Implementation of the Belenios Receipt-Free+ Online Voting Protocol

Thomas Clarke 1162927
Supervisor:David Galindo

MSci Computer Science

Department of Computer Science
University of Birmingham
September 12, 2016

Contents

List of Figures

Abstract

This report presents the implementation of an online voting protocol called BeleniosRF+, informing the reader of cryptographic protocols that enables the voting protocol to have a property called receipt freeness, among other important online voting properties. Among these cryptographic protocols is the use of asymmetric bilinear pairings that allows the randomisation of signature on ciphertexts while still allowing verification. To achieve this a library that enables the use of pairing friendly elliptic curves is used, resulting in election algorithms that can be used in online voting services.

1 Introduction

This aim of the work in this report is that of the implementation of the BeleniosRF+ online voting protocol, accompanied with the cryptographic methods that are used to design this online voting protocol and how it has evolved from previous protocols, whereby a property called receipt-freeness (RF) is a core focus. Accompanied with running time evaluations between the BeleniosRF+ and previous BeleniosRF protocol.

E-Voting or online voting, which can refer to either where specific voting machines are used in polling stations (which have been in main elections) or where voters can vote from their own device and anywhere, via the internet, and this version is which is addressed in this report. Online voting immediately has many advantages over government voting practices seen, for example, in the United Kingdom, where a voter still has to go to a physical polling station and use paper ballots. After a voter casts their vote they have no way of confirming their vote was counted or that the final result produced has not been tampered with, other than trusting that the rules and regulations set out have been followed at the polling stations. Online voting offers the voters and other parties confidence by allowing voters to verify their vote and allowing general parties to verify the results of the election, unlike paper ballots. The correct means of accomplishing this is an ongoing research field in computer security and cryptography, as there has to be assurances that peoples votes are kept private and that their votes remain as they were when the ballots were cast and, in most cases trust in specific parties is still needed as it is with paper ballots, but with online voting it allows the advantages mentioned. BeleniosRF+ is a protocol adapted from previously published papers, such as Helios [3] and Helios-C [27] that have been implemented into complete online voting services. As stated above the aims of this project was to implement the protocol which could be used in a fully implemented online voting service and not the implementation of the voting application itself. Through implementing this protocol a main problem in online voting will be visible which is where a protocol needs to ensure voter privacy, while allowing verification of the votes, a required property that can be in conflict.

To outline to the reader what they can expect from the rest of this report the following is brief description of the contents of each section. The background is where cryptographic protocols and primitive are presented, which are used, normally in conjunction, by the papers stated above to implement the protocols. The next section is Online Voting, where a deeper insight into online voting is provided via examples of previous protocols, and how the election protocols presented have been adapted from each other until the current BeleniosRF+ protocol, which has been implemented in this project. Next is the Implementation, where the implementation of the protocol is discussed, including the library used, software engineering practices followed and testing that was performed to ensure correct functionality. The results and evaluation section follows, presenting the results of the comparison for the running times of BeleniosRF+ with BeleniosRF, and discusses the reliability of the implementation, including its robustness. Project management gives a

2 Background

2.1 Elliptic Curves and Bilinear pairing

The following is brief introduction to elliptic curves and asymmetric bilinear groups with pairing friendly elliptic curve. Enough detail is given to understand their use in the report and follow along to the methods used in the implementation stage in section 4. We do not cover the mathematics involved as only a general understanding was needed for this project and it would be beyond the scope.

Discrete Logarithm Problem. A set F is a field if it has two binary operations $+$ and $\cdot$ where $(F, +)$ is a commutative group and given F^* a set that contains all the elements of F except the identity element for $(F, +)$, denoted as $F\backslash\{0\}$ as 0 is the identity element of $+$, then $(F^*, \cdot)$ is a commutative group, too. Lastly, the binary operator $+$ and $\cdot$ are both left and right distributive. A finite field is denoted as F_p. The multiplicative group $\mathbb{Z}_p^*$ is a finite set under the modulo operation and given p being prime all its elements are invertible modulo p. It is a cyclic group meaning there is a $g \in \mathbb{Z}_p^*$ that generates the whole set where $\mathbb{Z}_p^* = \{g^{p-1} \bmod = 1, g^1, g^2, \cdots, g^{p-2}\}$ and this g is called the generator.

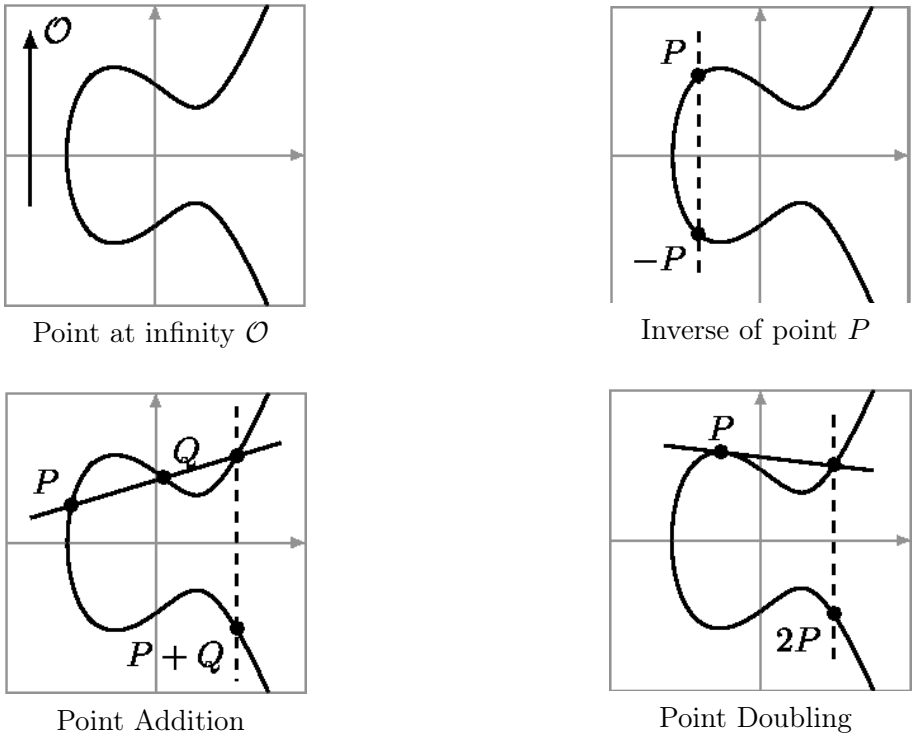
The discrete logarithm problem is given $g, h \in \mathbb{Z}_p^*$ where g is the generator and where $h = g^x \bmod p$ and $x \xleftarrow{\$} \mathbb{Z}_{p-1}$ find x, given g and h.

2.2 Elliptic Curves

Unlike many cryptographic schemes that use the hard problem of discrete log or factoring large numbers, elliptic curves introduce the hard problem of the elliptic curve discrete logarithm problem [20], though the hardness of the problem is associated with the underlining curves and groups used, just like the discrete logarithm problem. An elliptic curve is for all pairs $(x, y) \in \mathbb{Z}_p$ where p is a large prime then an elliptic curve is defined with $y^2 \equiv x^3 + ax + b \pmod p$, where $a, b \in \mathbb{Z}_p$ and $4a^3 + 27b^2 \not\equiv 0 \pmod p$, a property that's important for elliptic curves being used for cryptography. For elliptic curves to be used for the discrete logarithm problem they need to have a cyclic group like the multiplicative group mentioned above. For elliptic curves the set of elements are points on the curve (x, y) and the group binary operations are called point addition and point doubling, which comes under the operation of $+$. The group operations have an identity known as the point at infinity denoted as $\mathcal{O}$ where a point $P + \mathcal{O} = P$ All points on the curve have an inverse and for point $P = (x, y)$ its inverse is $-P = (x, -y)$. Point addition, denoted $+$, of two points if $P + Q = R$ where to find R you'd find the line through points P and Q and third point of intersection the curve $-R$ and R is the mirror of $-R$ along the x-axis. Point Doubling is doubling a point P where the tangent of P goes through a second intersection $-R$ point on the curve and its mirror is $R = 2P$. The group operations are closed, so for each P, Q on the curve $P + Q = R$ where R is always a point on the curve. They are associative meaning $(P + Q) + R) = P + (Q + R)$ and commutative so $P + Q = Q + P$. As the group additive operation satisfies all these laws it is a commutative group. Figure 1 shows examples of the group operations.

The mathematical equations to compute the group operations are done modulo a prime p, which is where the discrete logarithm is used for the points on the curve, where the elliptic curve E is defined over a finite field F_p giving $E(F_p)$. The elliptic curve discrete logarithm problem is defined as given a point P that generates all the points on the curve, point $Q = xP$, where $x \xleftarrow{\$} \mathbb{Z}_n$, where n is the order of the curve (number of points on it), find x given Q and P. For this to be practical the computation of xP has to be efficient, which is where point multiplication

is used. Point multiplication is adding a point to itself x times, for example, with xP above it would be $\underbrace{\{P + P + \cdots + P\}}_{\text{x times}}$. Using the point double and add algorithm, which is analogous to the square and multiply algorithm used for computing modular exponentiations, using x being 26 as an example then the algorithm would involve four point doubles and 2 point additions. Elliptic curves can also be used over smaller primes with, for example, 256 bits. This is because the attacks on the elliptic curve discrete logarithm problem are less efficient than the standard discrete logarithm problem, only being able to find the discrete logarithm in $\sqrt{2^p} = 2^{p/2}$



Point at infinity $\mathcal{O}$ | Inverse of point P

Point Addition | Point Doubling

Figure 1: Elliptic Curve Group Operations [2]

2.2.1 Bilinear Pairings

Biliniear pairings [7] are possible with specific pairing friendly elliptic curves. They are a relatively young property used in modern cryptographic protocols, but pairings were first published by Weil in 1940, known as Weil Parings. A pairing is be decribed as:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$$

and is a mapping, e, which takes a pairing over two groups $\mathbb{G}_1$ and $\mathbb{G}_2$, which are usually called the source groups and maps them into a group $\mathbb{G}_T$, known as the target group. Bilinear pairings have the following properties, where $\forall P \in \mathbb{G}_1$, $Q \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_n^*$

$$\begin{aligned} e(aP, bQ) &= e(P, Q)^{ab} = e(bP, aQ) \\ e(P + R, Q) &= e(P, Q) \cdot e(R, Q), where R \in \mathbb{G}_1 \\ e(P, R + Q) &= e(P, Q) \cdot e(P, R), where R \in \mathbb{G}_2 \\ e(g_1, g_2) &\neq 1 \end{aligned}$$

The first property is where the scalars, a and b, can be removed from the individual element and applied to the mapping result, or switched, both resulting in equality. The fourth property is that of non-degeneracy, whereby the opposite, degenerative, maps everything to 1. Bilinear

pairings can be symmetric and asymmetric. The mapping should also be computationally efficient algorithm. Symmetric bilinear pairs is where $\mathbb{G}_1 = \mathbb{G}_2$ and with elliptic curves they are restricted to using supersingular curves and is the mapping $e : E(\mathbb{F}_p) \times E(\mathbb{F}_p) \to \mathbb{F}^*_{p^a}$ where $E(\mathbb{F}_p)$ is an elliptic curve over the finite field $\mathbb{F}_p$, which are additive groups and $\mathbb{F}^*_{p^a}$ is a multiplicative finite field extension group, with embedding degree a. Asymmetric pairings is where $\mathbb{G}_1 \neq \mathbb{G}_2$ are over non-supersingular elliptic curves with the mapping $e : E(\mathbb{F}_p) \times E(\mathbb{F}_{p^a}) \to \mathbb{F}^*_{p^a}$. For any $P \in E(\mathbb{F}_p)$ which is a point on the curve ($E(\mathbb{F}_p)$ which is the over the ground field ($E(\mathbb{F}_p)$ and any $Q \in E(\mathbb{F}_{p^a})$ which is a point on another curve $E(\mathbb{F}_p)$ over the extension field. With the definition provided above you can construct equalities such as:

$$e(aP + bR, Q) = e(P + R, Q)^{ab} = e(P, aQ) \cdot e(R, bQ) \tag{1}$$

2.3 Security Assumptions

The following are security assumption used throughout this report and in the researched papers cited. They're not completely proven true, they are conjectures, but they're in the realm of Computer Science where disproving such assumptions would be of great significance, including, and probably most important, the finding of an efficient solution to the discrete logarithm problem.

Definition 1. *CDH The Computational Diffe Hellman assumption (CDH) [21] is given a cyclic group $\mathbb{Z}^*_p$, one of its generators g and the random elements $a, b \xleftarrow{\$} \mathbb{Z}_p$. Given the triple (g, g^a, g^b), it's infeasible to compute g^{ab}.*

CDH relies on the hardness of the discrete logarithm problem, as if it is possible to compute a from g^a efficiently then CDH would be easy as $g^a b$ would be $(g^b)^a$.

Definition 2 (DDH)**.** *The Decisional Diffe Hellman assumption (DDH) [6] is given a group $\mathbb{G}_q$, its generator g and three random elements $a, b, c \xleftarrow{\$} \mathbb{Z}_q$. Given two tuples (g, g^a, g^b, g^z) and (g, g^a, g^b, g^{ab}) it's infeasible to distinguish them.*

DDH relies on the hardness of CDH, as if $g^a b$ can be computed then it can be compared to g^z. DDH can be easy if specific groups are not used, for example, with specific pairing friendly curves it's easy to check whether $z = ab$ by comparing the pairing $e(P, zP)$ with (aP, bP).

Definition 3 (CDH$^+$)**.** *The Advanced computational Diffe Hellman Assumption (CDH$^+$) [5] given two cyclic groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order p, their generators g_1, g_2, respectively, the bilinear mapping $e : (g_1 \times g_2) \to \mathbb{G}_T$ and two random scalars $a, b \xleftarrow{\$} \mathbb{Z}_p$. Given the tuple $(g_1, g_2, g_1^a, g_2^a, g_1^a)$ it's computationally infeasible to compute g_1^{ab}.*

Definition 4 (SXDH)**.** *The Symmetric External Diffe Hellman Assumption (SXDH) [8] is given two cyclic groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order p, their generators g_1, g_2, respectively, the bilinear mapping $e : (g_1 \times g_2) \to \mathbb{G}_T$. It states that the DDH assumption holds in both $\mathbb{G}_1$ and $\mathbb{G}_2$*

Definition 5 (IND-CPA)**.** *Indistinguishable Under Chosen Plaintext Attack is where a challenger generates a keypair (pk, sk) and gives the adversary the public key pk. The attacker can encrypt any polynomial number of messages which the challenger will decrypt. Eventually the adversary submits two unique message m_0 and m_1. The challenger randomly chooses $b \xleftarrow{\$} \{0, 1\}$ and sends the adversary the encryption of c_b of m_b. A public encryption scheme is IND-CPA when the adversary can only guess whether c_b is the encryption of m_0 or m_1 with negligible probability.*

2.4 ElGamal

Taher ElGamal published a paper in 1985 [15] called *"A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms"*, which is now mainly known as the ElGamal encryption scheme. It used the work in *"New Directions In Cryptography"* [14], which is now known as the

Diffe-Hellman public key distribution system, to create a public key encryption and signature scheme. The hardness of its core security relies on the discrete logarithm problem. ElGamal requires a large prime p to form the multiplicative group $\mathbb{Z}_p^*$. $g \in \mathbb{Z}_p^*$ generates the subgroup $\mathbb{G}_q$ where q is a smaller prime that divides $p-1$ and ensures Elgamal is secure in terms of the DDH assumption. The key generation algorithm is where the generator g is raised to the power of a uniformly randomly chosen number $d \xleftarrow{\$} \mathbb{Z}_q$ giving $pk \equiv g^d \pmod{p}$. It is efficient to compute the public key pk, but computationally hard to compute the private key d from g and pk, it is computationally infeasible for a large enough prime. Elgamal encryption scheme has the following algorithms:

$$Enc(pk, m_1, r) : (c_1 \equiv g^r \pmod{p}, c_2 \equiv m \cdot pk^r \pmod{p} \tag{2}$$

$$Dec(x, (c_1, c_2)) : m \equiv c_2 \cdot c_1^{-d} \pmod{p} \tag{3}$$

The encryption algorithm, Enc produces ciphertext pair, (c_1, c_2), and it is therefore twice the size of the given plaintext message $m \in \mathbb{G}_q$. The r in 2 is the reason why Elgamal is probabilistic and therefore a message has many ciphertexts. The decryption algorithm 3 expanded results in m as expanded it's $m \equiv m \cdot (g^r)^d \cdot (g^r)^{-d} \pmod{p}$, where $-d$ is the multiplicative inverse in $\mathbb{Z}_p^*$. Simplifying further to $m \equiv m \cdot g^{rd+(-rd)} \pmod{p}$, it's easy to see how they cancel and leave the original plaintext message.

2.5 Homomorphic Encryption Scheme

Elgamal is a homomorphic encryption scheme [12], which is where computations on a ciphertext(s) will result in the same computations on the plaintext when decrypted. In the case of Elgamal, it means that if you have two ciphertext pairs $(c_{1,1}, c_{2,1}), (c_{2,1}, c_{2,2}) \in \mathbb{G}_q \times \mathbb{G}_q$ and their corresponding plaintexts $m_1, m_2 \in \mathbb{G}_q$, computations between $(c_{1,1}, c_{2,1})$ and $(c_{2,1}, c_{2,2})$ would produce the same computations between m_1 and m_2 after decryption.A more formal example of multiplicative homomorphism, using Enc as the encryption algorithm, is:

$$\begin{aligned} Enc(pk, m_1, r_1) \cdot Enc(m_2, r_2) &= (g^{r_1}, m_1 \cdot g^{xr_1})(g^{r_2}, m_2 \cdot g^{xr_2}) \\ &= (g^{r_1+r_2}, (m_1 \cdot m_2) \cdot g^{x(r_1+r_2)}) \\ &= Enc(pk, m_1 \cdot m_2, r_1 + r_2) \end{aligned}$$

Homomorphic encryption is useful for voting [12], as it allows computations to be carried out on the encrypted votes, such as tallying a final result without decrypting each vote. As stated above the homomorphism with the current Elgamal encryptions scheme is multiplicative and therefore the group operation on the message space is multiplication modulo p. This could not be used for tallying as a vote of 0 and vote of 1 would become 0. For tallying to function it must be able to add the votes and therefore additive homomorphism is needed. To achieve this, instead of m being an element of $\mathbb{G}_q$, it's an element in the message space $\mathbb{Z}_q$ and the group operation on the messages is addition modulo q. It replaces the message m with g^m in the Elgamal encryption algorithm.

$$\begin{aligned} Enc(pk, g^{m_1}, r_1) \cdot Enc(g^{m_2}, r_2) &= (g^{r_1}, m_1 \cdot g^{xr_1})(g^{r_2}, m_2 \cdot g^{xr_2}) \\ &= (g^{r_1+r_2}, (g^{m_1} \cdot g^{m_2}) \cdot g^{x(r_1+r_2)}) \\ &= (g^{r_1+r_2}, (g^{m_1+m_2}) \cdot g^{x(r_1+r_2)}) \\ &= Enc(pk, g^{m_1+m_2}, r_1 + r_2) \end{aligned} \tag{4}$$

As seen in 4, the decrypted result is the generator g to the power of the addition, but as it's an element of the subgroup $\mathbb{G}_q$ the discrete log has to found. This is okay for smaller numbers, as lookup tables can be previously computed or efficient algorithms can be used, for example Pollard's rho algorithm for logarithms, which has a running time complexity of $\mathcal{O}(\sqrt{n})$, where n would be the number of voters.

2.6 Signatures on Randomisable Ciphertexts

2.6.1 Randomisable Ciphertext

Ciphertext Randomisation is where specific encryption schemes can have the property of randomisation, which is where given a ciphertext a new ciphertext can be created without knowing the message, using the public key pk and randomness r. This requires in addition to the encryption and decryption algorithms, Enc and Dec, a randomisation algorithm $Rand$, where the following holds $Dec(x, c' = Rand(pk, c = Enc(pk, m, r), r')) = m$. Furthermore, it should be computationally infeasible to determine that c' is a result of applying $Rand$ to c.

Randomisable Elgamal. Elgamal has a randomisation algorithm:

$$Rand(pk, (c_1, c_2), r') : c' = (c_1 \cdot g^{r'}, c2 \cdot pk^{r'} \tag{5}$$

Using the algorithm in 5 a message can be encrypted into ciphetext c using randomness r and then c can be randomised into a fresh ciphertext c' using randomness r'. This can all be done by a party that does not need to know the secret key, x, and still enables decryption of the original message as $Dec(x, c') = m \cdot (g^x)^{r+r'} \cdot (g^{r+r'})^{-x}$.

2.6.2 Signature Randomisation

Signatures on randomisable ciphertexts is the taking of a ciphertext and randomising it into a fresh ciphertext (section 2.6.1) and creating a new fresh signature that's valid on that fresh ciphertext, which is still verifiable under the original signature verification key.

Asymmetric Waters signature Scheme Asymmetric Waters signature Scheme is a primitive by Blazy *et al.* [5], which is a variant of Waters' Signature Scheme [28]. It allows the creation of a signature σ on message m whereby the signature can be randomised to signature σ' and can still verified. Blazy *et al.* shows this scheme to be secure under the CDH$^+$ assumption. It is defined by 6 algorithms:

$\mathsf{Setup}(1^\lambda, 1^k)$**:** requires a pairing friendly asymmetric groups mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where g_1 and g_2 are generators of groups $\mathbb{G}_1$ and $\mathbb{G}_2$ which have prime order p. k is the length of the message to be signed, where the message is $m = (m_1, \ldots, m_k) \in \{0,1\}^k$. The message is hashed using *Waters Hash* $\mathcal{F}(m)$, which requires a vector $\boldsymbol{u}$ of length $k+1$, $\boldsymbol{u} = (u_0, \ldots, u_k) \xleftarrow{\$} \mathbb{G}_1^{k+1}$. The *Waters Hash* is defined as:

$$\mathcal{F}(m) := u_0 \prod_{i=1}^{k} u_i^{m_i}.$$

Lastly, $z \xleftarrow{\$} \mathbb{G}_1$ to give the final returned parameters $pp := (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, z, \boldsymbol{u})$

$\mathsf{SKeyGen}(pp)$: where the public verification key vk and private signing key sk are generated using parameters pp. Using a $x \xleftarrow{\$} \mathbb{G}_1$, $X_1 = g_1^x$ and $X_2 = g_2^x$ to define the verification keys and $Y = z^x$ to define the signing key. Returning $vk = (pp, X_1, X_2)$ and $sk = (pp, Y)$

$\mathsf{Sign}(pp, sk = Y, m; s)$**:** Signs message m using signing key sk and a random $s \xleftarrow{\$} \mathbb{Z}_p$. Returns signature σ.

$$\sigma = (\sigma_1 = Y \cdot \mathcal{F}(m)^s, \sigma_2 = g_1^s, \sigma_3 = g_2^s)$$

$\mathsf{Verif}(vk = (pp, X_1, X_2), m, \sigma)$: Uses the mapping e to compare pairings and returns a 1 if all comparisons are equal ro 0 otherwise:

$$e(\sigma_1, g_2) = e(z, X_2) \cdot e(\mathcal{F}(m), \sigma_3) \qquad e(\sigma_2, g_2) = e(g_1, \sigma_3)$$

$\mathsf{Random}((pp, X_1, X_2), F, \sigma; s')$**:** Takes a hashed message F and the signature on it σ and returns a randomised signature σ' still verifiable with $vk = (X_1, X_2)$

$$\sigma' = (\sigma_1 \cdot F^{s'}, \sigma_2 \cdot g_1^{s'}, \sigma_3 \cdot g_2^{s'})$$

This scheme is randomisable as $\mathsf{Random}((pp, X_1, X_2), F, \sigma; s') = \mathsf{Sign}(sk = (pp, Y), m; s + s')$ and this the verification can be shown for first equality check in Verif:

$$\begin{aligned} e(\sigma_1', g_2) &= e(z, X_2) \cdot e(\mathcal{F}(m), \sigma_3') \\ = e(\sigma_1 \cdot F^{s'}, g_2) &= e(z, g_2^x) \cdot e(\mathcal{F}(m), \sigma_3 \cdot g_2^{s'}) \\ = e(Y \cdot \mathcal{F}(m)^s \cdot F^{s'}, g_2) &= e(z, g_2^x) \cdot e(\mathcal{F}(m), g_2^s \cdot g_2^{s'}) \\ = e(z^x \cdot \mathcal{F}(m)^{s+s'}, g_2) &= e(z, g_2^x) \cdot e(\mathcal{F}(m), g_2^{s+s'}) \\ = e(z \cdot \mathcal{F}(m), g_2)^{x \cdot (s+s')} &= e(z, g_2)^x \cdot e(\mathcal{F}(m), g_2)^{(s+s')} \end{aligned}$$

This shows how the hash F can still be verified after another random s' is applied and its equality is possible due to the example of the properties showed in 1 and for the second equality check in Verif (albeit skipping some expansions):

$$\begin{aligned} e(\sigma_2', g_2) &= e(g_1, \sigma_3') \\ = e(g_1^s \cdot g_1^{s'}, g_2) &= e(g_1, g_2^s \cdot g_2^{s'}) \\ = e(g_1 \cdot g_1, g_2)^{s+s'} &= e(g_1, g_2 \cdot g_2)^{s+s'} \end{aligned}$$

Our implementation of verify is displayed in code snippet 6.

2.7 Signatures on Randomisable Ciphertexts

Instead of being able to verify a message after creating a fresh signature, it's also possible to create a fresh signature on a ciphertext that has been randomised, that again, is still verified under the verification key vk.

Asymmetric Waters Signature with Elgamal Encryptions. Blazy *et al.* also shows how the Asymmetric Waters Scheme can be used with Elgamal ciphertexts, while still upholding the Elgamal randomisation property, by creating a fresh signature upon it. First, Elgamal has to be redefined [5,17], and instead of being defined with the subgroup $\mathbb{G}_q$, it has to use pairing friendly groups and the parameters $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$. It is still IND-CPA under the hardness of the DDH assumption, but now with group $\mathbb{G}_1$, as briefly discussed in section 2.3 the DDH assumption holds when specific elliptic curves are used. The Elgamal ciphertext will be the encryption of $\mathcal{F}(m)$ where $m = (m_1, \ldots, m_k) \in \{0,1\}^k$ rather than an element in $\mathbb{G}_1$. The algorithms added and altered are as follows:

$\mathsf{Setup}(1^\lambda, 1^k)$**:** As Setup defined in 2.6.2, returning parameters pp

$\mathsf{EKeyGen}_{\mathcal{E}}((p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2))$**:** $d \xleftarrow{\$} \mathbb{Z}_p$. $P = g_1^d$. Return $(pk = P, dk = d)$

$\mathsf{SKeyGen}(pp)$**:** As $\mathsf{SKeyGen}$ defined in 2.6.2, returning (vk, sk)

$\mathsf{Encrypt}(pp, pk, vk, m; r)$**:** $m \in \{0,1\}^k$. $r \xleftarrow{\$} \mathbb{Z}_p$ Returns ciphertext c

$$c = (c_1 = g_1^r, c_2 = pk^r \cdot \mathcal{F}(m))$$

$\mathsf{Sign}(pp, sk = Y, pk = P), c = (c_1, c_2); s)$**:** Signs message c using signing key sk and a random $s \xleftarrow{\$} \mathbb{Z}_p$. Returns signature σ.

$$\sigma = (\sigma_1 = c_1^s, \sigma_2 = Y \cdot c_2^s, \sigma_3 = g_1^s, \sigma_4 = g_2^s, \sigma_5 = P^s)$$

$\mathsf{Random}((pp, vk = (X_1, X_2), pk = P), c, \sigma; r', s')$**:** $r', s' \xleftarrow{\$} \mathbb{Z}_p$. Returns $c' = (c_1', c_2')$ and $\sigma' = (\sigma_1', \sigma_2', \sigma_3', \sigma_4', \sigma_5')$

$$c_1' = c_1 \cdot g_1^{r'} \qquad c_2' = c_2 \cdot P^{r'}$$
$$\sigma_1' = \sigma_1 \cdot c_1^{s'} \cdot \sigma_3^{r'} \cdot g_1^{r' \cdot s'} \qquad \sigma_2' = \sigma_2 \cdot c_2^{s'} \cdot \sigma_5^{r'} \cdot P^{r' \cdot s'}$$
$$\sigma_3' = \sigma_3 \cdot g_1^{s'} \qquad \sigma_4' = \sigma_4 \cdot g_2^{s'} \qquad \sigma_5' = \sigma_5 \cdot P^{s'}$$

$\mathsf{Verif}(pp, vk = (X_1, X_2), pk = P, c, \sigma)$**:** Return 1 iff the following holds:

$$e(\sigma_1, g_2) = e(c_1, \sigma_4) \qquad e(\sigma_2, g_2) = e(z, X_2) \cdot e(c_2, \sigma_4)$$
$$e(\sigma_3, g_2) = e(g_1, \sigma_4) \qquad e(\sigma_5, g_2) = e(P, \sigma_4)$$

It is the case that $\mathsf{Random}(pp, vk = (X_1, X_2), pk = P, c, \sigma; r', s') = \mathsf{Sign}(pp, sk = Y, pk = P, c; s + s')$ where $\mathsf{Encrypt}(pp, pk = P, vk, m; r + r')$ [5] and using the verification pairings equality $e(\sigma_2, g_2) = e(z, X_2) \cdot e(c_2, \sigma_4)$ as an example, we can show how the signature on the Elgamal ciphertext randomisation can be verified

$$\begin{aligned}
e(\sigma_2{}', g_2) &= e(z, X_2) \cdot e(c_2', \sigma_4{}') \\
= e(\sigma_2 \cdot c_2^{s'} \cdot \sigma_5^{r'} \cdot P^{r' \cdot s'}, g_2) &= e(z, g_2^x) \cdot e(c_2 \cdot P^{r'}, \sigma_4 \cdot g_2^{s'}) \\
= e(Y \cdot c_2^s \cdot (\mathcal{F}(m) \cdot P^r)^{s'} \cdot (P^s)^{r'} \cdot P^{r' \cdot s'}, g_2) &= e(z, g_2^x) \cdot e(\mathcal{F}(m) \cdot P^r \cdot P^{r'}, g_2^s \cdot g_2^{s'}) \\
= e(z^x \cdot \mathcal{F}(m)^{s+s'} \cdot P^{(r \cdot s)+(r \cdot s')+(s \cdot r')+(r' \cdot s')}, g_2) &= e(z, g_2^x) \cdot e(\mathcal{F}(m) \cdot P^{r+r'}, g_2^{s+s'}) \\
= e(z^x \cdot \mathcal{F}(m)^{s+s'} \cdot P^{(r+r')(s+s')}, g_2) &= e(z, g_2^x) \cdot e(\mathcal{F}(m) \cdot P^{(r+r')}, g_2)^{(s+s')} \\
= e(z^x \cdot (\mathcal{F}(m)^{s+s'} \cdot P^{(r+r')(s+s')}), g_2) &= e(z, g_2^x) \cdot e((\mathcal{F}(m) \cdot P^{(r+r')})^{(s+s')}, g_2) \\
= e(z^x \cdot (\mathcal{F}(m)^{s+s'} \cdot P^{(r+r')(s+s')}), g_2) &= e(z, g_2^x) \cdot e((\mathcal{F}(m)^{(s+s')} \cdot P^{(r+r')(s+s')}), g_2)
\end{aligned}$$

and finally, using the property of $e(P \cdot R, Q) = e(P, Q) \cdot e(R, Q), where R \in \mathbb{G}_1$, we can see how these pairings would be equal and verified (for this specific check).

2.8 Zero Knowledge Proofs

A Interactive Zero Knowledge proof system allows a party, called the prover, to prove to another party, the verifier, that a given statement is true, under some secret. This secret is not revealed to the verifier and therefore zero knowledge of the secret is gathered through this interaction. These proofs are useful to online voting as they enable a party to provide a proof that they decrypted votes correctly and another party can then verify this. A zero knowledge proof should have three properties; completeness, soundness and zero-knowledge. **Completeness** is whereby an honest prover will be able to verify that their statement is true, given their secret, to an honest verifier with high probability. **Soundness** is whereby an adversary (cheating prover) without knowledge of the secret can not convince the verifier that their proof is true, other than a negligible probability. **Zero-knowledge** is where the verifier does not obtain any information

regarding the secret from the honest prover, other than that the prover's statement is true.

They are a set of sigma protocols, which, using definitions in [4, 10], are a three step protocol between a prover and a verifier and is defined as $\Sigma = (\mathsf{Prove}_\Sigma, \mathsf{Verify}_\Sigma)$ where Prove_Σ is an algorithm taking inputs w and Y, where w is known as the witness (e.g. a secret key) and $Y \in \{0,1\}^*$ is the statement, used by the prover to prove statement Y and Verify_Σ is another algorithm used by a verifier, taking inputs Y, com, ch and res, to verify if the statement. In the case of IZK proofs these algorithms would be interactive and the three steps of the protocol are the prover initialises by sending a commitment com, which the verifier receives and responds with a uniformly random challenge ch from a challenge set (e.g. this could be the group $\mathbb{Z}_p$) and, lastly, the prover sends their response res whereby the verifier then, using Verify_Σ, accept or rejects the proof (generated from this interaction) for the statement Y.

2.8.1 Schnorr's Identification Protocol

Schnorr's Indentification Protocol [26] underlines many zero-knowledge proofs, as it's a protocol which enables a prover to prove they have knowledge of a secret key x, proof of a discrete logarithm $x = \log_g(g^x)$

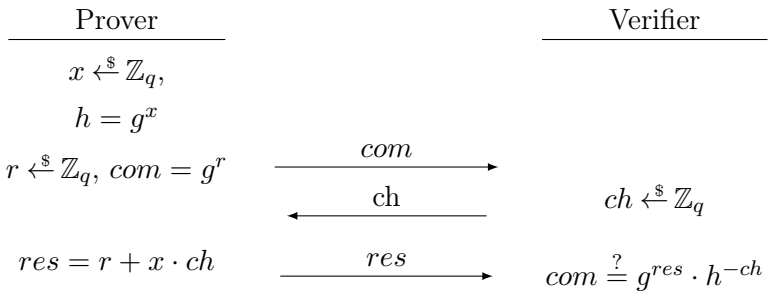


Figure 2: Schnorr Identification Protocol [26]

Three step protocol..A cyclic group G_q is used with generator g. Firstly the prover randomly selects r in $\mathbb{Z}_q$ and sends the commitment g^r (mod q) to the verifier. r is used to ensure each protocol does not reveal any information regarding the secret, x, via challenges sent from the veriifier. Secondly, the verifier sends their challenge, c, a random element in $\mathbb{Z}_q$, which is to ensure that the prover really has knowledge of the secret and is not replaying. Thirdly, the prover sends $s = r + xc \pmod q$. The verifier is able to verify that $a = g^s \cdot h^{-c}$, as $g^s \cdot h^{-c} = g^{r+xc} \cdot g^{-xc} = a = g^r$ and it proves knowledge of discrete log x as both are using same base generator g.

The Schnorr identification of has a property of special soundness...

2.8.2 Chaum Pedersen Protocols

The Chaum-Pedersen protocol uses the underling properties of the Schnorr protocol.

Logarithm Equality. The Chaum-Pedersen discrete logarithm equality sigma protocol enables a prover to prove that two elements $g, h \in \mathbb{G}_q$ both are using the same discrete logarithm x, giving the language $\mathcal{L}_{EqDl} = (g, h, y_1, y_2) | \log_g y_1 = \log_h y_2$ [10]. As stated above this is useful for online elections as the party that tallies the result can prove that the decryption of the encrypted done using the private key, which is the discrete logarithm of the public key that was used by voters to encrypt their votes. Parties can verify this proof under the assurance that the a correct result (given the proof of decryption) has been computed, all without leaking information regarding the private key.

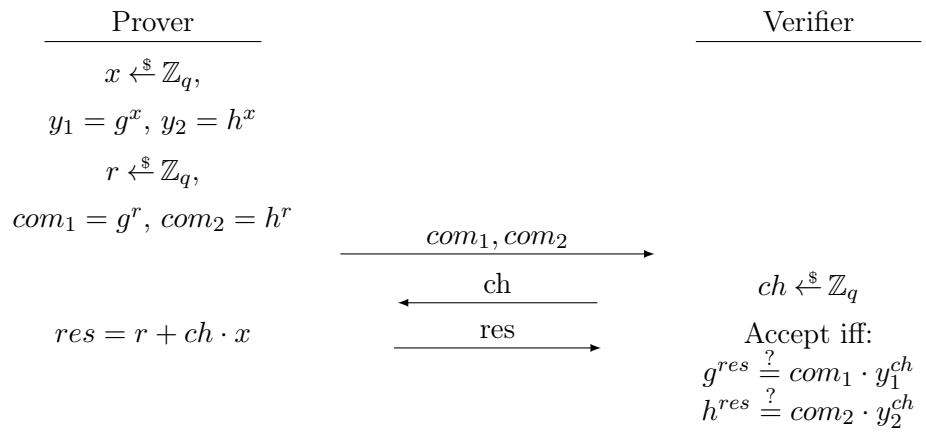


Figure 3: Chaum Pedersen Logarithm Equality [9]

The verifier will accept, given an honest prover, as $g^{res} = g^{(r+ch \cdot x)} = com_1 \cdot y_1^{ch} = g^r \cdot g^{x \cdot ch} = g^{r+x \cdot ch)} \pmod{p}$ and the same for the second equality check, but for the case of h.

2.8.3 Non-Interactive Zero-Knowledge Proofs

A disadvantage of the protocols above is that they are interactive and in many applications, especially with online voting, these kind of interactive proofs would be problematic. For instance, the Chaum-Pedersen proof of discrete logarithm equality can be used for proof of decryptions, but if this was applied this to a online voting service the server would have to constantly process this protocol with a party that wanted to verify the result, which would be inefficient and could be a doorway for Denial of Service Attacks. This introduces Non-Interactive Zero Knowledge proofs (NIZK) and usefulness in such systems.

Fiat-Shamir Transformation The Fiat-Shamir transformation [4, 16] is whereby an interactive zero knowledge proof is transformed into Non-Interactive one by using a collision resistant hash function. In addition to $\Sigma = (\mathsf{Prove}_\Sigma(w, Y), \mathsf{Verify}_\Sigma(Y, com, ch, res)$, there is $H : \{0,1\}^* \to Ch$ which is a hash function that takes an arbitrary string of bits and outputs an element in the challenge set for Σ. The algorithms are now defined as $\mathsf{FS}_\Sigma = (\mathsf{Prove}(w, Y), \mathsf{Verify}(Y, ch, res)$ where Prove is composed of $com \leftarrow \mathsf{Verify}_\Sigma(Y, com, ch, res)$ and then $ch \leftarrow H(Y, com)$, followed by the second stage of Prove_Σ with the challenge hash ch to calculate the response res. Returns the pair (ch, res). Verify computes com from (Y, ch, res) and then runs $\mathsf{Verify}_\Sigma(Y, com, ch, res)$ accepting or rejecting Y.

Using the Fiat-Shamir transformation, the Chaum-Pedersen Σ protocol for proving the equality of discrete logarithm, described above, can be transformed into a NIZK. As with IZK protocol requires a cyclic group $\mathbb{G}_q$ and $g, h \in \mathbb{G}_q$, secret key x, $y_1 = g^x$ and $y_2 = h^x$. The prover generates a uniformly random $r \in \mathbb{Z}_p$ and creates the two commitments $com_1 = g^r$ and $com_2 = h^r$ and as there is now no interaction (other than the prover sending the statement and proof in a single pass), the prover generates the challenge $ch \leftarrow H(g, h, y_1, y_2, com_1, com_2)$, calculates the $res = r + x \cdot ch$ and outputs (ch, res). The verifier instead of checking that $g^{res} \stackrel{?}{=} com_1 \cdot y_1^{ch}$ and $h^{res} \stackrel{?}{=} com_2 \cdot y_2^{ch}$, computes the commitments with $com_1 = g^{res}/y_1^{ch}$ and $com_2 = h^{res}/y_2^{ch}$ and then checks $ch \stackrel{?}{=} H(g, h, y_1, y_2, com_1, com_2)$. Even though the prover now handles the challenge, the protocol is still complete and soundness as the challenge is a collision resistant hash over the statement Y and com, which are (g, h, y_1, y_2) and the (com_1, com_2), respectively. Therefore the prover can not be malicious in terms of the statement they're proving and one they hash, as the verifier will hash the statement and the recomputed commitments.

NIZK Disjunctive Chaum-Pedersen The Chaum-Pedersen protocol for proving discrete logarithms, as described above, can be used so that a prover can prove that they have encrypted a 0 or a 1, without revealing which one was encrypted and is known as Disjunctive Chaum-Pedersen. Using [4, 10], it's defined as $\mathsf{DisjProof}_F(g, pk, R, S) = (\mathsf{DisjProve}_F, \mathsf{DisjVerify}_F)$ and is an NIZK proof of whether an Elgamal ciphertext $c = (R = g^r, S = pk^r \cdot g^m)$ is an encryption of $m = 0$ or $m = 1$, where F is a collision resistant hash function. It uses [11] and the discrete logarithm equality proof language $\mathcal{L}_{EqDl}$, showing that $(g, pk, R, S) \in \mathcal{L}_{EqDl}$ or $(g, pk, R, S \cdot g^{-1}) \in \mathcal{L}_{EqDl}$. $\mathsf{DisjProve}_F(g, pk, R, S, r)$ take the generator g, public key pk, ciphertext R and S and the randomness r used in the encryption and returns the proof $\pi \leftarrow (ch_0, ch_1, rep_0, rep_1)$. For the case of $m = 1$, the prover fakes a proof $g, pk, R, S) \in \mathcal{L}_{EqDl}$ by randomly choosing $(ch_0, rep_0) \xleftarrow{\$} \mathbb{Z}_p \times \mathbb{Z}_p$ and setting $U_0 = g^{rep_0}/R^{ch_0}$ and $V_0 = pk^{rep_0}/S^{ch_0}$. It then sets the real proof, by randomly choosing $u_1 \xleftarrow{\$} \mathbb{Z}q$ and setting $U_1 = g^{u_1}$ and $V_1 = pk^{u_1}$. It then computes the challenge $ch \leftarrow F(g, pk, R, S, U_0, V_0, U_1, V_1)$. The disjunction is then $ch_1 = ch - ch_0$ and $rep_1 = u_1 + ch_1 \cdot r$ and π is outputted as defined above. $\mathsf{DisjVerify}_F(g, pk, R, S, \pi)$ outputs accept or reject depending on $ch_0 + ch_1 \stackrel{?}{=} F(g, \mathrm{pk}, R, S, \frac{g^{res_0}}{R^{ch_0}}, \frac{\mathrm{pk}^{res_0}}{S^{ch_0}}, \frac{g^{res_1}}{R^{ch_1}}, \frac{\mathrm{pk}^{res_1}}{(S \cdot g^{-1})^{ch_1}})$.

3 Online Voting

This section gives a deeper insight into online voting, the current and past application/protocols accompanied with the underlining methods that provides a given functionality and security. It also introduces problems in online voting and the solutions to these. Lastly, it details the BeleniosRF+, which is the protocol that was implemented and is presented in section 4.

To begin, online voting should satisfy two main properties, privacy and verifiability. Privacy is composed of three sub properties:

Ballot Privacy: Is that no one should be able to learn how a voter voted.

Receipt-Freeness: Is that a voter should not be able to prove how they voted, which prevents vote selling and helps limit coercion.

Coercion Resistant: If a voter is under temporary coercion from an adversary, they should be able to cast their desired vote after the coersion has concluded.

Verifiability is composed of three sub properties:

Individual Verifiabiity: Which is that an individual voter should be able to confirm that their ballot is in the published public ballot-box

Universal Verifiability: Which is that anyone can verify the election result corresponds to ballots in the public ballot-box.

Eligible Verification Which is that only a voter who is eligible to vote in an election should be able to.

The problem with online-voting is that privacy and verifiability are in opposition to each other, an online voting system should ensure a voters' vote stays private, but a voter should be able to verify their vote has been counted. For example, receipt-freeness states that a voter should not be able to prove how they voted, but individual verifiability states that a voter should be able to verify how they voted. A question to answer is, how can a voter verify how they voted without being able to view their specific vote. The solution to this and other problems is presented by looking through the evolution from Helios to BeleniosRF+.

3.1 Election Syntax

A voting service is built up of individual parties each with their own specific tasks and a syntax was used in [27] to outline this:

Election Administrator: The Election Administrator, denoted by ε, is responsible for setting up elections. It publishes the IDs for voters who are allowed to vote in the election, the candidates of the election, and the result function for the election. The result function is how the results of the election are calculated and it is usually by adding up the votes for each candidate.

Registrar: The registrar, denoted by $\mathcal{R}$, generates secret credentials for each voter and distributes them, while registering the public credential to the corresponding voter.

Trustee: The trustee, denoted by $\mathcal{T}$, tallies the final result of the election and publishes it to the Ballot-Box

Ballot Box Manager: The Ballot Box manager, denoted by $\mathcal{B}$, is the voting server. It is responsible for storing valid ballots in the Ballot-Box, **BB**, after any additional processing. It will publish the Public Bulletin Board, **PBB**, which is the public version of **BB** with sensitive information omitted, for anyone to view.

Voters: The eligible voters, denoted by $id_1, ..., id_\tau$, send their ballot to the bulletin board.

3.2 Helios to Belenios+

Helios. Helios [3] is a web based online election introduced in 2008 and improvements have been made since its inception when research showed weakness in the protocol; the latest updated being 4.0 in 2012. It allows users to create elections or referendums, identified by a unique URL id, upload a *.csv* file of all the eligible voters, each are required to have a unique id and email address. The Helios server will send an email to each voter providing them with a password to be used for their authentication to vote vote. In Helios 1.0 the server is the only authority, acting as the bulletin board and trustee and the Helios paper stated "Trust no one for integrity, trust Helios for privacy" ([3]), which is implying that for a voter to trust their vote will remain private they have to trust Helios itself and in terms of the election syntax (section 3.1 this is the Ballot Box Manager. There is individual verifiability as a voter can verify their vote in the public bulletin board; this is achieved by comparing the hash the voter received when encrypting their ballot, with the hash of the ciphertext in the public bulletin board, which gives the voter a receipt for their election and not upholding the Receipt-Freeness property of election privacy. As all the trust is put in Helios (voting server) it can possibly stuff ballots, which is the act of inserting ineligible ballots into the ballot box, and remain undetected, therefore manipulating the result.

Helios generates a new Elgamal keypair for each election that is created by the Election Administrator. The public key is used by voters to encrypt their votes, which are accompanied with NIZK Disjunctive Chaum-Pedersen proofs (section 2.8.3) that the voter has encrypted $m = 0$ or $m = 1$. When the voting time period ends, the ballots can be tallied. Helios approaches this by first using a Sako-Kilian mixnet [25] to anonymise the votes, which is the process of unlinking a voter from their associated ciphertext. This is defined in Helios, and other systems, as shuffling. A mixnet is a permutation and reencryption of the ciphertexts and therefore depends on encryption scheme being homomorphic, in this case Elgamal re-encryption (section 2.6.1) is used. Figure 4 displays a simple example of a mixnet, where A - B represent four ciphertexts being inputted and with each rectangular block the ciphertexts are being reencrypted with a new r and then randomly permuted.

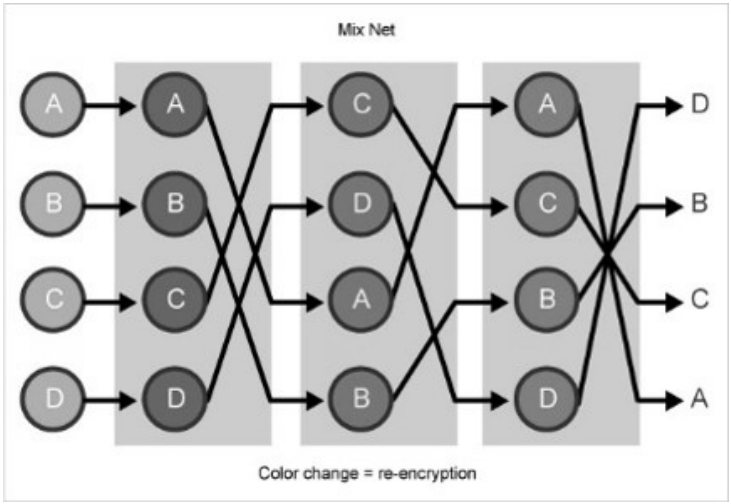


Figure 4: Mixnet Anonymization Example [23]

After the ciphertexts are outputted after shuffling a proof of correctness is created for the integrity of the shuffle, for Helios this is a NIZK Fiat-Shamir proof. Helios allows auditors to

verify this proof and after a set time the shuffled ciphertexts are individually decrypted and NIZK Chaum-Pedersen Discrete Logarithm proofs (section 2.8.3) of decryption are computed for each decrypted vote. An example is as follows: The protocol is proving that $\log_g y_1 = \log_h y_2$ and for the Elgamal ciphertext $(R = g^r, S = pk^r \cdot g^m)$ of message m, the discrete logarithm parameter correspond as $g = g$, $h = R = g^r$, $y_1 = pk = g^x$ and $y_2 = S \cdot g^{-m} = pk^r \cdot g^m \cdot g^{-m}$. Using the protocol defined in section 2.8.3 the prover outputs the pair $(ch = H(g, R, pk, S \cdot g^- m, com_1 = g^{r'}, com_2 = R^{r'}), res = r' + x \cdot ch)$. This proof of decryption is verified as $ch = H(g, R, pk, S \cdot g^{-m}, com_1 = g^{r'+x \cdot ch}/pk^{ch} = g^{r'+x \cdot ch}/g^{x \cdot ch} = g^{r'}, com_2 = R^{res}/(S \cdot g^{-m})^{ch} = (g^r)^{r'+x \cdot ch}/g^{x \cdot r \cdot ch} = g^{r \cdot r' + r \cdot x \cdot ch)}/g^{x \cdot r \cdot ch} = g^{r \cdot r' + r \cdot (x \cdot ch)}/g^{x \cdot r \cdot ch} = R^{r'}$. Now some verifying the decryption can be sure (with an adversary only cheating with negligible probability) that the decryption was performed correctly and the g^m is the original vote. Finally, when the votes are decrypted they are tallied and result published.

Helios-C. Helios-C [27] is an adaptation on Helios and an election service using Helios-C at its core and can be found at [18]. It outlined a more specific construction of an election service that the individual parties (e.g. Trustee, Bulletin Board, etc.) should follow and introduced the use of a registrar, allowing them to introduce or expand on notions of security. Figure 5 from Helios-C shows an example of such a construction. Helios-C means Helios with credentials, and is the result of adding a registrar, which as stated in section 3.1, is responsible for allocating credentials for each voter. These credentials are actually signing keys, whereby a unique pair of public and private keys is given to the voter, and the public key is stored in the bulletin board. This is the only role of the registrar and therefore all knowledge of the credentials can be deleted, but not before making the public keys publically available. This registrar adds an additional authority to the election, as each voter now signs their ballot with their secret key and they can verify their vote is in the Public Bulletin Board, but also other parties can use the voters public verification keys to ensure that the ballots in the PBB all verify (they haven't been altered) and also compare the list of credentials L with the PBB list.

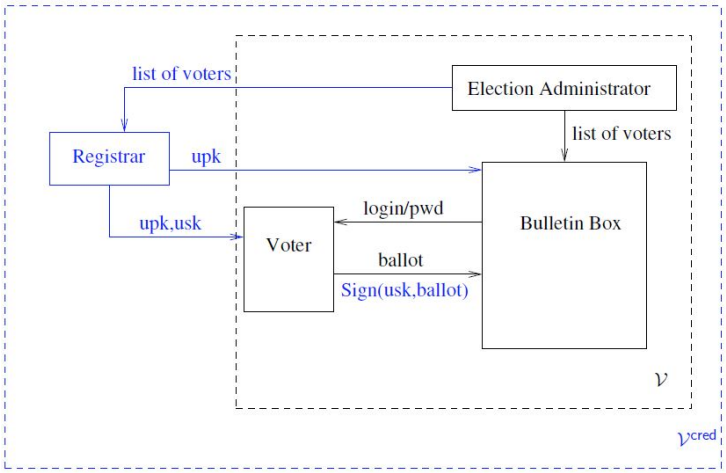


Figure 5: Helios-C Construction Example [27]

Helios-C defined a set of election algorithms [27] and are a part of the whole election syntax used. The algorithms are used in many voting protocols to define separate generic functions that they should execute. The algorithms are then open to more specific definitions, depending on the voting protocol being considered. A high level definition for Helios-C is given, though is used by BeleniosRF, too, as it follows the same structure:

Setup(1^λ): This using security parameter 1^λ, generates an election keypair (pk, dk), outputting this and other parameters necessary for the encryption and signing schemes used.

Register(id): Using a voter id as input it will output the voters signing keypair (upk, usk), where the verification key uvk will then be adding to a list of credentials L and $(uvk, usk$ given to the voter.

Vote(id, upk, usk, v)): Ran by voters to cast their $v \in \mathcal{V}$ and sends the ballot b to the voting server.

Valid(BB, b): Checks that the ballot b belong to an eligible voter and is well formed.

Append(BB, b): Adds ballot b to the Bulletin Board **BB**.

Publish(BB): Takes the Bulletin Board BF as input and outputs the Public Bulletin Board (**PBB**)

VerifyVote(PBB, id, upk, usk, b): Typically ran by the voter to verify their vote will be included in the tally and therefore whether it's in the Public Bulletin Board (**PBB**)

Tally(BB, dk): This takes the Bulletin Board (**BB**) and private key dk, which are decrypted and tallied to produce a result. Outputs the result and proof on that result.

Verify(PBB, r, Π): Takes the Public bulletin Board (**PBB**), the result r of the election produced by a trustee and the proof of correct decryption for that result Π. Checks whether Π is correct for the result r.

As with Helios, Helios-C uses the Elgamal encryption scheme for its election, given its homomorphic property. For the introduction of voters now signing their ballots, the Schnorr signature scheme [26] is used. An election Administrator will create an election, again providing the list of eligible voters, now this list will be sent to the registrar which will generate signing key pairs for each eligible voter and email them directly. When a voter votes they, as with Helios, encrypt their vote using Elgamal with the election public key pk and provide an NIZK Disjunctive proof that they've voted $m = 0$ or $m = 1$. Now, the voter creates a signature on the ciphertext for their vote using their secret key. The voter submits their ballot $b = (upk, (c, \pi), \sigma)$ to the voting server, which then validates the ballot by, firstly, verifying the upk given is in the list of eligible voters L for that election. Secondly, verifies the disjunctive proof and lastly verifies the signature on the ballot. The ballot is accepted if these three checks are passed.

When the voting period ends a trustee can tally the result. The trustee validates the ballots and then uses the homomorphic property of Elgamal (section 2.5) to compute the result ciphertext C_Σ from all the individual voter ciphertexts which is $(R_\Sigma, S_\Sigma) = () \prod_{b \in \mathbf{BB}} R_b, \prod_{b \in \mathbf{BB}} S_b$. The result g^r is retrieved with the decryption $S_\Sigma \cdot R_\Sigma^{-dk}$. Given the example of two votes $m_1 = 0$ and $m_2 = 1$ giving two ciphertexts $c_1 = (g^{r_1}, pk^{r_1} \cdot g^0)$ and $c_2 = (g^{r_2}, pk^{r_2} \cdot g^1)$, the decryption would be $(pk^{r_1} \cdot g^0 \cdot pk^{r_2} \cdot g^1) \cdot (g^{r_1} \cdot g^{r_2})^{-dk} = (pk^{r_1+r_2} \cdot g^{0+1}) \cdot (g^{r_1+r_2})^{-dk} = g^{dk \cdot (r_1+r_2)} \cdot g^{-dk \cdot (r_1+r_2)} \cdot g^{0+1}) = g^{0+1}$. Giving the result $g^r = g^1$ and as discussed in section 2.5 this means the trustee has to compute the discrete log to obtain r, which can be done efficiently for small numbers, taking time $\sqrt{\tau}$ where τ is the number of voters in the election and r is between 0 and τ. $r = \tau$ being the case where every voter voted a 1. Finally the trustee creates a proof of decryption for the result ciphertext which proves that $\log_g pk = \log_{R_\Sigma} S_\Sigma \cdot (g^r)^{-1}$, which can be computed together, too, instead of computing individual proofs for each vote.In Helios the bulletin board and trustee (which are under one party, the server, unless the option for multiple trustees is used) have to be solely trusted, but with Helios-C the registrar or bulletin board can be dishonest but not simultaneously [27], which they state as having strong verifiability if it has individual and universal verifiability.

BeleniosRF BeleniosRF [24] uses receipt-freeness, where a voter, even a dishonest one, can not prove how they voted, but are still able to verify their vote is in the final published ballot-box, with proofs that their vote has not been spoiled. Unlike Helios and Helios-C, which uses the cyclic subgroup $\mathbb{G}_q$ of $\mathbb{Z}_p$, we now use groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ with the mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, which are asymmetric bilinear groups possible with pairing friendly elliptic Curves (section 2.1). The asymmetric bilinear groups are required because of the use of Asymmetric Waters Signature Scheme [5] for signatures on randomised ciphertexts (section 2.6.2) and for the Groth-Sahai proofs, to allow randomising, which is essential for receipt-freeness. Receipt-freeness is a key property to election privacy as it truly allows a voters' vote to remain private, given the honesty of the voting server and Trustee. Due to this receipt-freeness where even the voter can not prove how they voted, scenarios such as a voter buyer are far more restrictive, as even if the voter wanted to willingly inform a vote buyer of their vote, it would not be able to be proved cryptographically when the vote buyer tries to view the specific vote in the public bulletin board. BeleniosRF also introduces this method of receipt-freeness without any additional actions of the voter, other than them voting, with a direct quote from the paper being "...a voter does not need to lie or produce fake credentials ... she simply has no way to prove how she voted." ([24]). The only link the voter has to their ballot when published is their verification key, which can only be used by them to ensure their vote was not altered, but given that the encryption of their vote has been randomised, they can simply not prove how they voted.

Groth-Sahai proofs [19] are used to prove commitments of the individual message bits in a voters vote and are used along side the Asymmetric Waters Signature Scheme to ensure signatures on unforgeable. [5] For case of BeleiosRF they are used to also prove specific relations of the ciphertexts given when a voter submits their ballot. As stated above they are also randomisable, which means that parties are able to verify the integrity of votes using the corresponding verification key, but also verify themselves that the ballot in the PBB contains a vote of only messages that are zero or one.

3.3 BeleniosRF+

BeleniosRF+ [17] removed the Groth-Sahai proofs used in BeleniosRF and replaced them with the NIZK disjunction proofs, that were used in Helios and Helios-C. This was because NIZK Disjunction proofs are far more efficient than the Groth-Sahai proofs, both in terms of size and running time, though the latter being the most improvement reason, given that a large amount of individual Groth-Sahai proofs have to be computed on the voters' device when they vote. Though with one advantage, there is a disadvantage, which is that NIZK Disjunction proofs can not be randomised and therefore can not be published in the public bulletin board along side the randomised ciphertext and its fresh signature, as the randomness use to encrypt them clientside could be used as a means of proving how a voter voted. This means that integrity of the vote can still be ensured with the voter verification key, and therefore the vote can not be changed and go undetected, but the ballot could be spoiled (making the vote invalid), which would not be detected until being tallied by a trustee, requiring some additional trust placed on the voting server. Basically it means that an election could be spoilt and a reelection would have to occur, but the result of the election could not be altered (providing the parties are honest and don't collude)

BeleniosRF+ uses the Elgamal encryption scheme with asymmetric bilinear mapping parameters and is IND-CPA, assuming the DDH assumption, given the hardness in group $\mathbb{G}_1$, the Asymmetric Waters Scheme which is randomisable and secure under the CDH$^+$ assumption. As stated above it now uses the NIZK Disjunction proof, but for this a change is needed in the voting algorithm which requires an extension to the $\mathsf{DisjProof}_F(g, pk, R, S) = (\mathsf{DisjProve}_F, \mathsf{DisjVerify}_F)$ into $\mathsf{DisjProof}^+_F(g, u, pk, R, S) = (\mathsf{DisjProve}^+_F, \mathsf{DisjVerify}^+_F)$ [17]. Where $\mathsf{DisjProve}^+_F(g, u, R, S, r)$ now proves $(g, pk, R, S) \in \mathcal{L}_{EqDl}$ or $(g, pk, R, S \cdot u^{-1}) \in \mathcal{L}_{EqDl}$ using [11]. The BeleniosRF+ ex-

tension is a follows, for the case of $(g, pk, R, S) \in \mathcal{L}_{EqDl}$ ($m = 0$). The prover makes a real proof for $(g, pk, R, S) \in \mathcal{L}_{EqDl}$ by choosing a random $u_0 \xleftarrow{\$} \mathbb{Z}_p$ and setting $U_0 = g^{u_0}$ and $V_0 = pk^{u_0}$. Then fakes a proof by randomly choosing $(ch_1, rep_1) \xleftarrow{\$} \mathbb{Z}_p \times \mathbb{Z}_p$ and setting $U_1 = g^{rep_1}/R^{ch_1}$ and $V_1 = pk^{rep_1}/(S \cdot u^{-1})^{ch_1}$. The challenge $ch = F(g, pk, R, S, U_0, V_1, U_1, V_1)$ and the proof $\pi = (ch_0 = ch - ch_1, ch_1, rep_0 = u_0 + ch_0 \cdot r, rep_1)$. $\mathsf{DisjVerify}^{+}_{F}(g, u, pk, R, S, \pi)$ accept or rejects for $ch_0 + ch_1 \stackrel{?}{=} F(g, pk, R, S, \frac{g^{res_0}}{R^{ch_0}}, \frac{pk^{res_0}}{S^{ch_0}}, \frac{g^{res_1}}{R^{ch_1}}, \frac{pk^{res_1}}{(S \cdot u^{-1})^{ch_1}})$. The implementation for this using elliptic curves and group $\mathbb{G}_1$ is in section 2.8.3.

3.3.1 BeleniosRF+ SRC Scheme

BeleniosRF+ takes the Signatures on Randomisable Ciphertexts by [5] (section 2.7) and extends it by adding additional parameters, ciphertexts and signatures, to become (Setup, $\mathsf{EKeyGen}$, $\mathsf{SKeyGen}$, $\mathsf{Encrypt}^{+}$, $\mathsf{Decrypt}^{+}$, Sign^{+}, Verify^{+} and Random^{+})[1] which is unforgeable under the CDH^{+} and is shown in figure 6. Firstly, $\mathsf{EKeyGen}$ produces two additional parameters $(h_1, h_2) \xleftarrow{\$} \mathbb{G}_1^2$ which is used in the function $H : \{0,1\}^* \to \mathbb{G}_1$ which is defined as

$$H(x) := h_1 \cdot h_2^{H'(vk)} \tag{6}$$

where $H' : \{0,1\}^* \to \mathbb{Z}_p$ is a collision resistant hash function.

[1] BeleniosRF+ refers to its SRC algorithms with an added $^{+}$ to differentiate that now the Elgamal encryption scheme and Asymmetric Waters Signatures scheme are being used together

$\mathsf{EKeyGen}(1^\lambda, 1^k)$:

$$pp := (\mathcal{G}, z, \boldsymbol{u}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda, 1^k);\ (h_1, h_2) \xleftarrow{\$} \mathbb{G}_1^2; \qquad Run(P, dk) \xleftarrow{\$} \mathsf{KeyGen}(\mathcal{G}, 1).$$

Return $pk := (pp, (h_1, h_2), P), dk$

$\mathsf{Encrypt}^+((pp, (h_1, h_2), P), vk = (pp, X_1, X_2), m; r)$: Compute, with H defined by (h_1, h_2) as

$$c_1 = g_1^r \qquad c_2 = \mathcal{F}(m) \cdot P^r \qquad c_3 = H(vk)^r$$

Return $c := (c_1, c_2, c_3)$

$\mathsf{Sign}^+(sk = (pp, Y), (pp, (h_1, h_2), P), (c_1, c_2, c_3); s)$: Return $\sigma := (\sigma_1, \ldots, \sigma_5)$, where

$$\sigma_1 = c_1^s \qquad \sigma_2 = Y \cdot c_2^s \qquad \sigma_3 = g_1^s \qquad \sigma_4 = g_2^s \qquad \sigma_5 = P^s$$

$\mathsf{Random}^+(vk, (pp, (h_1, h_2), P), (c_1, c_2, c_3), \sigma; (r', s'))$: Set:

$$\begin{aligned}
&{c_1}' := c_1 \cdot g_1^{r'} && {c_2}' := c_2 \cdot P^{r'} && {c_3}' := c_3 \cdot H(vk)^{r'} \\
&{\sigma_1}' := \sigma_1 \cdot c_1^{s'} \cdot \sigma_3^{r'} \cdot g_1^{r' \cdot s'} && {\sigma_2}' := \sigma_2 \cdot c_2^{s'} \cdot \sigma_5^{r'} \cdot P^{r' \cdot s'} && {\sigma_3}' := \sigma_3 \cdot g_1^{s'} \\
&{\sigma_4}' := \sigma_4 \cdot g_2^{s'} && {\sigma_5}' := \sigma_5 \cdot P^{s'}
\end{aligned}$$

Return $c := (c_1', c_2', c_3')$ and $\sigma' := (\sigma_1', \sigma_2', \sigma_3', \sigma_4', \sigma_5')$

$\mathsf{Verify}^+((pp, X_1, X_2), (pp, (h_1, h_2), P), (c_1, c_2, c_3), \sigma)$: Return 1 iff the following holds:

$$\begin{aligned}
&e(\sigma_1, g_2) = e(c_1, \sigma_4) && e(\sigma_2, g_2) = e(z, X_2)\, e(c_2, \sigma_4) \\
&e(\sigma_3, g_2) = e(g_1, \sigma_4) && e(\sigma_5, g_2) = e(P, \sigma_4)
\end{aligned}$$

$\mathsf{Decrypt}^+(dk, (pp, (h_1, h_2), P), vk, (c_1, c_2, c_3))$:

$$Let\ F = \mathsf{Decrypt}(dk, c = (c_1, c_2));$$

Browse $\mathcal{M}$ and return the first m with $\mathcal{F}(m) = F$.

Figure 6: BeleniosRF+ [17] SRC Scheme

$\mathsf{Encrypt}^+$ now produces an additional ciphertext $c_3 = H(vk)^r$, where H is the function defined above and $vk = (pp, X_1, X_2)$; it is used to hash the voters' verification key to an element in $\mathbb{G}_1$, which is used to link the voter to that ciphertext c. Random^+ randomises c_3 too, with $c_3' = c3 \cdot H(vk)^r$. It adds the $\mathsf{Decrypt}^+$ algorithm which, using Elgamal, decrypts $F = (c_1, c_2)$ and then brute forces $\mathcal{F}(m)$ where $m \in \mathcal{M}$ resulting in the vote $v = m$ when the first $F = \mathcal{F}(m)$.

3.3.2 BeleniosRF+ Election Algorithms

Below is the final BeleniosRF+ election algorithms, where the more generic algorithms presented in section 3.2 are now defined more specifically for this particular election protocol. The algorithm can be seen to be utilising its SRC scheme and implementing checks to ensure the security of the protocol and proofs to ensure everything can be verifiable by an external party. Using the definitions in BeleniosRF+ [17] the algorithms are:

$\mathsf{Setup}(1^\lambda, 1^k)$: $pp := (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, z, \boldsymbol{u}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda, 1^k);\ (h_1, h_2) \xleftarrow{\$} \mathbb{G}_1^2$
For $i = 1, \ldots, k$ run $(P_i, dk_i) \xleftarrow{\$} \mathsf{KeyGen}(\mathcal{G}, 1)$.

Define $\boldsymbol{P} = (P_1, \ldots, P_k),\ P = \prod_{i=1}^{k} P_i,\ dk = \sum_{i=1}^{k} d_i$.
Return $\mathbf{pk} := (pp, (h_1, h_2), P, \boldsymbol{P})$, $\mathbf{sk} := dk$.

$\mathsf{Register}(id)$**:** On (implicit) input $\mathbf{pk} = (pp, h, P, \boldsymbol{P})$, return $(upk_{id}, usk_{id}) \xleftarrow{\$} \mathsf{SKeyGen}(pp)$.

$\mathsf{Vote}(id, upk, usk, v)$**:** is used by a voter to create a ballot b for vote $v \in \mathbb{V}$, by computing, with H defined by h:

$$c_1 = g_1^r \qquad\qquad c_3 = H(upk)^r$$

Compute partial Elgamal encryptions of $m_i \in \{0, 1\}$, where $m = (m_1, \ldots, m_k)$,
For $i = 1, \ldots, k$

$$c_{2,i} = P_i^r \cdot u_i^{m_i} \qquad\qquad \pi_i = \mathsf{DisjProve}_F^+(g_1, u_i, c_1, c_{2,i}, r)$$

Let $\boldsymbol{c_2} = (c_{2,1}, \ldots, c_{2,k}),\ \boldsymbol{\pi} = (\pi_1, \ldots, \pi_k)$. Define

$$c_2 := u_0 \prod_{i=1}^{k} c_{2,i} = \mathcal{F}(m) P^r$$

Let $c := (c_1, c_2, \boldsymbol{c_2}, c_3, \boldsymbol{\pi})$. Compute $\sigma \leftarrow \mathsf{Sign}^+(usk, \mathbf{pk}, (c_1, c_2.c_3))$. Return $b = (id, upk, c, \sigma)$.
Notice that (c_1, c_2, c_3) thus obtained enjoys the same distribution as $\mathsf{Encrypt}^+((pp, h, P), vk = (pp, X_1, X_2), m; r)$.

$\mathsf{Valid}(BB, b)$**:** first checks that the ballot b is *valid*, i.e., that it is well-formed and the signature is correct. Formally, it parses b as (id, upk, c, σ) and $c = (c_1, c_2, \boldsymbol{c_2}, c_3, \boldsymbol{\pi})$ checks if

1. id corresponds to an eligible voter from ID and upk corresponds to the registration of user id;
2. Verify $c_2 \stackrel{?}{=} \boldsymbol{c_2}$:

$$c_2 \stackrel{?}{=} \prod_{i=1}^{k} c_{2,i}$$

3. $\mathsf{Verify}^+(upk, \mathbf{pk}, (c_1, c_2, c_3), \sigma) = 1$;
4. For $i = 1, \ldots, k$: $\mathsf{DisjVerify}_F^+(g_1, u_i, c_1, c_{2,i}, \pi_i) = 1$.

If any step fails, it returns $\perp$; otherwise, it returns $\top$.

$\mathsf{Append}(BB, b(id, upk, c, \sigma))$**:** parses b as (id, upk, c, σ) and $c = (c_1, c_2, \boldsymbol{c_2}, c_3, \boldsymbol{\pi})$. Randomises $((c_1, c_2, c_3), \sigma)$ resulting in $((c_1, c_2, c_3), \sigma)$ as $((c_1', c_2', c_3'), \sigma') \leftarrow \mathsf{Random}^+(upk, \mathbf{pk}, (c_1, c_2, c_3), \sigma)$, a randomised ciphertext with a new verifiable signature. The updated ballot $\mathsf{b} = (id, upk, (c_1', c_2', c_3'), \sigma')$ is then appended to BB.

$\mathsf{Publish}(BB)$**:** Every element $\mathsf{b} = (id, upk, (c_1', c_2', c_3'), \sigma')$ in BB is adapted to be published in the PBB by removing the elements id, c_3 and σ_5, resulting in the publishable ballot $\hat{b} := (upk, (c_1', c_2'), (\sigma_1', \sigma_2', \sigma_3', \sigma_4'))$m which is then added to PBB. After all ballot are made publishable then PBB is returned.

$\mathsf{VerifyVote}(PBB, id, upk, usk, b)$**:** looks for $\hat{b}$ in PBB that contains upk. If none exists, it returns $\perp$. For the entry $\hat{b} := (upk, (pp, X_1, X_2), (c_1, c_2), (\sigma_1, \sigma_2, \sigma_3, \sigma_4))$ if

$$e(\sigma_1, g_2) = e(c_1, \sigma_4) \qquad e(\sigma_2, g_2) = e(z, X_2) \cdot e(c_2, \sigma_4) \qquad e(\sigma_3, g_2) = e(g_1, \sigma_4)$$

then return $\top$, else return $\perp$.

$\mathsf{Tally}(BB, \mathbf{sk})$**:** consists of the following steps.

1. Parse each ballot $b \in BB$ as $b = (id^{(b)}, upk^{(b)}, (c_1^{(b)}, c_2^{(b)}, c_3^{(b)}), \sigma^{(b)})$
2. $id^{(b)}$ corresponds to an eligible voter from set of the ids, ID, for the election and $upk^{(b)}$ corresponds to the registration of user id.
   The signature of each ciphertext is verfieid with $\mathsf{Verify}^{+}(upk^{(b)} = (pp, X_1, X_2), \mathbf{pk} = (pp, (h_1, h_2), P), (c_1^{(b)}, c_2^{(b)}, c_3^{(b)}), \sigma^{(b)})$, output $(r = \perp, PBB, \Pi_d = \emptyset)$. If any verification fails then $(r = \perp, \boldsymbol{PBB}, \Pi_d = \emptyset)$ is output.
3. The ciphertexts are anonymised using shuffling, resulting in $(c_1'^{(b)}, c_2'^{(b)})_{b \in BB}$ and a proof of
4. Each ciphertext in the anonymised table $(c_1'^{(b)}, c_2'^{(b)})_{b \in BB}$ is decrypted with Elgamal $\mathsf{Decrypt}$ returning F. m is found by brute force, in which the first $m \in \mathcal{M}$ which results in $\mathcal{F}(m) = F$ is the vote. When all the anonymised ciphertexts are decrypted and their corresponding $m = (m_1, \ldots, m_k)$ is found, then the result is computed using vector addition giving $r = (t_1, \ldots, t_k)$. A Fiat-Shamir proof decryption is computed for each F and its corresponding ciphertext, represented as Π_d for all proofs.
5. Output (r, PBB, Π_d).

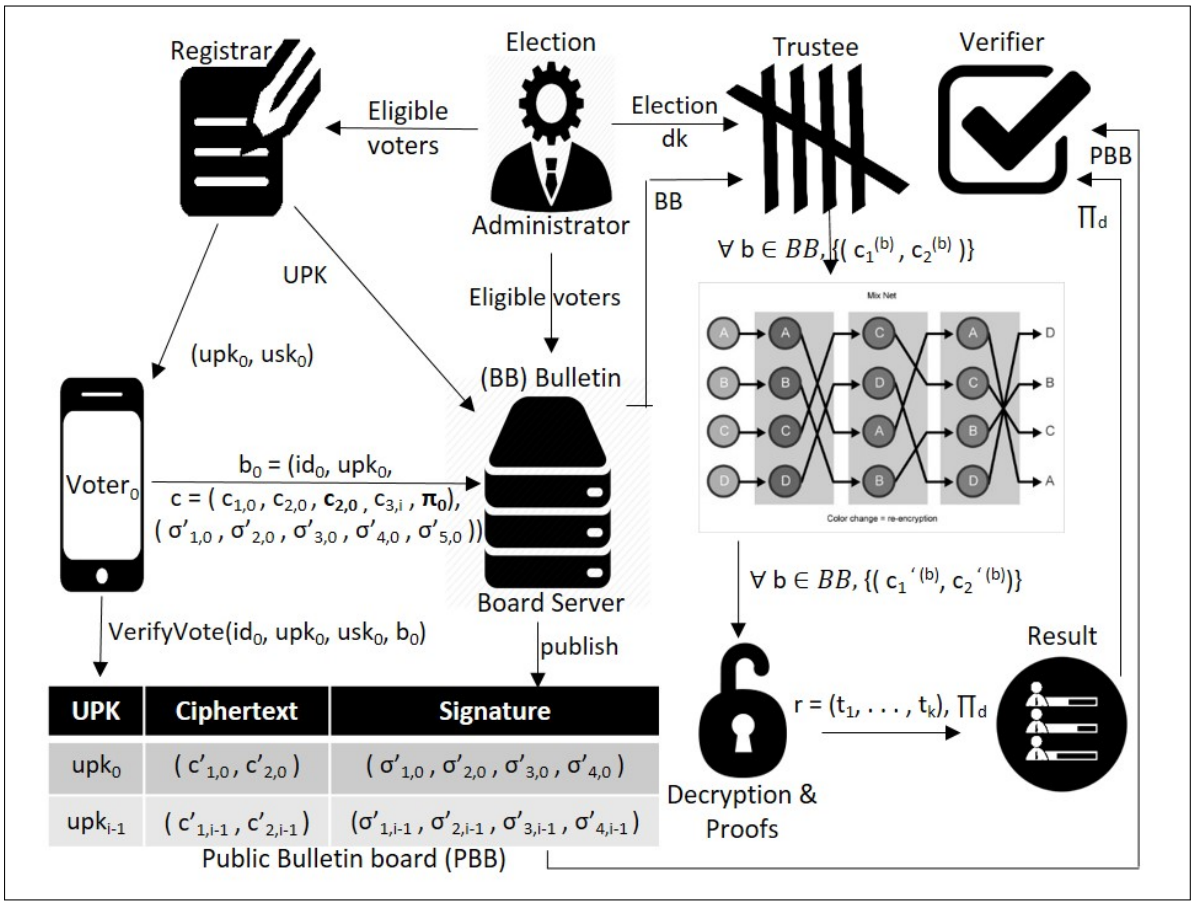


Figure 7: BeleniosRF+ Protocol Example

4 Implementation

This section displays the implementation of the BeleniosRF+ protocol, explained in section 3.3.1, showing the most fundamental and core features of the implementation. Even though most of the schemes are implemented in both Java and Javascript, the example presented are where that language is most likely to be used during the election protocol, for example, the NIZK disjunction prove algorithm (section 2.8.3) is more commonly used client side when the voter casts their vote, which would be Javascript, but the NIZK logarithm equality prove algorithm (section 2.8.3)) is most commonly used server side, which would be Java.

Multiplicative to Elliptic Curve Additive Operations. A note to the reader. Previous sections have shown the protocols over multiplicative groups $\mathbb{Z}_p$, subgroups $\mathbb{G}_q$ or the groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ with the bilinear mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, but multiplicatively. This section, given it's the actual implementation using elliptic curves, uses the additive operations, as described in section 2.1 on elliptic curves. For example, for computing $c_{2,i}$ in the voting algorithm (section 3.3.1) it's presented as $c_{2,i} = P_i^r \cdot u_i^{m_i}$ which using elliptic curve additive operations is $c_{2,i} = r \cdot P_i + m_i \cdot u_i$, where $+$ is the elliptic curve point addition and $\cdot$ is elliptic curve point multiplication. r, m_i are scalars and P_i, u_i are points on the curve. In the text the notation P^a or aP will be used for point multiplication and $P \cdot Q$ or $P + Q$ point addition.

4.1 Milagro-Crypto

The milagro-crypto library [22] by miracl was used for the implementation, as it offered the elliptic curve pairing friendly (section 2.1) functionality required in the BeleiosRF papers. I used the Javascript and Java libraries, for the browser and server functionality, respectively. There are some classes and their methods I shall explain to help with the understanding of the code snippets displayed in this report and for when/if reviewing the source code, as they are used throughout the implementation.

`ROM` is used throughout the milagro-crypto library and my implementation, as it acts as a configuration file. A selection of important parameters it defines are: `ROM.CURVETYPE` which is the elliptic curve type to use, with the main options being MONTGOMERY_FRIENDLY, WEIERSTRASS, EDWARDS and MONTGOMERY. `ROM.modulus` being the modulus which is the prime of the finite field, but I referenced this via `AbstractAsymmetricBilinearGroupParams.getP()` in the implementation and p throughout the paper. `ROM.CURVE_Order` which is the order of the elliptic curve, i.e the number points on the curve. This is referenced via `AbstractAsymmetricBilinearGroupParams.getN()` and n throughout this paper. `ROM.CURVE_Gx` and `ROM.CURVE_Gy` which represent the x and y coordinates of a point on the curve g_1, which is a generator (section 2.1) and in this case are for the elliptic curve $E(\mathbb{F}_p)$. `ROM.CURVE_Pxa`, `ROM.CURVE_Pxb`, `ROM.CURVE_Pya` and `ROM.CURVE_Pya` are used, again x and y are for the point on the curve g_2 that generates all points (section 2.1), but in this case for $E(\mathbb{F}_{p^a})$. a and b are elements in $\mathbb{F}_p$ used to represent an element in $\mathbb{F}_{p^2}$, as is the a and b for y, but different elements.
`BIG` is used to store large integers of maximum 2^{256}, which is large enough for elliptic curves, as the prime used for these curves is 254 bits in size, but for when these numbers are multiplied together, and therefore need double the amount of bits, then `DBIG` is used which allows integers with a maximum of 2^{512}. `BIG` has useful methods for when needing to do arithmetic within the multiplicative finite field, where the modulus is the curve order n, rather than the usual elliptic curve mod p) arithmetic (e.g. point doubling and point multiplication) such as `BIG.modmul(a, b, m)` which takes two BIGs, a and b and uses the BIG multiplication method on them which returns a `DBIG` and then returns its congruence as a `BIG` using m as the modulus. `MPIN.hashit(...)` is used to hash some bitstring of length n which can be used to create a `BIG` using `BIG.fromBytes(...)` or by mapping the hash to a point on the curve $E(\mathbb{F}_p)$ using

`HASH.mapit(...)`. Lastly, a method supplied by `BIG` that is paramount for the security of the implementation is `BIG.randomnum(BIG q,RAND rng)`, where q is large integer, for example, at least 2^256 bits in size, and rng is Milagro's random number generator that is seeded with an external source of entropy (atleast 16 bytes) which is then hashed, processed through pseudo random number generator and then hash again, to produce psuqedo random numbers indistinguishable from random. In section 3 when references are made to randomness and leakage, this is that randomness, where the external source of entropy is as vital to the security of the encryption scheme itself.

`ECP` and `ECP2` are used to represent points on the curves for $E(\mathbb{F}_p)$ and $E(\mathbb{F}_{p^a})$, respectively. They both have methods `this.add(...)`, `this.dbl(...)` and `this.mul(...)`. The first is used to add a point P' to point P, the second is the doubling of point P and the third is where a point P is multiplied by a BIG e. The methods used are `PAIR.ate(ECP2 Q, ECP P)`, `PAIR.ate2(ECP2 Q, ECP q, ECP2 Q', ECP P')` and `PAIR.fexp(F12 m)`. The first is used to find the mapping $e(Q,P)^2 \in \mathbb{F}^*_{p^a}$ and the second is for $e(Q,P) \cdot e(Q',P') \in \mathbb{F}^*_{p^a}$. The element in $\mathbb{F}^*_{p^a}$ is not the actual final result and therefore `PAIR.fexp(...)` is used to perform the final exponentiation and return another `F12`. This separate call is due to the amount of memory used during these computations. This section of the library is paramount for using the properties of bilinear pairing and their use in the implementation of the Asymmetric Waters Signature Scheme [5].

These classes and methods are used throughout our implementation, as they are the core of elliptic curve cryptography and asymmetric bilinear groups. They are used to setup encryption schemes, generate keys, encrypt, sign and verify; all of which is abundant in the implemention.

4.2 Structure

Many of the same algorithms were implemented in both Javascript and Java, as it provided the ability to test sections of the voting protocol solely on a client device, where a html page to test different parameters was placed in a public online cloud storage and client specific tests were able to be ran on different devices and browsers. In terms of the client side, it allows for future expansions or adaptations that could be added to the voting protocol, or cases where specific parties on ran on a client device, rather than a more powerful server, such as a registrar generating credentials, which could be done used the `AWSS.setup()` and `AWSS.sKeyGen()` in*asymmetricWaterSignatureScheme.js*. Javascript has Javascript objects which allow the mapping of a property name to some other Javascript variable. The code snippet 1 shows a parameters used abundantly in the Javascript implementation, where here the variables, which are after the colons, are `n` is the order of the order, `p` is the prime of the finite field that the elliptic curve is over, `g1` is a generator for group $\mathbb{G}_1$, `g2` is the generator for group $\mathbb{G}_2$ and `rng` is the random number generator used. The values before the colons are property names which map to the specific variable. To access the generator `g1` I would use `asymGrpParams.g1`.

```
1 var  asymGrpPrms = {
2              n: n,
3              p: p,
4              g1: g1,
5              g2: g2,
6              rng: eMpinAuth.getPrng_()
7 }
```

Code Snippet 1: JS Object of Asymmetric Group Parameters Example

As these values are use throughout the implementation and they need to be easily passed on other sections, such as the Elgamal, Asymmetric Waters Signature and SRC implementations.

[2] P and Q are flipped in the mapping, which is part of the notation for Ate pairings [13] and used by the milagro-crypto [22] library with their method argument ordering

This can be achieved easily in javascript as the Javascript object `asymGrpParams` can be added to another Javascript object with other variables. The code snippet 2 shows how the group parameters are added to the Asymmetric Waters Signature parameters and `g1` would now be accessed with `asymWaterSigParams.asymGrpParams.g1`. The necessary parameters are given as input to the setup methods such as `AWSS.setup(...)` `SRC.eKeyGen(...)` and this offers flexibility in the case where the parameters were to change, the user would just have to use the same property names. There is addition flexibility, which is taken advantage of, in that I don't the Javascript object given doesn't have to have the exact same structure, it could even have more parameters than needed, it just needs to have the variables used with the correct property names used for accessing. For comparison this is similar where in Java a subclass would be given as argument in a method which is expecting the superclass; the method will still be able to access the variables needed (stated by the superclass), but the subclass could have additional variables.

```
var asymWaterSigParams = {
  asymGrpParams : asymGroupParams,
    z: z,
    usigs: usigs,
    k: k
 }
```

Code Snippet 2: Asymmetric Waters Signature JS Object Example

The same easy access of the different parameter was needed in the Java library too, but Java's closest object to that of Javascript object would be a mapping object, but they would be an additional object to be passed around and would have introduced too much overhead. Instead I make use of Java's class inheritance features, where classes subclass other classes. The code snippet 3 displays the abstract class `AbstractAsymmetricBilinearGroupParams` which is used similarly to that of the Javascript object for the asymmetric group parameters and has the same parameters, where each represent the same as before. The class is abstract and therefore restricted from objects being an instance of it and therefore classes are meant to subclass this providing the parameters to the constructor, this offers the flexibility of using different parameters in the future. For example the Elgamal class methods have an `AbstractAsymmetricBilinearGroupParams` instance as an argument enabling the use of any parameters. Using static methods and giving the parameters as arguments, among the other variables needed for that specific method, also follows the algorithm design set out in the papers, whereby they take some input and then return some output, where instances of classes are only use for the parameters inputted and returned.

```
public abstract class AbstractAsymmetricBilinearGroupParams extends Params {

  private BIG p;
  private BIG n;
  private ECP g1;
  private ECP2 g2;

  public AbstractAsymmetricBilinearGroupParams(BIG p, BIG n, ECP g1, ECP2
      g2, RAND rng) {

    super(rng);
    this.p = p;
    this.n = n;
    this.g1 = g1;
    this.g2 = g2;
  }

  //Getters and setters for the field variables p, n, g1 and g2.
}
```

Code Snippet 3: AbstractAsymmetricBilinearGroupParams Class

4.3 Implementing the Building Blocks

The building blocks were implemented first so they could be used solely on their own, but also by making use of helper methods, which in many cases, higher level functions would call instead, such as the SRC scheme, which then would be used itself by even higher level functions, such as the BeleniosRF+ algorithms (E.g. Register, Vote, Valid, etc.). It was implemented in this way as it enforces reusability and code separation, offering advantages when testing and helps someone using the library understand the core of a given method. During testing if there was bug then there would be confidence that it was the high level function, when successful testing was performed on the called building block functions. As displayed in figure 6 and as explained in its section, the Signatures on Randomisable Ciphertexts scheme uses the building blocks mentioned, but it doesn't use them in their standard implementation and they're usually extended, for example, the Asymmetric Waters Signature produces the signature $\sigma = (\sigma_1, \sigma_2, \sigma_3)$, but the SRC scheme extends this and produces $\sigma = (\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5)$ when signing and this another reason for implementing the building block with the helper methods. Lastly, as stated, it helps a user understanding, as it gives the user the knowledge of specifically what schemes are being used, rather than just having the SRC scheme implement the whole algorithm for e.g. $\mathsf{Encrypt}^+$ in one method, when it actually utilises the Elgamal scheme.
5

```
1  public static G1KeyPair keyGen(AbstractAsymmetricBilinearGroupParams params)
       {
2
3  // the private key
4  BIG dk = BIG.randomnum(params.getN(), params.getRng());
5
6  ECP pk = params.getG1();
7
8  // pk = g^dk (point addition and doubling - i.e g + g + ... + g, dk
9  // times)
10 pk = pk.mul(dk);
11
12 // where pk is point ECP and dk is a big integer
13 return new G1KeyPair(pk, dk);
14
15 }
```

Code Snippet 4: Elliptic Curve Key Generation

Key Generation. Code Snippet 4 is how an Elgamal keypair is created, where a $g \in E(\mathbb{F}_p)$ is point multiplied against a randomly generated scalar $dk \in \mathbb{Z}_n$ (the private key), where n is the curve order. dk is uniformly randomly integer obtained by calling `BIG.randomnum(...)` and is any integer between 0 and $2^{254} - 1$ creating the public key $pk \in E(\mathbb{F}_p)$ and thus the keypair (pk, dk). This computation is very efficient, which is guaranteed as a result of the addition and multiplication elliptic curve operations, point doubling and point multiplication, respectively, which have to be efficient to compute to uphold as cryptographic scheme, including that finding dk in computationally infeasible with large enough numbers, n in this case. If this algorithm was ran and only pk was returned, it would be computationally infeasible for the user to compute dk, under the elliptic curve discrete logarithm problem (section 2.1). The same principle here for key generation is used throughout the scheme when generating other keys or just random points in the group $\mathbb{G}_1$ or $\mathbb{G}_2$. For example, during the Setup algorithm in the Asymmetric Waters Signature Scheme, random numbers are generated and point multiplied against the elliptic curves generator to create elements such as $\boldsymbol{u} = (u_0, \ldots, u_k) \xleftarrow{\$} \mathbb{G}_1^{k+1}$, which is the exact same method of how a public key is generated, but in this case the random number can be discard afterwards. This process was not separated into a separate method, even though some aspects are reused, as

it was a small amount of code and was not worth the overhead of calling the methods, but also for the case stated above where in many cases the random number used would not need to be returned.

Elgamal Randomisation. The code snippet 5 is used for randomising ciphertext as described in 2.6.1, which is the core component of a vital feature of BeleniosRF+ as it's used to reencrypt voter ciphertexts, preventing them from proving how they voted by using some specific randomness on their client device (receipt-freeness). `randomise(...)` takes the parameters for group $\mathbb{G}$, the election public key pk and an Elgamal ciphertext pair c_1, c_2 as argument. The method uses a helper method `randomiseCiphertext` as this method is used throughout the implementation, as it's more flexible, for example, in the BeleniosRF+ SRC scheme the ciphertext is a triple (c_1, c_2, c_3). `randomCiphertext(...)` takes the ciphertext element in $E(\mathbb{F}_p)$ you wish to randomise, $g \in E(\mathbb{F}_p)$ which was part of the original encryption, for example, pk for c_2 and thirdly r, the randomness to use, which is given as an argument, as the same r needs to be used for the other ciphertext elements. For the case of `randomCiphertext(`c_2`, `pk`, `r`)` the returned result would be $c_2 + pk \cdot r$, the point addition of point c_2 with the point multiplication of r and point pk. Having the Elgamal standard ciphertext randomisation use the helper method too, even for such a small amount of code, as it reduces the error rate during implementation.

```
1  // c . g1^r' where g is an point in group G1
2  public static ECP randomiseCiphertext(ECP c, ECP g, BIG r) {
3
4    ECP cRand = new ECP();
5
6    cRand.copy(c);
7
8    cRand.add(g.mul(r));
9
10   return cRand;
11 }
12
13 public static ECPPair randomise(AsymmetricBilinearGroupParams params, ECP p
       k, ECPPair c, BIG r) {
14
15   if (r == null) {
16
17     r = BIG.randomnum(params.getN(), params.getRng());
18   }
19
20   ECP c3 = randomiseCiphertext(c.getEcp1(), params.getG1(), r);
21
22   ECP c4 = randomiseCiphertext(c.getEcp2(), pk, r);
23
24   return new ECPPair(c3, c4);
25
26 }
```

Code Snippet 5: Elgamal Randomisation

Asymmetric Waters Signature Scheme Verification. Asymmetric Waters Signature Scheme [5] was the second building block implemented and the primitive algorithms: `setup(...)`, `sKey Gen(...)`, `sign(...)`, `verify(...)` and `random(...)` were implemented as in section 2.6.2) with the addition of helper methods, which are used by the algorithms stated and to be used externally by other methods. One method being `watersHash(ECP[] usigs, BitEnum[] message)` which took the usigs i.e $(u_0, \cdots, u_k) \xleftarrow{\$} \mathbb{G}_1^{k+1}$ outputted from `setup(...)` (which was among other parameters) and the message which is given as a k length array of `BitEnum`'s, which was a

simple Enum implemented to only allow the integer values zero and one.

Code snippet 6 is the Asymmetric Waters Signature Scheme verification algorithm written in Java, discussed in section 2.6.2. It displays how I used helper methods such as `verifyPairs(...)` and `verifySingleWithDoublePairs(...)`, to again allow this building block to be reused. As the section states, the Verif algorithm makes two comparisons (both of which have to be true), $e(\sigma_2', g_2) = e(g_1, \sigma_3')$ and $e(\sigma_1, g_2) = e(z, X_2) \cdot e(\mathcal{F}(m), \sigma_3)$, where the `verify(...)` method is its implemented equivalent and it takes the public Asymmetric Waters Scheme parameters using an instance of `AsymmetricWatersSignatureSchemePublicParams`, the Waters hash F of the message an `ECP` and the signatures an instance of `AsymmetricWatersSignature`. In the method the first comparison is calling the `verifyPairs(...)`, which takes four points, where here $(P1 = \sigma_2 and P2 = g_1) \in \mathbb{G}_1^2$ which is $E(\mathbb{F}_p)^2$ and $(Q1 = g_2, Q2 = \sigma_3) \in \mathbb{G}_2^2$ which is $E(\mathbb{F}_{p^{12}})^2$. `PAIR.ate(Q1, P1)` is called to retrieve the optimal ate pairing returning an instance of the `FP12` class, final exponentiation is performed (separately for memory issues) returning another `FP12`, which is a representation of $\mathbb{F}^*_{p^{12}}$. The same is done for points P2 and Q2 and then the equality of the two `FP12` is checked and if it returning true if it is and false otherwise. `verifySingleWithDoublePairs` is used to check the second comparison and uses the `PAIR.ate2(Q2, P2, Q3, P3)` which is the optimal R-ate for double pairings and, in this case, returns the mapping for $e(z, X_2) \cdot e(\mathcal{F}(m), \sigma_3)$ which is compared against the mapping for $e(\sigma_1, g_2)$. I wrote `verify(..)` with the if statement as if the first check fails it will return false, given the second check does not need to be computed.

```
1  public static boolean verifyPairs(ECP P1, ECP2 Q1, ECP P2, ECP2 Q2) {
2
3    // Ate pairing e(Q, P) takes a point Q = (xQ, yQ) âĹĹ E(Fp12) and a point
4    // P = (xP, yP) âĹĹ E(Fp)
5    FP12 map1 = PAIR.ate(Q1, P1);
6
7    map1 = PAIR.fexp(map1);
8
9    FP12 map2 = PAIR.ate(Q2, P2);
10
11   map2 = PAIR.fexp(map2);
12
13   return map1.equals(map2);
14
15 }
16
17 public static boolean verifySingleWithDoublePairs(ECP P1, ECP2 Q1, ECP P2, E
       CP2 Q2, ECP P3, ECP2 Q3) {
18
19   FP12 map1 = PAIR.ate(Q1, P1);
20
21   map1 = PAIR.fexp(map1);
22
23   FP12 map2 = PAIR.ate2(Q2, P2, Q3, P3);
24
25   map2 = PAIR.fexp(map2);
26
27   return map1.equals(map2);
28
29 }
30
31 public static boolean verify(AsymmetricWatersSignatureSchemePublicParams asy
       mWaterSigPubParams, ECP F, AsymmetricWatersSignature sigs) {
32
33   if (AsymmetricWatersSignatureScheme.verifyPairs(sigs.getSig2(), asymWaterS
         igPubParams.getG2(), asymWaterSigPubParams.getG1(),
34   sigs.getSig3())) {
```

```
35
36      return AsymmetricWatersSignatureScheme.verifySingleWithDoublePairs(sigs.
            getSig1(), asymWaterSigPubParams.getG2(),
37      asymWaterSigPubParams.getZ(), asymWaterSigPubParams.getVk2(), F, sigs.ge
            tSig3());
38
39    }
40
41    return false;
42  }
```

Code Snippet 6: Asymmetric Waters Signature Scheme Verification

4.4 Implementing the Signatures on Randomisable Ciphertexts Scheme

After the building blocks were implemented, more importantly in this case the Elgamal Scheme and Asymmetric Waters Signature Scheme, then the SRC scheme could be implemented (section 3.3.1), given that it combines these schemes into a specific randomisation scheme. All algorithms of the SRC scheme were implemented. `eKeyGen(...)` that took an integer k, that represents the number of message bits as in BeleniosRF+, and returned a `SRCSchemeSecret Params(asymWaterSigParams, elgamalKeypair.getPk(), a new HashPoints(h1, h2), elga malKeypair.getDk())`, which held the Asymmetric Waters scheme parameters, the elgamal public and private keys and parameters h_1 and h_2 used for hashing a voters' verification key. `encrypt Plus(SRCSchemePublicParams srcSchemePublicParams, AsymmetricWatersSigVerification Keys vk, BitEnum[] message, BIG r)` enabling users to send their message, which would be waters hashed and encrypted with the election public key. As with how the parameters are given in the algorithms implemented in BeleniosRF+, which are defined in section 3.3.1, the same is achieved in the implementation by using the `SRCSchemePublicParams` class.

Code snippet 7 is a vital part of the BeleniosRF+ implementation as it is its randomisation service, the function that allows the voters' vote to be published and reveal no information about their vote, referred to as the Random^+ algorithm in figure 6. Lines 3-6 show the r' that's used to randomise the ciphertexts (c_1, c_2, c_3) and the s' that's used to randomise the signatures on ciphertexts (c_1, c_2, c_3). From lines 8-16 the `ElgamalAsymmetricGroup.randomiseCiphertext(...)` helper method is used to individually randomise c_1, c_2 and c_3, rather than randomising then directly in that method, they are then ready to be returned as (c'_1, c'_2, c'_3). Lines 17-37 is then the randomising of $(\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5)$ on the ciphertexts (c_1, c_2, c_3). Where `AsymmetricWatersSig natureScheme` helper method `randomiseSig(...)` is used, which is overloaded[3] as different signatures are elements in $\mathbb{G}_1$ and some elements in $\mathbb{G}_2$ and therefore `ECP` and `ECP2` are used, respectively. `AsymmetricWatersSignatureScheme.randomiseSig(ECP sig, ECP g, BIG randomS)` follows the structure of randomisation that would occur in the Asymmetric Waters Signature scheme, which is $\sigma'_i = \sigma_i \cdot a^{s'}$ where $a \in \mathbb{G}_1$ or $a \in \mathbb{G}_2$, and therefore the method takes a signature `sig` that will be point added to `g`, an element in $\mathbb{G}_1$, that will first be point multiplied against a random scalar `randomS`.

Note though the helper method does not do anything more than what the Asymmetric Waters Schemes sets out and that the SRC randomisation method is implemented to utilise the `ran domiseSig(...)` method, for example, lines 28-31 show the randomisation of signature σ_1 to σ'_1 is $\sigma_1 \cdot c_1^{s'} \cdot \sigma_3^{r'} \cdot g_1^{r' \cdot s'}$ and `randomiseSig(...)` is first used to compute $\sigma_1 \cdot c_1^{s'}$, where σ_1 is `sig` the signature provided, c_1 is `g` the element in $\mathbb{G}_1$ and s' is `randomS` the random scalar, and

[3]Overloading is a feature in Java where multiple methods can have the same name as long as there's more method arguments or at least one method argument is a different type. They can not have different return types though.

the returned result is stored in `sig1Rand`. Then $\sigma_3^{r'} \cdot g_1^{r' \cdot s'}$ is computed, again, using `randomiseSig(...)`, where $\sigma_3^{r'}$ is `sig`, g_1 is `g` and $(r' \cdot s')$ is `randomS`, and the result is immediately point added to the result previously stored in `sig1Rand`. To return the randomised ciphertexts and signature the `SRCSchemeCiphertextAndSignatures` class is used, which takes `SRCSchemeCiphertext` instance and a `SRCSchemeSignature` instance, which are two separate classes as different section of the SRCScheme or BeleniosRF+ algorithms sometimes require just the cipphertexts or signature and this approach allowed easier access.

```
1  public static SRCSchemeCiphertextAndSignatures randomisePlus(SRCSchemePublic
       Params srcSchemePubParams,
2          AsymmetricWatersSigVerificationKeys vk, SRCSchemeCipherText c, SRCSche
             meSignature sigs) {
3
4       BIG s = BIG.randomnum(srcSchemePubParams.getN(), srcSchemePubParams.getR
            ng());
5
6       BIG r = BIG.randomnum(srcSchemePubParams.getN(), srcSchemePubParams.getR
            ng());
7
8       ECP c1Rand = ElgamalAsymmetricGroup.randomiseCiphertext(c.getC1(), srcSc
            hemePubParams.getG1(), r);
9
10      ECP c2Rand = ElgamalAsymmetricGroup.randomiseCiphertext(c.getC2(), srcSc
            hemePubParams.getPk(), r);
11
12      // hashes verification key (a string of bits) to point on curve H
13      ECP hashPoint = srcHash(srcSchemePubParams, vk);
14
15      ECP c3Rand = ElgamalAsymmetricGroup.randomiseCiphertext(c.getC3(), hashP
            oint, r);
16
17      // sig1 + c1 * s + sig3 * r + g1 * r * r
18      BIG rMulSMOD = BIG.modmul(r, s, srcSchemePubParams.getN());
19
20      ECP sig1Rand = AsymmetricWatersSignatureScheme.randomiseSig(sigs.getSi
            g1(), c.getC1(), s);
21
22      sig1Rand.add(AsymmetricWatersSignatureScheme.randomiseSig(PAIR.G1mul(sig
            s.getSig3(), r), srcSchemePubParams.getG1(), rMulSMOD));
23
24      // sig2 + c2 * s + sig5 * r + P * r * s
25      ECP sig2Rand = AsymmetricWatersSignatureScheme.randomiseSig(sigs.getSi
            g2(), c.getC2(), s);
26
27      sig2Rand.add(AsymmetricWatersSignatureScheme.randomiseSig(PAIR.G1mul(sig
            s.getSig5(), r), srcSchemePubParams.getPk(), rMulSMOD));
28
29      // sig3 + g1 * s
30      ECP sig3Rand = AsymmetricWatersSignatureScheme.randomiseSig(sigs.getSi
            g3(), srcSchemePubParams.getG1(), s);
31
32      // sig4 + g2 * s
33      ECP2 sig4Rand = AsymmetricWatersSignatureScheme.randomiseSig(sigs.getSi
            g4(), srcSchemePubParams.getG2(), s);
34
35      // sig5 + pk * s
36      ECP sig5Rand = AsymmetricWatersSignatureScheme.randomiseSig(sigs.getSi
            g5(), srcSchemePubParams.getPk(), s);
37
38      SRCSchemeCipherText ciphertextRand = new SRCSchemeCipherText(c1Rand, c2R
            and, c3Rand);
39
```

```
40      SRCSchemeSignature signatureRand = new SRCSchemeSignature(sig1Rand, sig2
            Rand, sig3Rand, sig4Rand, sig5Rand);
41
42      return new SRCSchemeCiphertextAndSignatures(ciphertextRand, signatureRan
            d);
43    }
```

Code Snippet 7: SRC Random$^+$

4.5 Implementing the BeleniosRF+ Election Algorithms

After the SRC scheme was implemented it could then be used by the BeleniosRF+ algorithms. All defined algorithms defined section 3.3.1 were implemented, albeit some missing parts that were not possible with the time, such as the shuffling and decryption proofs in the Tally algorithm.

Voting. Code snippet 8 is the Vote algorithm in BeleniosRF+ and its primary target is to be ran on the voters device. The function takes the BeleniosRF+ parameters (public in this case), the voters id, voters' signing keypair and their vote as function arguments. A developer implementing a real election service would use this function on the clientside, after the voter had received their identification, signing keys and selected their vote via some interactive means. Line 3 is the randomly generated number, the one which a corrupt voter can not use with BeleniosRF+ to form a receipt, due to the randomisation service. Line 5 is where the `SRC_SCHEME.encryptPlus(...)` is used to encrypt the voters choice, which hashes it with waters hash and uses the random `BIG`, `r`, to encrypt, where the public encryption key $\boldsymbol{P} = (P_1, \ldots, P_k)$, k generated public keys which are pointed added $P = \prod_{i=1}^{k} P_i$ as defined in the BeleniosRF+ Setup algorithm. Continuing on line 5 the last argument given is a boolean `returnC2` set to false, that can be used to inform `encryptPlus(...)` to only compute and return the ciphertexts c_1 and c_3 (saving unnecessary computation), and used in this case because the voting algorithm computes c_2 separately, so that it can provide proofs for each message bit in v, but c_2 ends up being equal to the c_2 that would have been returned from `encryptPlus(...)` provided the P stated above is used; this can be seen in section 3.3.1 and in the current code snippet. Line 13 is a forloop where $(0 \leq i < k)$ where on lines 22-24 $c_{2,i} = P_i^r \cdot u_{i+1}^{m_i}$ and on lines 28-29 $\pi_i = \mathsf{DisjProve}_F^+(g_1, u_{i+1}, c_1, c_{2,i}, r)$. With each loop the most recently computed $c_{2,i}$ is added to the previously computed $c_{2,i}$'s, where the first addition before the forloop (line 9) was the addition of u_0. Therefore when the loop ends there are k $c_{2,i}$ partial encryptions, k π_i corresponding to each partial encryption and $c_2 := u_0 \prod_{i=1}^{k} c_{2,i}$ which is equal to $\mathcal{F}(m)P^r$. On line 34 `SRC_SCHEME.signPlus(...)` is used to sign (c_1, c_2, c_3) with the voters' signing key. Line 36 shows the final ballot which is returned.

```
1  vote : function(beleniosRFParams, id, upk, usk, v){
2
3    var r = BIG.randomnum(beleniosRFParams.asymGrpParams.n, beleniosRFParams.a
         symGrpParams.rng);
4
5    var c = SRC_SCHEME.encryptPlus(beleniosRFParams, upk, v, r, false);
6
7    var c2 = new ECP();
8
9    c2.add(beleniosRFParams.asymWaterSigParams.usigs[0]);
10
11   var c2Partials = []; var proofs = [];
12
13   for (i = 0; i < beleniosRFParams.asymWaterSigParams.k; i++) {
14
15     var m = new BIG(0);
16
17     if (v[i] == 1) {
18
```

```
19        m = new BIG(1);
20      }
21
22      c2Partials[i] = PAIR.G1mul(beleniosRFParams.pubKeys[i], r);
23
24      c2Partials[i].add(PAIR.G1mul(beleniosRFParams.asymWaterSigParams.usigs[i
            + 1], m));
25
26      c2.add(c2Partials[i]);
27
28      proofs[i] = NIZK_Disjunction.disjProveUsig(beleniosRFParams.asymGrpParam
            s, beleniosRFParams.asymWaterSigParams.usigs[i+1],
29        beleniosRFParams.pubKeys[i], v[i], c.c1, c2Partials[i], id, r);
30  }
31
32    var cWithC2Partials = {c1 : c.c1, c2 : c2, c2Partials: c2Partials, c3 : c.
          c3, proofs: proofs};
33
34    var sigs = SRC_SCHEME.signPlus(beleniosRFParams, usk, cWithC2Partials);
35
36    return {id : id, upk: upk, c : cWithC2Partials, sigs: sigs };
37  }
```

Code Snippet 8: Client Voting

4.6 Testing

The building blocks, SRC scheme and BeleniosRF+ election algorithms were tested for cases where the tests would be a success and for when they should fail. Tests were performed on both the Javascript and Java implementations. The way in which I structured my testing made it easy for me to retest a method/class after I made some alteration, as I was able to simply rerun the tests and determine whether it was now resulting in a success or failure.

Bilinear Map Testing. Code snippet 9 displays a `@Test` in `BilinearityTests` which was to show that the implementation was correct in terms of upholding the bilinear pairing properties (section 2.2.1). This test in particular, called `bilinearityCoeffFlipTest()`, was testing the $e(bQ, aP) = e(aQ, bP)$ property where $Q \in E(\mathbb{F}_{p^a})$, $P \in E(\mathbb{F}_p)$. Line 20 is for $e(bQ, aP)$ where `PAIR.fexp(PAIR.ate(bQ, aP))` computes the mapping to $(\mathbb{F}^*_{p^a})$ with `PAIR.fexp(...)` being the final exponentiation. Line 22 is the same as above but for the mapping $e(aQ, bP)$ where a and b have been swapped. The `PAIR.fexp(...)` called on lines 20 and 22 return `FP12` objects, which are elements in $\mathbb{F}^*_{p^a}$). They are compared and an assert true is used to verify their equality. The tests passed with an average test runtime of 0.26 seconds.

```
1  //e(aR, bS) = e(bR, aS)
2  @Test
3  public void () {
4
5  AsymmetricBilinearGroupParams params = new AsymmetricBilinearGroupParams();
6
7  RAND rng = InitRandom.initRandom();
8
9  BIG a = BIG.randomnum(params.getN(), rng);
10 BIG b = BIG.randomnum(params.getN(), rng);
11
12 ECP2 bQ = params.getG2().mul(b);
13
14 ECP aP =  params.getG1().mul(a);
15
```

```
16  ECP2 aQ = params.getG2().mul(a);
17
18  ECP bP =  params.getG1().mul(b);
19
20  FP12 g1MulAG2MulAPair = PAIR.fexp(PAIR.ate(bQ, aP));
21
22  FP12 g1MulBG2MulBPair = PAIR.fexp(PAIR.ate(aQ, bP));
23
24  assertTrue(g1MulAG2MulAPair.equals(g1MulBG2MulBPair));
25
26  }
```

Code Snippet 9: Bilinear Pairing Test

Elgamal Randomisation Tests. The test unit class `ElgamalAsymmetricGroupTests` which was used to test the class `ElgamalAsymmetricGroup` functionality, which is to encrypt, decrypt and randomise Elgamal ciphertexts (section 5). The code snippet 10 is testing the Elgamal randomisation, where a message is encrypted, randomised and then decrypted. The decrypted message is compared with the original message to ensure it's functioning correctly. `setup()` is ran before every test, which generates a new keypair, as well as a new message by multiplying the generator g_1 with a random scalar. Given the DDH assumption (section 2.3) which Elgamal is secure under, given the hardness in $\mathbb{G}_1$, when the message m is encrypted with some randomness r and then randomised with some unknown r', the first encryption can not be distinguished from the encryption when randomised. These assumptions are not easy to test and therefore if testing is used to show the encryption scheme has been implemented correctly, then we can be confident that it is secure, under these security assumptions.

```
1
2  private AsymmetricBilinearGroupParams params;;
3  private ECP m;
4  private G1KeyPair keypair;
5
6  @Before
7  public void setup() {
8
9    this.params = new AsymmetricBilinearGroupParams();
10
11   BIG r = BIG.randomnum(params.getN(), params.getRng());
12
13   m = new ECP();
14   m.copy(params.getG1());
15   m = m.mul(r);
16
17   this.keypair = ElgamalAsymmetricGroup.keyGen(params);
18
19 }
20
21 @Test
22 public void randomiseDecTest() {
23
24   ECPPair c = ElgamalAsymmetricGroup.encrypt(this.params, this.keypair.getP
         k(), this.m, null);
25
26   ECPPair cRand = ElgamalAsymmetricGroup.randomise(this.params, this.keypai
         r.getPk(), c, null);
27
28   ECP decM = ElgamalAsymmetricGroup.decrypt(this.params, cRand, this.keypai
         r.getDk());
29
30   assertTrue(m.equals(decM));
```

```
31
32 }
```

Code Snippet 10: Elgamal Randomisattion Test

NIZK Discrete Logarithm Equality Tests. Code snippet 11 was testing the NIZK Discrete Logarithm equality proof, as defined in section 2.8.3 and which is used in the BeleniosRF+ Tally algorithm (section 3.3.1) for proving correct decryption of the vote which is the Waters hash, by computing the proof and then verifying. In `setup()` `params` the asymmetric parameters are retrieved (line 4), `keypair` an Elgamal keypair is generated (line 6) and `message` a random message of bits of length k (line 8). In the `@Test watersHashMessageCorrectDecryptionTest()` the `message` is Waters hashed (line 18) and then encrypted (line 20). `NIZKLogarithmEquality.prove(AbstractAsymmetricBilinearGroupParams params, ECP g1, ECP y1, ECP g2, ECP y2, BIG x)`, where `g1` and `g2` are `ECP` and therefore are elements in group $\mathbb{G}_1$, `y1` is `g1` multiplied by scalar `x`, as is `g2`. The method is called with following arguments `NIZKLogarithmEquality.prove(params, params.getG1(), keypair.getPk(), c.getEcp1(), y2, keypair.getDk())`, where `y2` as seen in lines 22-24 is the second ciphertext c_2/S multiplied by F^{-1}, which with the additive operations is $S - F$) as seen on line 24. The same proof is then given as input, along with the same arguments, including $S - F$), as it's is a proof of correct decryption that's being verified and not just that `y1` and `y2` have the same exponential (scalar) x.

This test was important given that the sole verifiability of the election results relies on the NIZK discrete logarithm equality methods being able to be used for proof of decryption and that they're working correctly. To further verify this, test were ran with parameters that should fail. One test, displayed in code snippet 12, was where the waters hash F had been changed and to check that this would fail, which detects whether the Trustee has replaced a ciphertext with the encryption of a new hashed message m' in an attempt to manipulate the result.. Proof of decryption is the only way to detect this as the Trustee uses the Tally algorithm which strips the ballot of all identifiers, which means a votes integrity can longer be assured with the verification key.

```
1  @Before
2  public void setup() {
3
4    params = new AsymmetricBilinearGroupParams();
5
6    keypair = ElgamalAsymmetricGroup.keyGen(params);
7
8    message = TestUtils.randomVote(k);
9  }
10
11 @Test
12 public void watersHashMessageCorrectDecryptionTest() {
13
14   AsymmetricWatersSignatureSchemeParams asymSigParams = AsymmetricWatersSign
         atureScheme.setup(params, k);
15
16   try {
17
18     ECP F = AsymmetricWatersSignatureScheme.watersHash(asymSigParams.getUsig
           s(), message);
19
20     ECPPair c = ElgamalAsymmetricGroup.encrypt(params, keypair.getPk(), F, n
           ull);
21
22     ECP y2 = new ECP(); y2.copy(c.getEcp2());
23
```

```
24      y2.sub(F);
25
26      LogEqualityProof equalityProof = NIZKLogarithmEquality.prove(params, par
            ams.getG1(), keypair.getPk(), c.getEcp1(), y2, keypair.getDk());
27
28      assertTrue(NIZKLogarithmEquality.verify(params, params.getG1(), keypair.
            getPk(), c.getEcp1(), y2, equalityProof));
29
30    } catch (WatersSignatureException e) { e.printStackTrace(); }
31  }
```

Code Snippet 11: NIZK Discrete Logarith Equality Test

```
1   @Test
2   public void waterHashDifferentFailTest() {
3
4     AsymmetricWatersSignatureSchemeParams asymSigParams = AsymmetricWatersSign
          atureScheme.setup(params, k);
5
6     try {
7
8       ECP F = AsymmetricWatersSignatureScheme.watersHash(asymSigParams.getUsig
            s(), message);
9
10      ECP diffF = AsymmetricWatersSignatureScheme.watersHash(asymSigParams.get
            Usigs(), TestUtils.randomVote(k));
11
12      ECPPair c = ElgamalAsymmetricGroup.encrypt(params, keypair.getPk(), F, n
            ull);
13
14      ECP y2 = new ECP(); y2.copy(c.getEcp2());
15
16      y2.sub(F);
17
18      LogEqualityProof equalityProof = NIZKLogarithmEquality.prove(params, par
            ams.getG1(), keypair.getPk(), c.getEcp1(), y2,
19      keypair.getDk());
20
21      ECP y2DiffF = new ECP();
22      y2DiffF.copy(c.getEcp2());
23
24      y2DiffF.sub(diffF);
25
26      assertTrue(! (NIZKLogarithmEquality.verify(params, params.getG1(), keypa
            ir.getPk(), c.getEcp1(), y2DiffF, equalityProof)));
27
28    } catch (WatersSignatureException e) {e.printStackTrace();}
29  }
```

Code Snippet 12: NIZK Discrete Logarith Equality Fail Test

NIZK Disjunction Tests. The NIZK Disjunction was tested to ensure it was verifying for the standard $\mathsf{DisjProve}_F$ algorithm (section 2.8.3) and for the BeleniosRF+ extended $\mathsf{DisjProve}_F^+$ algorithm (section 3.3.1) where the bits of the message m were being muliplies with points u_i instead of g_1. It was also tested for cases where it should fail and these tests in particular made enabled an error to be spotted in the `challengeHash(ECP g, ECP pk, ECP R, ECP S, ECP U0, ECP V0, ECP U1, ECP V1, String id)` method which is used for the hash F in $\mathsf{DisjProve}_F^+$ and $\mathsf{DisjVerify}_F^+$ and because the same method is used in both algorithms the tests with normal values did not fail. In a full online voting service $\mathsf{DisjProve}_F^+$ would be ran on the voter device and $\mathsf{DisjVerify}_F^+$ on the voting server. This was a result, too, of not (despite being attempted)

testing between Javascript and Java (discussed further in section 5).
The code snippet 13 is testing that the implementation of $\mathsf{DisjVerify}_F^+$ fails when a currupt message is used, a message that is not a zero or a one, which simulates a malicious voter trying to manipulate an election result. Lines 10-18 an Elgamal encryption is computed with u^5 instead of u^0 or u^1. On line 20 a proof is created for $m = 1$, but where in terms of the ciphertext this is not the case. The proof is then verified to check that it does return false, which is expected.

```
1  @Test
2  public void disjUsigVoteOneCorrupMTest() {
3
4    BitEnum honestM = BitEnum.ONE;
5
6    BIG dishonestM = new BIG(5);
7
8    ECP u = params.getG1().mul(BIG.randomnum(params.getN(), params.getRng()));
9
10   BIG r = BIG.randomnum(params.getN(), params.getRng());
11
12   ECP c1 = params.getG1().mul(r);
13
14   ECP c2 = keypair.getPk().mul(r);
15
16   c2.add(u.mul(dishonestM));
17
18   DisjunctionProofIncRS proofIncRS = NIZKDisjunction.prove(params, u, keypai
         r.getPk(), honestM, c1, c2, id, r);
19
20   assertTrue(!NIZKDisjunction.verify(params, u, keypair.getPk(), c1, c2, id,
          proofIncRS.getProof()));
21 }
```

Code Snippet 13: NIZK Currupt message test

SRC Scheme Tests. The SRC scheme was tested, again, both with data values that should return true and false when being verified. The test in code snippet 14 tested the SRC scheme randomisation, which was essentially a test that the Asymmetric Waters Signature scheme block combined with the Elgamal randomisation scheme functioned correctly.

```
1  @Test
2  public void randomiseAndVerify() {
3
4  try {
5
6    SRCSchemeCipherText c = SRCScheme.encryptPlus(srcSchemePublicParams, asymS
         igKeys.getVK(), message, null, true);
7
8    SRCSchemeSignature sigs = SRCScheme.signPlus(srcSchemePublicParams, asymSi
         gKeys.getSk(), c);
9
10   SRCSchemeCiphertextAndSignatures cAndSigs = new SRCSchemeCiphertextAndSign
         atures(c, sigs);
11
12   SRCSchemeCiphertextAndSignatures randCAndSigs = SRCScheme.randomisePlus(sr
         cSchemePublicParams, asymSigKeys.getVK(), c, sigs);
13
14   assertTrue(SRCScheme.verifyPlus(srcSchemePublicParams, asymSigKeys.getVK()
         , randCAndSigs.getCiphertexts(), randCAndSigs.getSignatures(), true));
15
16 } catch (Exception e) { e.printStackTrace();}
17 }
```

Code Snippet 14: SRC Scheme Randomisation Test

5 Results and Evaluation

A core strategy I followed throughout my implementation was that after I implemented each building block that could be functional completely on its own, was to always test afterwards. My aim was to test the functionality of the class or method I had implemented, so I could be confident in the quality in the piece of software before moving on to other building blocks, or even higher level functionality that would be using those building blocks. This was an important decision I made early on, which was advantageous, as it was able to find mistake I had made in coding early on.

Reliability and Robustness. From a cryptographic means the algorithms have been shown to work when given correct data, but also have been tested with data where the algorithms should return false. Testing has shown that building blocks implemented and the SRC scheme built upon them function how they are expected, for example the SRC scheme test in section 4.6 presents that when a ciphertext is randomised and the signature on it, then the signature has been shown to still be verifiable on the ciphertext. The implemented SRC random method could be ran n times and still be verifiable.

Some subclasses of `AbstractAsymmetricBilinearGroupParams` is given as input to many methods, for example, `SRCSchemePublicParams` that actually extends `AsymmetricWatersSignatureSchemeParams` which then finally extends `AbstractAsymmetricBilinearGroupParams`. This enabled many methods to get all of the constant parameters for each election in one instance, where many of such instances are returned from other setup methods and thus limiting the possible errors made by users, which could result from having to maintain many individual parameters. Therefore given these object and the other

In many methods with loops that depend on the input, for example, with the implementation of the Vote algorithm the message bit array and usig array should be length of k and $k + 1$ respectively, then these lengths are not checked in the algorithm, but could be easily verified before calling the algorithm. When the lengths of the input are essential for security then they're checked, for example, in the Valid algorithm the c_2 partial encryptions array and their corresponding proofs array are both checked against the length of k, as in this case the inputs would be form untrusted sources. Another case is when decrypting in the Tally algorithm where the Waters Hashes have to be decrypted and a message may not be able to be found which does not match the Waters Hash, and therefore for this specific case an exception is used called `NoWatersHashMessageFound` which is throw when this occurs, though this should never happens as the message bits are verified int he Valid algorithm.

Performance. As discussed in section 3.2 BeleniosRF had the disadvantage of having an increased runtime on client devices due to the expensive Groth-Sahai proofs that were used, results in the BeleniosRF paper [24] presenting results where when using a 2016 iPhone model device the voting for $k = 25$ candidates took 22 seconds and therefore BeleniosRF was adapted to BeleniosRF+ where it used the the more efficient NIZK Fiat-Shamir proofs (section 2.8.3 and 3.3.1). I was able to run the BeleniosRF voting algorithm using an implementation at [1] and then run the BeleniosRF+ implementation to compare the result, and to observe whether there was a substantial improvement in the voting running times. As table 8 displays, a Macbook Air with a 1.8GHz i5 CPU and an iPhone 6s were used to, where multiple browsers were tested, which were Chrome, Firefox and Safari. The runtime for k being 1, 5, 10, and 25 was testing, with the table showing the results for k being 5 and 25. Ten tests were ran for each device, for each browser, for each k and the average was used. Immediately by viewing the Total Voting time column that BeleniosRF+ has a significant reduction in running time compared to Be-

leniosRF. For Macbook Air and averaging over the three browsers for $k = 5$ BeleniosRF+ is 78% and $k = 25$ it is, again, on average 78% faster when voting. For the iPhone 6s, averaging across all browsers with $k = 5$, BeleniosRF+ is 76% faster than BeleniosRF and for $k = 25$ is 75% faster. For example if k was 25 and the voter was using the chrome browser on the IPhone 6s, when voting using BeleniosRF+ they would wait on average 2.3 seconds, whereas when voting with BeleniosRF they would wait on average 9.3 seconds, which is significant gap for in terms of what users given todays technology expect. It's also worth noting that these times are only how long it takes to cryptographically compute the voting protocol given the necessary parameters, but in a fully implemented election service the voter would first have to wait for parameters to be download, webpages, etc over a network, further adding to the waiting time.

Improvements. Many parts the implementation presented could be used by someone to implement a complete online voting service with BeleniosRF+ as its backbone, only requiring the algorithms to be adapted for use in a database for when a voter *upk* and *id* needs checking and the Tally algorithm also requires shuffling and decryption proofs. The evidence displayed shows that most of the algorithms are functional and could instantly be used and individually, too, for other applications due to way the schemes were implemented offering schemes such as Elgamal Randomisation and Asymmetric Waters Signatures.

As mentioned above to be used the algorithms would require some extensions, and an improvement would be to add database support to the implementaton so people could easily use it with a given database, for example, making using of Java's hibernate packages. The code could be reviewed to make sure secure programming techniques were followed, by which I mean ensuring that no algorithms leak any information via side channels, though this would be dependent on the libraries that are used too, in particular the Milagro-Crypto library. A second would be to ensure that the methods were completely multithreaded safe, a necessity if being ran in real world election service, where there would be simultaneous voting. A substantial amount of testing was performed but even more vigorous testing could be applied, such as testing further security properties of the BeleniosRF+ protocol, such as testing the javascript and Java together so it is more realistic interpretation of how they would actually be used. An attempt was made to do this type of testing, by calling Javascript via Java, but unfortunately there were bugs and project timeline did not allow for finding a solution to the problem.

Version	Hardware	Platform	Browser	k	User Key Gen	Encryption	Proof	Signature	Voting Total
BeleniosRF	1.8GHz i5 4GB	OS 10.11	Chrome	5	68	64	3388	101	3553
BeleniosRF+	1.8GHz i5 4GB	OS 10.11	Chrome	5	73	136	490	104	729
BeleniosRF	1.8GHz i5 4GB	OS 10.11	Chrome	25	72	65	14203	101	14369
BeleniosRF+	1.8GHz i5 4GB	OS 10.11	Chrome	25	73	455	2322	100	2877
BeleniosRF	1.8GHz i5 4GB	OS 10.11	FireFox	5	47	45	2266	69	2380
BeleniosRF+	1.8GHz i5 4GB	OS 10.11	Firefox	5	51	89	334	70	493
BeleniosRF	1.8GHz i5 4GB	OS 10.11	FireFox	25	49	43	9224	66	9333
BeleniosRF+	1.8GHz i5 4GB	OS 10.11	Firefox	25	49	308	1589	69	1966
BeleniosRF	1.8GHz i5 4GB	OS 10.11	Safari	5	80	73	2497	90	2659
BeleniosRF+	1.8GHz i5 4GB	OS 10.11	Safari	5	64	113	397	78	589
BeleniosRF	1.8GHz i5 4GB	OS 10.11	Safari	25	66	71	9133	87	9292
BeleniosRF+	1.8GHz i5 4GB	OS 10.11	Safari	25	54	343	1770	76	2187
BeleniosRF	iPhone 6s	iOS 10	Chrome	5	48	45	2316	69	2430
BeleniosRF+	iPhone 6s	iOS 10	Chrome	5	56	103	372	82	557
BeleniosRF	iPhone 6s	iOS 10	Chrome	25	47	45	9212	73	9330
BeleniosRF+	iPhone 6s	iOS 10	Chrome	25	60	362	1861	87	2310
BeleniosRF	iPhone 6s	iOS 10	Firefox	5	47	44	2193	69	2306
BeleniosRF+	iPhone 6s	iOS 10	Firefox	5	56	108	373	80	560
BeleniosRF	iPhone 6s	iOS 10	Firefox	25	47	44	8871	69	8983
BeleniosRF+	iPhone 6s	iOS 10	Firefox	25	54	360	1831	77	2268
BeleniosRF	iPhone 6s	iOS 10	Safari	5	48	45	2244	70	2359
BeleniosRF+	iPhone 6s	iOS 10	Safari	5	56	101	373	77	551
BeleniosRF	iPhone 6s	iOS 10	Safari	25	48	44	9109	71	9223
BeleniosRF+	iPhone 6s	iOS 10	Safari	25	58	366	1826	76	2269

Figure 8: Voting Runtime on Client Devices (ms)

6 conclusion

This report presented cryptographic protocols such as the Elgamal encryption scheme with randomisation, the Asymmetric scheme which can sign messages and randomise the signature, Signatures on Randomisable Ciphertexts (SRC) scheme which combines Elgamal and Asymmetric waters signature to enable the randomisation of ciphertexts and the signatures on them and Non-Interactive proofs for prove a message is a zero or a one, or the equality of discrete logarithms. This was then followed online voting protocols that utilised some or all of the protocol presented, highlighting the BeleniosRF+ protocol which uses its SRC scheme to introduce the property of receipt-freeness. Then the implementation of the cryptographic protocols, the SRC scheme and the BeleniosRF+ protocol. The numerous tests of the algorithms showed that a voter would be able to verify their vote, while maintaining the privacy of the vote. Using runtime performance tests presented in the evaluation, it showed that BeleniosRF+ allowed far more efficient voting, while maintaining the election properties, which is an important result as the future of online voting will be providing the key online voting properties that should alway be present but also having services that are efficient.

Appendices

A Package Structure and Running the Software

The implemented software is located in the SVN repository:

`https://codex.cs.bham.ac.uk/svn/projects/2015/tgc127/`

The Javascript implementation:

`https://codex.cs.bham.ac.uk/svn/projects/2015/tgc127/Implementation/ClientSide/`

The Java implementation:

`https://codex.cs.bham.ac.uk/svn/projects/2015/tgc127/Implementation/ServerSide`.

The Miracl Milagro-Crpyto [22] library for Javascript and Java:
`https://codex.cs.bham.ac.uk/svn/projects/2015/tgc127/Implementation/ClientSide/WebApp/js/milagro_crypto/`

`https://codex.cs.bham.ac.uk/svn/projects/2015/tgc127/Implementation/ServerSide/BeleniosRF/milagro-crypto/`

The Java files are .java source files and can either be compiled along with the milagro-crypto library or both can be added to the build path of your workspace. The testing in `https://codex.cs.bham.ac.uk/svn/projects/2015/tgc127/Implementation/ServerSide/BeleniosRF/test/` can then be ran or the class and algorithms can be individually used by importing *beleniosrf*. The Javascript files can be ran in any modern browser and there are individual testing .html pages located at `https://codex.cs.bham.ac.uk/svn/projects/2015/tgc127/Implementation/ClientSide/WebApp/`. The Javascript files have to be in the root of your *index.html* file and added as script types in your *index.html*.

References

[1] Beleniosrf client voting implementation. https://dl.dropboxusercontent.com/u/16959859/bel-100/JS/index.html,, 2016.

[2] Latex tikz elliptic curve diagrams. http://www.iacr.org/authors/tikz/, 2016.

[3] O. d. M. Ben Adida and O. Pereira. Helios voting system. http://www.http://heliosvoting.org.

[4] D. Bernhard, O. Pereira, and B. Warinschi. *How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios*, pages 626–643. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[5] O. Blazy, G. Fuchsbauer, D. Pointcheval, and D. Vergnaud. *Signatures on Randomizable Ciphertexts*, pages 403–422. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[6] D. Boneh. *The Decision Diffie-Hellman problem*, pages 48–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[7] D. Boneh. Bilinear pairings in cryptography. https://www.youtube.com/watch?v=F4x2kQTKYFY, May 2013.

[8] D. Boneh, X. Boyen, and H. Shacham. *Short Group Signatures*, pages 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[9] D. Chaum and T. P. Pedersen. *Wallet Databases with Observers*, pages 89–105. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[10] V. Cortier, D. Galindo, S. Glondu, and M. Izabachène. Distributed elgamal à la pedersen: Application to helios. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 131–142, New York, NY, USA, 2013. ACM.

[11] R. Cramer, I. Damgård, and B. Schoenmakers. *Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols*, pages 174–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[12] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'97, pages 103–118, Berlin, Heidelberg, 1997. Springer-Verlag.

[13] A. J. Devegili, M. Scott, and R. Dahab. *Implementing Cryptographic Pairings over Barreto-Naehrig Curves*, pages 197–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[14] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 22(6):644 – 654, 1976.

[15] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 31(4):469 – 472, July 1986.

[16] A. Fiat and A. Shamir. *How To Prove Yourself: Practical Solutions to Identification and Signature Problems*, pages 186–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.

[17] D. Galindo. Beleniosrf+: Practical receipt-free e-voting, 2016.

[18] S. Glondu. Belenios. http://www.belenios.org/, September 2016.

[19] J. Groth and A. Sahai. *Efficient Non-interactive Proof Systems for Bilinear Groups*, pages 415–432. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[20] K. Kuppa. Introduction to elliptic curves. https://www.youtube.com/watch?v=njNPgMmghMo, January 2013.

[21] U. M. Maurer. *Towards the Equivalence of Breaking the Diffie-Hellman Protocol and Computing Discrete Logarithms*, pages 271–281. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[22] Miracl. Milgaro-crypto. https://github.com/miracl/milagro-crypto/, August 2016.

[23] W. V. Process. Mixnet example. https://wombat.factcenter.org/wp-content/uploads/2011/05/mixnet.jpg, 2016.

[24] G. F. Pyrros Chaidos, Véronique Cortier and D. Galindo. Beleniosrf: A non-interactive receipt-free electronic voting scheme. In editor, editor, *booktitle*.

[25] K. Sako and J. Kilian. *Receipt-Free Mix-Type Voting Scheme*, pages 393–403. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[26] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[27] S. G. V. Cortier, D. Galindo and M. Izabach'ene. Election verifiability for helios under weaker trust assumptions. *ESORICS*, 2014.

[28] B. Waters. *Efficient Identity-Based Encryption Without Random Oracles*, pages 114–127. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.