The literature offers many papers on encryption MixNets with both RSA and ElGamal, with RSA usually adopted in conjugation with decryption MixNets and ElGamal usually adopted in encryption MixNets.

An encryption MixNet that makes use of the ElGamal re-encryption property shown in section 2.2.2 is often referred to as a re-encryption MixNet. Since the re-encryption property maintains the same ciphertext length, the output of an ElGamal re-encryption MixNet has constant size. When encrypting the whole ciphertext without re-encryption properties, the size of the ciphertext increases as the the size of the message grows at every encryption step. In the context of voting systems, the input of the mix network is a set of ciphertexts and thus, for cryptosystems that support re-encryption (e.g., ElGamal), the most natural choice for a MixNet is one with uses the same cryptosystem as the one used for the ballot encryption.

To the best of my knowledge all paper presenting decryption MixNets make use of RSA, lacking examples of ElGamal decryption MixNets.

When using a $\ell-\ell$ threshold encryption scheme each mix node is essential to the success of the whole mix procedure. An exception is constituted by cryptosystems that support re-encryption (e.g., ElGamal), in which a missing re-encryption simply amount to a missing shuffling, where the ciphertext can still be decrypted. This limitation constitutes the main reason why Re-encryption MixNets (especially with ElGamal) are the most popular option, and why in general, Re-encryption MixNets are chosen over decryption MixNets.

Since decryption MixNets perform both decryption and shuffling, at the end of the process the ballots are visible in plain text. Encryption MixNets re-encrypt and shuffle the ballots sets, so they require an additional decryption phase.

The MixNet approach has been employed in several contexts such as Tor and cryptocurrency mixing. Tor, acronym of "The Onion Router" is a software that allows a user to reach a server trough a path over many proxy nodes, each of which removes one encryption layer, until the plaintext message reaches the destination. The last node, also called *exit* node is the only server in possess of the plaintext request, which is then executed and the response forwarded back to the sender.

Assumption of trusted server [Re-]encryption MixNets remove the association between vote and voter by shuffling and [re-]encrypting ballot sets.
In the literature, the privacy guarantee is built upon an underlying implicit assumption of a honest server. If the server receiving the encrypted votes has knowledge of the combined private key at any time, it can decrypt the original ballot set, voiding the anonymization process.

Table 2 summarizes several MixNet options and their characteristics. The second to last column of the table indicates whether a MixNet design is tolerant to a skipped node during the mix process, a constraint that can be relaxed by means of a $t - \ell$-threshold encryption scheme.

Cryptosystem	Technique	Non-Increasing size	S_k not shared	Tolerant to skipped mix node	Tolerant to arbitrary mix node path order	Notes
RSA	Enc	×	×	×	×	
RSA	Dec	✓	✓	×	×	⋆
ElGamal	Enc	×	×	×	×	
ElGamal	Re-Enc	✓	×	✓	✓	⋆
ElGamal	Dec	✓	✓	×	✓	Not proposed

Table 2: Comparison of MixNets. Once again, the "Not proposed" is with respect to to the best of my knowledge. The combinations identified by ⋆ are the most popular options in the literature.

2.3.1 Proving the mix

Each node of a mix network has to hide the parameters used in order to provide anonymity to the ballots. This secrecy goal contrasts with the idea of proving correctness and auditability. It is thus needed a *Zero knowledge proof*, a procedure to verify the correctness of the mix without revealing the parameters used to generate it.

In [10] Sako and Kilian proposed an extension to the original MixNet design capable of achieving universal verifiability by providing a cryptographic proof of each mix. After each mix, the mix node produced n additional shadow mixes and it is challenged with n bits: if $c_i = 0$ the node has to prove equality of shadow mix i to the original ballot set B by revealing the parameters used to generate the shadow mix. If $c_i = 1$ the node has to prove equality of shadow mix i to the primary mix B' by providing the parameters necessary to convert B_i into the primary mix B'. Figure 7 provides a visual interpretation of the process just described.

A honest mix node knows the parameters needed to convert the original ballot set B into the primary mix B', thus it can answer correctly when challenged. A dishonest node will only be able to answer to one of the answers correctly and will get away with probability 2^{-n} where n is the number of times it's questioned. Clearly, as n grows, the probability of fooling the system decreases. In [4] Adida proposes $n = 80$.

A non-interactive process: Fiat-Shamir Heuristic Since in the process described above, each mix node (claimer) has to prove the correctness of its work to each challenger, the amount

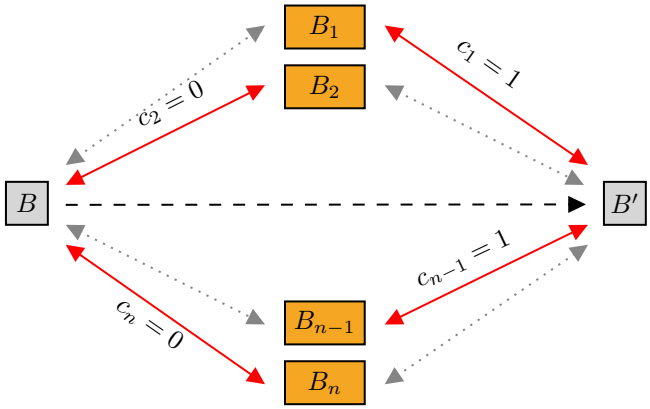

Figure 7: Sometimes "L/R" is used as notation instead of "0/1" to indicate whether the mix node has to prove equality to the original ballot set B (left) or to the primary mix B' (right).

of computation grows with the number of challengers.

As shown in 2.2, this proof can be transformed into a non-interactive process by making use of techniques such as the Fiat-Shamir Heuristic in which the infeasibility of cheating is based on the pre-image resistance property of an hash function. A one-way cryptographic function is applied to the n shadow copies and n bits are extracted from the digest to be used as if they were chosen by a verifier in a interactive process. A cheating mix node would have to generate n shadow copies which can be proven to be equivalent to either B or B' and whose hash digest has a binary representation matching the chosen equivalences, which is an unfeasible task. This unfeasible problem allows the mix node to produce a single proof of its work based on said bits and will not be required to produce a new proof for each challenger.

2.3.2 Proving an ElGamal Re-encryption MixNet

A re-encryption ElGamal MixNet with m ciphertexts fed as input can be verified

Each shadow mix $B_{i=1\ldots n}$ is performed by re-encrypting the ciphertexts with m random values $R_i = \left[r_1^i, \ldots, r_m^i\right]$ as shown in section 2.2.2 and then by shuffling with a permutation π_i.

Let B' be the primary mix with m re-encryptions with random values in R' and a permutation π'.

The parameters needed to prove equality of a shadow mix B_i to the primary mix B' can be recovered by reversing the shuffling π_i of the shadow mix, by combining the sets of random values R_i and R' and by applying the primary shuffling π':

$$\pi' \left[R' - \pi_i^{-1}\left(R_i\right)\right]$$

The process is shown in figure 8.

2.4 Existing End to End verifiable Voting Systems

Several end to end verifiable voting systems have been proposed, many of whom use the techniques presented in the previous sections.

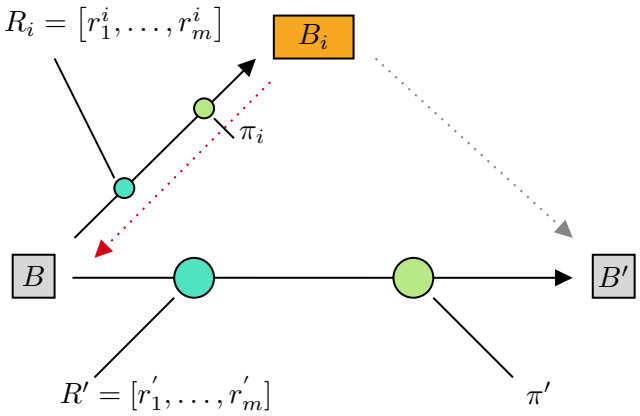

Figure 8: Proof of a Re-encryption ElGamal MixNet. The parameters needed to prove equality of a shadow mix B_i to the primary mix B' (gray arrow) can be obtained by reversing the shadow mix generation (red arrow) and combining them with the parameters of the primary mix.

In the next sections we will cover two voting system approaches, one based on paper ballots (Prêt à Voter) and one based on ballotless elections (Helios).

2.4.1 Prêt à Voter

Prêt à Voter is a *paper-based* end-to-end verifiable voting system proposed in 2005 by Chaum, Ryan and Schneider [12].

Given a list of candidates in alphabetical order, each ballot is generated with said list shifted by θ positions with the secret amount θ computed as the sum $\mod v$ of values provided by k trustees. Each trustee provides two values k_{2i}, k_{2i+1}:

$$\theta = \sum_{i=0}^{2k-1} d_i \mod v$$

This set is computed in advance of the voting process, making sure that there are enough ballots for all voters to vote. The value p is encrypted into a code called "Onion" which can be decrypted if all trustees cooperate. The printed ballot is composed of two columns, one containing the candidate list and the other containing slots for the voter choice and the onion that allows to recover the shift value θ. The first part containing the candidate list is removed once the voter has expressed its vote on the second one.

Since the order of the candidates can only be recovered with the value θ, the ballot alone does not provide any information about the picked candidate until the onion is decrypted.

Once the voting phase ends, a decryption MixNet allows to recover the value of θ, which is used to find the name associated with the voter's mark.

The 2005 paper proposed ballot going through a decryption MixNet with each teller removing one layer of encryption and a later 2006 version [13] proposed the use of re-encryption MixNets.

The technique employed by this voting system can be considered a direct derivation of a shift cipher with secret key θ.

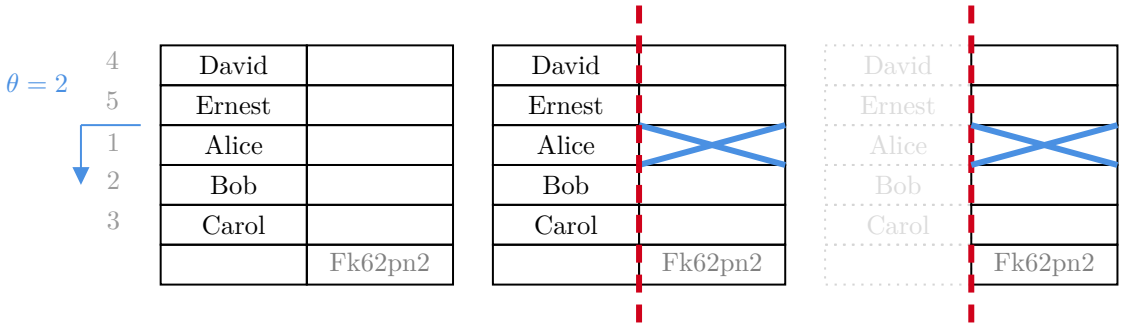

Figure 9: Filling of a PRÊT À VOTER ballot. The onion "Fk62pn2" is the encryption of the shift value $\theta = 2$ used in the candidate list column. Once the voter has expressed its preference on the second column, the two parts are separated.

2.4.2 Helios

HELIOS is a popular open-source voting system by Ben Adida, built upon the Benaloh's simple Verifiable Voting Protocol [3] based on the Sako-Kilian MixNet[10]. This voting system does not provide a paper ballot, making it an *electronic* end-to-end verifiable voting system. A modified version of this voting system is currently used by our University for academic elections. This system is designed as a centralized software, which acts as registrar, as bulletin board and as mix node, making it weaker than some alternatives presented in the next sections.

The admin can edit the questions, the list of trustees and the details of the election up to the moment he freezes it, generating an hash value of all the parameters. This hash value is the published and the election parameters can be checked to be unchanged at any time by recomputing the digest.

The election can either be limited to a finite set of voters, uploaded as a tabular comma-separated values (csv) file, or can open to any visitor of the website.

A number of trustee can be added by the creator of the election and each one is responsible for uploading its public key before the voting phase starts and for uploading its secret key when the mix phase is complete. Both the public key and the secret key are computed according to the $\ell - \ell$-threshold scheme presented in 2.2.3 which requires all trustees to cooperate honestly. The system allows the creator of the election to create an automatic trustee impersonated by the server itself, if trusted, which will perform the same operations as a human trustee.

Before the election is frozen, the admin can register a trustee starting from its email address, which is used to send the password needed to authenticate. The user receiving the email will be able to authenticate through a dedicate login page and to generate or reuse an existing ElGamal keypair.

Once the election day arrives, all voters can access the booth section of the website, where they can choose from a set of candidates to form their ballot. The election defines a set of questions, each with a number of possible answers and two numerical values indicating the minimum and maximum number of choices. This voting scheme allows neither sorting nor voting constraints. The ballot page can be previewed before the voting phase starts, allowing to try and verify all operations previous to the casting phase.

Once the voter has filled it ballot, the latter sealed by a ElGamal encryption, a hash value h is generate and the voter is presented two choices: the ballot can either be audited, revealing all the encryption parameters used or can be cast by sending it to the server which will display it in a public bulletin board. By choosing the audit option the voter can verify the correct encryption of its ballot, allowing the voter to perform the sealing operation from scratch with different random keys. The hash value h is published on the bulletin board, allowing a voter to verify the presence of the sent ballot.

When the date specified as the closing time for the election comes, the server stops accepting ballots and starts the anonymization process that makes use of re-encryption ElGamal MixNet. Each mix is followed by a public proof of correctness. Once the set of ballots is anonymized, the platform waits for all trustees to upload their private key, needed for the decryption of each ballot. At this point all ballots are visible in their plaintext version and the tally process takes place, producing a report of the result of the election.

As pointed out in the paper, there is no way for an observer to detect neither replacement nor insertion of a ballot by a malicious bulletin board, an issue called *ballot stuffing*. The *auditability* property of this system can thus only be guaranteed by many voters performing the audit of their ballot. This issue was theoretically addressed in Helios-C [6] and implemented in Belenios [9, 8], where ballots are signed by the voter with a secret key unknown to the bulletin board.

In the original paper Adida pointed out how this voting system, like the majority of remote voting systems, does not offer *incoercibility* as an observer could join the voter. Instead of hiding this limitation, the latter was highlighted by providing the voter a procedure to share the parameters used to encrypt its ballot.

This version of Helios only supports election where the voter can pick k out of n without keeping track of their order without describing the ballot encoding in details.

Helios is written in Python and makes use of the Flask module to provide a web interface allowing the voters to encrypt and cast their vote using a browser. The front end of the website constitutes the voting device of a voter and is composed of a series of JavaScript files.

Since a pure JavaScript implementation of the required cryptographic operations showed poor performances, Adida proposed the use of a Java Virtual Machine responsible for the operations on big integers and a technology called LiveConnect to access said operations from the JavaScript context.

Helios with homomorphic encryption Even though the original Helios paper proposed the use of MixNets, the support was eventually dropped and nowadays it implements ElGamal homomorphic cryptography. This change was motivated by the need of applying different weights to ballots according to the title of the voter [5].

In his homomorphic encryption version, Helios encodes a ballot composed of k out of n candidates as k binary values. If the question allows the voter to pick up to n preferences, the final ballot will be composed of $k \leq n$ values indicating positive choice (1) and $n - k$ values indicating negative choice (0).

Homomorphic encryption allows to aggregate these binary values while they are in their encrypted form, as shown in 2.2.1, and to only decrypt the resulting value to get a tally.

In [5] Adida highlights the issue of having a central authority decrypting the tally with the entire reconstructed secret key.

2.4.3 Zeus

ZEUS [7] is a fork of HELIOS developed by Tsoukalas et al. that aim at improving the latter in numerous ways.

The paper of this voting system highlights the limitations of homomorphic encryption, technique that only allows to perform aggregated tallying.

The main goal behind ZEUS is to support *Single transferable votes* (STVs), a type of election in which candidates are *ranked* by the voter, with the first choice being the main one the following names being backup options in case the first candidate is disqualified. In order to support this electoral process, ZEUS reverted the change from MixNets to homomorphic encryption made by HELIOS developers. The homomorphic encryption implementation is still part of the code in order to benefit from updates made to the fork's parent HELIOS, and it's simply not used.

When editing an election, the admin can change the *election type* among several supported options. Depending on the election type the voter has to either sort candidates (e.g., STV's), pick from a list of candidates from multiple parties or pick an option for multiple questions.

ZEUS also uses a different ballot encoding approach than HELIOS in which all $n!$ possible permutations of n candidates are enumerated and the identifier of the corresponding permutation is used as the ballot to cast. Once decrypted, the ballot is decoded by performing the enumeration once again and by recovering the original choice of the voter.

2.4.4 Helios-C

HELIOS-C by Cortier et al. [6] is a derivation of HELIOS which proposes a design providing full correctness. This variant addresses the possibility of a malicious bulletin board to insert arbitrary votes by signing each vote with the voter's private key.

The paper proposes two separate entities responsible for the voter registration and for the election management, assumed *not simultaneously malicious*. Because of this assumption this voting scheme can be considered *strongly* verifiable. Once the voter registers to vote, the registration entity generates and provides him a keypair (Pk, Sk). Once the ballot b is filled and encrypted, the voter signs it with its private key Sk and sends both the signature $S = sign\,(Sk, b)$ and the verification key Pk to the bulletin board. A malicious board would still be able to alter a cast vote, replacing it with an arbitrary value, without being able to sign it lacking knowledge of the secret key Sk provided by the registrar.

This paper proposes using the ElGamal $t - \ell$-threshold encryption scheme shown in section 2.2.4.

2.4.5 Belenios

An implementation of the HELIOS-C protocol is BELENIOS [9, 8], written in the Ocaml programming language.

This voting system makes use of Homomorphic encryption, while highlighting the limitations of this technique and suggesting future MixNet integration.

Similarly to HELIOS-C, BELENIOS addresses the case in which a malicious bulletin board would be able to add arbitrary votes by making voters sign their ballots. The keypairs for the ballot signature are generated by the registrar and sent to the voters by mail.

The design of this voting systems relies on four actors with four distinct roles: the bulletin board, the registrar, the voter and the used device, and the decryption trustee.

This voting system implements the ElGamal $t - \ell$-threshold encryption scheme proposed in the HELIOS-C design.

The voting setup is composed of three phases: election setup, voting phase and tally phase. In the third last step, the tally phase, all p ciphertexts are combined using the exponential Elgamal encryption procedure described in equation (5):

$$\text{res}_e = \prod_{i=1}^{p} c_i$$

This combined value is then decrypted by sending it to the decryption trustees i, each of whom performs a partial decryption with its share of the secret key dk_i and returns the result $\text{res}_e^{dk_i}$ with proof of correctness pok.

Unlike HELIOS, this voting system addresses the issue highlighted in 2.3 by preventing any entity of the system from being in possession of the combined private key of the election.

The *verifiability* of the voting system is guaranteed if the voting device is honest and if the server and the registrar are not simultaneously malicious, regardless of the honesty of the decryption trustees. The *privacy* of the voting system is achieved if the voting device, the server, the registrar are honest and at most t decryption trustees are corrupted.

These properties have been verified both theoretically and practically by means of EasyCrypt, a toolset for automatic theorem proving.

Two variants of Belenios were proposed in the same paper, with the first called BELENIOSRF offering receipt-freeness and verifiability and the second called BELENIOSVS providing privacy and verifiability, even with a dishonest voting device.

Assumptions	Helios	Helios-C	Belenios	BeleniosRF	BeleniosVS
Individual Verifiability	✓	✓	✓	✓	✓
Universal Verifiability	✓	✓	✓	✓	✓
Eligibility Verifiability	×	Registrar and bulletin board not simultaneously malicious			
Ballot secrecy (privacy)	Honest bulletin board + Honest voting device	$< t$ malicious trustees + Honest voting device			$< t$ malicious trustees
Incoercibility	×	×	×	×	×

Table 3: Summary of the properties of mentioned Electronic End to End verifiable voting systems.

3 Kairos

In this chapter we present KAIROS, a new framework partially based on HELIOS and its derivations.

The goal of this framework is to provide a generalized the voting scheme by providing a modular structure which can be easily extended for new cryptosystems, anonymization methods and protocols.

By providing Peer to Peer capabilities also, the role of a human trustee can be partially or entirely replaced by trusted servers, which can deal with interactive processes such as the $t - \ell$ threshold encryption scheme more easily than humans.

As highlighted before in 2.3, the anonymization process that makes uses of re-encryption MixNets can be rendered useless once the combined private key is reconstructed: if the server has the initial bulletin board with the association between voter and encrypted vote, once this key is reconstructed and the votes are decryptable, the *ballot secrecy* property is voided. The assumption we want to drop is that a honest server, an a honest system administrator, will not use the private key. This can be achieved by having more server interact, none of which has knowledge of the full decryption key, which is the same approach suggested by HELIOS-C and implemented in BELENIOS.

While the initial goal was to simply extend HELIOS with features, the software was almost entirely rewritten from scratch due an old monolithic structure which makes use of dated technologies and coding styles that make maintenance complex. In the new software the front end side and the back end side are nicely separated and they communicate through REST requests in JSON format. Because of this separation, any developer is free to rewrite each component in a different language independently. Also, by not mixing languages together in the same file, the system benefits from more expert developers in each language.

The application is composed of two main DOCKER containers: an APACHE2 web server and a MYSQL8 database server.

The back end is a PHP 7.4 application, based on the popular LARAVEL framework which provides common features not worth reinventing.

LARAVEL allows to define jobs to be executed in queues. While a single queue is currently used to avoid focusing on concurrency issues, this choice can be revisited in future by simply adding a second worker responsible for a second queue. LARAVEL also allows to perform time based tasks by making CRONTAB execute a PHP script every minute. For development reasons the CRONTAB behaviour is emulated by an endless loop handled by DOCKER. Even though there are many alternatives to Laravel which are capable of handling more requests per second, the latter was chosen because of the code quality, the clean project structure and the big community supporting it.

The front end is implemented in VUE, a popular JavaScript framework, recently chosen by Wikipedia developers.

With the exception of the distributed bulletin board feature, which will be presented later, KAIROS offers the same properties as BELENIOS, requiring the bulletin board and the registrar entity not to be simultaneously malicious.

Chapter structure In the next sections we will cover the implementation of the KAIROS structure.

In section 3.3 a new modular structure will be presented. Said structure generalizes the voting scheme allowing to pick different question types, ballot encoding formats, anonymization methods and cryptosystems.

In section 3.6.2 we will cover a new MixNet obtained by combining some properties presented in the previous chapter.

In section 3.8 we will cover the Peer to Peer capabilities of this framework.

3.1 Single Page Application

In HELIOS only few parts of the site are loaded in a asynchronous manner through JQuery templates, while the most part uses Server-Side rendering. In KAIROS the whole application is handled by the client side, following a popular trend called *Single Page Application* (SPA) where only the first static file (INDEX.HTML) is loaded from the server, while the whole page structure is handled by a JavaScript framework, in our case: Vue. This approach does not require the server to render every HTML page dynamically as the pages are compiled and shipped in a big JavaScript file, usually called APP.JS.

By having a separation between front end and back end, adding features such as Internalization becomes much easier as the back end code is not affected.

Thrust model of a web based application In the context of a web based voting system, the JavaScript script acts as the voting device of a voter.

Since the whole structure is handled by the JavaScript file, which is static and whose signature can be published, it can be easily audited.

```
<script
    src="https://kairos/js/app.min.js"
    integrity="sha384-aJ21OjlMXNL5UyIl/
       XNwTMqvzeRMZH2w8c5cRVpzpU8Y5bApTppSuUkhZXN0VxHd"
    crossorigin="anonymous"></script>
```

Figure 10: Example of a JavaScript asset included specifying its integrity sign.

Even though a JavaScript file can be provided by a trusted third-party service like a Content Delivery System (CDN) and signed as shown in figure 10, the HTML page that includes is still generated by a server, which is free to include additional code, potentially malicious.

Even though we can't guarantee an honest content of the page, we can still audit the page by periodically checking the provided content.

When designing a voting system that runs in the browser we can consider the client side applications acting as voting devices only as trustfully as the servers providing them.

3.2 Authentication

HELIOS authenticates users with the standard HTTP session which makes use of cookies. In order to adapt to the Single Page Application structure, this stateful approach has been replaced by a stateless one, which makes use of tokens. Once the user logs in with either credentials or an

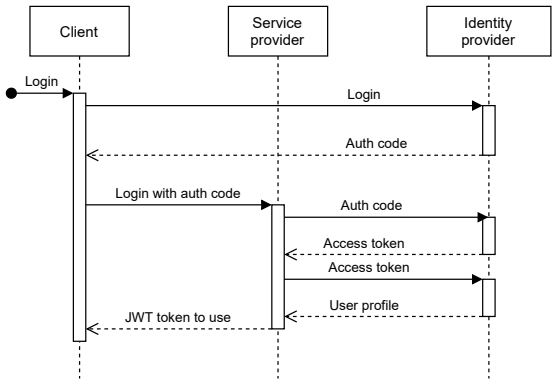

Figure 11: Login with an Identity provider.

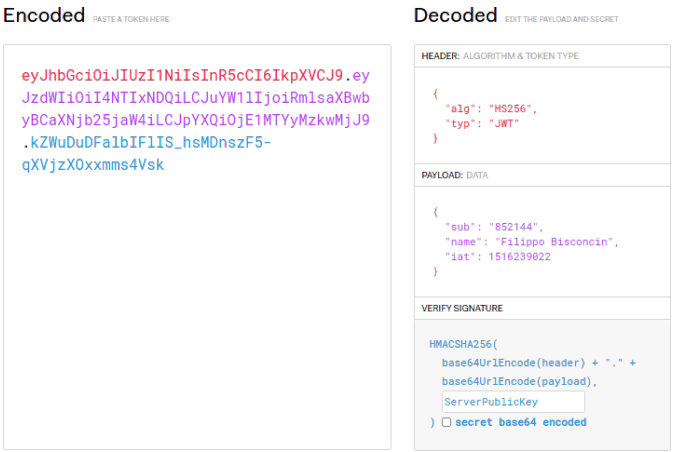

Figure 12: Breakdown of a JWT token provided by JWT.IO.

identity provider, a JSON Web Token (JWT) containing a field that identify a user is generated and returned to the client, which will specify it in each request via the Authentication HTTP Header. One benefit of JWT tokens is the fact that there is no need for them to be stored on the server as they are signed and can be sent by the client in each request. Any entity with knowledge of the public key of the issuer can thus verify the validity of the token.

A JWT token is a structure composed of three sections: the first section describes the algorithm and the token itself, the second contains the actual payload and the third contains details about the signature. The string representation of a token with a sample payload and its decoded structure can be seen in figure 12.

Moving from cookies to headers also, simplifies the process of implementing new clients not based on browsers.

Similarly to HELIOS, KAIROS allows an user to authenticate through identity providers. An example of identity provider is Google, which allows a service provider (KAIROS) to retrieve first name, last name and email of a user through Google People, one of its services.

Figure 11 shows the sequence of requests for the login with the Google identity provider,

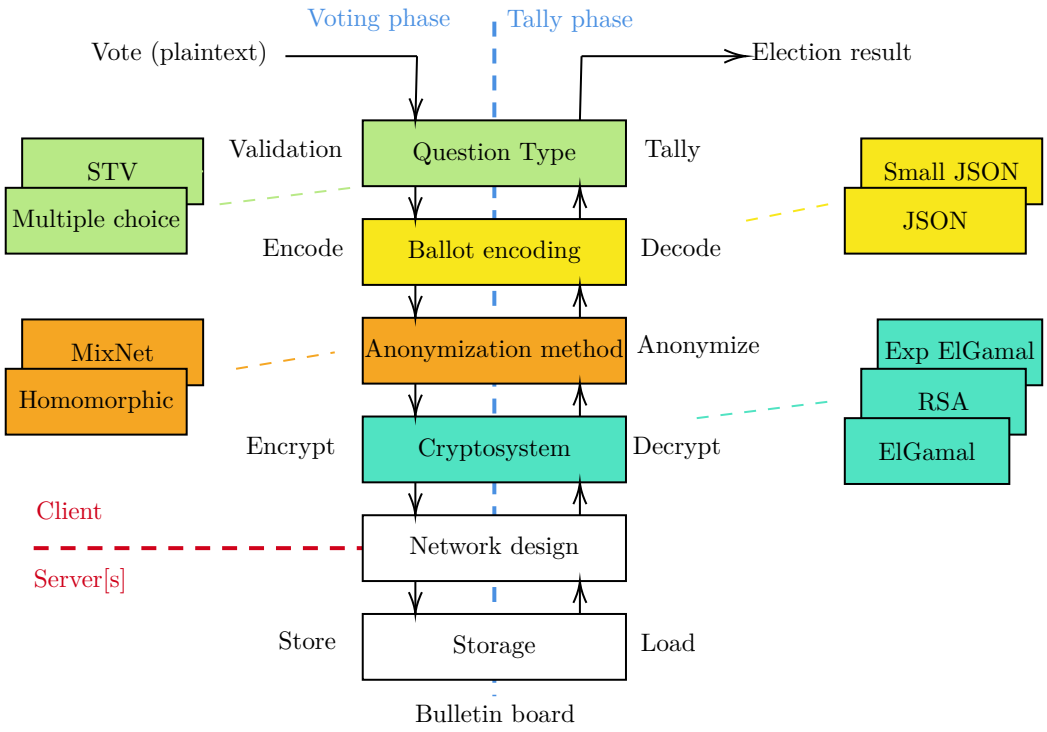

Figure 13: Modular structure.

whose steps are described below.

Once the user chooses to login in using an identity provider, the JavaScript code opens the authorization page in a new window, waiting for the user to confirm the app can access its data. Once the user gives its permission, a request is made to the identity provider's server which returns a redirect response to a redirect URL specified during the window creation. This process is handled through a Vue third-party module which is responsible for opening the window and closing it once the redirect response has been received. During the redirect response, an *auth code* is returned, which is then extracted by the client and sent to the KAIROS server, which exchanges it for an access token and uses the latter for retrieving the user profile. Once the profile has been received, the server can use the email of the user and its unique identifier on the identity provider to either register or log the user in. In both cases, a JWT token is generated and returned to the client.

Other identity providers can be used by simply creating a dedicate class responsible for retrieving the user information.

3.3 Modular structure

In order to generalize the structure of the voting system, KAIROS handles interchangeable modules with common interfaces though a massive polymorphic class hierarchy. Figure 13 represents the chosen structure with encrypted votes being fed from the top. On the left half of the picture the arrows describe the path taken by a ciphertext from the booth to storage. The dashed red

line shows the separation between operations being performed on the client side (voting device) and on the server side (bulletin board). The right part describes how votes are loaded from storage and fed to each layer up to the tally step.

Proceeding from the higher level to the lower we have several layers:

1. **Question type**: Each module implements a different type of question and defined the tally procedure to use. This layer will be covered in section 3.4.
2. **Ballot encoding**: Each module describes how a set of choices is converted in one or more encrypted values and its decryption. This level is responsible of defining how the election result is extracted and declared. This layer will be covered in section 3.5.
3. **Anonymization method**: Each module describes how a set of votes is altered in order to prevent an observer from associate a decrypted vote to a voter. This layer will be covered in section 3.6.
4. **Cryptosystem**: Each module implements the technique used to encrypt the ballot. This layer will be covered in section 3.7.
5. **Network structure**: Modules of this layer describes how a set of messages is exchanged between the voting device and one or more peer servers. The two implemented modules will be described in section 3.8, in the context of Peer to Peer.
6. **Storage engine**: Each module defines how the ballots are stored during the process. Kairos makes use of two modules, one which handles the main MySQL Database and a second one that stores ballot in a SQLite file, a more portable option that simplifies the process of sharing a set of tables. This second engine will be briefly described in section 3.4 when describing the modules responsible for different question types.

Since these levels have dependencies, there are constraints that have to be met when changing these parameters when the election is created or edited. As an example, a ballot encoding format may only be used in conjunction with a particular cryptosystem. This is the case for Homomorphic encryption, which only supports tallying procedures that operate on aggregated results and does not require an anonymization step as the votes are not singularly decrypted.

These modules can be chosen in the election editor when the election is created.

3.4 Question types

Several voting systems described in the second chapter were created from scratch mainly because of a type of question not compatible with existing systems. As previously mentioned Zeus was created to fulfill the need of using *Single Transferable Votes* (STVs) which was incompatible with the homomorphic properties used in Helios.

One of the goal of this project was to provide an elastic structure capable of handling multiple kinds of questions and tally procedures. This pursuit of a generic approach requires us to define a common data structure which is capable of handling every possible ballot.

The following paragraph will describe in details the approach used in Kairos, where decrypted ballots are stored as records in a database table.

Tally on the database

This section describes a procedure which requires ballots to be decrypted into a sequence of IDs representing answers thus is clearly incompatible with homomorphic ciphertexts of Exponential ElGamal. The proposed mechanism allows for any kind of question structure that can be converted into a tabular structure. The way the sequence of numbers is extracted from the plaintext, once decrypted, is described in the next section.

In order to generalize the election process to the point of supporting many question types, we can think of the tally procedure as a query on database table containing the decrypted votes.

The first step required to implement a new question type is to define the procedure that takes a set of answers and returns a tabular representation suitable for a database. The most general approach for a question that accepts up to n answers is to define n columns and to fill only the required one. This structure allows to store answers in any order and any combination. This method amounts to a flattening of the data structure representing the set of answers to each question.

Let T be the final table with all decrypted ballots, if the election defines q questions with question i allowing up to n_i answers, T will have $n_1 + \ldots + n_t$ columns.

By storing decrypted ballots in a database table, the set of possible tallying modalities that can be obtained without changes the PHP structure widens drastically, with the only limitations given by the DBMS itself. Instead of storing T in the main database, KAIROS stores each tally in a separate SQLITE file to simplify shipping and sharing.

Since the tally is performed by the DBMS itself, the procedure benefits from the massive internal optimizations on queries and will likely take less time than an equivalent operation performed by the web server. The robustness of popular DBMS's also gives a sturdy guarantee of the correct handling of the table's content and the query execution correctness.

A small price to pay for this gain in generalization is the complexity of the tally, which can now only make use of the set of operations provided by the DBMS and can make simple operations complex. As an example, counting how many times each ID of an answer appears in the table in any order and without duplicates would require a couple of straightforward PHP loops while the SQL query equivalent is syntactically much more complex as the maximum number of answers grows. The SQL query of the last example can be seen at the bottom of figure 15. By replacing the SQLITE database with a different DBMS, one could decide to translate a custom tally function into a stored function which makes use of loops and variables to obtain a result closer to the PHP equivalent implementation.

The traditional tally procedure in which a voter can choose one candidate and the winning candidate is the one who received more votes can be easily translated into a query on table T with one column containing the ID of the voted candidate. Figure 14 shows an example of the table generated from a single question accepting up to three choices. Empty (null) cells represent unspecified choices. All three columns have foreign keys on a table used to store the answers to the first question.

The sixth and the ninth rows represent two empty ballots, a case of ballot filling that results problematic when adopting traditional ballot encoding.

By using foreign keys we can also exploit the integrity constraints features offered by the DBMS that prevent a ballot containing an invalid candidate ID from being tallied. For each question we can create a dedicate table filled with all the possible answers and use foreign keys.

	id	q_1_a_1	q_1_a_2	q_1_a_3
	Filter	Filter	Filter	Filter
1	1	1	NULL	NULL
2	2	2	NULL	NULL
3	3	2	NULL	NULL
4	4	2	NULL	NULL
5	5	3	NULL	NULL
6	6	NULL	NULL	NULL
7	7	2	NULL	NULL
8	8	1	NULL	NULL
9	9	NULL	NULL	NULL
10	10	2	3	NULL
11	11	2	NULL	NULL
12	12	3	NULL	NULL
13	13	1	3	NULL
14	14	2	NULL	NULL
15	15	1	2	3

Figure 14: Table containing the decrypted ballots. In this example the question q_1 accepts up to three answers a_1, a_2, a_3.

Single Transferable Votes are supported by KAIROS: the voter specifies a ranking of n candidates which are stored in n columns of T in the same order. Selecting the first non disqualified candidate of each ballot amounts to performing a query joining the candidate table with a join condition that filters disqualified candidates out and selecting the first non-null column of each record.

Different policies can be defined on the insertion of a ballot with invalid ID of the answer into T such as discarding only the invalid ID or the whole ballot. The first option does not prevent vote selling as a voter could sell its vote to candidate 1 by specifying a second invalid recognizable ID in any other answer while maintaining a valid ballot, thus the second policy is preferable and has been implemented in KAIROS.

The resulting query has obviously to be frozen during the election freeze operation and the tallying server has to be disqualified if it can't provide a valid proof of correctness. In KAIROS we store the generated query as a SQL view directly into the database.

All these operations can be verified by simply publishing the database and all the queries, which can be easily executed by an observer. KAIROS provides a link which makes the browser download the entire SQLITE database.

When the voter opens the election booth page, each question is rendered by dynamically loading the appropriate question component. While the traditional multiple choice question type only requires a series of checkboxes with a constraint on the maximum number of positive choices, the component responsible for STVs displays a list of candidates which can be sorted with a simple drag and drop operation. Figure 16 shows the two components used for two questions of an election.

Constraints as query clauses Previously mentioned constraints on the ballot choices can be easily formulated as filters applied to the tally query, which can be translated into the SQL lexicon with a set of WHERE clauses.

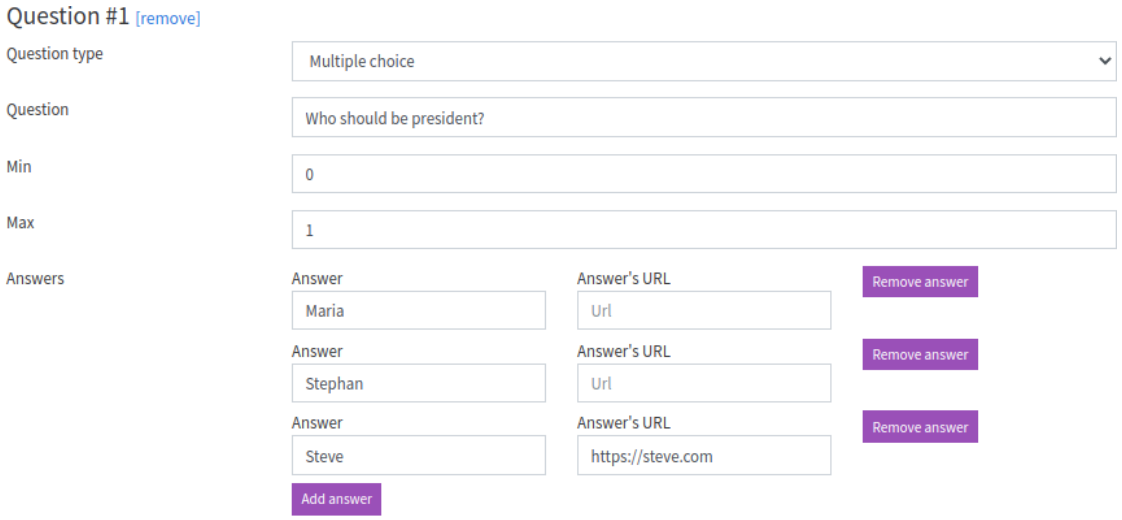

Figure 15: The question editor shows the tally query generated by the class that implements the chosen question type. Since the question allows to pick up to one answer, we only need a column, called "q_1_a_1".

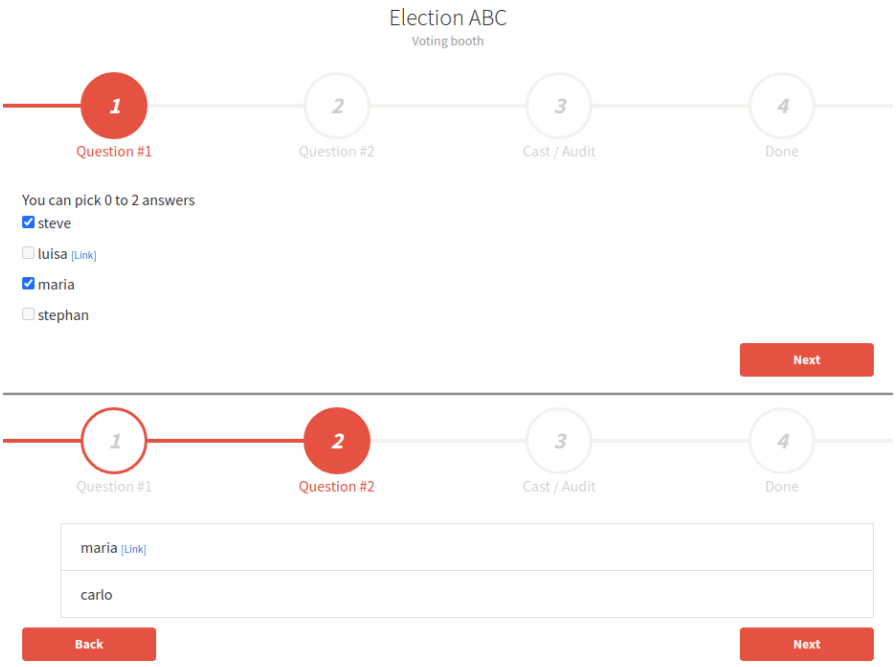

Figure 16: The two questions of an election: the first is a multiple choice question while the second is a STV question.

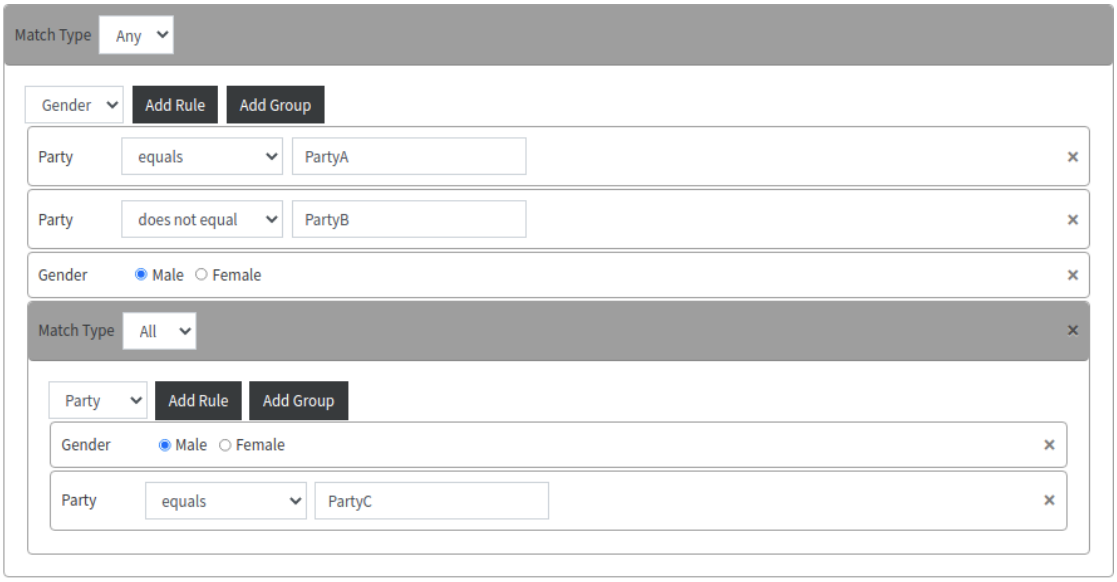

Figure 17: Question editor which generates the filtering clause used in the tally query.

By providing a way of specifying additional attributes for each answer (e,g,. gender, age, party) we allow an user to define constraints based on said attributes.

Designing a *usable* voting system requires an interface easy to use that allows to define constraints without any coding skill. Figure 17 shows an example of the traditional rule editor offered by many websites to define constraints.

The image also includes the dynamically generated SQL filtering clause that will be inserted in the tally query.

We avoid any custom low-level constraint definition by the user, which would also require deep validation and sanitization. The limitations of this approach are now given by the usability constraint of the user interface.

While the structural requirements needed for this feature are already in place, the rule builder of figure 17 is currently hidden merely because of a poor user experience that requires further improvements.

3.5 Ballot formats

In order to provide an elastic and general voting framework, we must take into account ballots with generic data structures of variable size. Designing a voting system capable of only receiving a fixed number of candidate choices would not provide generalization.

Whereas Helios and Zeus use the binary encoding and enumeration of all possible ballots aforementioned, we propose the use of a single numeric representation of the JSON encoding of the ballot. Once the JSON string has been converted into a numeric format, this value can be encrypted (e.g, using ElGamal) and, once the vote is decrypted, it has to be converted back into string, then decoded into the original object/array format. At this point the ballot is inserted as a record in the database table containing all ballots described in the previous section.

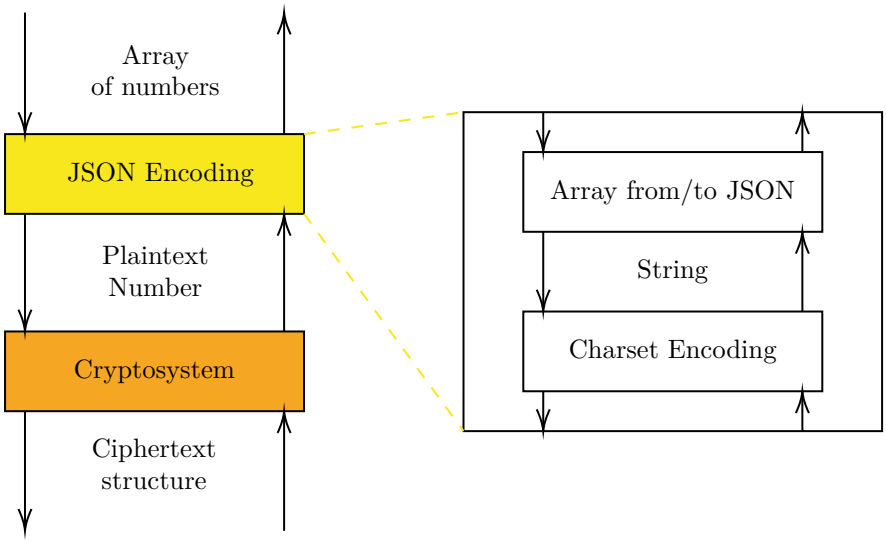

Figure 18: Ballot format.

A set of n_i candidate chosen for question i can be represented as a list of the n_i IDs of said candidates: $C^i = \left[c_1^i, \ldots, c_{n_i}^i\right]$. For a ballot with m questions, the representation will be $C = \left[C^1, \ldots, C^m\right] = \left[\left[c_1^1, \ldots, c_{n_1}^1\right], \ldots, \left[c_1^m, \ldots, c_{n_m}^m\right]\right]$, whose JSON encoding can be translated into numeric representation.

Depending on the charset of the string, the numeric representation can have different bit count. A string encoded with the UTF-8 encoding can require from one to four bytes per character [25].

Custom 4 bit charset Even though JSON allows us to encode and nest objects, because of the tabular structure of a relational DBMS we can ignore objects and strings only focusing on arrays of integers. [1] This constraint allows us to only consider 13 characters, which can be encoded with the custom charset shown in Table 4 in which only 13 out of the 16 characters offered by an hexadecimal alphabet are used. As a convention, the three unused characters represent three invalid characters which would corrupt the JSON value and thus have not to be used. This representation only requires 4 bits per character, allowing for bigger ballots than those representable with a heavier charset and thus allowing for elections with more questions and answers.

KAIROS provides two ballot encoding modules: the first encodes the JSON string by means of the UTF-8 charset and the second makes use of the custom charset just mentioned.

char	0..9	[	]	,	!	@	#
hex	0..9	a	b	c	d	e	f

Table 4: Proposed alphabet.

[1] When using document-based DBMS's one could decide to provide nesting capabilities ad to store ballots as documents.

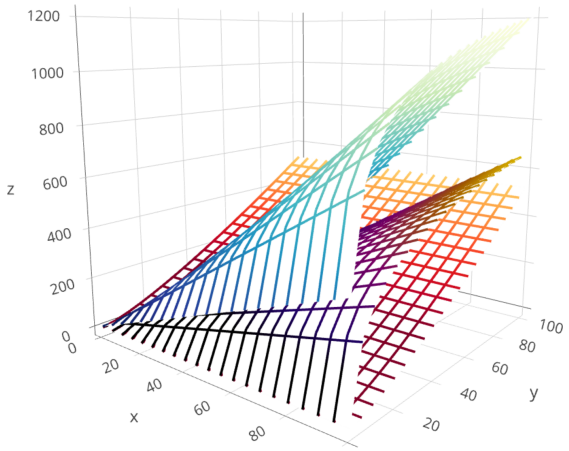

Figure 19: Bit length (z axis) required to encode k answers (x axis) out of n, the maximum number of answers (y axis) for a single question. The blue surface indicates the custom encoding just mentioned, the purple one indicates the encoding used in Zeus and the red one represents the one used in Helios. The 2D representation of these surfaces is hard to visualize so the plot has been rotated and thus the origin of the vector space can be found at the left of the picture.

Bit length of the representation When designing an encoding algorithm we have to take into account the maximum plaintext length supported by the cryptosystem.

The number of characters needed to encode k_i out of n_i answers for question i is $O(l_i)$ with

$$l_i = 2 + (k_i - 1) + k_i \log_{10} n_i$$

as the final plaintext will be composed of a leading "[" and trailing "]", $k - 1$ commas and k times the maximum length of the ID. Using the proposed charset the bit length is $4l$. With an ElGamal parameter p of 2048 bits the number of answers that can be handled for a question is approximately 160. While this result can be considered a sufficiently high amount of answers, this limitation can be removed by splitting the plaintext up in multiple parts and submitting multiple ciphertexts.

Figure 19 shows clearly how the custom encoding just mentioned and the one proposed in Zeus scale with the number of actual answers whereas the encoding used in Helios scales with the number of all possible answers, including the non-picked ones. Despite this custom encoding representing a smaller number of answers than the one used in Zeus, the first is more straightforward than the latter and does not require the enumeration of all possible combinations of candidates.

Storage space As mentioned in 2.2.1, homomorphic encryption requires one ciphertext for each answer to each question while ballot encoding techniques such as the one proposed in Zeus or the one just presented only need one, saving a considerable amount of storage space that depends on the number of answers.

Preventing vote selling A voter capable of altering the voting machine, would be able to encrypt a recognizable JSON string that would allow to sell the vote. Without any further measure a voter could add extra spaces and encode "[1, 2, 3]" instead of the expected "[1, 2, 3]" maintaining the same decoding. In case some subtle type conversion is in place, also the string "[1,′ 2′, 3]" would decode to the same valid vote string.

This can be avoided by a strict check on the JSON decryption. Let v be the plaintext vote. If $D(E(v))$ does not strictly match v or if the elements are not integers the vote is rejected. This assumes an identical implementation of JSON serialization on both the bulletin board and the voting device.

This amount to the check performed by election official on ballots which are excluded if made recognizable by signs.

3.6 Anonymization methods

Both homomorphic encryption and MixNets are implemented in Kairos and can be selected as the method to use for each election separately. The Blind Signature technique has not been implemented yet, but could be easily integrated by creating an additional module.

Figure 21 shows the election editor of Kairos which allows to specify which cryptosystem and anonymization method to use. As mentioned before, anonymization methods are tightly related to cryptosystems and thus some constraints are defined on the choice of the two options. Homomorphic encryption can only be picked when the chosen cryptosystem is Exponential ElGamal while MixNets follow the opposite logic and are only available in conjunction with ElGamal and RSA.

Each anonymization method is handled by a dedicate class which has to implement common methods require by a AnonymizationMethod interface. Figure 20 shows the UML diagram of the hierarchical implementation in which we have a class for homomorphic encryption, one for encryption MixNets, one for decryption MixNets and one for an hybrid approach which will be presented in the next sections.

Each class has to define callbacks to run at specific times of the election phase such as AfterVotingPhaseStops(), which is called when the voting phase is concluded. When using Homomorphic encryption, this callback is responsible for aggregating the ciphertexts and for decrypting the output of the tally. When using MixNets instead, this callback has to invoke the more complex mix process phase.

3.6.1 MixNets

All three kinds of MixNets share common methods such as those responsible for storing a mix to storage and for the corresponding loading. Each MixNet class provides methods needed to generate a proof of a mix and to verify them. Each mix is identified by a record in the database containing a reference to the previous mix, if any, and a reference to the entity who generated it. This last attribute is needed to describe a sequential mix process over multiple trustees, concept which will be described in section 3.8.

Figure 22 shows four mixes generated in a sequential manner by three peers (trustees 45, 46, 47) in two separate chains, each providing a link to download a public JSON file containing both the primary mix and all shadow mixes, as shown in figure 23. Figure 23 shows how the mix was

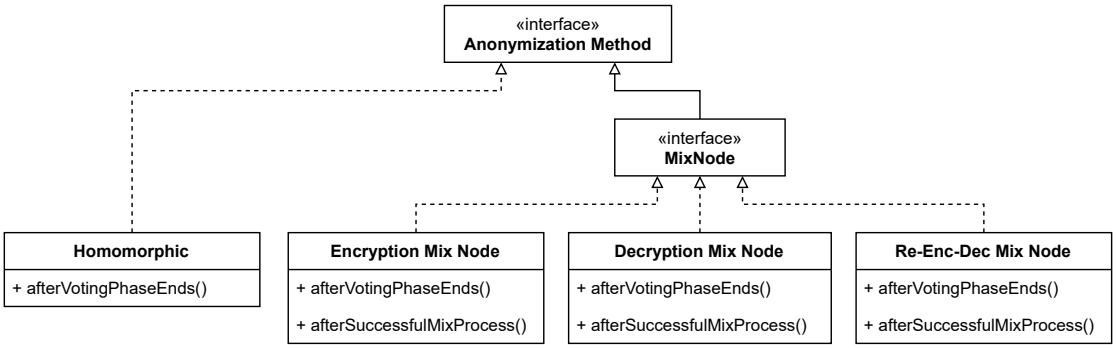

Figure 20: Anonymization methods.

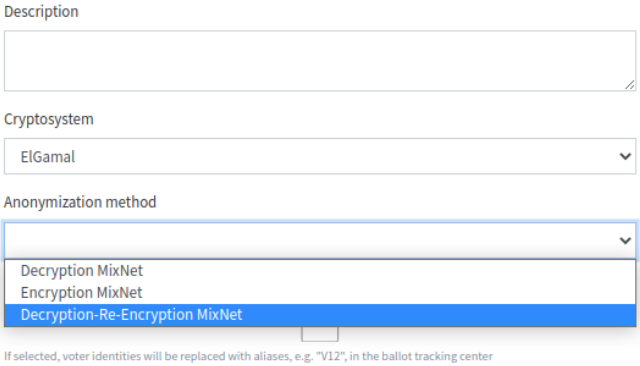

Figure 21: The election editor allows to choose both the cryptosystem and the anonymization method.

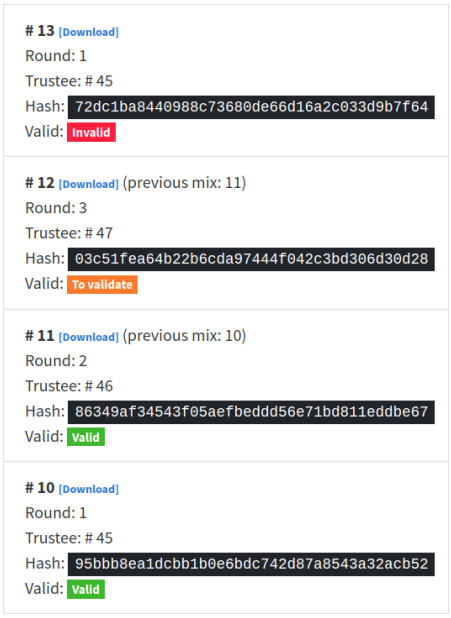

Figure 22: Example of four re-encryption mixes generated by three peers in two chains. The second (#12) and the third (#11) mixes also display the ID of the previous mix and they are generated once the previous peer has sent a mix proved correct.

fed 25 ciphertexts, produced a primary mix with the same number of ciphertexts and 80 shadow mixes starting from a set of 80 challenge bits obtained with the Fiat-Shamir heuristic. It is important to note that the parameter set of the primary mix was set to null after the generation of all proofs.

3.6.2 Re-Encryption-Decryption ElGamal MixNet

As previously mentioned in the Helios implementation, the server itself is not guaranteed to behave honestly and could use the private keys to decrypt the original bulletin board where each vote also contains a reference to the voter.

In section 2.3 we compared different MixNet approach and we saw how the literature favorites re-encryption MixNets with ElGamal and decryption MixNets with RSA. Table 2 summarized the characteristics of different designs, showing how a Decryption MixNet with ElGamal would yield a more elastic solution than its RSA equivalent when making use of a $t - \ell$-threshold encryption scheme. Such a design appears to be original to the best of my knowledge.

In the typical scenario of a single server running a re-encryption MixNet, before the tally takes place, all the secret keys x_i are sent to it by the trustees in order to decrypt the anonymized ballot set. The server is trusted not to use this key on the anonymized ballot set fed as input to the MixNet voiding the anonymization process, a massive assumption. Even with valid code running on the server, an user could log in and extract the combined private key, which can be used on the public bulletin board.

Avoiding trustees from sharing their private keys before the tally collides with the approach used in re-encryption MixNets. This can be solved by making each trustee partially decrypt

```
{
  "election_uuid": "5a4d7ac0-f525-4488-9b80-3cd5733ccaf1",
  "challenge_bits": "10001100111000001101110010101100110111110110100001111000011111111001110100111110",
  "original_ciphertexts": [ … ], // 25 items
  "primary_mix": {
    "election_uuid": "5a4d7ac0-f525-4488-9b80-3cd5733ccaf1",
    "ciphertexts": [ … ], // 25 items
    "parameter_set": null
  },
  "shadow_mixes": [ … ], // 80 items
  "parameter_sets": [ … ], // 80 items
  "proofs": [ … ] // 80 items
}
```


Figure 23: JSON file containing the primary mix, the shadow mixes and all needed parameters of mix #10 of figure 22.

the ballot set sequentially without sharing its key, thus adopting a decryption MixNet. With this new approach each trustee never shares its private key and has control over the decryption phase.

Sequential partial ElGamal decryption The traditional ElGamal decryption of equation (2) can be performed in a sequential manner:

$$\begin{aligned} m =& \beta \left[\prod_{t=1}^{\ell} \alpha^{x_t} \right]^{-1} \\ =& \beta \left[\alpha^{x_1}\right]^{-1} \cdots \left[\alpha^{x_\ell}\right]^{-1} \end{aligned}$$

As shown, the decryption can be executed in a sequential way where the first trustee computes (10) and passes the result over to the next one, which will repeat the same operation. Once the sequence of t steps has been completed, β will contain the decrypted value m.

$$(\alpha_1, \beta_1) = \left(\alpha, \beta \left[\alpha^{x_1}\right]^{-1}\right) \tag{10}$$

This sequential approach works with a generic $t - \ell$ threshold encryption scheme.

It is important to point out that since α is required by all trustees it has to be passed forward, however since its values does not change between mix stages it voids the mix procedures as one vote can be traced back. As shown in (10), $\beta_1 \neq \beta$ while $\alpha_1 = \alpha$. This technique alone is thus not suitable for a MixNet, as it would void the ballot secrecy property.

This issue can be resolved applying an additional re-encryption step, as seen in 2.2.2, which allows to alter both α, β without altering their decryption. Since the value of α is now changing, the mix process is no longer voided.

By combining the partial decryption seen in 3.6.2 with the re-encryption property seen in 2.2.2, we obtain a new MixNet approach: a Re-Encryption-Decryption MixNet. This process is equivalent to a re-encryption MixNet and a decryption MixNet that operate in serial.

The re-encryption has to be performed with respect to the combined public key obtained by the public key shares of the mix nodes that haven't performed the partial decryption yet. Since server i performs the re-encryption step before the partial decryption with $y_i = Sk_i$, the first has to be performed with respect to the combined key obtained by combining y_i with all public

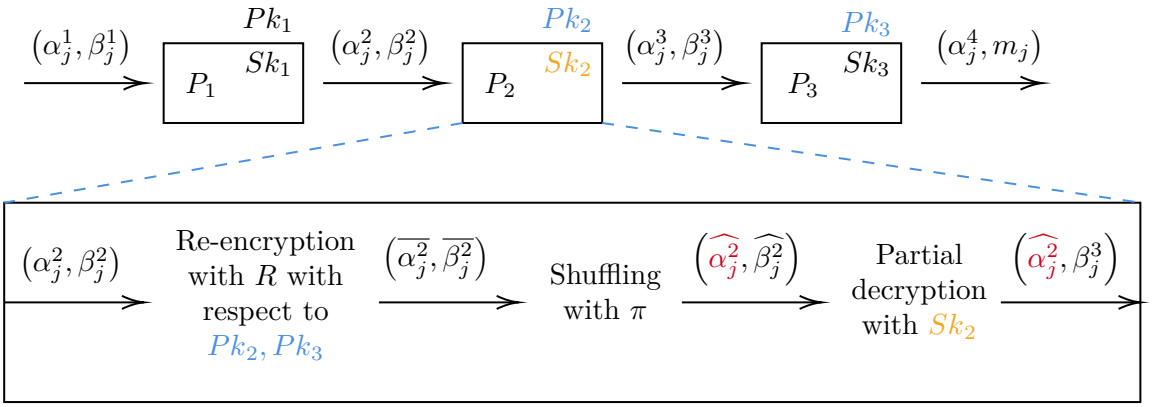

Figure 24: Re-Encryption-Decryption MixNet with three nodes P_1, P_2, P_3 and n ballots encrypted as n ElGamal ciphertexts (α_i, β_i) with $i = 1 \dots n$.

keys of the next servers:

$$\prod_{k=i}^{P} y_k$$

After the partial decryption each peer updates the public key as the combination of the shares of those peers that have still to perform the decryption, that is the currently public key divided by its own public key share. Once all peers have performed their tasks, the public key will be 1, allowing to detect whether the mixed sequence has reached completion without additional information.

Figure 24 shows an example of a MixNet that implements this new design. The values fed as input are ElGamal ciphertext encrypted with a combined secret key obtained from Sk_1, Sk_2, Sk_3 The values of α marked in red indicate where the issue described in 3.6.2 occurs. By adding a re-encryption we can prevent an observer from tracing $\widehat{\alpha_j^2}$ back to α_j^2, achieving anonymity. At the end of the decryption chain the original vote m_j appears as the value of β.

Although this MixNets is very similar to a traditional re-encryption MixNet, this slight modification brings an important security upgrade to the traditional approach as secret keys are not shared. This also has a massive impact on a distributed MixNet, such as the one presented in the next sections, performed by many peers sequentially.

Proving the mix The proof of this is obtained by combining the re-encryption MixNet proof seen in section 2.3.2 with the proof of decryption seen in 2.2.

Once the primary mix has been generated by performing re-encryption, shuffling and partial decryption, n additional shadow mixes are generated by only performing re-encryption and shuffling.

A sequence of n challenge bits c_i is provided or generated by means of the Fiat-Shamir heuristic and the shadows mixes are proved equal to the original ballot set (left) if $c_i = 0$ and to

the primary mix (right) if $c_i = 1$ as in the traditional proof.

The left equivalence proof amounts to undoing the shuffling and the re-encryption steps obtaining a value that has to be equal to β_j.

The right equivalence requires to perform all the operations of the left equivalence with an additional proof of decryption seen in section 2.3.2 that allows Sk_1 not to be shared.

Analytical proof The following is the proof of correctness by induction of the proposed design based on two cases of a mix network with only one mix node ($P = 1$) and of a mix network with two or more mix nodes ($P \geq 2$). By combining these two cases it is possible to prove that the output of a Re-Encryption-Decryption MixNet with arbitrary number of nodes is the plaintext m.

Mix network with one mix node Given a mix network of one node ($P = 1$) and its public key $y_1 = g^{x_1} \mod p$, the encryption of a message m with encryption randomness r is

$$(\alpha, \beta) = (g^r, \quad m y_1^r) \mod p$$

where the public key y is composed only of those nodes which have not performed the decryption yet, thus only y_1:

$$y = \prod_{i=1}^{P=1} y_i = y_1$$

The re-encryption of (α, β) with re-encryption randomness r_1 is

$$(\alpha', \beta') = (\alpha g^{r_1}, \quad \beta y^{r_1})$$

The partial decryption of (α', β') with secret key share x_1 is

$$\begin{aligned}
(\alpha_1, \beta_1) &= \left(\alpha', \quad \beta' \left[\alpha'^{x_1}\right]^{-1}\right) \\
&= \left(\alpha g^{r_1}, \quad \beta y_1^{r_1} \left[(\alpha g^{r_1})^{x_1}\right]^{-1}\right) \\
&= \left(g^r g^{r_1}, \quad m y_1^r y_1^{r_1} \left[(g^r g^{r_1})^{x_1}\right]^{-1}\right) \\
&= \left(g^{r+r_1}, \quad m y_1^{r+r_1} \left[\left(g^{r+r_1}\right)^{x_1}\right]^{-1}\right) \\
&= \left(g^{r+r_1}, \quad m g^{x_1[r+r_1]} \left[g^{x_1[r+r_1]}\right]^{-1}\right) \\
&= \left(g^{r+r_1}, \quad m\right)
\end{aligned}$$

which contains m as the value of β_1, as expected.

Mix network with two or more mix nodes Given a mix network of $P \geq 2$ nodes and their public keys $y_1 = g^{x_1}, \ldots, y_P = g^{x_P}$, the encryption of a message m with encryption randomness r is

$$(\alpha, \beta) = \left(g^r, \quad m \left[\prod_{i=1}^{P} y_i \right]^r \right) \mod p$$

where the public key y is composed only of those nodes which have not performed the decryption yet:

$$y = \prod_{i=1}^{P} y_i$$

(Server 1) The re-encryption of (α, β) with re-encryption randomness r_1 is

$$(\alpha', \beta') = (\alpha g^{r_1}, \quad \beta y^{r_1})$$

(Server 1) The partial decryption of (α', β') with secret key share x_1 is

$$\begin{aligned}
(\alpha_1, \beta_1) &= \left(\alpha', \quad \beta' \left[\alpha'^{x_1} \right]^{-1} \right) \\
&= \left(\alpha g^{r_1}, \quad \beta \left[\prod_{i=1}^{P} y_i \right]^{r_1} \left[(\alpha g^{r_2})^{x_1} \right]^{-1} \right) \\
&= \left(g^r g^{r_1}, \quad m \left[\prod_{i=1}^{P} y_i \right]^{r} \left[\prod_{i=1}^{P} y_i \right]^{r_1} \left[(\alpha g^{r_1})^{x_1} \right]^{-1} \right) \\
&= \left(g^{r+r_1}, \quad m \left[\prod_{i=1}^{P} y_i \right]^{r+r_1} \left[(g^r g^{r_1})^{x_1} \right]^{-1} \right) \\
&= \left(g^{r+r_1}, \quad m \left[\prod_{i=1}^{P} y_i \right]^{r+r_1} \left[g^{x_1 [r+r_1]} \right]^{-1} \right) \\
&= \left(g^{r+r_1}, \quad m \left[\prod_{i=1}^{P} g^{x_i} \right]^{[r+r_1]} \left[g^{x_1 [r+r_1]} \right]^{-1} \right) \\
&= \left(g^{r+r_1}, \quad m \left[\prod_{i=2}^{P} g^{x_i} \right]^{[r+r_1]} \right) \\
&= \left(g^{r+r_1}, \quad m \left[\prod_{i=2}^{P} y_i \right]^{[r+r_1]} \right)
\end{aligned}$$

which is equivalent to the decryption performed by a MixNet of $P - 1$ nodes of plaintext m encrypted with voting randomness $r + r_1$.

On top of proving the correctness of the output of a mix node, it is paramount to prove the hardness of tracing the output mix back to the input, issue that would void the mix procedure.

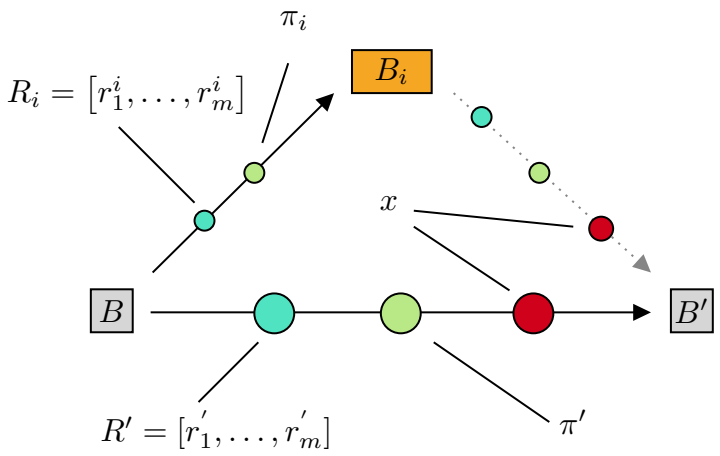

Figure 25: Re-Encryption-Decryption MixNet obtained by adding a partial decryption step (red dots) to a regular Re-Encryption MixNet.

The four problems that have to be addressed when using the Sako-Kilian design [10] are:

1. Tracing the primary mix back to the input must be hard without the parameter set.
2. Tracing the primary mix back to a shadow mix must be hard without the parameter set.
3. Tracing a mix shadow back to the input must be hard without the parameter set.
4. Finding the correlation between two shadow mixes must be hard when the two provided parameter sets prove equivalence to the opposite sides.

Points (1) and (2) are both guaranteed by the hardness of the decisional Diffie Hellman Assumption (DDH) problem, the underlying assumption of ElGamal shown in 2.2. Points (3) and (4) are both guaranteed by the proof of the original Sako-Kilian Re-Encryption MixNet design [10].

3.7 Cryptosystems

In order to achieve generalization, a big hierarchical structure was created, capable of generalizing multiple cryptosystems with different data structures. KAIROS implements RSA, ElGamal and Exponential ElGamal cryptosystems. Each cryptosystem is implemented as a series of classes, all of whom implement high level common interfaces such as CRYPTOSYSTEM, PUBLIC KEY or KEY PAIR. Interfaces and polymorphism provide a high generalization level, allowing cryptosystems to be replaced or added without any change to the rest of the infrastructure required.

Due to the elasticity of ElGamal and its Exponential variant, these cryptosystems were given the highest priority and were fully developed and tested. The classes that define Exponential ElGamal extends those responsible for regular ElGamal, overriding specific methods such as ENCRYPT() according to the details shown in 2.2.1. Figure 26 shows the hierarchical structure for ciphertexts, one of the many needed to fully generalize each cryptosystem.

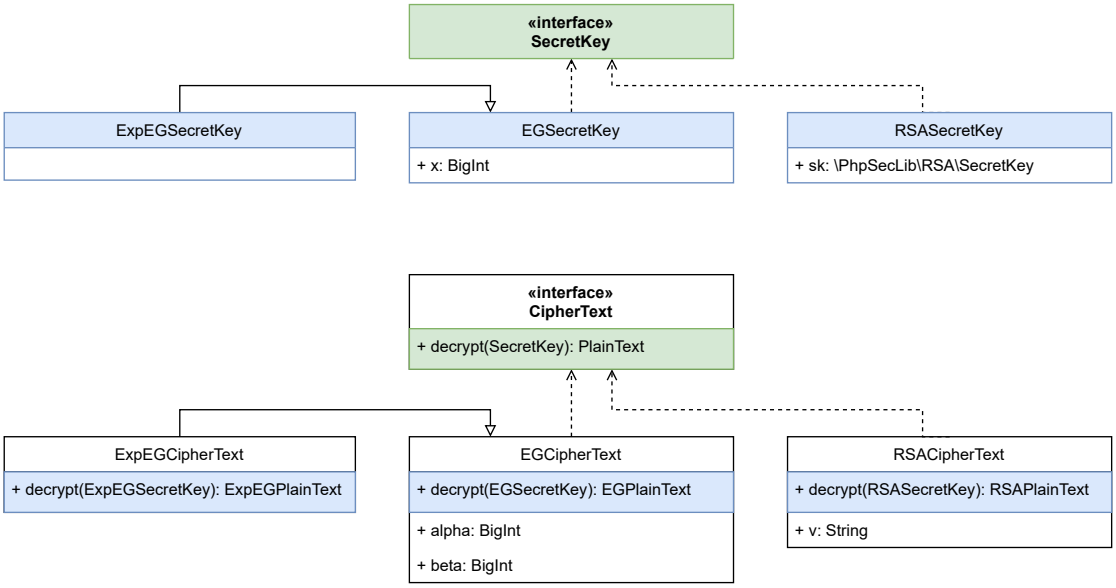

Figure 26: UML diagram of the hierarchical structure defined for ciphertexts and cryptosystems in general.

Server side encryption While PhpSecLib supports RSA, and thus the new classes are simple wrappers, ElGamal is not implemented so its classes define the latter cryptosystem entirely.

The BigInt operations are handled by the data types offered by the PhpSecLib library [15].

Each cryptosystem has a dedicate class that represents a ciphertext of said cryptosystem and a dedicate casting class responsible for serialization and unserialization of ciphertext objects. This serialization methods are used to store and load a ciphertext to and from the database and to communicate with the front end side. As an example, while a RSA ciphertext is represented as a single string, the ElGamal's caster has to deal with the pair (α, β) and thus has to convert the pair to string during serialization and vice versa during unserialization. The serialized string also contain the identifier of the cryptosystem which allow to recover the class which has to be instantiated when the value is unserialized.

Depending on the context, the serialization can ignore redundant fields such as the cryptosystem parameters or the public key of an object. This is the case for the ciphertexts, where the public key is stored in the election record and including it in every ciphertext would be a waste of memory and time.

Client side encryption Each back end PHP class that represents a cryptosystem requires an equivalent JavaScript version which is mainly responsible for the encryption phase that takes place during the ballot seal process.

The voting booth in the front end part of the voting system is responsible for encrypting the ballot before submitting it to the server.

In order to perform BigInt operations efficiently, *Helios* relies on a Java Virtual Machine accessed through LiveConnect. These dated technologies have been replaced by BigInt structures provided by all major browsers. Nowadays all major browser, with the exception of Internet

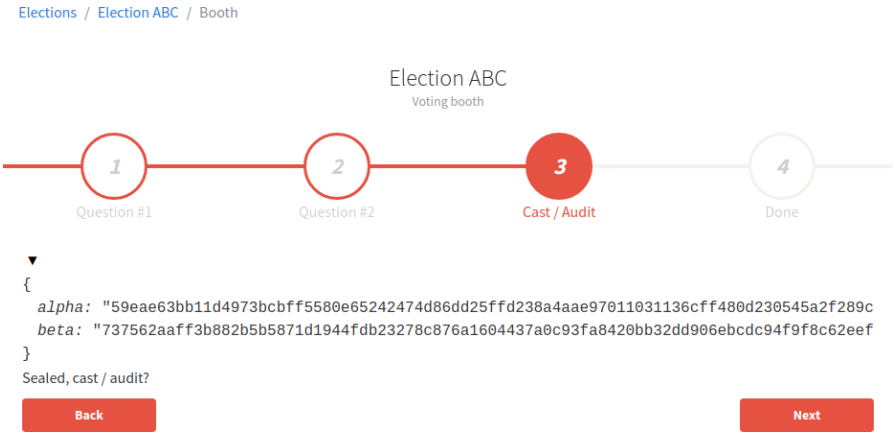

Figure 27: Once the ballot is sealed the UI shows the vote encrypted with the chosen cryptosystem. The example in the picture shows a vote encrypted with ElGamal.

Explorer, support the BigInt type object natively [14]. While operations such as ModPow are not available out of the box, those have been imported from third-party Node modules.

Every JavaScript cryptosystem class implements the same serialization features of the equivalent PHP class.

Figure 27 shows the representation of a sealed ballot encrypted with ElGamal. Note that the voting booth calls common methods implemented by all cryptosystems, an thus the UI displays whatever object is returned by the cryptosystem class. In case of RSA, the encrypted ballot would a single long string.

3.8 Peer2Peer

This section covers a P2P implementation capable of performing the tasks described in the previous sections in a distributed environment with multiple peers cooperating.

Whereas Helios deals with users acting as trustees, many operations could be automatized by having said tasks performed by servers. An example of an operation that is hard to carry out with human trustees is the protocol needed for the $t - \ell$-threshold encryption scheme presented in the second section, as it requires a number of interactive exchanges and proofs.

The MixNet approach has also a strong relation with the concept of communicating peers, due to its nature and shape. As an example, by transforming the centralized design into a P2P network, each node can be the verifier of a mix generated another node.

The Peer to Peer implementation described below comes from a natural translation of the Pedersen's distributed key generation algorithm (DKG) seen in section 2.2.4 expanded to account for all operations needed to manage an election.

This distributed approach requires all peer to share the same code and since each node has the whole set of features at its disposal, it can manage its own elections while perhaps being a trustee for an election created by another peer. As an example, in the context of elections for Universities, one scenario could be of each university being a peer in a network of relatively trusted entities.

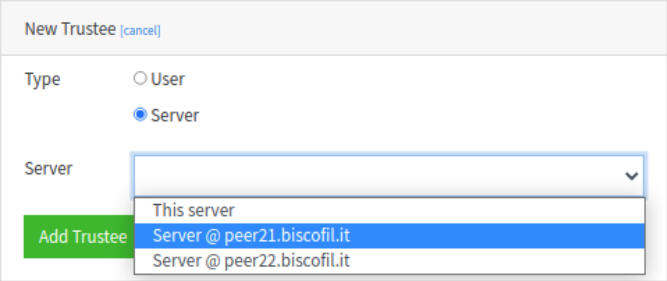

Figure 28: Modal that allows to create a trustee for an election on the server with third level domain "Peer20". The list of servers contains the server itself and two other servers "Peer21" and "Peer22".

For now we only consider peers acting as trustees that interact to anonymize, if needed, and decrypt a ballot set provided by the election creator that acts a bulletin board.

The whole application was developed on a local machine and periodically copied to three Virtual Private Servers (VPS) servers rented around the globe to simulate a realistic scenario of usage that requires to deal with latency. In order to ship the whole application more easily, tools like DOCKER and DOCKER-COMPOSE were used. The three servers were assigned a third level domain name managed through CloudFlare and a SSL certificate issued by LET'S ENCRYPT by means of the CERTBOT tool.

Working with a distributed system requires to deal with numerous issues, with some of them requiring a significant part of the current university program. Some of these issues are described in the next paragraphs below.

Unique identifiers Since we need to identify a model with the a unique identifier across many server without collisions, an issue that had to be addressed is the discrepancy among different databases of the set of unique keys. KAIROS uses UUIDv5 identifiers generated by hashing a concatenation of a standard namespace UUID and the unique domain name followed by the URL path containing the local identifier of the identified object.

When a peer B is added by peer A as trustee for an election, a copy of the election record is copied to B's database and the UUID generated by the creator has to be unique in both databases allowing for retrieval of the same record on both servers.

Coherent time values Scheduled operations have to be executed at the same time by every peer so the format of the time must be coherent across all servers. In KAIROS all time values are store in the database in their UTC values as the LARAVEL framework automatically converts the values with respect to the timezone of the server.

Peer server trustees KAIROS generalizes the concept of *trustee*: whereas HELIOS considers trustees as users, this new design allows peers to be picked as automatic trustees. Figure 28 shows the modal that allows to create a new trustee for an election. The user that manages the election can pick from the list of server the server has already performed an handshake with.

Figure 29 shows the user interface with the selector that allows to specify the value of t for the $t-\ell$-threshold encryption scheme. This selector is only visible when the cryptosystem chosen

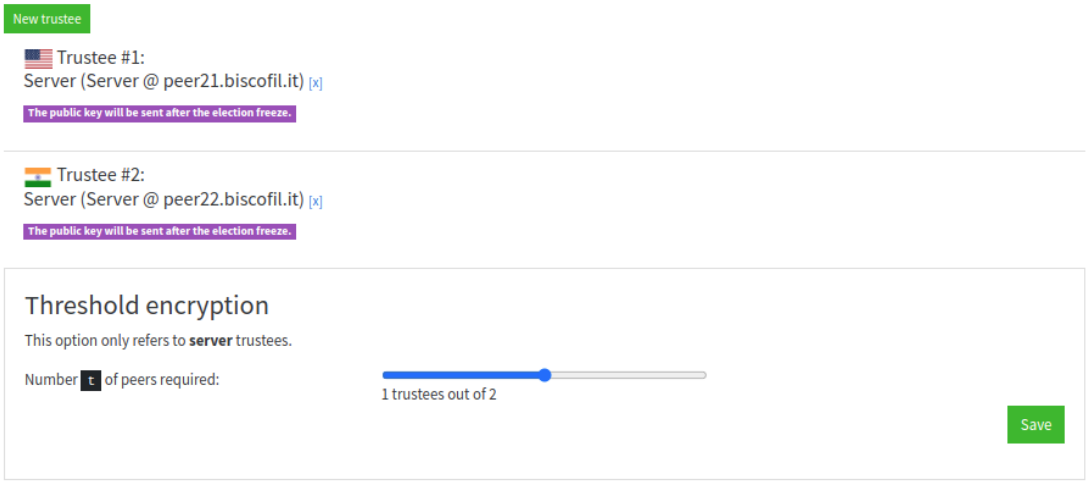

Figure 29: Selector for the number t of servers out of ℓ that have to share their secret key.

for the election supports $t - \ell$-threshold encryption.

In case $t = \ell$ all servers have to share their private key shares without any margin for dishonest or malfunctioning peer. In this first case, when the election is frozen, each peer is only required to generate a keypair of the picked cryptosystem and to share its public key.

In case $t < \ell$ there is some margin for $\ell - t$ peers not to be heaving honestly. In this second case each peer is also required to perform some operations needed to reconstruct keys with only t trustees. In case the ElGamal cryptosystem is chosen, these operations amount to generating a polynomial and to send a broadcast to every other peer as described in 2.2.4.

Peers with different Kairos versions Even if not yet addressed, when dealing with multiple servers managed by different institutions, the assumption of having the same version of the software on all peers may not be correct. By having an election with peers running different versions of the code we could end up with undesired outcomes and errors.

This issue can be addressed by including a version number in the project and by comparing it with server added as peers for an election.

These errors occurred even during the development of KAIROS, when the PHP code was automatically copied to all servers, due to a subtle caching of the code executed by DOCKER.

Network monitoring page In order to simplify the debugging of multiple servers without monitoring several consoles, a section was added to the platform to monitor realtime messages shared by all servers through an external WebSocket service. The interface is shown in figure 30 and displays an animation of the path taken by messages on the world map. In the picture two HEARTBEAT messages are captured as they are traveling from the German server to the Indian and Californian peers.

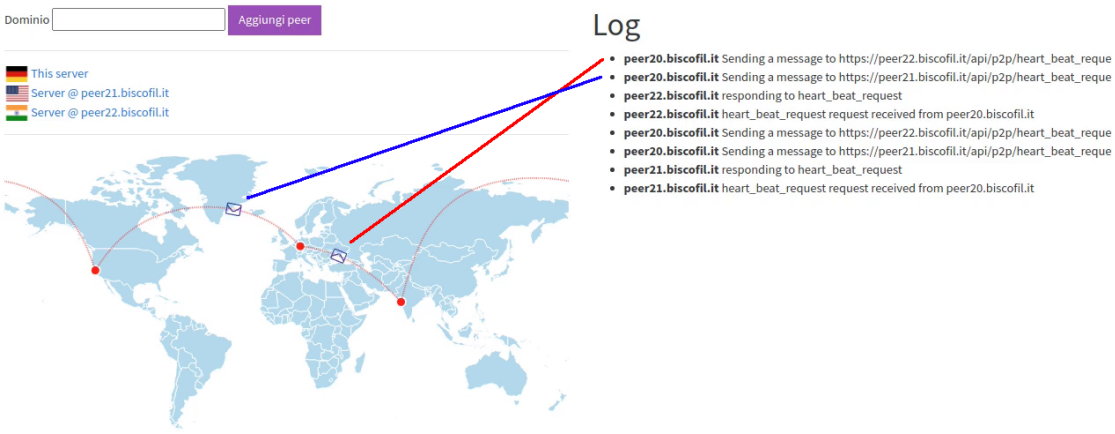

Figure 30: Monitoring page displaying some messages exchanged between servers. The red lines represent of the connections between three servers rented in India, California and Germany. This interface was provided by the German server.

3.8.1 P2P with HTTP and WebSocket

When working with multiple servers, a communication protocol has to be defined. The ideal approach would be of using an existing communication solution, as in the case of a DOCKER container responsible for the network related tasks, capable of working in an arbitrarily shaped mesh network. Building these features upon an external service however results in a more complex development phase as the KAIROS framework has to adapt to the existing communication solution. Due to time constraint KAIROS temporarily handles the communication between servers with a relatively low-level approach, dealing directly with HTTP requests and WebSocket connections, discarding many existing protocols and paradigms. In the future this temporarily workaround has definitively to be replaced with a more long lasting and more optimized solution. Many of the encountered issues also, have already been solved in existing solutions and are not worth being solved from scratch.

KAIROS defines two modules responsible for the communication between servers which have to provide ways to receive and send messages.

The first module makes use of the HTTP protocol and has to deal with bidirectional communication as the protocol is based on the request-response design. The HTTP module works in a non blocking manner by receiving the incoming requests directly from the web server itself. The HTTP protocol is based on the TCP protocol of the transport layer, and does not provide broadcast features. In order to perform a broadcast with this first HTTP module, the module sends multiple unicast messages.

The second module makes use of the WebSocket protocol, an alternative to HTTP that is capable of broadcast [24]. Unlike the HTTP module, listening for incoming messages with WebSocket requires to establish a connection and to keep it open by means of an endless loop.

The greater speed of WebSocket with respect to HTTP comes at the cost of abandoning the request-response design of HTTP, which would requires a custom infrastructure for memorizing sent messages and identifying them later on when a response message arrives.

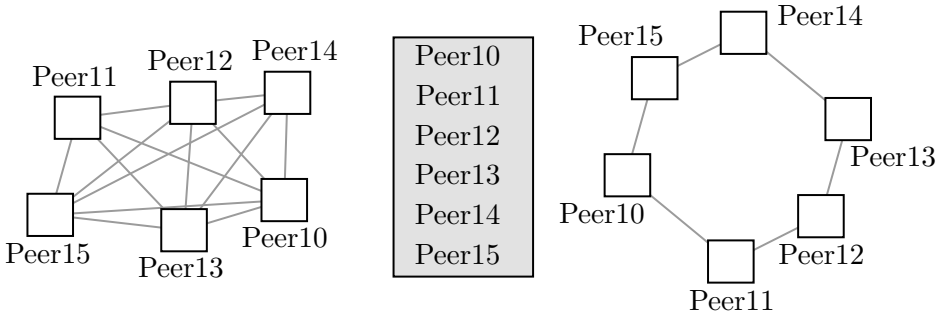

Figure 31: An order can be established among all network nodes by computing a value starting from their unique identifiers. Since we are working with web servers we can use the domain name.

WebSocket also handles private channels for which a client has to provide an authentication token in order to subscribe. One could decide to make the n peers communicate privately by means of $\binom{n}{2}$ private WebSocket channels for private connections replacing HTTP entirely.

While LARAVEL provides a WebSocket server out of the box, in order to avoid each server from subscribing to every other server's channel, the module makes use of an external commercially available WebSocket service.

When performing broadcasts of particular messages, as in the key generation seen in 2.2.4, we want the guarantee of an identical message being sent to every subscribed peer. Whereas this could be achieved with HTTP and a complex chain of signatures by each receiving peer, by using a single external WebSocket server the guarantee can be considered fulfilled by moving to an assumption of honesty of the third-party service.

Due to time constraints, the implementation of KAIROS currently makes use of the HTTP module for every communication.

Ring Network Since the set of ℓ peers for an election is fixed, we can sort the list of servers according to the alphabetical order of their domain names and create a circular path agreed upon by all nodes. The first peer of the list is assigning ID 0 and all subsequent operations are achieved by performing computations in arithmetic modulo $\ell - 1$. This order will be used in the distributed algorithms presented in the next sections.

This "ring" structure is nothing more than a sorting as the messages are still exchanged directly between peers.

3.8.2 A structure for a message

In order to allow messages to be sent with any transport protocol we can generalize them as classes with serialization an unserialization methods. When a sender wants to send a message to a destination, it creates an instance of the message class and sends it to the destination by specifying the transport module. The chosen module calls the serialization method of the message and proceeds to send the obtained string to the destination according to the implemented protocol. Once the destination server receives the message it extracts the identifier of the message type and calls the unserialization method of said class, retrieving an object identical to the one

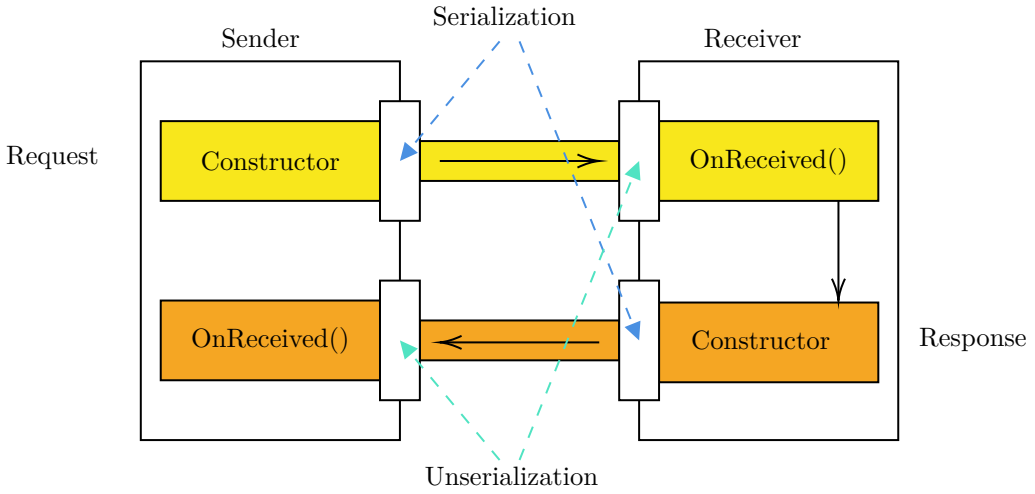

Figure 32: When using HTTP requests the request-response design is modeled with a pair of unidirectional messages.

serialized by the sender. The destination then executes the OnReceived() method of the object, which performs some operation and returns a response message, if needed.

When using HTTP to communicate with another server, we have both a request and a response that will be sent back. Figure 32 shows how these two classes are used by the request sender and by the destination peer.

When B receives a HTTP request that requires to perform a time consuming task whose output has to be sent back to the request sender A, an empty response is returned and the task is performed asynchronously by a queue. The result of the task will be sent back to A trough a second HTTP request initiated by B.

Kairos implements several message types, each composed of a request message and a response message, which will be needed for the operations described in the next sections:

- AddMeToYourPeers: used to perform a handshake with a server, exchanging public keys and other attributes
- WillYouBeAElectionTrusteeForMyElection: used to ask a server to be a trustee for an election
- Freeze1IAmFreezingElection, Freeze2IAmReadyForFreeze, Freeze3CommitFail: used for the three phase commit election freeze protocol
- ThisIsMyThresholdBroadcast: used during the distributed freeze protocol to exchange threshold broadcasts and shares of a $t - \ell$-threshold encryption scheme
- ThisIsMyMixSetRequest: used to send a mix generated by a mix node to other peers
- ThisIsMySecretKeyRequest: used by anonymization methods that require peers to exchange their secret keys

3.8.3 Peer handshake and Authentication

In order for servers to authenticate to each other, they use a JWT token received during during the handshake protocol.

In the next sections we will see scenarios in which peers need each other's parameters, like when peer A has to verify a message was sent by B by checking the validity of its signature. Operations like these require an initial exchange of parameters such as the RSA public key of a peer, which is performed during an handshake that makes use of a AddMeToYourPeers message.

Each peer has a set of known peers, initially only containing itself. Peer A sends a request to B with its domain and its RSA public key Pk_A. B verifies the claimed domain by resolving it and by ensuring that the resolved IP matches the IP of the sender of the request. The response of the request contains the RSA public key of B, Pk_B. Other parameters are exchanged during the handshake, such as the timezone of the peer, the email address to use for information and the name of the voting platform. These public keys will be used in several contexts, such as signing JWT token and messages to prove their authenticity.

When a peer is added, a call to an external geolocation service is made to retrieve some geographical attributes such as the approximate GPS coordinates and country flag. While these values are only used for graphical reasons, in future one could decide to utilize the GPS coordinates of a series of server to determine the optimal path across them to minimize latency.

3.8.4 Distributed election freeze

In order to achieve consistency in the network, operations such as the election freeze have to be agreed upon by all peers.

In order to implement a distributed freeze, we make use of a three-phase commit protocol, represented in figure 33, coordinated by the creator A of the election .

In case the coordinator A is also a trustee and if the $t-\ell$-threshold encryption scheme is used, the first message also carries the broadcast and the share for peer B.

Once a peer B receives the first message, it has to confirm whether it is ready for the commit or not.

If a $t-\ell$-threshold encryption scheme is used, each peer has to broadcast and exchange share with every other peer but the coordinator. To optimize the number of messages between a pair of peers B, C we can exploit the request-response structure of HTTP by making the peer with lower label (domain) wait for the request made by the peer with higher label. When a peer receives the first message, a delay is added before proceeding to the exchange of shares in order to account for peer which might not have received the first message yet.

When using a $\ell-\ell$-threshold encryption scheme the peer has no task to perform and can immediately reply to the first message.

Once a peer is ready it sends a "ready" message back to the coordinator, which will wait for all peers to reply. With $t-\ell$-threshold encryption, in order to prevent a malicious peer from sending different broadcasts to different peers preventing the ballot set decryption, each message sent back to the coordinator also includes the list of received broadcasts, each signed with the secret key of the sender C. The coordinator A then checks if all peers are ready and if all broadcasts are valid. By having the broadcast signed by the owner, the coordinator can make sure that B is not trying to disqualify C by claiming it provided a false broadcast. If the signature is valid and

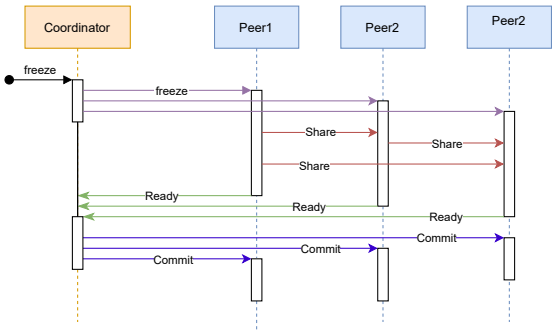

Figure 33: Freeze with a three-phase commit procedure.

the broadcast does not match the one received by other peers the actor to blame is the owner C whereas the opposite scenario with an invalid signature and a non matching broadcast has B as actor to blame.

If all checks are valid, the coordinator finalizes the transaction by sending a commit message to all peers, otherwise a fail message is broadcasted.

The whole freeze operation has a timeout timer that, once the time is over, cancels the freeze operation.

In a mesh network this freeze operation requires $3n + \frac{n(n-1)}{2} + 3n + 3n$ messages.

There exist alternative protocols for this kind of distributed operations, which can offer more robustness. Digging into the state of the art algorithms in distributed networks although is out of the scope of this thesis and thus similar improvements will require a code revision in the future. Luckily for developers, the modular structure of the project would allow to integrate new communication approaches without performing big changes to the rest of the codebase.

3.8.5 MixNets and P2P

A distributed system can benefit the anonymization phase that makes use of MixNets, especially when an interactive process is needed, like in the case of $t - \ell$-threshold encryption described in 2.2.4.

Whereas Helios makes the server perform a number of mixes and then waits for the secret keys of the trustees to be shared, in Kairos each peer server trustee is responsible for performing its own mix, to be shared with other peers later on.

When adopting a decryption approach, either a traditional decryption MixNet or the one proposed in 3.6.2, each trustee has exclusive control over its secret key.

Since the initial mix has to use the set of cast votes as input, the mix process has be initiated by a peer that acts as a bulletin board.

Once a mix M_A and the corresponding proofs have been generated by peer A, a message containing the description of M_A is sent to every other peer B. Once peer B has received this message it is free to download the file containing the mix and the proofs from A's server which can later be loaded and verified. The validity of the mix is then stored in the corresponding record of the database, whose representation is shown in figure 22.

After a mix M_A has been proven valid by the next peer B of the sequence, the mix is used as

input for the generation of another mix M_B. This operation continues until enough peer servers have performed their mix: in the case of a $t-\ell$-threshold encryption, this amounts to a chain of t valid mixes performed sequentially by t peer servers.

Depending on the choice of the anonymization method, the sequence of mix nodes through which a ballot set has to go through can either be strict or more elastic. Whereas ElGamal Re-Encryption MixNets allow for any path across the ℓ servers, an RSA decryption MixNet requires a ballot set to go through a set of mix nodes according to the sequence of encryption reverse.

Distributed ElGamal Re-Encryption mix: elastic sequence We now describe the most general protocol for a distributed mix, temporarily ignoring the need for a fixed path across servers. This is the case for a traditional ElGamal re-encryption MixNet, in which the mix process can be repeated an arbitrary number of times by any server. This sequence can be changed as the ElGamal re-encryption procedure is *idempotent*.

The sequence of interactions between servers, which has intuitively to be initiated by a server acting as bulletin board, is represented in figure 34.

We now consider a sequence of peers sorted according to the order presented in 3.8.1. Let the bulletin board server be peer $i = 0$.

The mix procedure starts with a bulletin board server i which uses the list of received votes as input for the mix algorithm.

After the mixes have been generated, both the primary mix and the shadow mixes are broadcasted to every other peer j. Peer j has then to generate the challenge bits which will then be sent back to peer i to generate its proofs. In case the Fiat-Shamir heuristic is used the "Generate proof" task (blue square) is performed directly after the mixing phase (green square) in a non interactive manner.

Regardless of the technique used to generate the challenge bits, the proofs are now at disposal of peer j which has to verify the correctness of the mix.

In case the mix generated by peer i is considered valid by peer $i+1$, the latter proceeds to generate another mix using the output of the received mix as input for another mix.

In case the mix is considered invalid, peer $i+1$ broadcasts a complaint against peer i and increments its local strike counter. Figure 35 shows one of many possible policies for dealing with invalid mixes: the algorithm in the picture shows how the mix of i is ignored by peer $i+1$ which then proceeds to generate a new mix staring from the previous mix generate by peer $i-1$.

Figure 36 shows a generic representation of the sequence of tasks performed by the three servers with the different amount of time required for each task highlighted. It can be seen how each mix generated by a server is verified by every other peer, with the latter proceeding to the generation of a derived mix if the peer is next in the sequence.

Since each mix is appended to the previous one with correct proof, the longest chain of mixes represents the network consensus. This structure resembles the one used in the Blockchain and in the Bitcoin protocol, where the consensus is established as the longest chain of valid blocks of transactions [23].

Distributed Decryption mix: strict sequence When using the threshold encryption scheme proposed by Chaum seen in 2.2.4, the set of peer servers I has to be fixed as each peer in it has to compute its private key share depending on which peers are in I. When using a re-encryption MixNet this set doesn't affect the mix procedure as the decryption is performed at a later stage.

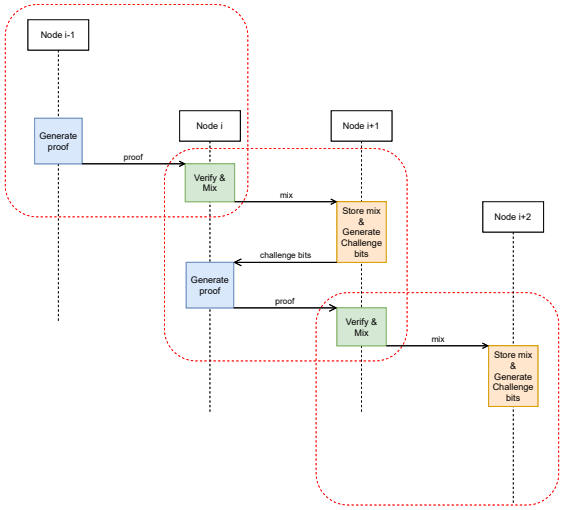

Figure 34: Peer i generating a valid mix and sending it to the next peer $i+1$. The red lines show the time span of three mix rounds.

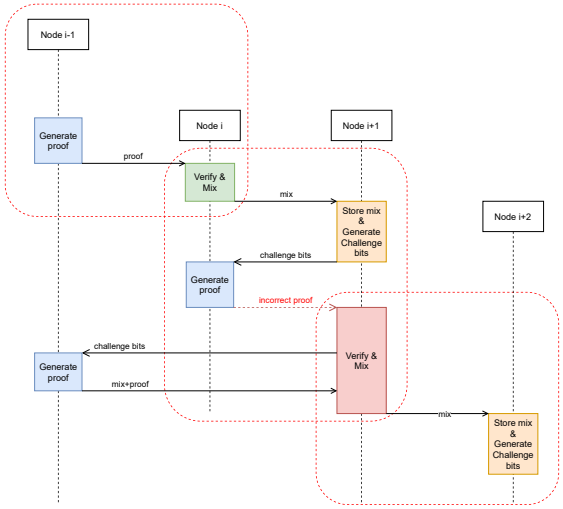

Figure 35: Peer i generating an invalid mix and sending it to the next peer $i+1$. The red lines show the time span of three mix rounds.

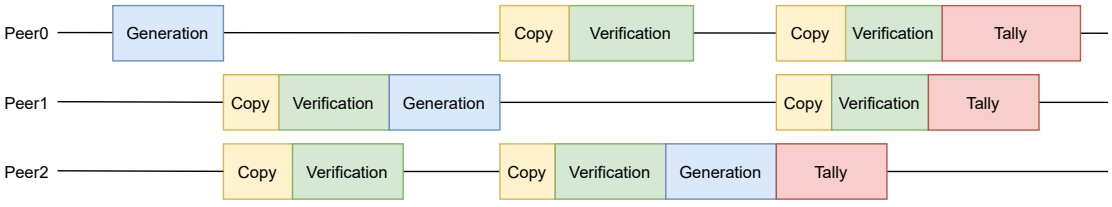

Figure 36: Time representation of the sequence of tasks performed by the three servers.

When working with a decryption MixNet instead, the set of mix nodes has to match the set I with an order that can either be equal or different than I according to the properties of the underlying cryptosystem. The reason for this strict sequence is that the decryption procedure is not *idempotent*.

Unlike the previous case thus, when a mix fails the mix procedure has to start over from a bulletin board server with a new set I constructed excluding the disqualified peers.

3.8.6 Voting though other peer's pages

Due to the proposed structure, each peer A is capable of providing the voting booth interface for any election it either created or received by B. A voter is thus free to vote for an election by visiting either A or B, which provide the same voting booth user interface. This amounts to submitting a vote to the same bulletin board through a different voting machine.

Cross origin requests When trying to submit a ballot from the user interface provided by server A to the server B acting as bulletin board, we have to deal with the *Cross-origin resource sharing* security mechanism implemented by browsers.

If a request to B is coming from the domain of server A, the browser makes a *CORS preflight* request to B with OPTION method and expects a Access-Control-Allow-Origin header to be returned indicating that A is allowed to make *cross origin* requests [22].

Kairos provides an endpoint for this preflight request that returns an Access-Control-Allow-Origin header that contains the lists of domains of the peer servers that act as trustees for the election.

3.8.7 Distributed bulletin board

Kairos makes use of a single bulletin board while providing multiple means of expanding this design in the future. The first peer server trustee is selected as bulletin board on creation ad will be responsible for storing received ballots.

This approach could be extended to the case in which multiple peer servers act as a separate bulletin boards. This second solution would be tolerant of failures of the server that created the election but would not prevent a malicious registrar from registering and authenticating illegitimate users.

Designing a distributed registrar structure would allow to relax several assumptions, up to the point of only trusting that t out of ℓ trustees are honest, a reasonable assumption heavily adopted in networks that require consensus such as cryptocurrencies built upon the Blockchain. This third option would be much more complex to implement, especially when the authentication of the voter makes us of external identity providers.

When the voter broadcasts its vote to all servers, an authentication scheme is required and since servers do not share memory, a way to share credentials would be needed.

Instead of distributing the whole list of all users and credentials to each server, we can make use of the JSON Web Token technology, having the authentication server sign its tokens with a RSA public key shared during peer handshake.

A peer server that receives a request to cast a vote has to verify the signature of the token, extract the id of the voter and register the vote.

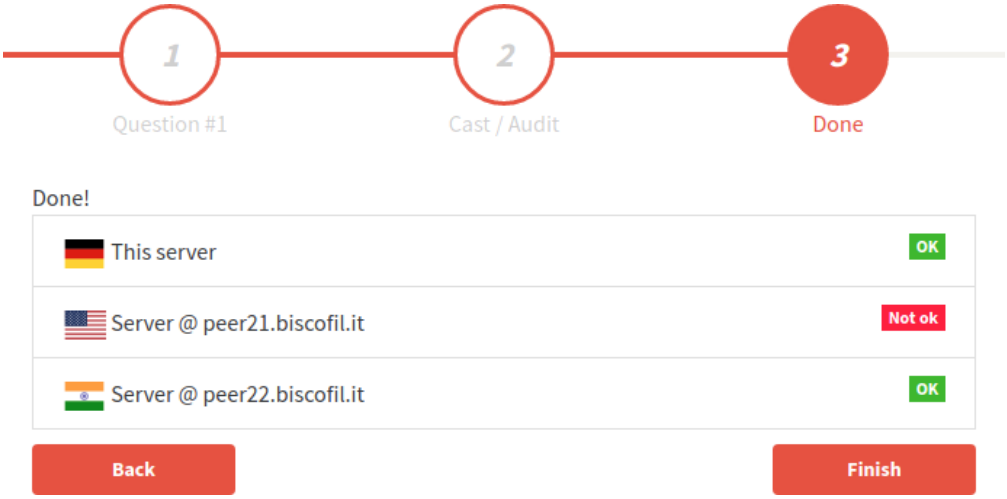

Figure 37: Election booth with multiple trustees acting as bulletin boards. The colored labels on the right indicate the outcome of the vote submission.

In case of an open election with a list of voters that changes over time, the traditional authentication flow with sessions would require a broadcast of the new voter record for each registration. By adopting JSON Web Tokens instead, we can postpone the check on voters until the voting phase ends, limiting the amount of traffic during the vote phase.

Even if all components required to post votes in multiple bulletin boards have already been implemented in KAIROS, the distributed algorithm has not been designed yet.

Since this scenario involves multiple peer servers acting as bulletin boards and thus having the list of votes at their disposal, multiple peers can initiate the mix process, preventing a single entity from blocking the anonymization step.

Vote broadcast and verification We can achieve fault tolerance on the vote submission by making each peer P_0 register a vote v not directly received by a voter if v was received and forwarded by at least t trustees $P_1, \ldots, P_t$. Peers could periodically exchange the list of votes they received. Even if still unused, KAIROS provides the data structure and the messages required for this feature already. Each vote recorded in the bulletin board of P also contains a list of trustees that have received and forwarded the vote to P_0. This list is represented by a *bitmask* where the i-th bit is 1 if the i-th trustees has received and forwarded the vote according to the trustee order define in 3.8.1 and is stored as a regular number.

Helios-C and Belenios proposed the signing of a ballot in order to prevent a malicious bulletin board from overwriting votes, approach that requires the registrar and the bulletin board not to be malicious simultaneously [6, 9]. By making multiple peers act as bulletin boards the signing step could be skipped and the assumption could be relaxes to the point of only requiring the registrar and less than t bulletin board peers not to be malicious simultaneously.

4 Experimental results

The following are numerical comparisons between Kairos and the most recent version of Helios with homomorphic encryption.

The time measurements of Helios have been taken by visiting the website of the platform and by temporarily overwriting the definition of specific JavaScript functions by means of the inspection tool of the browser inserting time measurement calls.

The measures of operations performed by the browser have been taken in Google Chrome v91 on a Ubuntu machine with 16Gb of Ram.

4.1 In-browser ballot seal

Figure 38 shows the comparison between the encryption time in milliseconds of a single plaintext in Helios and Kairos, with the first making use of custom BigInt implementations and the latter making use of the BigInt standard of modern browsers.

Whereas Helios's implementation appears to be significantly more consistent, the browser's builtin implementation appears to take less time in the average case.

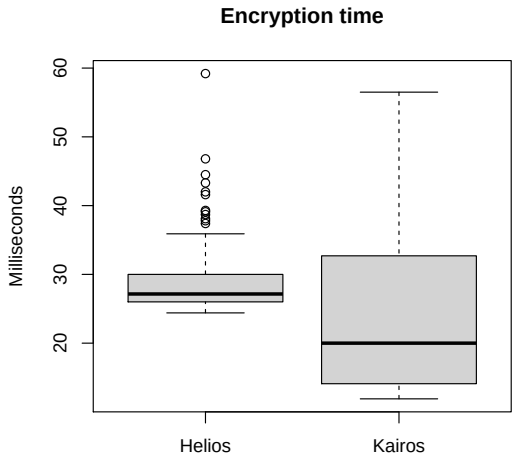

Figure 38: Encryption time comparison.

When the ballot is sealed, depending on the way the set of answers is converted into one or more ciphertexts, the time required can vary significantly. As mentioned in the sections before, Helios encrypts a plaintext for every answer whereas Kairos, similarly to Zeus, encrypts a single plaintext containing the whole ballot. This very significant difference is clearly noticeable when making use of the voting ballot interface.

Figure 39: Seal time comparison. The line represents a linear fit on the data points.

Figure 39 shows the total seal time for a ballot of an election with increasing number of answers. As expected, the seal time of a ballot in Kairos is constant, whereas Helios's scales linearly with the number n of possible answers as each one is represented by a dedicate binary plaintext and encrypted.

When adopting the homomorphic encryption technique, the voter has also to prove that the ciphertexts are valid since a malicious actor could submit the encryption of a very high number, giving himself a high voting power. In order to prevent this, the voting device in Helios has to generate a cryptographic proof indicating that each ciphertext represents either a zero or a one and, in general, all ciphertexts combined should represent a maximum value of one. The impact of these additional tasks is quite evident in the plot of figure 39.

4.2 Peer to Peer and distributed mix

Since the MixNet support was eventually dropped from Helios, the measures reported below have no external reference and are only compared between servers running the same code.

When moving from a centralized software to a distributed design, a significant loss in performances has to be expected. Dealing with a network requires to exchange data through messages, operation that can be time consuming, especially for communications that involve a big amount of data, such as sending mixes and proofs.

Figure 40 shows the comparison of the amount of time required by three servers to perform the distributed Re-Encryption-Decryption mix with an increasing number of ballots with 20 shadow mixes each. The three tasks require a time that scales linearly with the number of ballots. The total time, obtained by summing the time required by the three tasks, scales linearly too. This linear relation is coherent with the set of operations performed on the set of ballots.

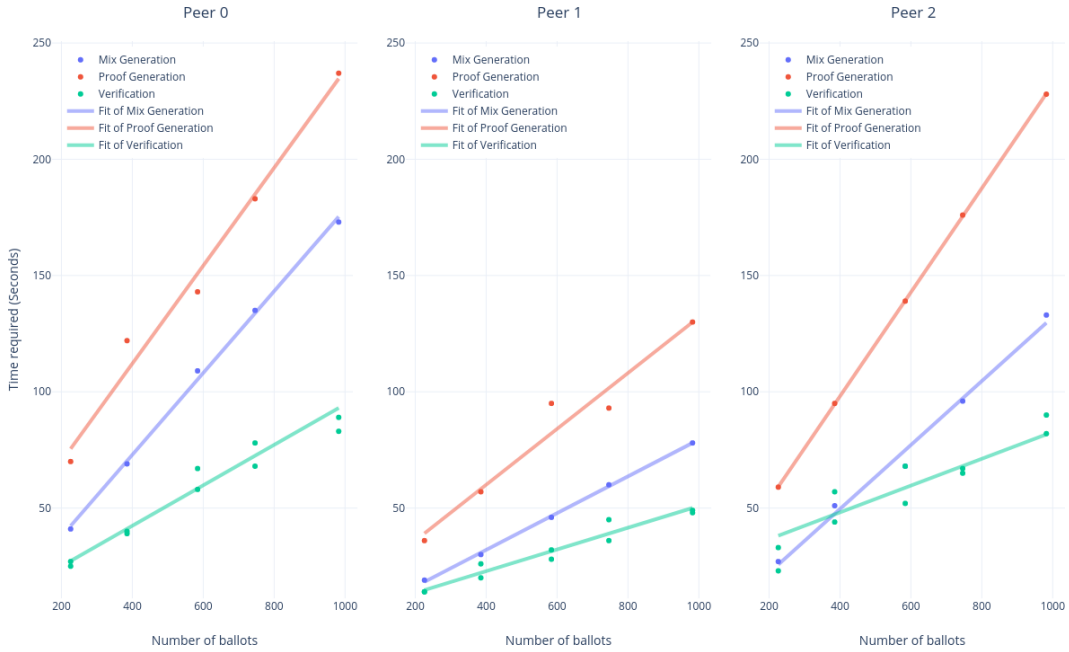

Figure 40: Time required by the three servers to execute the distributed mix procedure. Blue points indicate the time required to generate 20 shadow mixes, red points indicate the time required to generate a proof for every shadow mix and green points indicate the time required to verify each proof. The lines represent linear fits on the data points.

In figure 40 the second server (Peer1) shows a speed significantly higher than the two peers. This difference turned out to be motivated by a different hardware configuration across the three peers. Despite of identical prices for the three servers provided by the same company, by digging into the hardware details it appears that the specifications vary significantly. The configuration of each server was obtained with the LSCPU tool, and the summary is shown in figure 41 for reference.

	Peer0	Peer1	Peer2
Architecture	x86_64	x86_64	x86_64
CPU op-mode(s)	32-bit, 64-bit	32-bit, 64-bit	32-bit, 64-bit
Byte Order	Little Endian	Little Endian	Little Endian
Address sizes	40 bits physical, 48 bits virtual	40 bits physical, 48 bits virtual	40 bits physical, 48 bits virtual
CPU(s)	1	1	1
On-line CPU(s) list	0	0	0
Thread(s) per core	1	1	1
Core(s) per socket	1	1	1
Socket(s)	1	1	1
NUMA node(s)	1	1	1
Vendor ID	AuthenticAMD	AuthenticAMD	GenuineIntel
CPU family	23	23	6
Model	1	49	85
Model name	AMD EPYC 7601 32-Core Processor	AMD EPYC 7542 32-Core Processor	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
Stepping	2	0	4
CPU MHz	2199.96	2899.998	2399.998
BogoMIPS	4399.92	5799.99	4799.99
Hypervisor vendor	KVM	KVM	KVM
Virtualization type	full	full	full
L1d cache	64 KiB	64 KiB	32 KiB
L1i cache	64 KiB	64 KiB	32 KiB
L2 cache	512 KiB	512 KiB	4 MiB
L3 cache	16 MiB	16 MiB	16 MiB
NUMA node0 CPU(s)	0	0	0

Figure 41: Hardware configurations of the three rented servers obtained with the LSCPU tool. Each of the three virtualized servers has 2Gb of Ram allocated.

By dividing the obtained results by the number of shadow mixes and by the number of ballots for each mix we obtain a normalized value representing the time required for a mix generation and proof for a single ballot. The results are shown in figure 42, which highlights once again the different performances of the three servers.

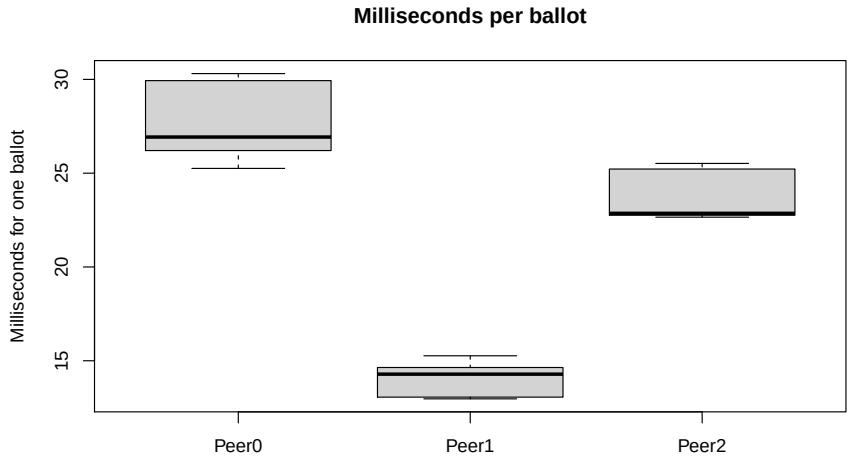

Figure 42: Normalized values for the time required for the mix of a single ballot.

By multiplying these values by the expected number of voters casting their ballots, by the desired number of shadow mixes and by the number of mixes, one could get an estimate of the time required by a peer server to execute the anonymization phase.

Summary The time required to seal a ballot, one of the biggest limitations of the current Helios design, shows a significant improvement in the Kairos implementation.

The relation between the time required by the distributed mix procedure and the size of the election looks reasonable.

Another interesting measure would be comparing the time required for a tally procedure that makes use of SQL functions and its PHP equivalent. The expected result would have the SQL implementation performing better up to a complexity point in which a traditional PHP script may become faster. In order to optimize the RAM usage, the records of the database could be fetched in batches with the *cursor* approach. Because of the relatively slow access to the database records, said complexity would have to be high enough to shadow the transfer time.

5 Conclusions

I personally consider KAIROS a very promising solution that, once fully implemented, could realistically benefit organizations such as Universities.

During the development of this project, one of the data center coordinator of our University shared a list of issues and limitations of the HELIOS voting system currently in use of our University. The properties offered by KAIROS ended up resolving all limitations mentioned in said list.

Developing KAIROS has been a massive challenge due to the size of the project which deals with many conceptual levels, from the most abstract ones to those more practical. Due to the high level design of KAIROS, which modular structure covers both high level features and low level aspects, many issues were encountered. Due to time constraints all these levels where implemented at a fast pace and would deserve a more in-depth study to account for edge cases and details not yet addressed.

By creating a framework elastic enough to support the addition of new question types and cryptosystems, we provide a platform than could realistically adapt to reasonable voting needs. This would allow institutions and developers to build new types of questions and features without creating a new voting system from scratch.

When comparing Homomorphic encryption and Mix Networks, the latter demonstrated to be the most elastic solution as it allows to adopt different types of questions and ballot encodings. This elasticity comes at the massive cost of performing the mix sequence in a possibly distributed environment, increasing the time required to release the election results.

The proposed Re-Encryption-Decryption MixNet design is a big security upgrade with respect to the traditional re-encryption MixNet due to the missing secret key exchange. The proposed MixNet design is the third design iteration, as the first two presented big security issues resolved along the way. The path throughout the design and proof of this proposal has been very challenging yet extremely rewarding when it turned out to work as expected.

ElGamal turned out to be the most elastic cryptosystem of those covered in this paper. The combination of the custom ballot encoding with database tally, the ElGamal cryptosystem and MixNets in particular, has shown to be the most elastic one, as the set of offered properties fulfills the whole range of covered needs.

The modules responsible for the communication between servers would definitively require a deeper analysis and would ideally be replaced by existing solutions that offer higher level interfaces. One viable option would be adapting this framework to make use of existing Blockchain solutions and smart contracts, task that would require a heavy reorganization of the project. Moving to smart contracts would also require to translate existing pieces of code to languages compatible with blockchain peers, segmenting the project into multiple parts.

When dealing with consensus and distributed algorithms, a popular option nowadays is to combine existing systems with a *proof of stake* (PoS) design that assigns a participating peer an amount of power proportional to the amount of a cryptocurrency money it owns. This concept is based on the idea that it would be economically costly for a peer to behave dishonestly.

The registrar / authentication entity remains a big bottleneck in all the designs mentioned so far. A viable option would be adopting the blockchain approach and making all peers cooperate at an earlier stage than the decryption phase. As an example, the registration of each voter would have to be translated into a dedicate transaction which should be registered and validated

by all peers.

The current design has all peers working on a unique set of voters. This structure could be easily extended from the current flat design to a hierarchical structure, where a big number of voters is segmented into groups of peers. This structure resembles the hierarchical management of traditional Country-level elections, where municipalities tally their sets of ballots and communicate the results to higher level entities such as districts, provinces and country which are responsible for the aggregation of said partial results. Because of the peer to peer capabilities which have already been implemented, this hierarchy can be achieved with very little modifications to the existing structure.

During the development phase, automatic tests with PHPUnit and coverage tests have been an essential tool to verify empirically the correctness of all the cryptographic operations. When dealing with modular operations and with very large prime numbers, the debugging phase has been extremely difficult. During the implementation of the $t - \ell$-threshold encryption scheme in particular, all operations were carried out with a proxy triplet (p, q, g) of very small integers in order to obtain results easier to validate manually.

The development of a distributed algorithm has definitely been a challenge, as the debugging and the testing phase were much more complex than the other features. Testing the distributed algorithm with a single software instance has been extremely problematic. The pairing of unique identifiers and integrity constraints of the database fields has played a critical role in this issue and resulted in the distributed algorithms to be tested manually copying the code to the three servers and inspecting the log files.

Summarizing, the design implemented in Kairos shows a promising set of features and techniques and would definitely benefit several organizations.

References

[1] Svetlana Z. Lowry, Poorvi L. Vora. Desirable Properties of Voting Systems. Available at: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=903961.

[2] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. Available at: https://dl.acm.org/doi/10.1145/358549.358563.

[3] Josh Benaloh. Simple Verifiable Elections. Available at: https://dl.acm.org/-doi/10.5555/1251003.1251008.

[4] Ben Adida. Helios: Web-based Open-Audit Voting. Available at https://www.usenix.org/legacy/event/sec08/tech/full_papers/adida/adida.pdf.

[5] Ben Adida, Olivier de Marneffe, Olivier Pereira, Jean-Jacques Quisquater. Electing a University President using Open-Audit Voting: Analysis of real-world use of Helios. Available at https://www.usenix.org/legacy/events/evtwote09/tech/full_papers/adida-helios.pdf.

[6] Véronique Cortier, David Galindo, Stéphane Glondu, Malika Izabachène. A generic construction for voting correctness at minimum cost - Application to Helios. Available at https://eprint.iacr.org/2013/177.pdf.

[7] Giorgios Tsoukalas, Kostas Papadimitriou, Panos Louridas, Panayiotis Tsanakas. From Helios to Zeus. Available at https://www.usenix.org/system/files/jets/issues/jets-0101-tsoukalas.pdf.

[8] Stéphane Glondu. Belenios specification - version 1.6. Available at http://www.belenios.org/specification.pdf

[9] Véronique Cortier, Pierrick Gaudry, Stéphane Glondu. Belenios: a simple private and verifiable electronic voting system. Available at https://hal.inria.fr/hal-02066930/document.

[10] Kazue Sako, Joe Kilian. Receipt-Free Mix-Type Voting Scheme. Available at https://link.springer.com/content/pdf/10.1007%2F3-540-49264-X_32.pdf.

[11] Pance Ribarski, Ljupcho Antovski. Mixnets: Implementation and Performance Evaluation of Decryption and Re-encryption Types. Available at https://ieeexplore.ieee.org/abstract/-document/6308057.

[12] Peter Y. A. Ryan, David Bismark, James Heather, Steve Schneider, Zhe Xia. The Prêt à Voter Verifiable Election System. Available at https://ieeexplore.ieee.org/document/5272310.

[13] Peter Y. A. Ryan, Steve Schneider. Prêt à Voter with re-encryption mixes. Available at https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.677.7734&rep=rep1&type=pdf.

[14] Standard built-in BigInt objects. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt

[15] PHP Secure Communications Library. Available at https://github.com/phpseclib/phpseclib

[16] Ronald Cramer, Rosario Gennaro, Berry Schoenmakers. A Secure and Optimally Efficient Multi-Authority Election Scheme. Available at https://www.win.tue.nl/~berry/papers/euro97.pdf.

[17] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, Tal Rabin. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. Available at https://link.springer.com/content/pdf/10.1007/s00145-006-0347-3.pdf.

[18] Mark A. Herschberg. Secure Electronic Voting Over the World Wide Web. Avaliable at https://dspace.mit.edu/handle/1721.1/43497.

[19] Atsushi Fujioka, Tatsuaki Okamoto, Kazuo Ohta. A practical secret voting scheme for large scale elections. Available at https://dl.acm.org/doi/10.5555/647091.713943.

[20] David Chaum. Blind signatures for untraceable payments. In 1982 Advances in Cryptology: Proceedings of CRYPTO '82. pp. 199-203.

[21] Technical Guidelines Development Committee. Voluntary Voting System Guidelines VVSG 2.0. Available at https://www.eac.gov/sites/default/files/TestingCertification/Voluntary_Voting_System_Guidelines_Version_2_0.pdf.

[22] Cross-Origin Resource Sharing (CORS). Available at https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[23] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Available at https://bitcoin.org/bitcoin.pdf, page 3.

[24] The WebSocket Protocol. Available at https://www.websocket.org/aboutwebsocket.html

[25] UTF-8, a transformation format of ISO 10646. Available at https://datatracker.ietf.org/doc/html/rfc3629, pp. 3-4.