Swiss Online Voting System

Cryptographic proof of Individual Verifiability

April 7, 2017

SCYTL SECURE ELECTRONIC VOTING

Swiss Online Voting System
Cryptographic proof of Individual Verifiability

R&S

April 7, 2017

Revision History

Revision	Date	Author(s)	Description
0.1	29.06.16	SG	First version of the protocol
0.2	12.07.16	SG	Notation changes, privacy game
0.3	08.11.16	SG	First version with CaI games
0.4	11.11.16	SG	Correction of CaI games after review
0.5	24.11.16	SG	Second version privacy game
0.6	29.11.16	SG	Put together definitions and demos after review
0.7	30.11.16	SG	Notation changes after review
0.8	05.12.16	SG	Privacy game refactor
1.0	07.12.16	SG	Complete version
1.1	22.12.16	SG	Complete version with revision history
1.2	22.02.17	SG	Second version draft
1.3	23.03.17	SG	Update with new CaI proofs in progress
2.0	07.04.17	SG, JP	New complete version

Abstract

This document describes Scytl's electronic voting protocol implemented in the Swiss Online Voting System, which is focused on the provision of privacy and cast-as-intended verifiability. The document presents the protocol, defines its security properties and provides a formal analysis. It also provides some specific details on the implementation of the system.

Keywords: electronic voting protocols, binding election, cast-as-intended verifiability, malicious voting client, choice codes.

1 Introduction

Switzerland has a long history on direct participation of its citizens in decision making processes. Besides traditional elections where voters choose their representatives in the Federal Assembly, citizens can participate in several other voting events. Citizens can propose popular voting initiatives on their own (after having obtained enough popular support by collecting signatures), and then parties and governments themselves (at the communal, cantonal or federal level) can organize referendums in order to ask the citizens for their opinion on a new law or a modification of the Constitution, among others. At the end, Swiss citizens have the chance to participate in 3-4 voting processes a year in average.

Remote electronic voting was first introduced in Switzerland in three cantons: Geneva, Zurich and Neuchâtel [17]. The first binding trials were held in 2004. Since then, 14 cantons have offered the electronic voting channel to their electors, which until recently has been restricted to be used by up to 30% of the eligible voters (10

Verifiability in remote electronic voting is traditionally divided in three types, which are related to the phase of the voting process which is verified [1]. The first step to audit is the vote preparation at the voting client application run in the voter's device. This application is usually in charge of encrypting the selections made by the voter prior to casting them to a remote server so that their secrecy is ensured. *Cast-as-intended* verification methods provide the voters with means to audit that the vote prepared and encrypted by the voting client application contains what they selected, and that no changes have been performed. *Recorded-as-cast* verification methods provide voters with mechanisms to ensure that, once cast, their votes have been correctly received and stored at the remote voting server. Finally, *counted-as-recorded* verification allows voters, auditors and third party observers to check that the result of the tally corresponds to the votes which were received and stored at the remote voting server during the voting phase.

Classically, cast-as-intended and recorded-as-cast verifiability are referenced as *individually verifiable* mechanisms, while counted-as-recorded is referenced to be a *universally* verifiable method. The explanation of this division is simple: with individual verifiability only the voter knows that she had actually cast a vote, and the intended content, so the verification process cannot be done by third parties (universal). On the other hand, anybody should be able to verify the correct outcome of the election given the votes in the ballot box.

However, in the regulation introduced by the Federal Council, the individual and universal verifiability requirements are introduced in a slightly different way. Specifically, the Federal Council defines two types of verifiability in the regulation for e-voting with an specific trusting model:

Individual verifiability requires cast-as-intended verifiability and recorded-as-cast is assumed according to the following trust model:

- The server side of the voting platform is trusted.
- The client side and the communication channel between the client side and the server side is not trusted.
- A part of the voters may not be trustworthy.

Under this scope, the Federal Council requirement regarding verifiability is that an attacker cannot change the voter intention, prevent a vote from being stored, or cast a vote on its own, without detection from an honest voter that follows the verification protocol.

While this seems similar to the *usual* union of cast-as-intended and recorded-as-cast verifiability done in the literature, it differs from it due to the fact that in this model the server side is trusted, which is not the case when talking in general about recorded as cast mechanisms [13]. We can refer to it as a recorded as cast verification based on trusting the voting server.

Complete verifiability requires individual and universal verifiability using the following trust model:

- The server side of the voting platform is not trusted. Instead, there exists a group of so called control components which interact with it and which is trusted as a whole, under the assumption that at least one of them is reliable (each sole control component is not trusted).
- Same assumptions than for individual verifiability apply for voters, client side, and channel between the client side and the server side. Trusted model for the recorded-as-cast verifiability will be based on trusting the control components instead of the voting server.
- Given proofs generated by the system that will be verified by auditors, at least one of the auditors and her technical aids (software or hardware tools) are trusted to behave properly.

Taking into account this trust model, the Federal Council verifiability requirements for this type are many: an attacker cannot change a vote before/after it is stored, or prevent a vote from being stored, delete it from the ballot box, as well as insert new votes, without voters or auditors noticing it. These correspond to the previous requirements for individual verifiability, taking into account that the trusted part of the system is not the server, but the control components which interact with it. Additionally, voters must have to be able to verify whether their voting credentials have been used to cast a vote in the system. Finally, auditors must receive a proof that the result of the election corresponds to the votes cast by eligible voters and accepted by the system during the voting phase. All these requirements have to be fullfilled while vote and intermediate results secrecy is preserved.

In this case, the requirements for complete verifiability cover the classic cast-as-intended, recorded-as-cast and counted-as-recorded concepts, plus additional features (such as that each voter can verify her participation or not in the election). Note that, by the definition provided, the recorded-as-cast verification may not be restricted to be verified by the voter, but also by auditors which inspect the votes registered by the trusted part of the system (the control components).

According to the report by the Federal Council, systems to be used for up to the 50% of electors are required to provide methods for individual verifiability, and systems for up to 100% of the electorate are required to provide complete verifiability, while also enforcing the separation of duties on operations impacting the privacy, integrity and verifiability of the system.

Besides its requirements, the different electorate extents for which the the electronic voting system can be used define the level of authorization to be passed. Specifically, systems to be used for more than the 50% of the electors have to provide both security and symbolic proofs which demonstrate that the system fulfills the claimed properties.

1.1 Our contribution

In this paper we present the protocol of what we call the Swiss Online Voting System, which is the platform developed by Swiss Post and Scytl that is used in several Swiss Cantons. This protocol provides cast-as-intended verification, according to the requirements of the Federal Council (*Federal Chancellery Ordinance on Electronic Voting (VEleS)*) for systems to be used by up to 50% of the electorate. The protocol has the particularity of only allowing voters to cast one vote through the electronic channel, and therefore gives provisions for ensuring that such vote is considered to be cast only in case that it represents the voter intention, by means of a confirmation phase executed by the voter. After this step, the voter receives a confirmation from the server, which informs of the correct storage of the vote.

The protocol is an evolution of the so-called *Norwegian voting protocol* [18, 19, 24] that was used in the Norwegian elections in 2011 and 2013. Importantly, it substantially improves the Norwegian scheme by not needing to rely on the strong assumption that two independent server-side entities do not collude to preserve voter privacy. Furthermore, the scheme also represents a great performance improvement of the voting client application compared with the original Puiggali-Guasch scheme [2], from which the Norwegian scheme was initially derived. Besides the presentation of the protocol, this paper also includes the definitions and the assumptions under which the security properties of the scheme are proven, as described below.

Particularly, the security proof of our scheme is focused on proving cast-as-intended verifiability, given that the "weak" recorded-as-cast verifiability which is also required for the Federal Council is included straightforwardly in the proof: the cast-as-intended verification method requires the participation of the server side for the generation of the verification information during the voting phase. Otherwise, the client side could forge the required information and cheat to the voter. Given the fact that this verification information is generated from the vote cast by the voter, this means that the vote has to reach the server side for the cast-as-intended verification method to work. Finally, due to the assumption of a trustworthy server side, it can be assumed that any vote that reaches the server side is correctly stored.

1.2 Proving the security properties of the protocol

The next metodology is followed in order to prove the security properties of the protocol by means of both security and symbolic proofs, according to the authorization requirements of electronic voting systems to be approved by the Federal Council to be used by up to 50% of the electorate:

1. **Scheme definition:** includes the presentation of the protocol based on

algorithm and workflow descriptions (Section 3) along with the cryptographic primitives and building blocks used (Section 2).

2. **Cryptographic proof:** consists in the demonstration of the fulfillment of the security properties defined in Section 4, by the protocol instantiation from Section 3. Specifically, the property of cast-as-intended verifiability is analyzed.

3. **Symbolic proof:** consists in a particular representation of the protocol and the properties to prove, in such a way that logic clauses can be used to check that the protocol fulfills the claimed properties. For the scope of this project, the representation of protocol and properties will be done in a machine-readable language, in order to use specific software (ProVerif) for the verification. The preparation and representation of the symbolic proof is provided in another deliverable.

4. **Relation to the implementation design:** Annex A presents implementation details and information about the particular set up for the Swiss Post project, which are intended to be able to easily check the adequacy of the protocol model against the implementation design (closer to the code itself).

1.3 Results

The protocol has been proven in Section 4 to provide cast-as-intended verifiability. Given the format of the choice codes, it is important to take into account some considerations: a vote cannot contain repeated voting options, and the number of voting options inside has to be fixed. In case the voter can perform a variable number of selections, the maximum will be included in the vote, filling those not selected by the voter with blank options (which are all different). This means that the voter will receive choice codes for both the options she selected and the voting options she left blank. Finally, only one vote is allowed per voter, and this has also effect on the security of the cast-as-intended verification mechanism, as it is shown in the analysis. These considerations are all enforced by the protocol implementation.

2 Building blocks

The voting protocol uses the following building blocks. A more detailed explanation of how these blocks are implemented and their specific configuration can be found in the *Cryptographic Tools Specification* document:

2.1 Encryption schemes

Public key encryption scheme. Formally, a public key encryption scheme is defined by the algorithms $(\mathsf{Gen_e}, \mathsf{Enc}, \mathsf{Dec})$: the key generation algorithm $\mathsf{Gen_e}$ receives as input a security parameter 1^λ and outputs a key pair composed by a public key pk_e and a private key sk_e, defines a message space $\mathcal{M}_{sp}$, a ciphertext space $\mathcal{C}_{sp}$ and a randomness space $\mathcal{R}_{sp}$ (in case of a probabilistic encryption

scheme); the encryption algorithm Enc takes as input a message $m \in \mathcal{M}_{sp}$ and a public key pk_{e}, and computes a ciphertext $c \in \mathcal{C}_{sp}$. In case the algorithm is probabilistic, it uses random values $r \in \mathcal{R}_{sp}$ for computing such ciphertext; the decryption algorithm Dec receives as input a ciphertext $c \in \mathcal{C}_{sp}$ and a private key sk_{e}, and outputs a message $m \in \mathcal{M}_{sp}$ or $\perp$ in case of error.

Our protocol uses the ElGamal encryption scheme [16]: The key generation algorithm $\mathsf{Gen_e}$ takes as input a subgroup $\mathbb{G}$ which has a generator g of order q of elements in $\mathbb{Z}_p^*$, where p is a safe prime such that $p = 2q + 1$ and q is a prime number. It outputs an ElGamal public/secret key pair $(pk_{\mathsf{e}}, sk_{\mathsf{e}})$, where $pk_{\mathsf{e}} \in \mathbb{G}$ such that $pk_{\mathsf{e}} = g^{sk_{\mathsf{e}}} \mod p$ and $sk_{\mathsf{e}} \in \mathbb{Z}_q$. The encryption algorithm Enc receives as input a message $m \in \mathbb{G}$ and a public key pk_{e}, chooses a random $r \in \mathbb{Z}_q$ and computes $c = (c_1, c_2) = (g^r, pk_{\mathsf{e}}^r \cdot m)$. The decryption algorithm Dec receives c and the private key sk_{e} and outputs $m = c_2/c_1^{sk_{\mathsf{e}}}$.

Symmetric key encryption scheme. A symmetric key encryption scheme is defined by the algorithms $(\mathsf{KGen_e^s}, \mathsf{Enc^s}, \mathsf{Dec^s})$: $\mathsf{KGen_e^s}$ receives as input a security parameter 1^λ and outputs a symmetric key k from the key space $\mathcal{K}_{sp}$; $\mathsf{Enc^s}$ takes as input a message $m \in \{0,1\}^\lambda$ and a key $k \in \mathcal{K}_{sp}$, and produces a ciphertext $c_s \in \{0,1\}^\lambda$; finally the decryption algorithm $\mathsf{Dec^s}$ takes as input a ciphertext $c_s \in \{0,1\}^\lambda$ and a key $k \in \mathcal{K}_{sp}$, and produces a decrypted message $m \in \{0,1\}^\lambda$.

Our protocol uses the AES encryption scheme in GCM mode [23] for authenticated encryption/decryption. Particularly, this encryption mode provides a mechanism for checking the authenticity of encrypted data in the following way: given a message m, a key k and an authentication information a, the ciphertext and the authentication tag are computed as $c \leftarrow \mathsf{Enc^s}(m; k)$, $t_a \leftarrow \mathsf{ghash}(c, a)$, where ghash denotes a keyed hash function. Then, given the ciphertext c, the authentication information a and the authentication tag t_a, the authenticated decryption algorithm first checks that $t_a = \mathsf{ghash}(c, a)$ and if so, it runs $\mathsf{Dec^s}(c; k)$ to obtain the plaintext message m. Otherwise, it returns *failure*. We use this encryption mode without any authentication information, but still provides a check on the authenticity of encrypted data.

As analysed in [22] and [5], the privacy and authenticity properties of this cryptosystem rely on the fact that the underlying block cipher cannot be distinguished from a random function in case the secret key is not known, which is the case for the AES encryption algorithm.

2.2 Key derivation functions

The protocol uses a password-based key derivation function defined by the algorithm δ which, on input a password string pwd, a security parameter 1^λ and a salt salt, derives a cryptographic key K. Specifically, we use the PBKDF2 algorithm specified in [20], which additionally receives a number of iterations c and the output length dkLen. This algorithm derives the cryptographic key K using a pseudo-random function based on the hash algorithm SHA2-256 iterated c times over the concatenation of the password and the salt. The security of this primitive relies on the one-way and collision-resistance properties of the underlying hash function.

2.3 Signature schemes

A signature scheme is defined by three probabilistic algorithms $\mathsf{Gen_s}, \mathsf{Sign}, \mathsf{Verify}$, that stand for key generation, signature generation and signature verification. $\mathsf{Gen_s}$ receives as input a security parameter 1^λ, outputs a signing key pair $(pk_\mathtt{s}, sk_\mathtt{s})$ and defines a message space $\mathcal{M}_{sp}$ and a signature space $\mathcal{S}_{sp}$; Sign receives a message $m \in \mathcal{M}_{sp}$ and the signing private key $sk_\mathtt{s}$, and outputs a signature $\psi \in \mathcal{S}_{sp}$; Verify receives the signing public key $pk_\mathtt{s}$, a message $m \in \mathcal{M}_{sp}$ and a signature $\psi \in \mathcal{S}_{sp}$, and outputs 1 if the verification succeeds, 0 otherwise.

Our scheme uses the RSA Probabilistic Signature Scheme (PSS) ([6,8,25]), which is an RSA system with hash variant that uses a random padding: $\mathsf{Gen_s}$ receives two primes p, q of similar bit-length $(\lambda/2)$ (which define a ring $\mathbb{Z}/n\mathbb{Z}$) and computes the public key $pk_\mathtt{s} = (n, e)$, where $n = pq$ and e is coprime with $\phi(n)$ ($\phi(n) = (p-1)(q-1)$). The private key $sk_\mathtt{s}$ takes the value of d, where $ed = 1 \bmod \phi(n)$. Sign takes as input a message m, which is not restricted to a specific space, and the private key $sk_\mathtt{s}$, and outputs $\psi = (\mathtt{ME}(m))^d \bmod n$, where $\mathtt{ME}$ denotes a transformation with random padding over $\mathsf{H}_s(m)$ and H_s denotes a hash function which maps strings to elements in $\mathbb{Z}_n$. Verify takes as input the public key $pk_\mathtt{s}$, the message m and the signature ψ, and checks that $\mathtt{ME}(m) = \psi^e \bmod n$. It outputs 1 if the verification is successful, 0 otherwise.

This signature scheme is preferred instead of schemes with deterministic paddings such as RSA-FDH [7] given that it provides a tighter security proof [8, 12] for signature unforgeability in the Random Oracle Model (ROM).

2.4 Non-interactive zero-knowledge proofs of knowledge

Our protocol uses the Fiat-Shamir [15] transformation to turn interactive zero-knowledge protocols, such as σ-proofs, into non-interactive, by using a hash function to compute the random challenge. The security of the resulting non-interactive zero-knowledge proof of knowledge (NIZKPK) is based on the assumption made in the ROM that a hash function behaves as a random oracle. Therefore the challenge has a resulting distribution similar to the original and the non-interactive version of the ZKPK maintains its properties [7].

A NIZKPK scheme is composed by the algorithms ($\mathsf{GenCRS}, \mathsf{NIZKProve}$, $\mathsf{NIZKVerify}, \mathsf{NIZKSimulate}$): The common reference string generation algorithm GenCRS generates the parameters of the NIZKPK scheme. It receives as input a security parameter 1^λ and, in some cases, a mathematical group $\mathbb{G}$, and it outputs a common reference string crs. $\mathsf{NIZKProve}$ is the proof generation algorithm. It receives as input a common reference string crs, a statement x and a witness w, and outputs a proof π. $\mathsf{NIZKVerify}$ is the verification algorithm. It receives as input the common reference string crs, the statement x and the proof π, and outputs 1 if the verification is successful, 0 otherwise. $\mathsf{NIZKSimulate}$ is a proof simulation algorithm. It receives as input a (false) statement x^* and outputs a simulated proof π^*.

Our implementation for NIZKPKs uses the Maurer framework [21] for a generalized implementation. This framework defines a common procedure for constructing interactive proofs for statements presenting an homomorphism ϕ such that $\phi(a; x) \to a^x$. We provide concrete example. Let's say that a Prover wants to prove that it knows x such that $\phi(a; x) \to a^x$:

- Prover computes witness $t = \phi(a; s) = a^s$, where s is selected at random from the same value space than x, and sends it to the verifier.
- Verifier provides a random challenge h.
- Prover computes $z = s + x \cdot h$ and provides it to the verifier.
- Verifier checks that $(a^x)^h \cdot t = a^z$.

This procedure can be turned into non-interactive by computing the challenge as the hash function (Fiat-Shamir heuristic [15]) over some of the elements that participate in the proof generation, such as the input statement $\phi(a; x)$, the original value a, and some auxiliary string. Moreover, including the initial witness into the hash for computing the challenge h the resulting proof is shorter (given that the output of the hash is shorter than the output of the function ϕ. The procedure then is as follows:

- Prover computes witness $t = \phi(a; s) = a^s$, where s is selected at random from the same value space than x.
- Prover computes the challenge h as $h \leftarrow \mathsf{H}(a, \phi(a; x), t, aux)$.
- Prover computes $z = s + x \cdot h$ and provides it to the verifier, together with h.
- Verifier computes $h' = \mathsf{H}(a, a^x, (a^x)^{-h} \cdot a^z, aux)$ and checks whether $h' = h$.

Following this method for the generation of NIZKPKs, we proceed to describe the three types of NIZKPKs used in the protocol:

Equality of discrete logarithms. This is a generalization of the Chaum-Pedersen proof system [11] which we denote as EqDL:

- $\mathsf{ProveEq}((a_1, a_2, \ldots, a_n, a_1^x, a_2^x, \ldots, a_n^x), x)$[1] takes at random a value s from $\mathbb{Z}_q$, computes $a_1^s, a_2^s, \ldots, a_n^s$, $h = \mathsf{H}(a_1, a_2, \ldots, a_n, a_1^x, a_2^x, \ldots, a_n^x, a_1^s, a_2^s, \ldots, a_n^s)$ and $z = s + x \cdot h$, being H a hash function which maps strings to elements in $\mathbb{Z}_q$. The output proof $\pi_{\mathtt{eqDl}}$ is (h, z).
- $\mathsf{VerifyEq}((a_1, a_2, \ldots, a_n, a_1^x, a_2^x, \ldots, a_n^x), \pi_{\mathtt{eqDl}})$ computes $a_1^{s'} = a_1^z \cdot (a_1^x)^{-h}, a_2^{s'} = a_2^z \cdot (a_2^x)^{-h}, \ldots a_n^{s'} = a_\eta^z \cdot (a_n^x)^{-h}$, and checks that $h = \mathsf{H}(a_1, a_2, \ldots, a_n, a_1^x, a_2^x, \ldots, a_n^x, a_1^{s'}, a_2^{s'}, \ldots, a_n^{s'})$. If the validation is successful, the algorithm outputs 1. Otherwise it outputs 0.
- $\mathsf{SimEq}(a_1, a_2, \ldots, a_n, a_1^*, a_2^*, \ldots, a_n^*)$ takes z^* and h^* at random from $\mathbb{Z}_q$ and forms the proof π^*. In this kind of proof, a programmed random oracle has to be used for simulation such that when the adversary asks for the value $H(a_1, a_2, \ldots, a_n, a_1^*, a_2^*, \ldots, a_n^*, a_1^{s'}, a_2^{s'}, \ldots, a_n^{s'})$ the oracle returns the value h^*.

Knowledge of encryption exponent. Based on the Schnorr identification protocol [27], it is used for proving knowledge of the encryption exponent of the

[1] Our NIZKPK schemes particularly do not use a common reference string, and therefore GenCRS is not executed.

ElGamal ciphertext c. We denote it as ExpP, and its construction is similar to the particular case of the NIZK proof of equality of discrete logarithms where $n = 1$. Therefore:

- $\mathsf{ProveExp}((g, c_1, c_2), r)$ takes at random a value s from $\mathbb{Z}_q$, computes g^s, $h = \mathsf{H}(g, c_1, c_2, g^s)$ and $z = s + r \cdot h$. The output is $\pi_{\mathtt{sch}} = (h, z)$.
- $\mathsf{VerifyExp}((g, c_1, c_2), \pi_{\mathtt{sch}})$ computes $g^{s'} = g^z \cdot (c_1^{-h}$, and checks that $h = \mathsf{H}(g, c_1, c_2, g^{s'})$. If the validation is successful, the algorithm outputs 1. Otherwise it outputs 0.

The combination of ElGamal encryption together with a proof of knowledge of the encryption exponent is known as Signed ElGamal, and has been shown to be NM-CPA secure in [9].

Correct decryption. proofs of correct decryption are also based on the Chaum-Pedersen protocol. However, we use a different notation than in EqDL for simplicity in the protocol description. We denote them as DecP and describe the following algorithms:

- $\mathsf{ProveDec}((c, m), sk_{\mathsf{e}})$ receives a ciphertext $c = (c_1, c_2)$ and a witness sk_{e}, where $c_1 = g^r$ and $c_2 = pk_{\mathsf{e}}^r \cdot m$, being $pk_{\mathsf{e}} = g^{sk_{\mathsf{e}}}$. It takes at random s from $\mathbb{Z}_q$, computes $(g^r)^s$, g^s, $h = \mathsf{H}(c, m, (g^r)^s, g^s)$ and $z = s + sk_{\mathsf{e}} \cdot h$. The is $\pi_{\mathtt{dec}} = (h, z)$.
- $\mathsf{VerifyDec}((c, m), \pi_{\mathtt{dec}})$ computes $(g^r)^{s'} = (c_1)^z \cdot (c_2/m)^{-h}$ and $g^{s'} = g^z \cdot pk_{\mathsf{e}}^{-h}$, and checks that $h = \mathsf{H}(c, m, (g^r)^{s'}, g^{s'})$. If the validation is successful, the algorithm outputs 1. Otherwise it outputs 0.
- $\mathsf{SimDec}(c, m^*)$ takes at random z^* and h^* from $\mathbb{Z}_q$ and forms the proof π^*. As in the previous proof, a programmed random oracle has to be used for simulation such, that when the adversary asks for the value $\mathsf{H}(c, m^*, (g^r)^{s'}, g^{s'})$ the oracle returns the value h^*.

The properties of NIZKPKs are *completeness*, *soundness* and *zero-knowledge* [14], [26]. Informally, completeness tells us that given a proof generated by an honest prover, the verifier will always succeed on the verification. Soundness means that in case a dishonest prover generates a proof over an incorrect statement, the verification will fail with overwhelming probability. Finally, the property of zero-knowledge implies that the outputs of the proving and the simulation algorithms are indistinguishable.

2.5 Representation of the voting options

The voting options $\{v_1, \ldots, v_k\}$ are chosen as small bit-length primes belonging to the group $\mathbb{G}$ defined for the ElGamal encryption scheme. We use the operations of product $\prod$ and factorization fact for encoding/decoding the selected voting options: a vote ν is the product of voting options chosen by the voter prior to the encryption. After the votes are decrypted, the individual voting options are recovered by factorizing the resulting value. Therefore, it has to be ensured that the product of t of such primes, where t is the number of selections

a voter can make, is smaller than p. Only pre-configured voting options which are represented as primes are considered in the protocol description and in the security analysis.

2.6 Pseudo-random functions

A function family is a map $F : T \times D \to R$, where T is the set of keys, D is the domain and R is the range. A pseudorandom function family (PRF) is a family of efficiently computable functions with the following property: a random instance of the family is computationally indistinguishable from a random function, as long as the key remains secret. The function $f_K(x) = y$ denotes a function f from a family F, parameterized by a key K.

Two keyed pseudo-random functions are used in the protocol: we denote by $f_k()$ an HMAC function composed by a SHA-256 hash function, parameterized by the symmetric key k. As detailed in [4], HMAC is a PRF whose resistance against collision is the one of the underlying hash scheme. Up to date, the collision probability of the SHA-256 hash function is considered to be negligible. The second pseudo-random function is the exponentiation function $g \to g^k$. This function is defined in our scheme for the group $\mathbb{G}$ (the same used in the ElGamal encryption scheme) and computed over the small primes representing the voting options.

2.7 Verifiable mixnet

A verifiable mixnet is composed by two algorithms: the algorithm Mix receives a set of ciphertexts $\mathcal{C} = \{c_1, \ldots, c_\ell\}$ as input, and outputs a set of ciphertexts $\mathcal{C}' = \{c'_1, \ldots, c'_\ell\}$ and a proof $\pi_{\texttt{mix}}$ of correct mixing. These ciphertexts correspond to the input values, randomly permuted and re-encrypted or partially decrypted, depending on the type of mixnet. The algorithm MixVerify receives as input two sets of ciphertexts $\mathcal{C}$ and $\mathcal{C}'$ and the proof of correct mixing $\pi_{\texttt{mix}}$, and outputs 1 or 0 depending on the result of the verification.

In our protocol we use the verifiable re-encryption mixnet proposed by Stephanie Bayer and Jens Groth [3]. This mixnet has been proven by its authors to be sound, meaning that MixVerify will output 0 given an incorrect execution of Mix with overwhelming probability, and zero-knowledge both in the standard model and in the random oracle model in case of using the Fiat-Shamir heuristic for making the proofs non-interactive.

3 Protocol

In this section we define a protocol where the system provides to the voter a proof of content of the vote she has cast, consisting on a set of Choice Codes which she can check in her voting card. In case the verification is successful, the voter proceeds to confirm her vote.

For simplicity, we do not take into account the provision of credentials to the voter, but we assume that she will be authenticated by some external means. The scheme can later be easily enhanced with the use of credentials (for example,

to digitally sign the vote). Appendix A explains how it is done in the OV Swiss system.

The following are the participants of the voting protocol:

- *Election Authorities*: they are in charge of configuring the election and tallying the votes to produce the election result.
- *Registrars*: they register the voters and provide them with information for participating in the election.
- *Voter*: they participate in the election by choosing their preferred options.
- *Voting Devices*: they generate and cast votes given the voting options selected by the voters.
- *Bulletin Board Manager*: it receives, processes and publishes the votes cast by the voters in the bulletin board BB, as well as the election results.
- *Auditors*: they are responsible for verifying the integrity of the procedures run in the counting phase.

We denote the set of valid votes by Ω, which is composed by any combination of voting options $\{v_1, \ldots, v_k\}$ which is valid according to the election rules. Both the set of valid votes and the counting function $\rho : (\Omega \cup \{\perp\})^* \rightarrow R$ are assumed to be defined in advance by the election authorities. Particularly, the present definition is based on a mixnet-based system, and therefore the set of possible results R is given by the multiset function ρ, which provides the cleartext votes cast by the voters in a random order [10].

The voting protocol uses an encryption scheme $(\mathsf{Gen_e}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{EncVerify})$, a signature scheme $(\mathsf{Gen_s}, \mathsf{Sign}, \mathsf{Verify})$, and three NIZKPK schemes: $(\mathsf{ProveExp}, \mathsf{VerifyExp})$, $(\mathsf{ProveEq}, \mathsf{VerifyEq}, \mathsf{SimEq})$ and $(\mathsf{ProveDec}, \mathsf{VerifyDec}, \mathsf{SimDec})$. Additionally, it uses two keyed pseudo-random functions: the HMAC function denoted by $f_k()$ and the exponentiation function in $\mathbb{G}$, which also has homomorphic properties; a verifiable mixnet with algorithms $(\mathsf{Mix}, \mathsf{MixVerify})$; a symmetric encryption scheme $(\mathsf{KGen^s_e}, \mathsf{Enc^s}, \mathsf{Dec^s})$; a derivation function δ and a hash function H.

We also define the following value spaces:

- $\mathcal{A}_{lc}$: space of possible values of long Choice and Vote Cast Codes, which is the order of the group $\mathbb{G}$ and is equal to q.
- $\mathcal{A}_{cc}$: space of possible values of short Choice Codes, which is 10^4 since they are 4-digit values.
- $\mathcal{A}_{bck}$: space of possible values of Ballot Casting Keys, which is 10^8 since they are 8-digit values.
- $\mathcal{A}_{vcc}$: space of possible values of short Vote Cast Codes, which is 10^8 since they are 8-digit values.

- $\mathcal{A}_{svk}$: space of possible values of Start Voting Keys, which is 32^{20} since they are 20 Base32-encoded characters.

The protocol is characterized by the following algorithms:

$\mathsf{Setup}(1^\lambda)$ receives as input a security parameter 1^λ and generates:

- An Electoral Board key pair $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}) \leftarrow \mathsf{Gen}_{\mathsf{e}}(1^\lambda)$. Alternatively, $\mathtt{EB}_{\mathtt{sk}}$ may consist of the shares of the generated private key sk_{e} if there are several Electoral Board members.
- A Codes secret key $\mathtt{C}_{\mathtt{sk}} \xleftarrow{\$} T$, where T denotes the set of possible keys of the PRF function f.
- A Vote Cast Code Signer key pair $(\mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Gen}_{\mathsf{s}}(1^\lambda)$.

The public outputs $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{pk}})$ are input to the next algorithms although not specified.

$\mathsf{Register}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$ takes as input a security parameter 1^λ and the private keys $\mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}$, and performs the following operations:

- Generates a Start Voting Key $\mathtt{SVK}_{\mathtt{id}} \xleftarrow{\$} \mathcal{A}_{svk}$.
- Generates a Voting Card ID $\mathtt{VC}_{\mathtt{id}} \leftarrow \delta(\mathtt{SVK}_{\mathtt{id}}, \mathtt{IDseed})$.
- Generates a keystore password $\mathtt{KSpwd}_{\mathtt{id}} \leftarrow \delta(\mathtt{SVK}_{\mathtt{id}}, \mathtt{KEYseed})$.

– Generation of Verification Card data:

- Generates the Verification Card key pair: $(\mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}) \leftarrow \mathsf{Gen}_{\mathsf{e}}(1^\lambda)$ (although this is formally an encryption key pair, it will be used differently in the next steps of the protocol).
- Computes the encryption of the Verification Card private key with the keystore password: $\mathtt{VCks}_{\mathtt{id}} \leftarrow \mathsf{Enc}^{\mathsf{s}}(\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}, \mathtt{KSpwd}_{\mathtt{id}})$.

- Verification Card codes generation:

  - Chooses at random a Ballot Casting Key $\mathtt{BCK}^{\mathtt{id}} \xleftarrow{\$} A_{bck}$.
  - For each voting option $v_i \in \{v_1, \ldots, v_k\}$ it computes a long Choice Code $\mathtt{CC}^{\mathtt{id}}_i = f_{\mathtt{C}_{\mathtt{sk}}}(v_i^{\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}})$ and a short Choice Code $\mathtt{sCC}^{\mathtt{id}}$ taken at random from the value space $\mathcal{A}_{cc}$. The short Choice Codes are checked to be unique in the set for one voter.
  - Computes a long Vote Cast Code $\mathtt{VCC}^{\mathtt{id}} = f_{\mathtt{C}_{\mathtt{sk}}}(((\mathtt{BCK}^{\mathtt{id}})^2)^{\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}})$ and a short Vote Cast Code $\mathtt{sVCC}^{\mathtt{id}}$ taken at random from the value space $\mathcal{A}_{vcc}$.
  - Computes the signature value of the short Vote Cast Code $S_{\mathtt{VCC}^{\mathtt{id}}} \leftarrow \mathsf{Sign}(\mathtt{sVCC}^{\mathtt{id}}, \mathtt{VCCs}_{\mathtt{sk}})$.
  - Computes the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ consisting on pairs *(reference value - encrypted code)* for the Choice Codes, as well as for the Vote Cast Code and signature:

$$\{[\mathsf{H}(\mathtt{CC}_i^{\mathtt{id}}), \mathsf{Enc}^{\mathsf{s}}(\mathtt{sCC}_i^{\mathtt{id}}; \mathtt{CC}_i^{\mathtt{id}})]\}_{i=1}^{k}$$
$$[\mathsf{H}(\mathtt{VCC}^{\mathtt{id}}), \mathsf{Enc}^{\mathsf{s}}((\mathtt{sVCC}^{\mathtt{id}} | S_{\mathtt{VCC}^{\mathtt{id}}}); \mathtt{VCC}^{\mathtt{id}})]$$

Finally it outputs the generated information: $\mathtt{SVK}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{VCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^{k}, \mathtt{CM}_{\mathtt{id}}$.

$\mathsf{GetID}(\mathtt{SVK}_{\mathtt{id}})$ On input the Start Voting Key $\mathtt{SVK}_{\mathtt{id}}$, it does the following actions:

– Generates the Voting Card ID $\mathtt{VC}_{\mathtt{id}} \leftarrow \delta(\mathtt{SVK}_{\mathtt{id}}, \mathtt{IDseed})$.

It returns the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$.

$\mathsf{GetKey}(\mathtt{SVK}_{\mathtt{id}}, \mathtt{VCks}_{\mathtt{id}})$ On input the Start Voting Key $\mathtt{SVK}_{\mathtt{id}}$ and the Verification Card keystore $\mathtt{VCks}_{\mathtt{id}}$, it does the following actions:

– Generates the keystore password $\mathtt{KSpwd}_{\mathtt{id}} \leftarrow \delta(\mathtt{SVK}_{\mathtt{id}}, \mathtt{KEYseed})$.
– Runs the $\mathsf{Dec}^{\mathsf{s}}$ algorithm with inputs $\mathtt{VCks}_{\mathtt{id}}$ and $\mathtt{KSpwd}_{\mathtt{id}}$, and recovers the Verification Card private key $\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}$.

It returns the Verification Card private key $\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}$.

$\mathsf{CreateVote}(\mathtt{VC}_{\mathtt{id}}, \{v_1, \ldots, v_t\}, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}})$ takes as input the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, a set of voting options selected by the voter $\{v_1, \ldots, v_t\}$ and the Verification Card key pair $(\mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}})$, and does the following:

- Computes the aggregation of the voter's selections: $\nu = \prod_{\ell=1}^{t}(v_\ell)$.
- Encrypts the previous result: $c = (c_1, c_2) \leftarrow \mathsf{Enc}(\nu, \mathtt{EB}_{\mathtt{pk}})$.
- Generates a proof of knowledge of the encryption exponent $\pi_{\mathtt{sch}} \leftarrow \mathsf{ProveExp}((g, c_1, c_2), r)$, where r is the encryption randomness used to compute c.
- Computes partial Choice Codes as $\{\mathtt{pCC}_\ell^{\mathtt{id}}\}_{\ell=1}^{t} = (v_1^{\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}}, \ldots, v_t^{\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}})$.
- Computes $\tilde{c} = (\tilde{c}_1, \tilde{c}_2) = (c_1^{\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}}, c_2^{\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}})$.
- Generates two NIZKPK proofs to prove that the voting options in the ciphertext c and the voting options used for computing the partial Choice Codes are the same:
  $\pi_{\mathtt{exp}} = \mathsf{ProveEq}((g, c_1, c_2, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \tilde{c}_1, \tilde{c}_2), \mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}})$ proves that $\tilde{c}$ is computed by raising the elements of c to the private key $\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}$ corresponding to the public key $\mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}$.
  $\pi_{\mathtt{pleq}} = \mathsf{ProveEq}\left(\left(g, \mathtt{EB}_{\mathtt{pk}}, \tilde{c}_1, \frac{\tilde{c}_2}{\prod_{\ell=1}^{t}(\mathtt{pCC}_\ell^{\mathtt{id}})}\right), r \cdot \mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}\right)$ which proves that $\tilde{c}$ is equivalent to the encryption of the product of partial Choice Codes $\{\mathtt{pCC}_\ell^{\mathtt{id}}\}_{\ell=1}^{t}$ under the Electoral Board public key $\mathtt{EB}_{\mathtt{pk}}$.

Let $\alpha \leftarrow c$, $\beta \leftarrow \{\mathtt{pCC}_\ell^{\mathtt{id}}\}_{\ell=1}^{t}$, $\gamma \leftarrow (\tilde{c}, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \pi_{\mathtt{sch}}, \pi_{\mathtt{exp}}, \pi_{\mathtt{pleq}})$. The output of this algorithm is the vote $\mathtt{V} = (\alpha, \beta, \gamma)$.

$\mathsf{ProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ receives as input a bulletin board BB, a Voting Card ID $\mathtt{VC}_{\mathtt{id}}$ and a vote $\mathtt{V}$, and proceeds to do the following checks: first it verifies that there is not already a vote in BB for the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$ and that the public key $\mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}$ in the vote belongs to that voter according to the corresponding entry in BB. The algorithm continues by validating the NIZKPK proofs $\pi_{\mathtt{sch}}, \pi_{\mathtt{exp}}, \pi_{\mathtt{pleq}}$ from the vote $\mathtt{V}$ running:

- $\mathsf{VerifyExp}((g, c_1, c_2), \pi_{\mathtt{sch}})$
- $\mathsf{VerifyEq}((g, c_1, c_2, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \tilde{c}_1, \tilde{c}_2), \pi_{\mathtt{exp}})$
- $\mathsf{VerifyEq}\left(\left(g, \mathtt{EB}_{\mathtt{pk}}, \tilde{c}_1, \frac{\tilde{c}_2}{\prod_{\ell=1}^{t}(\mathtt{pCC}_{\ell}^{\mathtt{id}})}\right), \pi_{\mathtt{pleq}}\right)$

In case any of the validations fail, it stops and outputs 0. Otherwise, it outputs 1.

$\mathsf{CreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}})$ takes as input the vote $\mathtt{V}$, the Codes secret key $\mathtt{C}_{\mathtt{sk}}$, the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$, and does the following actions:

- Computes a long Choice Code value $\overline{\mathtt{CC}_{\ell}^{\mathtt{id}}} = f_{\mathtt{C}_{\mathtt{sk}}}(\mathtt{pCC}_{\ell}^{\mathtt{id}})$ for each of the partial Choice Codes $\{\mathtt{pCC}_{\ell}^{\mathtt{id}}\}_{\ell=1}^{t}$ in $\mathtt{V}$.
- Checks that for each Choice Code $\overline{\mathtt{CC}_{\ell}^{\mathtt{id}}}$, $\mathsf{H}(\overline{\mathtt{CC}_{\ell}^{\mathtt{id}}}) \in \{\mathsf{H}(\mathtt{CC}_{i}^{\mathtt{id}})\}_{i=1}^{k}$, where $\ell = 1, \ldots, t$, and for each one it recovers the short value $\overline{\mathtt{sCC}_{\ell}^{\mathtt{id}}}$ running the decryption algorithm $\mathsf{Dec}^{\mathsf{s}}$ with $\overline{\mathtt{CC}_{\ell}^{\mathtt{id}}}$ as the symmetric key. In a positive case, the output of the algorithm is the set of decrypted short Choice Codes $\{\overline{\mathtt{sCC}_{\ell}^{\mathtt{id}}}\}_{\ell=1}^{t} = (\overline{\mathtt{sCC}_{1}^{\mathtt{id}}}, \ldots, \overline{\mathtt{sCC}_{t}^{\mathtt{id}}})$. Otherwise, the output is $\perp$.

$\mathsf{GetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}})$ takes as input the bulletin board BB, a Voting Card ID $\mathtt{VC}_{\mathtt{id}}$ and the Codes secret key $\mathtt{C}_{\mathtt{sk}}$, and does the following actions:

- Checks whether there is an entry in the bulletin board corresponding to the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$.
- Retrieves the vote $\mathtt{V}$ from the bulletin board corresponding to the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$. If there is no vote, the algorithm stops and returns $\perp$.
- Retrieves the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ from the bulletin board.
- Computes a long Choice Code value $\overline{\mathtt{CC}_{\ell}^{\mathtt{id}}} = f_{\mathtt{C}_{\mathtt{sk}}}(\mathtt{pCC}_{\ell}^{\mathtt{id}})$ for each of the partial Choice Codes $\{\mathtt{pCC}_{\ell}^{\mathtt{id}}\}_{\ell=1}^{t}$ in $\mathtt{V}$.
- Checks that for each Choice Code $\overline{\mathtt{CC}_{\ell}^{\mathtt{id}}}$, $\mathsf{H}(\overline{\mathtt{CC}_{\ell}^{\mathtt{id}}}) \in \{\mathsf{H}(\mathtt{CC}_{i}^{\mathtt{id}})\}_{i=1}^{k}$, where $\ell = 1, \ldots, t$, and for each one it recovers the short value $\overline{\mathtt{sCC}_{\ell}^{\mathtt{id}}}$ running the decryption algorithm $\mathsf{Dec}^{\mathsf{s}}$ with $\overline{\mathtt{CC}_{\ell}^{\mathtt{id}}}$ as the symmetric key. In a positive case, the output of the algorithm is the set of decrypted short Choice Codes $\{\overline{\mathtt{sCC}_{\ell}^{\mathtt{id}}}\}_{\ell=1}^{t} = (\overline{\mathtt{sCC}_{1}^{\mathtt{id}}}, \ldots, \overline{\mathtt{sCC}_{t}^{\mathtt{id}}})$. Otherwise, the output is $\perp$.

$\mathsf{Confirm}(\mathtt{VC}_{\mathtt{id}}, \mathtt{V}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}, \mathtt{BCK}^{\mathtt{id}})$ receives as input a Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, a vote $\mathtt{V}$, the Verification Card private key $\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}$ and the voter's Ballot Casting Key $\mathtt{BCK}^{\mathtt{id}}$, and outputs a confirmation message: $\mathtt{CM}^{\mathtt{id}} = ((\mathtt{BCK}^{\mathtt{id}})^2)^{\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}}$.

$\mathsf{ProcessConfirm}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}})$ receives as input a bulletin board BB, a Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, a confirmation message $\mathtt{CM}^{\mathtt{id}}$, the Codes secret key $\mathtt{C}_{\mathtt{sk}}$ and the Vote Cast Code Signer public key $\mathtt{VCCs}_{\mathtt{pk}}$, and performs the following steps:

- Checks that there is a vote entry in BB for the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, and that it has not been confirmed yet.
- Computes the long Vote Cast Code as $\overline{\mathtt{VCC}^{\mathtt{id}}} = f_{\mathtt{C}_{\mathtt{sk}}}(\mathtt{CM}^{\mathtt{id}})$.
- Takes the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ from the bulletin board, looks for the pair $[\mathsf{H}(\mathtt{VCC}^{\mathtt{id}}), \mathsf{Enc}^{\mathsf{s}}((\mathtt{sVCC}^{\mathtt{id}} | S_{\mathtt{VCC}^{\mathtt{id}}}); \mathtt{VCC}^{\mathtt{id}})]$ for which $\mathsf{H}(\mathtt{VCC}^{\mathtt{id}})$ is equal to the computed value $\mathsf{H}(\overline{\mathtt{VCC}^{\mathtt{id}}})$ and recovers the short Vote Cast Code $\overline{\mathtt{sVCC}^{\mathtt{id}}}$ and the signature $\overline{S_{\mathtt{VCC}^{\mathtt{id}}}}$, using the decryption algorithm $\mathsf{Dec}^{\mathsf{s}}$ with $\overline{\mathtt{VCC}^{\mathtt{id}}}$ as the key.
- Checks that the retrieved short Vote Cast Code is correct by running $\mathsf{Verify}(\mathtt{VCCs}_{\mathtt{pk}}, \overline{\mathtt{sVCC}^{\mathtt{id}}}, \overline{S_{\mathtt{VCC}^{\mathtt{id}}}})$.

In case all the verifications succeed, the output of the algorithm is the pair $(\overline{\mathtt{sVCC}^{\mathtt{id}}}, \overline{S_{\mathtt{VCC}^{\mathtt{id}}}})$. Otherwise, the output is $\perp$.

$\mathsf{Tally}(\mathsf{BB}, \mathtt{EB}_{\mathtt{sk}})$ takes as input the bulletin board BB and the Electoral Board private key $\mathtt{EB}_{\mathtt{sk}}$, and runs several processes:

- Cleansing:
  - Validation of votes in the bulletin board: it runs $\mathsf{ProcessVote}$ for all the votes in the bulletin board which have a pair Vote Cast Code $\overline{\mathtt{sVCC}^{\mathtt{id}}}$ and signature $\overline{S_{\mathtt{VCC}^{\mathtt{id}}}}$ stored, and then it runs $\mathsf{Verify}(\mathtt{VCCs}_{\mathtt{pk}}, \overline{\mathtt{sVCC}^{\mathtt{id}}}, \overline{S_{\mathtt{VCC}^{\mathtt{id}}}})$, discarding those for which this validation failed.
- Mixing:
  - The list of ciphertexts $\mathcal{C} = \{c_1, \ldots, c_n\}$ is composed by extracting the ciphertexts from the votes which passed the previous validations, and it is passed as input to the mixnet, which runs the Mix algorithm and outputs a list of mixed ciphertexts $\mathcal{C}'$ and a proof of correct mixing $\pi_{\mathtt{mix}}$.
- Decryption of ciphertexts and recovery of voting options:
  - For each ciphertext $c'_i \in \mathcal{C}'$, $\mathsf{Dec}(c'_i, \mathtt{EB}_{\mathtt{sk}})$ is run to obtain ν_i and $\mathsf{ProveDec}((c'_i, \nu_i), \mathtt{EB}_{\mathtt{sk}})$ is run to produce a decryption proof $\pi_{\mathtt{dec}_i}$. Then $\mathsf{fact}(\nu_i)$ is run to output the factors $\{v_{i\ell}\}_{\ell=1}^{t}$ composing ν_i, for which it is tested that the combination of voting options $\{v_{i\ell}\}_{\ell=1}^{t} \in \Omega$. Otherwise, the whole factorized vote is discarded.

The outputs are the list of decrypted and factorized votes $r = \{(v_{11}, \ldots, v_{1t}), \ldots, (v_{n1}, \ldots, v_{nt})\}$ and the information Π necessary to verify the tally process, consisting on the proof of correct mixing $\pi_{\texttt{mix}}$, the list of mixed votes $\mathcal{C}'$ and the list of decryption proofs $\Pi_{\texttt{dec}} = \{\pi_{\texttt{dec}_1}, \ldots, \pi_{\texttt{dec}_n}\}$.

$\mathsf{VerifyTally}(\mathsf{BB}, r, \Pi)$ takes as input the bulletin board BB, the tally result r and the proof Π of correct tally. Initially it performs the same validations as Tally: it runs $\mathsf{ProcessVote}$ for all the votes in the bulletin board which have a pair Vote Cast Code $\overline{\texttt{sVCC}^{\texttt{id}}}$ and signature $\overline{S_{\texttt{VCC}^{\texttt{id}}}}$ stored, and then it runs $\mathsf{Verify}(\texttt{VCCs}_{\texttt{pk}}, \overline{\texttt{sVCC}^{\texttt{id}}}, \overline{S_{\texttt{VCC}^{\texttt{id}}}})$, discarding those for which the validations are not successful. Then it checks the tally operations:

- It extracts the ciphertexts c from the votes which have passed the previous validations and composes the list $\mathcal{C}$ with them.
- Then it parses Π as $(\pi_{\texttt{mix}}, \mathcal{C}', \Pi_{\texttt{dec}})$ and verifies that the mixing was correct by running $\mathsf{MixVerify}(\mathcal{C}, \mathcal{C}', \pi_{\texttt{mix}})$.
- Finally it checks that the decryption of each ciphertext in $\mathcal{C}'$ was correct by running $\mathsf{VerifyDec}$ from the NIZKPK scheme, using as input the following tuple: the ciphertext $c'_i \in \mathcal{C}'$, the product ν of the corresponding set of voting options $\{v_{i\ell}\}_{\ell=1}^{t}$ from $\mathcal{R}$, and the proof $\pi_{\texttt{dec}_i}$ from $\Pi_{\texttt{dec}}$.

If all the validations are successful, the process outputs 1. If any validation fails, it outputs 0.

The algorithms execution is organized in the following phases:

Configuration phase: in this phase, the election authorities set up the public parameters of the election such as the list of voting options $\{v_1, \ldots, v_k\}$, the set of valid votes (combinations of voting options) Ω and the result function ρ. They also run the Setup algorithm and publish the resulting Electoral Board public key $\texttt{EB}_{\texttt{pk}}$, the Vote Cast Code Signer key $\texttt{VCCs}_{\texttt{pk}}$ and the empty voter list ID in the bulletin board. The Electoral Board private key $\texttt{EB}_{\texttt{sk}}$ is kept in secret by the Electoral Board members, the Codes secret key $\texttt{C}_{\texttt{sk}}$ and the Vote Cast Code Signer private key $\texttt{VCCs}_{\texttt{sk}}$ are provided to the registrars, and the Codes secret key $\texttt{C}_{\texttt{sk}}$ is also provided to the bulletin board manager.

Registration phase: in this phase the voters are registered to vote in the election. For each voter, the registrars use the Codes secret key $\texttt{C}_{\texttt{sk}}$ and the Vote Cast Code Signer private key $\texttt{VCCs}_{\texttt{sk}}$ to run the $\mathsf{Register}$ algorithm. The list ID is updated with the tuple $(\texttt{VC}_{\texttt{id}}, \texttt{VC}_{\texttt{pk}}^{\texttt{id}}, \texttt{VCks}_{\texttt{id}}, \texttt{CM}_{\texttt{id}})$ in the bulletin board. The rest of the generated values $(\texttt{SVK}_{\texttt{id}}, \texttt{BCK}^{\texttt{id}}, \texttt{sVCC}^{\texttt{id}}, \{v_i, \texttt{sCC}_i^{\texttt{id}}\}_{i=1}^{k})$ are provided to the voter in a voting card.

Voting phase: this phase consists of several steps:

1. The voter provides $\texttt{SVK}_{\texttt{id}}$ to the voting device, which runs the GetID algorithm to derive the Voting Card ID $\texttt{VC}_{\texttt{id}}$ and ask to the Bulletin Board Manager for the Verification Card keystore $\texttt{VCks}_{\texttt{id}}$. Then it runs the

GetKey algorithm to open it and recover the Verification Card private key $\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}$. At that point the voting device is prepared to create a vote.

2. The voter provides the set of selected voting options $\{v_1, \ldots, v_t\} \in \Omega$ to the voting device. The voting device runs the CreateVote algorithm, producing a vote $\mathtt{V}$.

3. The vote $\mathtt{V}$ is sent to the bulletin board manager together with the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$.

4. Upon reception of $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$, the bulletin board manager runs the ProcessVote algorithm to verify the incoming vote. In case the result is 1, the bulletin board is updated with the pair $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ and the process continues. Otherwise, the process stops and the voting device receives an error message.

5. The bulletin board manager runs the CreateCC algorithm using the voter's Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ published in the bulletin board, and the Codes secret key $\mathtt{C}_{\mathtt{sk}}$. In case the execution is successful, it sends back to the voting device the generated short Choice Codes, which are shown to the voter. Otherwise, the process stops and the voter receives an error message.

6. After the previous step, the voter can call to GetCC through the voting device as many times as required, in order to get the Choice Codes corresponding to her stored vote.

7. The voter then compares that the received Choice Codes match those in her voting card linked to her selections. In case the verification is satisfactory, the voter provides her Ballot Casting Key $\mathtt{BCK}^{\mathtt{id}}$ to the voting device, which generates a confirmation message $\mathtt{CM}^{\mathtt{id}}$ using the Confirm algorithm, that is sent to the bulletin board manager with the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$. Otherwise the voter does not confirm her vote, and she can opt to vote through another channel.

8. The bulletin board manager then runs ProcessConfirm using as input the received confirmation message $\mathtt{CM}^{\mathtt{id}}$. In case the operation is successful (the output is different from $\perp$), it updates the bulletin board with the retrieved Vote Cast Code $\overline{\mathtt{sVCC}^{\mathtt{id}}}$ and signature $\overline{S_{\mathtt{VCC}^{\mathtt{id}}}}$, and sends the Vote Cast Code to the voter, who checks it matches the Vote Cast Code $\mathtt{sVCC}^{\mathtt{id}}$ in her voting card. Otherwise, an error message is sent.

9. After the previous step, the voter can request to the bulletin board manager to retrieve and show the value $\mathtt{sVCC}^{\mathtt{id}}$ as many times as she requires until the end of the election.

   At this point, in case the Vote Cast Code received by the voter is different from the one expected or an error has been returned, the voter may try to confirm her vote from another device or complain to the authorities, who may check if a vote has been indeed confirmed by that voter in the system. In case it has not, the voter may try to confirm her vote from another device or cast her vote through another channel.

Counting phase: in this phase, the election authorities run the interactive protocol Tally on BB using the Electoral Board private key $\mathtt{EB}_{\mathtt{sk}}$, obtaining and publishing in the bulletin board the result r and the proof Π. The auditors run the VerifyTally protocol using as input the contents in the bulletin board. In case their output is 1, the result r is announced to be fair. Otherwise, an investigation is opened to detect the reason of failure.

A voting protocol as defined above is *correct* if, when the four phases are run the result r output by the Tally algorithm is equal to the evaluation of the counting function ρ over the voting options corresponding to the votes cast and confirmed by the voters.

4 Security Analysis

In this section we define the notion of cast-as-intended verifiability for the protocol of the Swiss Online Voting System described in Section 3, and we prove it fulfills it.

We prove the security of our scheme using a game-based demonstration, where we end up relating the security of the protocol to some specific properties of the cryptographic primities which implement it. The games are played between a Challenger, which represents the honest part of the system, and an Attacker, which represents the dishonest party. During the games, we define which is the information the Attacker has access to, and which are the queries that can be made to the Challenger, to execute steps of the protocol. During the development of the games, bold text is used to mark changes between transitions for a better identification.

Although the games are represented with one voting option for simplicity, they can be extended to ballots with multiple options as far as all of them are different and the number of voting options the voter can select is fixed. This means that, in case the voter can select *up to* t voting options and does not select all of them, the rest of the ballot will be filled with blank options which are all different, and the voter will receive choice codes corresponding to such blank options besides her selections.

First of all, we informally introduce the trust assumptions we make on the scheme regarding privacy and integrity:

- **Voter:** the voter is assumed to follow the audit processes indicated and to object in case of any irregularity.
- **Election authorities and registrars:** in order to simplify the analysis, we consider that the election authorities and the registrars behave properly, in the sense that they generate correct and valid information, and that they do not divulge secret information to unintended recipients. In order to enforce this property, both election authorities and registrars can be distributed among a set of trustees which compute the required information using multiparty computation algorithms.
- **Voting device:** from the point of view of privacy, the voting device is trusted not to leak the randomness used for the encryption of the voter's

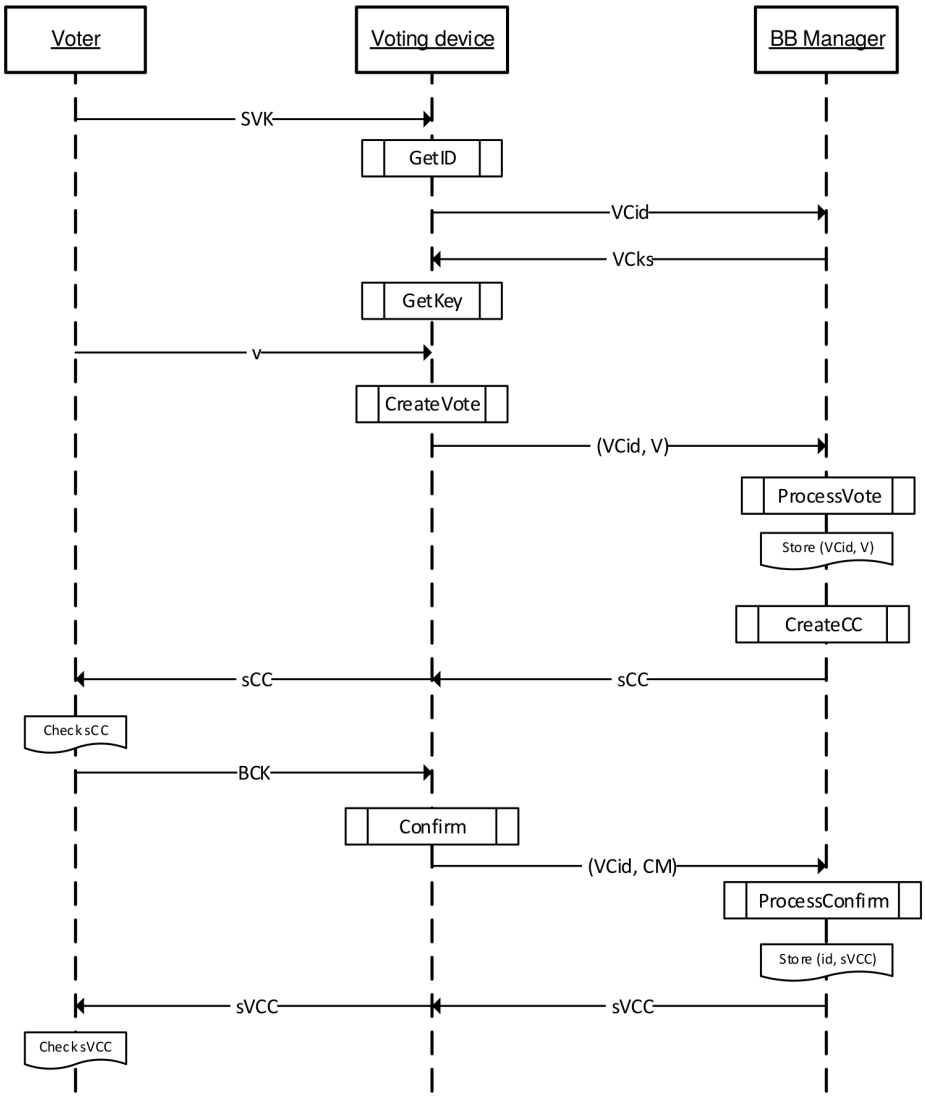

Figure 1: Workflow of the voting phase

choices. While this assumption may seem too strong, it is in fact needed in any voting scheme where the voting options are encrypted at the voting device (no pre-encrypted ballots are used) and the vote is not cast in an anonymous way. However, from the point of view of integrity, we consider that a malicious voting device may ignore the selections made by the voter and put other content in the ballot to be cast, or even reject to cast a vote.

- **Bulletin Board Manager:** the bulletin board manager is trusted to accept and post on the bulletin board only correct votes and confirmations. No assumptions are done in the case of privacy.
- **Auditors:** they are assumed to honestly transmit the result of their verification. However, we assume them to be curious and try to find out the content of voter's votes from the information they get.

4.1 Cast-as-Intended verifiability

A voting system as defined in Section 3 is defined to be cast-as-inteded verifiable if any of these situations can happen only with a negligible probability:

- A voter tries to cast a vote using her voting device, but the voting device decides to change her selection for a different one of its interest. The voter does not detect the modification although she follows the verification protocol.
- The voter casts a vote, performs the verification process and proceeds to confirm her vote, but the voting device refuses to send the confirmation message. The voter does not notice that her vote has not been confirmed, although she follows the verification protocol.
- The voter starts the voting process, but changes her mind and does not cast any vote. The voting device ignores her decision and casts a vote on behalf of the voter. A similar case is that the voting device changes the voter selection, the voter detects it after executing the verification protocol and rejects to confirm that vote, but the voting device confirms it anyway.

We define 3 games, each one representing one situation: in the first 2 the Attacker tries to *subvert* the actions of the voter, while still having to get the correct verification data so that the voter does not notice any change in the expected application flow. In the third game the scope of the Attacker is to act without the collaboration of the voter and still get a valid Ballot Casting Key for which a valid Vote Cast Code is generated by the Challenger (one for which the published signature validates, so that this vote is accepted in the tally phase). The probability of an Attacker succeeding on breaking the cast-as-intended verifiability property is defined as the maximum probability of an Attacker winning in any of the 3 games.

Then we prove the following theorem:

Theorem 1 *Let* $(\mathsf{ProveEq}, \mathsf{VerifyEq}, \mathsf{SimEq})$ *be a sound NIZKP scheme,* $f_k()$ *be a collision-resistant pseudorandom function, the symmetric encryption scheme*

$(\mathsf{KGen}_\mathsf{e}^\mathsf{s}, \mathsf{Enc}^\mathsf{s}, \mathsf{Dec}^\mathsf{s})$ *be modeled as a pseudo-random function, the voter's Verification Card key pairs be uniformly sampled from the key space and the size of the group* $\mathbb{G}$ *be much larger than the number of voters, the hash function* H *and the underlying hash function of the signature scheme be collision-resistant. Then the Attacker advantage in the protocol described in Section 3 is negligible when trying to defeat the protocol's cast-as-intended verifiability property in the random oracle model.*

4.1.1 Game A

The Attacker casts a valid vote different from the intended by the voter without detection.

In this game, the adaptive Attacker is given the opportunity to register both honest and corrupt voters. While for corrupt voters the Attacker is provided with all the registration information including the mapping among voting options and Choice Codes, for honest voters the Attacker receives only the information necessary to cast a vote. The public registration information from all voters is published by the Challenger on the bulletin board. The Attacker has also the ability to cast votes, which are processed by the Challenger to return the corresponding Choice Codes.

The objective of the Attacker in this game is, for any registered honest voter, to obtain a Choice Code which matches that of the voting option selected by the Challenger, while casting a vote with a different voting option inside. The game is presented in three phases. At the end of them, the Attacker has to provide the information required to win.

Configuration:

Challenger: Compute $(\mathtt{EB}_\mathtt{pk}, \mathtt{EB}_\mathtt{sk}, \mathtt{C}_\mathtt{sk}, \mathtt{VCCs}_\mathtt{pk}, \mathtt{VCCs}_\mathtt{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\mathtt{EB}_\mathtt{pk}, \mathtt{VCCs}_\mathtt{pk}, \mathsf{ID})$ in BB
Keep $(\mathtt{EB}_\mathtt{sk}, \mathtt{C}_\mathtt{sk}, \mathtt{VCCs}_\mathtt{sk})$

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{Register}(1^\lambda, \mathtt{C}_\mathtt{sk}, \mathtt{VCCs}_\mathtt{sk})$
If generated $\mathtt{VC}_\mathtt{id} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_\mathtt{id}$ to ID_h
Select $x_\mathtt{id} \xleftarrow{\$} \{v_1, \ldots, v_k\}$
Post $(\mathtt{VC}_\mathtt{id}, \mathtt{VC}_\mathtt{pk}^\mathtt{id}, \mathtt{VCks}_\mathtt{id}, \mathtt{CM}_\mathtt{id})$ in BB
Provide $\mathtt{SVK}_\mathtt{id}, x_\mathtt{id}$ to the Attacker
Keep $\{v_i, \mathtt{sCC}_i^\mathtt{id}\}_{i=1}^k$

$\mathcal{O}$**Challenger_RegisterCorrupt**(): $\mathsf{Register}(1^\lambda, \mathtt{C}_\mathtt{sk}, \mathtt{VCCs}_\mathtt{sk})$
If generated $\mathtt{VC}_\mathtt{id} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_\mathtt{id}$ to ID_c
Post $(\mathtt{VC}_\mathtt{id}, \mathtt{VC}_\mathtt{pk}^\mathtt{id}, \mathtt{VCks}_\mathtt{id}, \mathtt{CM}_\mathtt{id})$ in BB
Provide $(\mathtt{SVK}_\mathtt{id}, \mathtt{BCK}^\mathtt{id}, \mathtt{sVCC}^\mathtt{id}, \{v_i, \mathtt{sCC}_i^\mathtt{id}\}_{i=1}^k)$ to the Attacker

Voting: The Attacker generates votes running the CreateVote algorithm, or any other algorithm of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_Vote**($\mathtt{VC_{id}}, \mathtt{V}$)**:** Run ProcessVote($\mathsf{BB}, \mathtt{VC_{id}}, \mathtt{V}$). If result is 1:
- add ($\mathtt{VC_{id}}, \mathtt{V}$) to BB,
- compute $\sigma \leftarrow$ CreateCC($\mathtt{V}, \mathtt{C_{sk}}, \mathtt{CM_{id}}$),
- send σ

Else send $\perp$

$\mathcal{O}$**Challenger_getCC**($\mathtt{VC_{id}}$)**:** Run
$\sigma \leftarrow$ GetCC($\mathsf{BB}, \mathtt{VC_{id}}, \mathtt{C_{sk}}$),
Send σ

At the end of the game, the Attacker provides $(\mathtt{VC}^*_{\mathtt{id}}, \sigma^*)$.

Define S_0^A as the event that $\exists\ (\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}) \in \mathsf{BB}$ s.t. $\mathsf{Dec}(\mathtt{V}) \neq x_{\mathtt{id}}$ [2] and $\sigma^* = \mathtt{sCC}^{\mathtt{id}}_{x_{\mathtt{id}}}$, for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We define the Choice Code advantage of an adversary as:

$$\mathrm{CC}\mathsf{adv}[\mathcal{A}] = \left| \Pr\{S_0^A\} - \frac{1}{|\mathcal{A}_{cc}|} \right|$$

4.1.2 Security demonstration of Game A:

The demonstrations in this game are oriented to prove that all the information the Attacker may have access to is independent from the Choice Code it has to generate, and therefore that it has no significant advantage with respect to guessing the value at random.

Game A.1 First we proceed with a game transition based on failure, in which we rule out the possibility that the mapping information generated for different voters is the same. Let this be the game in which the algorithm Register is substituted by the algorithm UniqueRegister. In this new algorithm, the Challenger keeps track of the mapping information generated for the voters, and checks in every new registration that existing mapping information is not assigned to a new voter.

> UniqueRegister($1^\lambda, \mathtt{C_{sk}}, \mathtt{VCCs_{sk}}, \mathcal{K}$) takes as input a security parameter 1^λ, the private keys $\mathtt{C_{sk}}, \mathtt{VCCs_{sk}}$ and a list $\mathcal{K}$ of generated mapping information, and performs the following operations:
>
> - Generates a Start Voting Key $\mathtt{SVK_{id}} \xleftarrow{\$} \mathcal{A}_{svk}$.
> - Generates a Voting Card ID $\mathtt{VC_{id}} \leftarrow \delta(\mathtt{SVK_{id}}, \mathtt{IDseed})$.
> - Generates a keystore password $\mathtt{KSpwd_{id}} \leftarrow \delta(\mathtt{SVK_{id}}, \mathtt{KEYseed})$.
>
> – Generation of Verification Card data:
>
> - Generates the Verification Card key pair: $(\mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}) \leftarrow \mathsf{Gen_e}(1^\lambda)$.

[2] For ease of notation, we obviate the fact that Dec works over the ciphertext c inside V.

- Computes the encryption of the Verification Card private key with the keystore password: $\texttt{VCks}_{\texttt{id}} \leftarrow \mathsf{Enc}^{\mathsf{s}}(\texttt{VC}^{\texttt{id}}_{\texttt{sk}}; \texttt{KSpwd}_{\texttt{id}})$.

- Verification Card codes generation:
  - Chooses at random a Ballot Casting Key $\texttt{BCK}^{\texttt{id}} \xleftarrow{\$} A_{bck}$.
  - For each voting option $v_i \in \{v_1, \ldots, v_k\}$ it computes a long Choice Code $\texttt{CC}^{\texttt{id}}_i = f_{\texttt{C}_{\texttt{sk}}}(v_i^{\texttt{VC}^{\texttt{id}}_{\texttt{sk}}})$ and a short Choice Code $\texttt{sCC}^{\texttt{id}}$ taken at random from the value space $\mathcal{A}_{cc}$. The short Choice Codes are checked to be unique in the set for one voter.
  - Computes a long Vote Cast Code $\texttt{VCC}^{\texttt{id}} = f_{\texttt{C}_{\texttt{sk}}}((\texttt{BCK}^{\texttt{id}})^{\texttt{VC}^{\texttt{id}}_{\texttt{sk}}})$ and a short Vote Cast Code $\texttt{sVCC}^{\texttt{id}}$ taken at random from the value space $\mathcal{A}_{vcc}$.
  - Computes the signature value of the short Vote Cast Code $S_{\texttt{VCC}^{\texttt{id}}} \leftarrow \mathsf{Sign}(\texttt{sVCC}^{\texttt{id}}, \texttt{VCCs}_{\texttt{sk}})$.
  - Computes the Codes Mapping Table $\texttt{CM}_{\texttt{id}}$ consisting on pairs *(reference value - encrypted code)* for the Choice Codes, as well as for the Vote Cast Code and signature:

$$\{[\mathsf{H}(\texttt{CC}^{\texttt{id}}_i), \mathsf{Enc}^{\mathsf{s}}(\texttt{sCC}^{\texttt{id}}_i; \texttt{CC}^{\texttt{id}}_i)]\}^k_{i=1}$$
$$[\mathsf{H}(\texttt{VCC}^{\texttt{id}}), \mathsf{Enc}^{\mathsf{s}}((\texttt{sVCC}^{\texttt{id}} | S_{\texttt{VCC}^{\texttt{id}}}); \texttt{VCC}^{\texttt{id}})]$$

- **Checks whether for the generated values $\{\mathsf{H}(\texttt{CC}^{\texttt{id}}_i)\}^k_{i=1}, \mathsf{H}(\texttt{VCC}^{\texttt{id}})$ there is any entry equal in the list $\mathcal{K}$ and that they are all different. If not, start the algorithm from the Verification Card data generation step. Otherwise, $\mathcal{K} = \mathcal{K} \cup (\{\mathsf{H}(\texttt{CC}^{\texttt{id}}_i)\}^k_{i=1}, \mathsf{H}(\texttt{VCC}^{\texttt{id}}))$ and the algorithm proceeds**.

Finally it outputs the generated information: $\texttt{SVK}_{\texttt{id}}, \texttt{VC}_{\texttt{id}}, \texttt{VC}^{\texttt{id}}_{\texttt{pk}}, \texttt{VCks}_{\texttt{id}}, \texttt{BCK}^{\texttt{id}}, \texttt{VCC}^{\texttt{id}}, \{v_i, \texttt{sCC}^{\texttt{id}}_i\}^k_{i=1}, \texttt{CM}_{\texttt{id}}$.

The resulting game is as follows:

Configuration:

Challenger: Compute $(\texttt{EB}_{\texttt{pk}}, \texttt{EB}_{\texttt{sk}}, \texttt{C}_{\texttt{sk}}, \texttt{VCCs}_{\texttt{pk}}, \texttt{VCCs}_{\texttt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\texttt{EB}_{\texttt{pk}}, \texttt{VCCs}_{\texttt{pk}}, \mathsf{ID})$ in BB
Keep $(\texttt{EB}_{\texttt{sk}}, \texttt{C}_{\texttt{sk}}, \texttt{VCCs}_{\texttt{sk}})$
Init $\mathcal{K}$ to an empty list.

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{UniqueRegister}(1^\lambda, \texttt{C}_{\texttt{sk}}, \texttt{VCCs}_{\texttt{sk}}, \mathcal{K})$
If generated $\texttt{VC}_{\texttt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\texttt{VC}_{\texttt{id}}$ to ID_h
Select $x_{\texttt{id}} \xleftarrow{\$} \{v_1, \ldots, v_k\}$
Post $(\texttt{VC}_{\texttt{id}}, \texttt{VC}^{\texttt{id}}_{\texttt{pk}}, \texttt{VCks}_{\texttt{id}}, \texttt{CM}_{\texttt{id}})$ in BB
Provide $\texttt{SVK}_{\texttt{id}}, x_{\texttt{id}}$ to the Attacker

Keep $\{v_i, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^{k}$

$\mathcal{O}$Challenger_RegisterCorrupt(): $\mathsf{UniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_c
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $(\mathtt{SVK}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^{k})$ to the Attacker

Voting: The Attacker generates votes running the $\mathsf{CreateVote}$ algorithm, or any other algorithm of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$Challenger_Vote($\mathtt{VC}_{\mathtt{id}}, \mathtt{V}$): Run $\mathsf{ProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V})$. If result is 1:
- add $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ to BB,
- compute $\sigma \leftarrow \mathsf{CreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}})$,
- send σ
Else send $\perp$

$\mathcal{O}$Challenger_getCC($\mathtt{VC}_{\mathtt{id}}$): Run
$\sigma \leftarrow \mathsf{GetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}})$,
Send σ

At the end of the game, the Attacker provides $(\mathtt{VC}_{\mathtt{id}}^*, \sigma^*)$.

Define S_1^A as the event that $\exists\ (\mathtt{VC}_{\mathtt{id}}^*, \mathtt{V}) \in \mathsf{BB}$ s.t. $\mathsf{Dec}(\mathtt{V}) \neq x_{\mathtt{id}}$ and $\sigma^* = \mathtt{sCC}_{x_{\mathtt{id}}}^{\mathtt{id}}$, for $\mathtt{VC}_{\mathtt{id}}^* \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_0^A\} - \Pr\{S_1^A\}| = \epsilon_{\text{key}},$$

which we consider to be negligible under the assumptions that the Verification Card private keys are uniformly chosen from the key space, and that the order of the group $\mathbb{G}$ is much higher than the number of voters, and the collision-resistance property of the hash function.

Game A.2 Then we proceed with a game transition based on indistinguishability. Let this be the game in which the $\mathsf{ProcessVote}$ algorithm from game A.1 is changed into $\mathsf{PerfectProcessVote}$, which *only* outputs 1 when there is not already a ballot in BB for the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$ and $\mathtt{V} = (\alpha_x, \beta_x, \gamma(\alpha_x, \beta_x))$ - that is, the encrypted options and the partial choice codes have been computed over the same voting option x.

$\mathsf{PerfectProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V}, \mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}, \mathtt{EB}_{\mathtt{sk}})$ receives as input a bulletin board BB, a Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, a vote $\mathtt{V}$, the Verification Card private key $\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}$, and the Electoral Board private key $\mathtt{EB}_{\mathtt{sk}}$, and proceeds to do the following checks:

- Check that $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$ and that $(\mathtt{VC}_{\mathtt{id}}, \tilde{\mathtt{V}}) \notin \mathsf{BB}$, where $\tilde{\mathtt{V}}$ is a vote that may be equal or different from $\mathtt{V}$ (that is, there is no vote for the $\mathtt{VC}_{\mathtt{id}}$ entry in BB).

- Extract c and $\pi_{\mathtt{sch}}$ from $\mathtt{V}$ and check that $\mathsf{VerifyExp}((g, c_1, c_2), \pi_{\mathtt{sch}})$ is correct (it outputs 1).
- Run $\hat{v} \leftarrow \mathsf{Dec}(c, \mathtt{EB}_{\mathtt{sk}})$.
- Compute $\hat{\mathtt{pCC}}^{\mathtt{id}} = \hat{v}^{\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}}$.
- Extract $\mathtt{pCC}^{\mathtt{id}}$ from $\mathtt{V}$ and check that $\hat{\mathtt{pCC}}^{\mathtt{id}} = \mathtt{pCC}^{\mathtt{id}}$.

If all the validations pass, the result is 1. Otherwise, the output is 0.

In order to be able to execute this algorithm, the Challenger needs to get access to the voter's Verification Card private key $\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}$, which it can do by running the GetKey algorithm with the information it already knows.

The resulting game is as follows:

Configuration:

Challenger: Compute $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathsf{ID})$ in BB
Keep $(\mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
Init $\mathcal{K}$ to an empty list.

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{UniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_h
Select $x_{\mathtt{id}} \xleftarrow{\$} \{v_1, \ldots, v_k\}$
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $\mathtt{SVK}_{\mathtt{id}}, x_{\mathtt{id}}$ to the Attacker
Keep $\{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k$

$\mathcal{O}$**Challenger_RegisterCorrupt**(): $\mathsf{UniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_c
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $(\mathtt{SVK}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k)$ to the Attacker

Voting: The Attacker generates votes running the $\mathsf{CreateVote}$ algorithm, or any other algorithm of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_Vote**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$**: Run** $\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}} \leftarrow \mathsf{GetKey}(\mathtt{SVK}_{\mathtt{id}}, \mathtt{VCks}_{\mathtt{id}})$
Run $\mathsf{PerfectProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}, \mathtt{EB}_{\mathtt{sk}})$. If result is 1:
- add $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ to BB,
- compute $\sigma \leftarrow \mathsf{CreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}})$,
- send σ

Else send $\perp$

$\mathcal{O}$**Challenger_getCC(**$\mathtt{VC}_{\mathtt{id}}$**):** Run
$\sigma \leftarrow \mathsf{GetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}})$,
Send σ

At the end of the game, the Attacker provides $(\mathtt{VC}^{*}_{\mathtt{id}}, \sigma^{*})$.

Define S_2^A as the event that $\exists\ (\mathtt{VC}^{*}_{\mathtt{id}}, \mathtt{V}) \in \mathsf{BB}$ s.t. $\mathsf{Dec}(\mathtt{V}) \neq x_{\mathtt{id}}$ and $\sigma^{*} = \mathtt{sCC}^{\mathtt{id}}_{x_{\mathtt{id}}}$, for $\mathtt{VC}^{*}_{\mathtt{id}} \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_1^A\} - \Pr\{S_2^A\}| = \epsilon_{\mathrm{zkp}},$$

where ϵ_{zkp} is the ZKP-advantage of an efficient algorithm against the soundness of the NIZKP scheme (which is negligible assuming the properties of NIZKPs).

Game A.3 This transition is also based on indistinguishability: Let this be the game in which the algorithms UniqueRegister, CreateCC and GetCC from game A.2 are changed into RndUniqueRegister and RndCreateCC and RndGetCC, in which each long Choice Code and long Vote Cast Code is generated by using a random oracle $\mathcal{O}_{lc}$ (instead of the PRF function $f_{\mathtt{C}_{\mathtt{sk}}}$) which maps elements from $\mathbb{G}$ (the space of partial Choice Codes ($v_i^{\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}}$) and of confirmation messages) to $\mathcal{A}_{lc}$ (the space of long Choice Codes and long Vote Cast Codes).

$\mathcal{O}_{lc}$:

- Given an initially empty list $\mathcal{T}_{lc}$
- Given an input $a \in \mathbb{G}$,

  if $a \notin \mathcal{T}_{lc}$: $b \xleftarrow{\$} \mathcal{A}_{lc}$ and add (a, b) to $\mathcal{T}_{lc}$,

  otherwise, $b \leftarrow \mathcal{T}_{lc}(a)$

Specifically, the new functions are defined as follows:

RndUniqueRegister$(1^{\lambda}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$ takes as input a security parameter 1^{λ}, the private keys $\mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}$ and a list $\mathcal{K}$ of generated mapping information, and performs the following operations:

- Generates a Start Voting Key $\mathtt{SVK}_{\mathtt{id}} \xleftarrow{\$} \mathcal{A}_{svk}$.
- Generates a Voting Card ID $\mathtt{VC}_{\mathtt{id}} \leftarrow \delta(\mathtt{SVK}_{\mathtt{id}}, \mathtt{IDseed})$.
- Generates a keystore password $\mathtt{KSpwd}_{\mathtt{id}} \leftarrow \delta(\mathtt{SVK}_{\mathtt{id}}, \mathtt{KEYseed})$.

– Generation of Verification Card data:

- Generates the Verification Card key pair: $(\mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}) \leftarrow \mathsf{Gen}_{\mathsf{e}}(1^{\lambda})$.
- Computes the encryption of the Verification Card private key with the keystore password: $\mathtt{VCks}_{\mathtt{id}} \leftarrow \mathsf{Enc}^{\mathsf{s}}(\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}; \mathtt{KSpwd}_{\mathtt{id}})$.

- Verification Card codes generation:

  - Chooses at random a Ballot Casting Key $\mathtt{BCK}^{\mathtt{id}} \xleftarrow{\$} A_{bck}$.

- For each voting option $v_i \in \{v_1, \ldots, v_k\}$ it computes a long Choice Code **querying the oracle:** $\mathtt{CC}_i^{\mathtt{id}} = \mathcal{O}_{lc}(v_i^{\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}})$, and a short Choice Code $\mathtt{sCC}^{\mathtt{id}}$ taken at random from the value space $\mathcal{A}_{cc}$. The short Choice Codes are checked to be unique in the set for one voter.
- Computes a long Vote Cast Code **querying the oracle:** $\mathtt{VCC}^{\mathtt{id}} = \mathcal{O}_{lc}((\mathtt{BCK}^{\mathtt{id}})^{\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}})$ and a short Vote Cast Code $\mathtt{sVCC}^{\mathtt{id}}$ taken at random from the value space $\mathcal{A}_{vcc}$.
- Computes the signature value of the short Vote Cast Code $S_{\mathtt{VCC}^{\mathtt{id}}} \leftarrow \mathsf{Sign}(\mathtt{sVCC}^{\mathtt{id}}, \mathtt{VCCs}_{\mathtt{sk}})$.
- Computes the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ consisting on pairs *(reference value - encrypted code)* for the Choice Codes, as well as for the Vote Cast Code and signature:

$$\{[\mathsf{H}(\mathtt{CC}_i^{\mathtt{id}}), \mathsf{Enc}^{\mathsf{s}}(\mathtt{sCC}_i^{\mathtt{id}}; \mathtt{CC}_i^{\mathtt{id}})]\}_{i=1}^{k}$$
$$[\mathsf{H}(\mathtt{VCC}^{\mathtt{id}}), \mathsf{Enc}^{\mathsf{s}}((\mathtt{sVCC}^{\mathtt{id}} | S_{\mathtt{VCC}^{\mathtt{id}}}); \mathtt{VCC}^{\mathtt{id}})]$$

- Checks whether for the generated values $\{\mathsf{H}(\mathtt{CC}_i^{\mathtt{id}})\}_{i=1}^{k}, \mathsf{H}(\mathtt{VCC}^{\mathtt{id}})$ there is any entry equal in the list $\mathcal{K}$ and that they are all different. If not, start the algorithm from the Verification Card data generation step. Otherwise, $\mathcal{K} = \mathcal{K} \cup (\{\mathsf{H}(\mathtt{CC}_i^{\mathtt{id}})\}_{i=1}^{k}, \mathsf{H}(\mathtt{VCC}^{\mathtt{id}}))$ and the algorithm proceeds.

Finally it outputs the generated information: $\mathtt{SVK}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{VCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^{k}, \mathtt{CM}_{\mathtt{id}}$.

$\mathsf{RndCreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}})$ takes as input the vote $\mathtt{V}$, the Codes secret key $\mathtt{C}_{\mathtt{sk}}$, the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$, and does the following actions:

- **It uses the oracle to compute a long Choice Code value** $\overline{\mathtt{CC}_\ell^{\mathtt{id}}} = \mathcal{O}_{lc}(\mathtt{pCC}_\ell^{\mathtt{id}})$ for each one of the partial Choice Codes $\{\mathtt{pCC}_\ell^{\mathtt{id}}\}_{\ell=1}^{t}$ in $\mathtt{V}$.
- Checks that for each long Choice Code $\overline{\mathtt{CC}_\ell^{\mathtt{id}}}$, $\mathsf{H}(\overline{\mathtt{CC}_\ell^{\mathtt{id}}}) \in \{\mathsf{H}(\mathtt{CC}_i^{\mathtt{id}})\}_{i=1}^{k}$, where $\ell = 1, \ldots, t$, and for each one it recovers the short value $\overline{\mathtt{sCC}_\ell^{\mathtt{id}}}$ running the decryption algorithm $\mathsf{Dec}^{\mathsf{s}}$ with $\overline{\mathtt{CC}_\ell^{\mathtt{id}}}$ as the symmetric key. In a positive case, the output of the algorithm is the set of decrypted short Choice Codes $\{\overline{\mathtt{sCC}_\ell^{\mathtt{id}}}\}_{\ell=1}^{t} = (\overline{\mathtt{sCC}_1^{\mathtt{id}}}, \ldots, \overline{\mathtt{sCC}_t^{\mathtt{id}}})$. Otherwise, the output is $\perp$.

$\mathsf{RndGetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}})$ takes as input the bulletin board BB, a Voting Card ID $\mathtt{VC}_{\mathtt{id}}$ and the Codes secret key $\mathtt{C}_{\mathtt{sk}}$, and does the following actions:

- Checks whether there is an entry in the bulletin board corresponding to the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$.
- Retrieves the vote $\mathtt{V}$ from the bulletin board corresponding to the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$. If there is no vote, the algorithm stops and returns $\perp$.
- Retrieves the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ from the bulletin board.
- **It uses the oracle to compute a long Choice Code value** $\overline{\mathtt{CC}_\ell^{\mathtt{id}}} = \mathcal{O}_{lc}(\mathtt{pCC}_\ell^{\mathtt{id}})$ for each one of the partial Choice Codes $\{\mathtt{pCC}_\ell^{\mathtt{id}}\}_{\ell=1}^{t}$ in $\mathtt{V}$.

- Checks that for each Choice Code $\overline{\mathtt{CC}_\ell^{\mathtt{id}}}$, $\mathsf{H}(\overline{\mathtt{CC}_\ell^{\mathtt{id}}}) \in \{\mathsf{H}(\mathtt{CC}_i^{\mathtt{id}})\}_{i=1}^{k}$, where $\ell = 1, \ldots, t$, and for each one it recovers the short value $\overline{\mathtt{sCC}_\ell^{\mathtt{id}}}$ running the decryption algorithm $\mathsf{Dec}^{\mathtt{s}}$ with $\overline{\mathtt{CC}_\ell^{\mathtt{id}}}$ as the symmetric key. In a positive case, the output of the algorithm is the set of decrypted short Choice Codes $\{\overline{\mathtt{sCC}_\ell^{\mathtt{id}}}\}_{\ell=1}^{t} = (\overline{\mathtt{sCC}_1^{\mathtt{id}}}, \ldots, \overline{\mathtt{sCC}_t^{\mathtt{id}}})$. Otherwise, the output is $\perp$.

The resulting game is the same as in A.2, but using the new algorithms RndUniqueRegister, RndCreateCC and RndGetCC.

Define S_3^A as the event that $\exists\ (\mathtt{VC}_{\mathtt{id}}^*, \mathtt{V}) \in \mathsf{BB}$ s.t. $\mathsf{Dec}(\mathtt{V}) \neq x_{\mathtt{id}}$ and $\sigma^* = \mathtt{sCC}_{x_{\mathtt{id}}}^{\mathtt{id}}$, for $\mathtt{VC}_{\mathtt{id}}^* \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_2^A\} - \Pr\{S_3^A\}| = \epsilon_{\mathrm{prf}},$$

where ϵ_{prf} is the PRF-advantage of an efficient distinguisher algorithm, which is negligible considering that $f_{\mathtt{C}_{\mathtt{sk}}}$ is a pseudo-random function.

Game A.4 This transition is also based on indistinguishability: Let this be the game in which the algorithms MoreRndUniqueRegister, MoreRndCreateCC and MoreRndGetCC are used, instead of RndUniqueRegister, RndCreateCC and RndGetCC. In these new algorithms, the encryptions of the short Choice Codes and short Vote Cast Code which are stored in the codes mapping table $\mathtt{CM}_{\mathtt{id}}$ corresponding to a voter are substituted by choosing random values from the same value space. Still, the Challenger will keep the list of short Choice Codes and the short Vote Cast Code and signature internally, so that they can be provided during the voting phase.

MoreRndUniqueRegister$(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$ takes as input a security parameter 1^λ, the private keys $\mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}$ and a list $\mathcal{K}$ of generated mapping information, and performs the following operations:

- Generates a Start Voting Key $\mathtt{SVK}_{\mathtt{id}} \xleftarrow{\$} \mathcal{A}_{svk}$.
- Generates a Voting Card ID $\mathtt{VC}_{\mathtt{id}} \leftarrow \delta(\mathtt{SVK}_{\mathtt{id}}, \mathtt{IDseed})$.
- Generates a keystore password $\mathtt{KSpwd}_{\mathtt{id}} \leftarrow \delta(\mathtt{SVK}_{\mathtt{id}}, \mathtt{KEYseed})$.

– Generation of Verification Card data:

  - Generates the Verification Card key pair: $(\mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}) \leftarrow \mathsf{Gen}_{\mathsf{e}}(1^\lambda)$.
  - Computes the encryption of the Verification Card private key with the keystore password: $\mathtt{VCks}_{\mathtt{id}} \leftarrow \mathsf{Enc}^{\mathtt{s}}(\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}; \mathtt{KSpwd}_{\mathtt{id}})$.

- Verification Card codes generation:

  - Chooses at random a Ballot Casting Key $\mathtt{BCK}^{\mathtt{id}} \xleftarrow{\$} A_{bck}$.
  - For each voting option $v_i \in \{v_1, \ldots, v_k\}$ it computes a long Choice Code querying the oracle: $\mathtt{CC}_i^{\mathtt{id}} = \mathcal{O}_{lc}(v_i^{\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}})$, and a short Choice Code $\mathtt{sCC}^{\mathtt{id}}$ taken at random from the value space $\mathcal{A}_{cc}$. The short Choice Codes are checked to be unique in the set for one voter.

- Computes a long Vote Cast Code querying the oracle: $\mathtt{VCC}^{\mathtt{id}} = \mathcal{O}_{lc}((\mathtt{BCK}^{\mathtt{id}})^{\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}})$ and a short Vote Cast Code $\mathtt{sVCC}^{\mathtt{id}}$ taken at random from the value space $\mathcal{A}_{vcc}$.
- Computes the signature value of the short Vote Cast Code $S_{\mathtt{VCC}^{\mathtt{id}}} \leftarrow \mathsf{Sign}(\mathtt{sVCC}^{\mathtt{id}}, \mathtt{VCCs}_{\mathtt{sk}})$.
- Computes the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ consisting on pairs *(reference value - encrypted code)* for the Choice Codes, as well as for the Vote Cast Code and signature:

$$\{[\mathsf{H}(\mathtt{CC}^{\mathtt{id}}_i), \mathsf{Enc}^{\mathsf{s}}(\mathtt{sCC}^{\mathtt{id}}_i; \mathtt{CC}^{\mathtt{id}}_i)]\}^k_{i=1}$$
$$[\mathsf{H}(\mathtt{VCC}^{\mathtt{id}}), \mathsf{Enc}^{\mathsf{s}}((\mathtt{sVCC}^{\mathtt{id}} | S_{\mathtt{VCC}^{\mathtt{id}}}); \mathtt{VCC}^{\mathtt{id}})]$$

- Checks whether for the generated values $\{\mathsf{H}(\mathtt{CC}^{\mathtt{id}}_i)\}^k_{i=1}, \mathsf{H}(\mathtt{VCC}^{\mathtt{id}})$ there is any entry equal in the list $\mathcal{K}$ and that they are all different. If not, start the algorithm from the Verification Card data generation step. Otherwise, $\mathcal{K} = \mathcal{K} \cup (\{\mathsf{H}(\mathtt{CC}^{\mathtt{id}}_i)\}^k_{i=1}, \mathsf{H}(\mathtt{VCC}^{\mathtt{id}}))$ and the algorithm proceeds.

Finally it outputs the generated information: $\mathtt{SVK}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{VCC}^{\mathtt{id}}, \{v_i, \mathtt{pCC}^{\mathtt{id}}_i, \mathtt{sCC}^{\mathtt{id}}_i\}^k_{i=1}, \mathtt{CM}_{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}}$.

$\mathsf{MoreRndCreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}}, \{\mathtt{pCC}^{\mathtt{id}}_i, \mathtt{sCC}^{\mathtt{id}}_i\}^k_{i=1})$ takes as input the vote $\mathtt{V}$, the Codes secret key $\mathtt{C}_{\mathtt{sk}}$, the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ **and the list of pairs partial / short Choice Codes** $\{\mathtt{pCC}^{\mathtt{id}}_i, \mathtt{sCC}^{\mathtt{id}}_i\}^k_{i=1}$, and does the following actions:

- It uses the oracle to compute a long Choice Code value $\overline{\mathtt{CC}^{\mathtt{id}}_\ell} = \mathcal{O}_{lc}(\mathtt{pCC}^{\mathtt{id}}_\ell)$ for each one of the partial Choice Codes $\{\mathtt{pCC}^{\mathtt{id}}_\ell\}^t_{\ell=1}$ in $\mathtt{V}$.
- Checks that for each long Choice Code $\overline{\mathtt{CC}^{\mathtt{id}}_\ell}$, $\mathsf{H}(\overline{\mathtt{CC}^{\mathtt{id}}_\ell}) \in \{\mathsf{H}(\mathtt{CC}^{\mathtt{id}}_i)\}^k_{i=1}$ from $\mathtt{CM}_{\mathtt{id}}$, where $\ell = 1, \ldots, t$.
  - **If so, it extracts the partial Choice Codes $\{\mathtt{pCC}^{\mathtt{id}}_\ell\}^t_{\ell=1}$ from $\mathtt{V}$ and obtains the corresponding short Choice Codes $\{\mathtt{sCC}^{\mathtt{id}}_\ell\}^t_{\ell=1}$ from the input list.**
  - Otherwise, the output is $\perp$.

$\mathsf{MoreRndGetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}}, \{\mathtt{pCC}^{\mathtt{id}}_i, \mathtt{sCC}^{\mathtt{id}}_i\}^k_{i=1})$ takes as input the bulletin board BB, a Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, the Codes secret key $\mathtt{C}_{\mathtt{sk}}$ **and the list of pairs partial / short Choice Codes** $\{\mathtt{pCC}^{\mathtt{id}}_i, \mathtt{sCC}^{\mathtt{id}}_i\}^k_{i=1}$, and does the following actions:

- Checks whether there is an entry in the bulletin board corresponding to the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$.
- Retrieves the vote $\mathtt{V}$ from the bulletin board corresponding to the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$. If there is no vote, the algorithm stops and returns $\perp$.
- Retrieves the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ from the bulletin board.
- It uses the oracle to compute a long Choice Code value $\overline{\mathtt{CC}^{\mathtt{id}}_\ell} = \mathcal{O}_{lc}(\mathtt{pCC}^{\mathtt{id}}_\ell)$ for each one of the partial Choice Codes $\{\mathtt{pCC}^{\mathtt{id}}_\ell\}^t_{\ell=1}$ in $\mathtt{V}$.

- Checks that for each long Choice Code $\overline{\mathtt{CC}_\ell^{\mathtt{id}}}$, $\mathsf{H}(\overline{\mathtt{CC}_\ell^{\mathtt{id}}}) \in \{\mathsf{H}(\mathtt{CC}_i^{\mathtt{id}})\}_{i=1}^k$ from $\mathtt{CM}_{\mathtt{id}}$, where $\ell = 1, \ldots, t$.
- **If so, it extracts the partial Choice Codes $\{\mathtt{pCC}_\ell^{\mathtt{id}}\}_{\ell=1}^t$ from $\mathtt{V}$ and obtains the corresponding short Choice Codes $\{\mathtt{sCC}_\ell^{\mathtt{id}}\}_{\ell=1}^t$ from the input list.** Otherwise, the output is $\perp$.

The resulting game is as follows:

Configuration:

Challenger: Compute $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathsf{ID})$ in BB
Keep $(\mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
Init $\mathcal{K}$ to an empty list.

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{MoreRndUniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_h
Select $x_{\mathtt{id}} \xleftarrow{\$} \{v_1, \ldots, v_k\}$
Keep the list of pairs $\{\mathtt{pCC}_i^{\mathtt{id}}, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k$
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $\mathtt{SVK}_{\mathtt{id}}, x_{\mathtt{id}}$ to the Attacker

$\mathcal{O}$**Challenger_RegisterCorrupt**(): $\mathsf{MoreRndUniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_c
Keep the list of pairs $\{\mathtt{pCC}_i^{\mathtt{id}}, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k$
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $(\mathtt{SVK}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k)$ to the Attacker

Voting: The Attacker generates votes running the $\mathsf{CreateVote}$ algorithm, or any other algorithm of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_Vote**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$: Run $\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}} \leftarrow \mathsf{GetKey}(\mathtt{SVK}_{\mathtt{id}}, \mathtt{VCks}_{\mathtt{id}})$
Run $\mathsf{PerfectProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V}, \mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}})$. If result is 1:
- add $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ to BB,
- compute $\sigma \leftarrow \mathsf{MoreRndCreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}}, \{\mathtt{pCC}_i^{\mathtt{id}}, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k)$,
- send σ
Else send $\perp$

$\mathcal{O}$**Challenger_getCC**$(\mathtt{VC}_{\mathtt{id}})$: Run
$\sigma \leftarrow \mathsf{MoreRndGetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}}, \{\mathtt{pCC}_i^{\mathtt{id}}, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k)$,
Send σ

At the end of the game, the Attacker provides $(\mathtt{VC}^*_{\mathtt{id}}, \sigma^*)$.

Define S_4^A as the event that $\exists\ (\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}) \in \mathsf{BB}$ s.t. $\mathsf{Dec}(\mathtt{V}) \neq x_{\mathtt{id}}$ and $\sigma^* = \mathtt{sCC}^{\mathtt{id}}_{x_{\mathtt{id}}}$, for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_3^A\} - \Pr\{S_4^A\}| = \epsilon_{\text{enc}},$$

where ϵ_{enc} is the advantage of an efficient algorithm that distinguishes the output of $\mathsf{Enc}^{\mathtt{s}}$ from a random function, which is negligible considering that $\mathsf{Enc}^{\mathtt{s}}$ can be seen as a pseudo-random function.

Finally, we can see that the values the Attacker has access to: the Choice Codes σ generated during the voting phase, the information posted in the bulletin board, as well as the Choice Codes produced by the registration algorithm and given to the Attacker for the corrupt voters; are independent from the expected value $\mathtt{sCC}^{\mathtt{id}}_{x_{\mathtt{id}}}$ the Attacker has to generate, and therefore $\Pr\{S_4^A\}$ is the probability of randomly guessing it, which is $\frac{1}{|\mathcal{A}_{cc}|}$. We can conclude that

$$\Pr\{S_0^A\} = \epsilon_{\text{key}} + \epsilon_{\text{zkp}} + \epsilon_{\text{prf}} + \epsilon_{\text{enc}} + \frac{1}{|\mathcal{A}_{cc}|},$$

and therefore

$$\text{CC}\mathsf{adv}[\mathcal{A}] = \epsilon_{\text{key}} + \epsilon_{\text{zkp}} + \epsilon_{\text{prf}} + \epsilon_{\text{enc}},$$

which is negligible given the assumption that the voters' keys are uniformly sampled from the key space, the order of the group $\mathbb{G}$ is larger than the number of voters, cryptographic hash functions are collision-resistant, NIZKP schemes are sound, and the used pseudo-random function and the symmetric encryption scheme are unidistingible from random distributions.

4.1.3 Game B

In this game, the adaptive Attacker can register both honest and corrupt voters. For corrupt voters, the Attacker receives all the registration information, including the Vote Cast Code. For honest voters, the Attacker receives the information required to cast and confirm a vote. The public registration information from all the voters is published by the Challenger on the bulletin board. The Attacker has the ability to cast votes and confirm them, and the Challenger processes them to return the corresponding messages.

The objetive of the Attacker in this game is, for any registered honest voter, to obtain a valid Vote Cast Code, in order to trick the voter into believing that her vote has been indeed confirmed.

Configuration:

Challenger: Compute $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathsf{ID})$ in BB
Keep $(\mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{Register}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_h
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $\mathtt{SVK}_{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k, \mathtt{BCK}^{\mathtt{id}}$ to the Attacker
Keep $\mathtt{sVCC}^{\mathtt{id}}$

$\mathcal{O}$**Challenger_RegisterCorrupt**(): $\mathsf{Register}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_c
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $(\mathtt{SVK}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k)$ to the Attacker

Voting: The Attacker generates votes and confirmations running the $\mathsf{CreateVote}$ and $\mathsf{Confirm}$ algorithms, or any other algorithms of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_Vote**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$**:** Run $\mathsf{ProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V})$. If result is 1:
- add $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ to BB,
- compute $\sigma \leftarrow \mathsf{CreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}})$,
- send σ
Else send $\perp$

$\mathcal{O}$**Challenger_getCC**$(\mathtt{VC}_{\mathtt{id}})$**:** Run
$\sigma \leftarrow \mathsf{GetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}})$,
Send σ

$\mathcal{O}$**Challenger_Confirm**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}})$**:** Run
$(\mu, S_\mu) \leftarrow \mathsf{ProcessConfirm}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}})$,
- if $(\mu, S_\mu) \neq \perp$: add (μ, S_μ) to $\mathtt{VC}_{\mathtt{id}}$ entry in BB,
Send μ

At the end of the game, the Attacker provides $(\mathtt{VC}^*_{\mathtt{id}}, \mu^*)$.

Define S_0^B as the event that $\mu^* = \mathtt{sVCC}^{\mathtt{id}}$ and $\nexists(\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}, \mu^*, S_\mu) \in \mathrm{BB}$ for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We define the Vote Cast Code advantage of an adversary as:

$$\mathrm{VCCadv}[\mathcal{A}] = \left| \Pr\{S_0^B\} - \frac{1}{|\mathcal{A}_{vcc}|} \right|$$

4.1.4 Security demonstration of Game B:

The demonstration proceeds similarly than in case of the game A, showing that the information the attacker learns is independent from the value $\mathtt{sVCC}^{\mathtt{id}}$ it has to generate. In fact, we are going to reuse part of the game hopping steps and transformed algorithms from game A.

Game B.1 As in game A, first we proceed with a game transition based on failure, in which we rule out the possibility that the mapping information generated for different voters is the same. Let this be the game in which the algorithm $\mathsf{Register}$ is substituted by the algorithm $\mathsf{UniqueRegister}$ defined in game A.1. In this new algorithm, the Challenger keeps track of the mapping information generated for the voters, and checks in every new registration that existing mapping information is not assigned to a new voter.

The resulting game is as follows:

Configuration:

Challenger: Compute $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathsf{ID})$ in BB
Keep $(\mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
Init $\mathcal{K}$ to an empty list

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{UniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_h
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $\mathtt{SVK}_{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k, \mathtt{BCK}^{\mathtt{id}}$ to the Attacker
Keep $\mathtt{sVCC}^{\mathtt{id}}$

$\mathcal{O}$**Challenger_RegisterCorrupt**(): $\mathsf{UniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_c
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $(\mathtt{SVK}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k)$ to the Attacker

Voting: The Attacker generates votes and confirmations running the $\mathsf{CreateVote}$ and $\mathsf{Confirm}$ algorithms, or any other algorithms of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_Vote**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$**:** Run $\mathsf{ProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V})$. If result is 1:
- add $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ to BB,
- compute $\sigma \leftarrow \mathsf{CreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}})$,
- send σ
Else send $\perp$

$\mathcal{O}$**Challenger_getCC**$(\mathtt{VC}_{\mathtt{id}})$**:** Run
$\sigma \leftarrow \mathsf{GetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}})$,
Send σ

$\mathcal{O}$**Challenger_Confirm**($\mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}$)**:** Run
$(\mu, S_\mu) \leftarrow \mathsf{ProcessConfirm}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}})$,
- if $(\mu, S_\mu) \neq \perp$: add (μ, S_μ) to $\mathtt{VC}_{\mathtt{id}}$ entry in BB,
Send μ

At the end of the game, the Attacker provides $(\mathtt{VC}^*_{\mathtt{id}}, \mu^*)$.

Define S_1^B as the event that $\mu^* = \mathtt{sVCC}^{\mathtt{id}}$ and $\nexists(\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}, \mu^*, S_\mu) \in$ BB for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_0^A\} - \Pr\{S_1^A\}| = \epsilon_{\text{key}},$$

which we consider to be negligible under the assumptions that the Verification Card private keys are uniformly chosen from the key space, the order of the group $\mathbb{G}$ is much higher than the number of voters, and the collision-resistance property of the hash function.

Game B.2 This is a transition based on indistinguishability. Let his be the game in which the algorithms UniqueRegister, CreateCC, GetCC and ProcessConfirm, from the game B.1 are changed into the algorithms RndUniqueRegister, RndCreateCC, RndGetCC and RndProcessConfirm (where the first 3 have been defined in game A.3), in which each long Choice Code and long Vote Cast Code is generated by using the random oracle $\mathcal{O}_{lc}$ from game A.3, which maps elements from $\mathbb{G}$ (the space of partial Choice Codes and of confirmation messages) to $\mathcal{A}_{lc}$ (the space of long Choice Codes and long Vote Cast Codes).

Specifically, the new function RndProcessConfirm is defined as follows:

> RndProcessConfirm(BB, $\mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}$) receives as input a bulletin board BB, a Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, a confirmation message $\mathtt{CM}^{\mathtt{id}}$, the Codes secret key $\mathtt{C}_{\mathtt{sk}}$ and the Vote Cast Code Signer public key $\mathtt{VCCs}_{\mathtt{pk}}$, and performs the following steps:
>
> - Checks that there is a vote entry in BB for the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, and that it has not been confirmed yet.
> - **It uses the oracle to compute a long Vote Cast Code value** $\overline{\mathtt{VCC}^{\mathtt{id}}} = \mathcal{O}_{lc}(\mathtt{CM}^{\mathtt{id}})$.
> - Takes the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ from the bulletin board, looks for the pair $[\mathsf{H}(\mathtt{VCC}^{\mathtt{id}}), \mathsf{Enc}^{\mathsf{s}}((\mathtt{sVCC}^{\mathtt{id}} | S_{\mathtt{VCC}^{\mathtt{id}}}); \mathtt{VCC}^{\mathtt{id}})]$ for which $\mathsf{H}(\mathtt{VCC}^{\mathtt{id}})$ is equal to the computed value $\mathsf{H}(\overline{\mathtt{VCC}^{\mathtt{id}}})$ and recovers the short Vote Cast Code $\overline{\mathtt{sVCC}^{\mathtt{id}}}$ and the signature $\overline{S_{\mathtt{VCC}^{\mathtt{id}}}}$, using the decryption algorithm $\mathsf{Dec}^{\mathsf{s}}$ with $\overline{\mathtt{VCC}^{\mathtt{id}}}$ as the key.
> - Checks that the retrieved short Vote Cast Code is correct by running $\mathsf{Verify}(\mathtt{VCCs}_{\mathtt{pk}}, \overline{\mathtt{sVCC}^{\mathtt{id}}}, \overline{S_{\mathtt{VCC}^{\mathtt{id}}}})$.

In case all the verifications succeed, the output of the algorithm is the pair $(\overline{\mathtt{sVCC}^{\mathtt{id}}}, \overline{S_{\mathtt{VCC}^{\mathtt{id}}}})$. Otherwise, the output is $\perp$.

The game B.2 is exactly as game B.1, but substituting these four algorithms. At the end of the game, the Attacker provides $(\mathtt{VC}^*_{\mathtt{id}}, \mu^*)$.

Define S_2^B as the event that $\mu^* = \mathtt{sVCC}^{\mathtt{id}}$ and $\nexists(\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}, \mu^*, S_\mu) \in \mathrm{BB}$ for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_1^B\} - \Pr\{S_2^B\}| = \epsilon_{\mathrm{prf}},$$

where ϵ_{prf} is the PRF-advantage of an efficient distinguisher algorithm, which is negligible considering that $f_{\mathtt{C}_{\mathtt{sk}}}$ is a pseudo-random function.

Game B.3 This transition is also based on indistinguishability: Let this be the game in which the algorithms MoreRndUniqueRegister, MoreRndCreateCC, MoreRndGetCC and MoreRndProcessConfirm are used, instead of RndUniqueRegister, RndCreateCC, RndGetCC and RndProcessConfirm from the previous game. In these new algorithms, the encryptions of the short Choice Codes and short Vote Cast Code and signature which are stored in the codes mapping table $\mathtt{CM}_{\mathtt{id}}$ corresponding to a voter are substituted by choosing random values from the same value space. Still, the Challenger will keep the list of short Choice Codes and Vote Cast Codes with their signatures internally, so that they can be provided during the voting phase if necessary.

While the first three algorithms have been already defined in game A.4, here we provide the implementation of the algorithm MoreRndProcessConfirm:

> MoreRndProcessConfirm$(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}})$ receives as input a bulletin board BB, a Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, a confirmation message $\mathtt{CM}^{\mathtt{id}}$, **the short Vote Cast Code** $\mathtt{sVCC}^{\mathtt{id}}$ **and its signature** $S_{\mathtt{VCC}^{\mathtt{id}}}$, and performs the following steps:
>
> - Checks that there is a vote entry in BB for the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, and that it has not been confirmed yet.
> - It uses the oracle to compute a long Vote Cast Code value $\overline{\mathtt{VCC}^{\mathtt{id}}} = \mathcal{O}_{lc}(\mathtt{CM}^{\mathtt{id}})$.
> - Takes the Codes Mapping Table $\mathtt{CM}_{\mathtt{id}}$ from the bulletin board and checks that there is an entry for the hash value of $\overline{\mathtt{VCC}^{\mathtt{id}}}$.
>
> In case all the verifications succeed, the output of the algorithm is the pair $\overline{\mathtt{sVCC}^{\mathtt{id}}}, \overline{S_{\mathtt{VCC}^{\mathtt{id}}}}$. Otherwise, the output is $\perp$.

The resulting game is as follows:

Configuration:

Challenger: Compute $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathsf{ID})$ in BB
Keep $(\mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
Init $\mathcal{K}$ to an empty list

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{MoreRndUniqueRegister}(1^\lambda, \mathtt{C_{sk}}, \mathtt{VCCs_{sk}}, \mathcal{K})$
If generated $\mathtt{VC_{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC_{id}}$ to ID_h
Keep the list of pairs $\{\mathtt{pCC}_i^{\mathtt{id}}, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k$
Post $(\mathtt{VC_{id}}, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \mathtt{VCks_{id}}, \mathtt{CM_{id}})$ in BB
Provide $\mathtt{SVK_{id}}, \{v_i, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k, \mathtt{BCK}^{\mathtt{id}}$ to the Attacker
Keep $\mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}}$

$\mathcal{O}$**Challenger_RegisterCorrupt**(): $\mathsf{MoreRndUniqueRegister}(1^\lambda, \mathtt{C_{sk}}, \mathtt{VCCs_{sk}}, \mathcal{K})$
If generated $\mathtt{VC_{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC_{id}}$ to ID_c
Keep the list of pairs $\{\mathtt{pCC}_i^{\mathtt{id}}, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k$
Post $(\mathtt{VC_{id}}, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \mathtt{VCks_{id}}, \mathtt{CM_{id}})$ in BB
Provide $(\mathtt{SVK_{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k)$ to the Attacker
Keep $\mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}}$

Voting: The Attacker generates votes and confirmations running the $\mathsf{CreateVote}$ and $\mathsf{Confirm}$ algorithms, or any other algorithms of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_Vote**$(\mathtt{VC_{id}}, \mathtt{V})$**:** Run $\mathsf{ProcessVote}(\mathsf{BB}, \mathtt{VC_{id}}, \mathtt{V})$. If result is 1:
- add $(\mathtt{VC_{id}}, \mathtt{V})$ to BB,
- compute $\sigma \leftarrow \mathsf{MoreRndCreateCC}(\mathtt{V}, \mathtt{C_{sk}}, \mathtt{CM_{id}}, \{\mathtt{pCC}_i^{\mathtt{id}}, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k)$,
- send σ
Else send $\perp$

$\mathcal{O}$**Challenger_getCC**$(\mathtt{VC_{id}})$**:** Run
$\sigma \leftarrow \mathsf{MoreRndGetCC}(\mathsf{BB}, \mathtt{VC_{id}}, \mathtt{C_{sk}}, \{\mathtt{pCC}_i^{\mathtt{id}}, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k)$,
Send σ

$\mathcal{O}$**Challenger_Confirm**$(\mathtt{VC_{id}}, \mathtt{CM}^{\mathtt{id}})$**:** Run
$(\mu, S_\mu) \leftarrow \mathsf{MoreRndProcessConfirm}(\mathsf{BB}, \mathtt{VC_{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}})$.
- If $(\mu, S_\mu) \neq \perp$: add (μ, S_μ) to $\mathtt{VC_{id}}$ entry in BB,
Send μ

At the end of the game, the Attacker provides $(\mathtt{VC}_{\mathtt{id}}^*, \mu^*)$.

Define S_3^B as the event that $\mu^* = \mathtt{sVCC}^{\mathtt{id}}$ and $\nexists(\mathtt{VC}_{\mathtt{id}}^*, \mathtt{V}, \mu^*, S_\mu) \in \mathrm{BB}$ for $\mathtt{VC}_{\mathtt{id}}^* \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_2^B\} - \Pr\{S_3^B\}| = \epsilon_{\mathrm{enc}},$$

where ϵ_{enc} is the advantage of an efficient algorithm that distinguishes the output of $\mathsf{Enc}^{\mathtt{s}}$ from a random function, which is negligible considering that $\mathsf{Enc}^{\mathtt{s}}$ is a pseudo-random function.

Game B.4 Let this be the game in which the algorithms $\mathsf{MoreRndUniqueRegister}$ and $\mathsf{MoreRndProcessConfirm}$ from the previous game B.3 are changed into $\mathsf{FakeMoreRndUniqueRegister}$ and $\mathsf{FakeMoreRndProcessConfirm}$, where the signature of the short Vote Cast Code $S_{\mathtt{VCC}^{\mathtt{id}}}$ is computed and verified using a random oracle $\mathcal{O}_h$ as follows:

- Given an initially empty list $\mathcal{T}_h$,

- Given an input short Vote Cast Code $\mathtt{sVCC}^{\mathtt{id}}$,

  if $\mathtt{sVCC}^{\mathtt{id}} \notin \mathcal{T}_h$: $\mathsf{H}'_s(\mathtt{sVCC}^{\mathtt{id}}) \xleftarrow{\$} \mathbb{Z}_n$[3], add $(\mathtt{sVCC}^{\mathtt{id}}, \mathsf{H}'_s(\mathtt{sVCC}^{\mathtt{id}}))$ to $\mathcal{T}_h$,
  otherwise $\mathsf{H}'_s(\mathtt{sVCC}^{\mathtt{id}}) \leftarrow \mathcal{T}_h(\mathtt{sVCC}^{\mathtt{id}})$

Then the signature is computed as $S_{\mathtt{VCC}^{\mathtt{id}}} = (\mathtt{ME}(\mathtt{sVCC}^{\mathtt{id}}))^d \bmod n$, where $\mathtt{ME}$ denotes a transformation with random padding over $\mathsf{H}'_s(\mathtt{VCC}^{\mathtt{id}})$ and d is the signing private key according to the definition in Section 2. The signature is verified by checking that $\mathtt{ME}(\mathtt{sVCC}^{\mathtt{id}})) = S^e_{\mathtt{VCC}^{\mathtt{id}}} \bmod n$, using the same H'_s function.

The Attacker has access to the same oracle $\mathcal{O}_h$ for verifying the signatures of short Vote Cast Codes.

The game B.4 is exactly as game B.3, but substituting these two algorithms. At the end of the game, the Attacker provides $(\mathtt{VC}^*_{\mathtt{id}}, \mu^*)$.

Define S_4^B as the event that $\mu^* = \mathtt{sVCC}^{\mathtt{id}}$ and $\nexists(\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}, \mu^*, S_\mu) \in \mathrm{BB}$ for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_3^B\} - \Pr\{S_4^B\}| = \epsilon_{\mathrm{h}},$$

where ϵ_{h} is the advantage of an efficient distinguisher algorithm between the hash function H_s and the random oracle $\mathcal{O}_h$, which is negligible in the random oracle model [7].

We can see that all the information the Attacker can get during the game is independent from the expected value $\mathtt{sVCC}^{\mathtt{id}}$, and therefore $\Pr\{S_4^B\}$ is the probability of guessing it at random, which is $\frac{1}{|\mathcal{A}_{vcc}|}$. We can conclude that

$$\mathrm{VCC}\mathsf{adv}[\mathcal{A}] = \epsilon_{\mathrm{key}} + \epsilon_{\mathrm{prf}} + \epsilon_{\mathrm{enc}} + \epsilon_{\mathrm{h}},$$

which is negligible given the assumption that the voters' keys are uniformly sampled from the key space, the order of the group $\mathbb{G}$ is larger than the number of voters, cryptographic hash functions are collision-resistant, the used pseudo-random function and the symmetric encryption scheme are unidistingible from a random distribution, and the properties of hash functions in the random oracle.

4.1.5 Game C

The Attacker confirms a valid vote without the collaboration of the voter. In this game, the Attacker can register honest and corrupt voters. For corrupt voters, the Attacker is provided the whole registration information including the Ballot Casting Key. However, for honest voters the Attacker is given the information required to send a vote, but not to confirm it. The public registration information from all the voters is, as in previous cases, published by the Challenger on the bulletin board. The Attacker has also the ability to cast and confirm votes by sending vote and confirmation messages that are processed by the Challenger.

[3]The hash output space for the signature scheme defined in Section 2.

The objective of the Attacker in this game is, for any registered honest voter, to successfully confirm a vote without the participation of such voter. The game is presented in three phases. At the end of them, the Attacker has to provide the information required to win.

Configuration:

Challenger: Compute $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathsf{ID})$ in BB
Keep $(\mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{Register}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_h
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $\mathtt{SVK}_{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}^k_{i=1}$ to the Attacker

$\mathcal{O}$**Challenger_RegisterCorrupt**(): $\mathsf{Register}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_c
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $(\mathtt{SVK}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}^k_{i=1})$ to the Attacker

Voting: The Attacker generates votes and confirmations running the $\mathsf{CreateVote}$ and $\mathsf{Confirm}$ algorithms, or any other algorithms of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_Vote**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$**:** Run $\mathsf{ProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V})$. If result is 1:
- add $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ to BB,
- compute $\sigma \leftarrow \mathsf{CreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}})$,
- send σ
Else send $\perp$

$\mathcal{O}$**Challenger_getCC**$(\mathtt{VC}_{\mathtt{id}})$**:** Run
$\sigma \leftarrow \mathsf{GetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}})$,
Send σ

$\mathcal{O}$**Challenger_Confirm**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}})$**:** Run
$(\mu, S_\mu) \leftarrow \mathsf{ProcessConfirm}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}})$,
- if $(\mu, S_\mu) \neq \perp$: add (μ, S_μ) to $\mathtt{VC}_{\mathtt{id}}$ entry in BB,
Send μ

At the end of the game, the Attacker provides $\mathtt{VC}^*_{\mathtt{id}}$.

Define S_0^C as the event that $\exists(\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}, \mu, S_\mu) \in \mathsf{BB}$ for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We define the Ballot Casting Key advantage of an adversary as:

$$\mathrm{BCKadv}[\mathcal{A}] = \left| \Pr\{S_0^C\} - \frac{1}{|\mathcal{A}_{bck}|} \right|$$

4.1.6 Security demonstration of Game C:

The development of the game C starts with a reduction from the ability of the Attacker to confirm a vote, to be able to guess the right confirmation message. After that, it proceeds similarly than in the case of games A and B, showing that the information the Attacker gets is independent from the confirmation message it has to guess. Finally, we will see that the best chance for the Attacker is to try to guess the valid Ballot Casting Key from which the confirmation message is constructed.

Game C.1 As in games A and B, we first proceed with a game transition based on failure, in which we rule out the possibility that the mapping information generated for different voters overlaps. Let this be the game in which the algorithm Register is substituted by the algorithm UniqueRegister defined in game A.1. In this new algorithm, the Challenger keeps track of the mapping information generated for the voters, and checks in every new registration that existing mapping information is not assigned to a new voter.

The resulting game is as follows:

Configuration:

Challenger: Compute $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathsf{ID})$ in BB
Keep $(\mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
Init $\mathcal{K}$ to an empty list

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{UniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_h
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $\mathtt{SVK}_{\mathtt{id}}, \{v_i, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k$ to the Attacker

$\mathcal{O}$**Challenger_RegisterCorrupt**(): $\mathsf{UniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_c
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $(\mathtt{SVK}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}_i^{\mathtt{id}}\}_{i=1}^k)$ to the Attacker

Voting: The Attacker generates votes and confirmations running the CreateVote and Confirm algorithms, or any other algorithms of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_Vote**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$**:** Run $\mathsf{ProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V})$. If result is 1:
- add $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ to BB,
- compute $\sigma \leftarrow \mathsf{CreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}})$,
- send σ

Else send $\perp$

$\mathcal{O}$**Challenger_getCC**$(\mathtt{VC}_{\mathtt{id}})$**:** Run
$\sigma \leftarrow \mathsf{GetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}})$,
Send σ

$\mathcal{O}$**Challenger_Confirm**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}})$**:** Run
$(\mu, S_\mu) \leftarrow \mathsf{ProcessConfirm}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}})$.
- If $(\mu, S_\mu) \neq \perp$: add (μ, S_μ) to $\mathtt{VC}_{\mathtt{id}}$ entry in BB,

Send μ

At the end of the game, the Attacker provides $\mathtt{VC}^*_{\mathtt{id}}$.

Define S_1^C as the event that $\exists(\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}, \mu, S_\mu) \in$ BB for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_0^C\} - \Pr\{S_1^C\}| = \epsilon_{\text{key}},$$

which we consider to be negligible under the assumptions that the Verification Card private keys are uniformly chosen from the key space, the order of the group $\mathbb{G}$ is much higher than the number of voters, and the collision-resistance property of the hash function.

Game C.2 Let this be the game in which the algorithm ProcessConfirm from game C.1 is substituted by PerfectProcessConfirm, which does the following:

> $\mathsf{PerfectProcessConfirm}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}, \mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}})$ receives as input a bulletin board BB, a Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, a confirmation message $\mathtt{CM}^{\mathtt{id}}$, **the voter's Ballot Casting Key** $\mathtt{BCK}^{\mathtt{id}}$**, the voter's Verification Card private key** $\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}$**, the voter's short Vote Cast Code** $\mathtt{sVCC}^{\mathtt{id}}$ **and signature** $S_{\mathtt{VCC}^{\mathtt{id}}}$, and performs the following steps:
>
> - Checks that there is a vote entry in BB for the Voting Card ID $\mathtt{VC}_{\mathtt{id}}$, and that it has not been confirmed yet.
> - **Checks that** $\mathtt{CM}^{\mathtt{id}} = ((\mathtt{BCK}^{\mathtt{id}})^2)^{\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}}$**. If so:**
>
> **Returns the short Vote Cast Code and signature** $(\mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}})$**.**
> - **Else, returns** $\perp$**.**

The new game is as follows:

Configuration:

Challenger: Compute $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathsf{ID})$ in BB
Keep $(\mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
Init $\mathcal{K}$ to an empty list

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{UniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_h
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $\mathtt{SVK}_{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k$ to the Attacker
Keep $\mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}}, \mathtt{VCks}_{\mathtt{VC}_{\mathtt{id}}}, \mathtt{SVK}_{\mathtt{id}}$

$\mathcal{O}$**Challenger_RegisterCorrupt**(): $\mathsf{UniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_c
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $(\mathtt{SVK}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k)$ to the Attacker
Keep $\mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}}, \mathtt{VCks}_{\mathtt{VC}_{\mathtt{id}}}, \mathtt{SVK}_{\mathtt{id}}$

Voting: The Attacker generates votes and confirmations running the $\mathsf{CreateVote}$ and $\mathsf{Confirm}$ algorithms, or any other algorithms of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_Vote**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$: Run $\mathsf{ProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V})$. If result is 1:
- add $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ to BB,
- compute $\sigma \leftarrow \mathsf{CreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}})$,
- send σ
Else send $\perp$

$\mathcal{O}$**Challenger_getCC**$(\mathtt{VC}_{\mathtt{id}})$: Run
$\sigma \leftarrow \mathsf{GetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}})$,
Send σ

$\mathcal{O}$**Challenger_Confirm**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}})$: Run
$\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}} \leftarrow \mathsf{GetKey}(\mathtt{SVK}_{\mathtt{id}}, \mathtt{VCks}_{\mathtt{id}})$,
Run $(\mu, S_\mu) \leftarrow \mathsf{PerfectProcessConfirm}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}, \mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}})$,
- if $(\mu, S_\mu) \neq \perp$: add (μ, S_μ) to $\mathtt{VC}_{\mathtt{id}}$ entry in BB,
Send μ

At the end of the game, the Attacker provides $\mathtt{VC}^*_{\mathtt{id}}$.

Define S_2^C as the event that $\exists(\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}, \mu, S_\mu) \in \mathrm{BB}$ for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_1^C\} - \Pr\{S_2^C\}| = \epsilon_{\mathrm{cm}},$$

where ϵ_{bck} is the probability of the Attacker finding a value $(\texttt{CM}^{\texttt{id}})^* \neq ((\texttt{BCK}^{\texttt{id}})^2)^{\texttt{VC}^{\texttt{id}}_{\texttt{sk}}}$ for which one of the following situations may happen during the execution of the original ProcessConfirm algorithm:

- $f_{\texttt{c}_{\texttt{sk}}}((\texttt{CM}^{\texttt{id}})^*) = \texttt{VCC}^{\texttt{id}}$.
- A long Vote Cast Code is generated as $f_{\texttt{c}_{\texttt{sk}}}((\texttt{CM}^{\texttt{id}})^*) = (\texttt{VCC}^{\texttt{id}})^* \neq \texttt{VCC}^{\texttt{id}}$ but still:
  - the entry $\mathsf{H}(\texttt{VCC}^{\texttt{id}}), \mathsf{Enc}^{\mathsf{s}}((\texttt{sVCC}^{\texttt{id}}|S_{\texttt{VCC}^{\texttt{id}}}); \texttt{VCC}^{\texttt{id}})$ for that Voting Card ID in the Codes Mapping Table $\texttt{CM}_{\texttt{id}}$ matches for the hash value of $(\texttt{VCC}^{\texttt{id}})^*$,
  - the pair $(\texttt{sVCC}^{\texttt{id}})^*, (S_{\texttt{VCC}^{\texttt{id}}})^*$ can be decrypted, such that $\mathsf{Verify}(\texttt{VCCs}_{\texttt{pk}}, (\texttt{sVCC}^{\texttt{id}})^*, (S_{\texttt{VCC}^{\texttt{id}}})^*) = 1$.

The probability of the first case is negligible given the collision resistance of the PRF function $f_{\texttt{c}_{\texttt{sk}}}()$ (which is that of the underlying hash function). The probability of the second case is related to the collision-resistance property of the hash function, and the security properties of the encryption function $\mathsf{Enc}^{\mathsf{s}}$, which tells us that the probability of decrypting a ciphertext with two different keys (and obtain the same message) is negligible, and the collision-finding probability of the underlying hash function from the signature scheme, which is also negligible. We can therefore conclude that ϵ_{cm} is negligible.

Game C.3 This game is the result of a transition based on indistinguishability. Let it be the game in which the algorithms UniqueRegister, CreateCC and GetCC from game C.2 are changed into the algorithms RndUniqueRegister, RndCreateCC and RndGetCC defined in game A.3, in which each long Choice Code and long Vote Cast Code is generated by using the random oracle $\mathcal{O}_{lc}$ (instead of the PRF function $f_{\texttt{c}_{\texttt{sk}}}$) which maps elements from $\mathbb{G}$ (the space of partial Choice Codes $(v_i^{\texttt{VC}^{\texttt{id}}_{\texttt{sk}}})$ and of confirmation messages) to $\mathcal{A}_{lc}$ (the space of long Choice Codes and long Vote Cast Codes).

The resulting game is the same than C.2, but changing these three algorithms. At the end of the game, the Attacker provides $\texttt{VC}^*_{\texttt{id}}$.

Define S_3^C as the event that $\exists(\texttt{VC}^*_{\texttt{id}}, \texttt{V}, \mu, S_\mu) \in \text{BB}$ for $\texttt{VC}^*_{\texttt{id}} \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_2^C\} - \Pr\{S_3^C\}| = \epsilon_{\text{prf}},$$

where ϵ_{prf} is the PRF-advantage of an efficient distinguisher algorithm, which is negligible considering that $f_{\texttt{c}_{\texttt{sk}}}$ is a pseudo-random function.

Game C.4 This transition is also based on indistinguishability: Let this be the game in which the algorithms MoreRndUniqueRegister, MoreRndCreateCC and MoreRndGetCC defined in game A.4 are used, instead of RndUniqueRegister, RndCreateCC and RndGetCC from the previous game. In these new algorithms, the encryptions of the short Choice Codes, short Vote Cast Code and signature which are stored in the codes mapping table $\texttt{CM}_{\texttt{id}}$ corresponding to a voter are substituted by choosing random values from the same value space. Still, the Challenger will keep the list of short Choice Codes and Vote Cast Codes and

signatures internally, so that they can be provided during the voting phase if necessary.

The new game is as follows:

Configuration:

Challenger: Compute $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
Initialize the empty voter lists $\mathsf{ID}_h, \mathsf{ID}_c, \mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$
Post $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathsf{ID})$ in BB
Keep $(\mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}})$
Init $\mathcal{K}$ to an empty list

Registration: The Attacker can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_RegisterHonest**(): $\mathsf{MoreRndUniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_h
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $\mathtt{SVK}_{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k$ to the Attacker
Keep $\mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}}, \mathtt{VCks}_{\mathtt{VC}_{\mathtt{id}}}, \mathtt{SVK}_{\mathtt{id}}$
Keep the list of pairs $\{\mathtt{pCC}^{\mathtt{id}}_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k$

$\mathcal{O}$**Challenger_RegisterCorrupt**(): $\mathsf{MoreRndUniqueRegister}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, \mathcal{K})$
If generated $\mathtt{VC}_{\mathtt{id}} \in \mathsf{ID}$, stop and return $\perp$. Otherwise:
Add $\mathtt{VC}_{\mathtt{id}}$ to ID_c
Post $(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}_{\mathtt{id}})$ in BB
Provide $(\mathtt{SVK}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, \{v_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k)$ to the Attacker
Keep $\mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}}, \mathtt{VCks}_{\mathtt{VC}_{\mathtt{id}}}, \mathtt{SVK}_{\mathtt{id}}$
Keep the list of pairs $\{\mathtt{pCC}^{\mathtt{id}}_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k$

Voting: The Attacker generates votes and confirmations running the CreateVote and Confirm algorithms, or any other algorithms of its choice, and can ask the Challenger to run the following algorithms several times:

$\mathcal{O}$**Challenger_Vote**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$**:** Run $\mathsf{ProcessVote}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{V})$. If result is 1:
- add $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ to BB,
- compute $\sigma \leftarrow \mathsf{MoreRndCreateCC}(\mathtt{V}, \mathtt{C}_{\mathtt{sk}}, \mathtt{CM}_{\mathtt{id}}, \{\mathtt{pCC}^{\mathtt{id}}_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k)$,
- send σ
Else send $\perp$

$\mathcal{O}$**Challenger_getCC**$(\mathtt{VC}_{\mathtt{id}})$**:** Run
$\sigma \leftarrow \mathsf{MoreRndGetCC}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{C}_{\mathtt{sk}}, \{\mathtt{pCC}^{\mathtt{id}}_i, \mathtt{sCC}^{\mathtt{id}}_i\}_{i=1}^k)$,
Send σ

$\mathcal{O}$**Challenger_Confirm**$(\mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}})$**:** Run
$\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}} \leftarrow \mathsf{GetKey}(\mathtt{SVK}_{\mathtt{id}}, \mathtt{VCks}_{\mathtt{id}})$,

Run $(\mu, S_\mu) \leftarrow \mathsf{PerfectProcessConfirm}(\mathsf{BB}, \mathtt{VC}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}, \mathtt{sVCC}^{\mathtt{id}}, S_{\mathtt{VCC}^{\mathtt{id}}})$,
- if $(\mu, S_\mu) \neq \perp$: add (μ, S_μ) to $\mathtt{VC}_{\mathtt{id}}$ entry in BB,
Send μ

At the end of the game, the Attacker provides $\mathtt{VC}^*_{\mathtt{id}}$.

Define S_4^C as the event that $\exists(\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}, \mu, S_\mu) \in \text{BB}$ for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_3^C\} - \Pr\{S_4^C\}| = \epsilon_{\text{enc}},$$

where ϵ_{enc} is the advantage of an efficient algorithm that distinguishes the output of $\mathsf{Enc}^{\mathsf{s}}$ from a random function, which is negligible considering that $\mathsf{Enc}^{\mathsf{s}}$ is a pseudo-random function.

Game C.5 Let this be the game in which the algorithm MoreRndUniqueRegister from the previous game C.4 is changed into FakeMoreRndUniqueRegister, where the signature of the short Vote Cast Code $S_{\mathtt{VCC}^{\mathtt{id}}}$ is computed using the random oracle $\mathcal{O}_h$ defined in game B.4, and the Attacker has access to the same oracle $\mathcal{O}_h$ for verifying the signatures of short Vote Cast Codes.

The game C.5 is exactly as game C.4, but substituting this algorithm. At the end of the game, the Attacker provides $\mathtt{VC}^*_{\mathtt{id}}$.

Define S_5^C as the event that $\exists(\mathtt{VC}^*_{\mathtt{id}}, \mathtt{V}, \mu, S_\mu) \in \text{BB}$ for $\mathtt{VC}^*_{\mathtt{id}} \in \mathsf{ID}_h$. We claim that

$$|\Pr\{S_4^C\} - \Pr\{S_5^C\}| = \epsilon_{\text{h}},$$

where ϵ_{h} is the advantage of an efficient distinguisher algorithm between the hash function H_s and the random oracle $\mathcal{O}_h$, which is negligible in the random oracle model [7].

We can see that all the information the Attacker can get during the game is independent from the expected value $\mathtt{CM}^{\mathtt{id}}$ it has to generate in order to successfully confirm a vote, and therefore $\Pr\{S_5^C\}$ is the probability of guessing it. Given that the Attacker knows the Verification Card private key $\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}$, $\mathtt{CM}^{\mathtt{id}} = ((\mathtt{BCK}^{\mathtt{id}})^2)^{\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}}$ and $\mathcal{A}_{bck} < \mathbb{G}$, the best chance for the Attacker is to guess the valid Ballot Casting Key, for which the probability is $\frac{1}{\mathcal{A}_{bck}}$. We can conclude that

$$\text{BCK}\mathsf{adv}[\mathcal{A}] = \epsilon_{\text{key}} + \epsilon_{\text{cm}} + \epsilon_{\text{prf}} + \epsilon_{\text{enc}} + \epsilon_{\text{h}},$$

which is negligible given the assumption that the voters' keys are uniformly sampled from the key space, the order of the group $\mathbb{G}$ is larger than the number of voters, cryptographic hash functions are collision-resistant, the used pseudo-random function and the symmetric encryption scheme are unidistingible from a random distribution without knowledge of the required key, the underlying hash function of the pseudo-random function and of the signature scheme are also collision-resistant, and given the properties of hash functions in the random oracle.

Finally, the advantage of an adversary for the cast-as-intended verification considering any of the attack vectors we have defined can be expressed as:

$$\mathrm{CaIadv}[\mathcal{A}] = \mathsf{max}(\mathrm{CCadv}[\mathcal{A}], \mathrm{VCCadv}[\mathcal{A}], \mathrm{BCKadv}[\mathcal{A}]),$$

which is negligible according to our analysis.

4.2 Conclusions

The probability of an adversary of breaking the cast-as-intended property depends on the attack vector it chooses. Given that we have shown that the advantage is negligible in any of the three cases, we can work directly with the probability of guessing specific values at random:

Attack 1 A voter tries to cast a vote using her voting device, but the voting device decides to change her selection for a different one of its interest. The voter does not detect the modification although she follows the verification protocol.

- The probability of the Attacker succeeding in this attack is $\frac{1}{|\mathcal{A}_{cc}|}$, under the assumption that the voter will notice the first time that the Choice Code returned is different than the one expected. Given that the short Choice Codes are 4-digit random values, this probability is 10^{-4}. In case of elections where voters can select more than one choice, the Attacker succeeds with probability 10^{-4t} in changing the value of t voting options without detection.

Attack 2 The voter casts a vote, performs the verification process and proceeds to confirm her vote, but the voting device refuses to send the confirmation message. The voter does not notice that her vote has not been confirmed, although she follows the verification protocol.

- The probability of success of the Attacker in this situation is $\frac{1}{|\mathcal{A}_{vcc}|}$, under the assumption that again, the voter will notice at once that the Vote Cast Code received does not match the value expected. The short Vote Cast Codes are 8-digit random values, and therefore this probability is 10^{-8}.

Attack 3 The voter starts the voting process, but changes her mind and does not cast any vote. The voting device ignores her decision and casts a vote on behalf of the voter. A similar case is that the voting device changes the voter selection, the voter detects it after executing the verification protocol and rejects to confirm that vote, but the voting device confirms it anyway.

- For this attack situation, the probability of success of the Attacker at the first try is $\frac{1}{|\mathcal{A}_{bck}|}$. The space of values of Ballot Casting Keys is the same as of short Vote Cast Codes, and therefore this probability is 10^{-8}. However, the system foresees that a voter may enter a wrong Ballot Casting Key by mistake, and gives 5 tries to confirm the vote before blocking the user. Therefore, the real success probability of an attacker is $\frac{5}{10^8}$.

Appendices

A Implementation details and relation to the base protocol

In this section we present the abstractions that have been done in the protocol model presented in Section 3 in relation to the implementation of the system used in the Swiss Online Voting System.

A.1 Entities

While the entities described in the protocol in Section 3 are intended to be as standard as possible and follow common descriptions in the literature, they can be mapped as follows in the Swiss Post particular setup:

- Election Authorities and Registrars are the same: voters are pre-registered to vote (in an already existing census) and the generation of voter information is done at the same time as the generation of the election configuration and by the same entity. Note that in Section 4, the same trust assumptions apply for both Election Authorities and Registrars, and therefore the security of the protocol is not affected by this particular setup.
- Bulletin Board Manager: although the description of the protocol refers to the publication of all election-related information in the Bulletin Board, in the current setup the information is not published, but stored by a voting server that contains a ballot box. The server side architecture is split into multiple micro-services, namely:
  - Extended Authentication,
  - Authentication Context,
  - Election Information Context,
  - Certificate Registry,
  - Vote Verification Context,
  - Voting Workflow,
  - Voter Materials,

  which are in charge of different tasks of the protocol.

  Additionally, the protocol steps to be run during the configuration and counting phases are run in an offline machine which receives election configuration data, and the ballot box after the election closes. The execution of these processes is also in charge of the Election Authorities.

  Given that auditors and internal parties (i.e., system administrators) may have access to both the server (containing the ballot box with the votes) and to the configuration and counting machines, in order to run and audit the election, we have considered important to treat the information as public for our security analysis, taking into account not only attacks from the outside, but also from the inside. As there is no Bulletin Board

in practice for publicly posting the information related to the election, the integrity and authenticity of data stored in and served by the server, the configuration and the counting machines is guaranteed thanks to the digital signatures generated by the different entities of the system:

- The Electoral Authorities digitally sign the related election configuration and voters' generated data which is generated in the configuration machine and is stored in the server (i.e., posted on the Bulletin Board).
- The voters digitally sign their votes and confirmation messages prior to sending them to the server, where they are stored in the ballot box for further integrity verification (see Section A.3).
- The server side services digitally sign the information they provide: authentication tokens, the ballot box, etc.

The integrity and authenticity of the information stored and provided by the server side relies on the key distribution process for producing such digital signatures, and the ability of auditors to check them (besides other information like proofs generated by the protocol algorithms) in order to assert that the result of the election is valid.

Assumptions regarding Bulletin Boards are also common in the literature: the requirement that only authorized entities can post in the Bulletin Board assumes some authentication mechanism for accessing it. In the same way, the requirement that the information cannot be changed or deleted once posted assumes some mechanism that prevents or detects it, like WORM systems (write-once, read-many), or continuous monitorization of the Bulletin Board contents.

A.2 Voter identifiers and key/value names

Given the micro-service architecture based on bounded contexts (where each own handles its own information and is responsible of a subset of the operations in an autonomous way) of the server side, the identity of the voter (or the voter-assigned resources) varies accross each one of the contexts. In this sense,

- Voting Card ID,
- Credential ID,
- Verification Card ID and
- Auth ID,

refer all to the same id (the voter identifier $\texttt{VC}_{\texttt{id}}$ in Section 3, to which we generally refer to as Voting Card ID) corresponding to the same voter. In the implementation, the one to one relationship between all these identifiers is stored in the Voter Materials context as part of the electoral roll information, and is put in the token issued to the voter after authentication, which will be included in all communications from the client side to the server side. This way each micro-service knows which (internal) id corresponds to that voter.

Besides that, other identifiers defined in the implementation description are: Voting Card and Verification Card sets, which have a one-to-one relation and refer to a set of credentials (credentials may be generated in different *batches*); Ballot IDs and Ballot Box IDs, which refer to different ballots with different voting options, and different ballot boxes where they may be stored; and finally the Election Event ID, which is used for distinguishing among different elections held in the same voting platform (concurrently or not).

The protocol model presented in Section 3 assumes one election with one ballot, one ballot box and one credential set, and therefore these identifiers are of no use in the protocol given that the information is adequately *split* by identifiers (votes are stored in separated ballot boxes according to their Ballot Box ID and Election Event ID, etc).

A.3 Authentication and private key provision

Details about the authentication layer have been deliberately omitted in previous sections for the sake of clarity, and given the fact that they are not relevant for proving the cast-as-intended verifiability of the system. The authentication layer managed by the electronic voting system is used not only to qualify a user as an authorized voter in the election, but also to transparently provide her with some cryptographic secrets, such as the Verification Card key pair $(\mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}})$ as described in the GetKey algorithm in Section 3.

Besides this key pair, the voter also retrieves a signature key pair $(pks_{\mathtt{id}}, sks_{\mathtt{id}})$. The signing private key is used to answer a challenge from the server, which proves that the voter was able to open the keystore and therefore that she is in posession of the correct Start Voting Key. The server then issues a digitally signed authentication token which grants that the voter has successfully passed this phase, and will be attached in all the communications from the voter to the server.

Voters will use their signing private keys to digitally sign their votes during the voting phase. Only votes signed with valid voter keys will be accepted by the system.

A.4 Vote correctness

The Codes Mapping Table generated by the protocol in order to retrieve the short Choice Codes and short Vote Cast Code for the voter is also used for checking other metadata contained in the vote which indicates whether the voters' choices are correct according to the election rules.

While in the protocol from Section 3 the only checked *rule* is that the voting options contained in the vote exist in the election (by checking that the hashes of the computed long Choice Codes correspond to entries in the Codes Mapping Table in the algorithm CreateCC), more information is checked in the practical setting. The details have not been included in the protocol description from Section 3 since this does not affect to the cast-as-intended verifiability property of our system.

Particularly, two tags are created at the time of configuration: the list and the candidate tag. The Election Authorities provide this information along with

the voting options: $\{(v_1 - \mathtt{tag}_1), \ldots, (v_k - \mathtt{tag}_k)\}$, and define the set of valid votes Ω to be those containing X list selections, and Y candidates.

During the creation of the vote to be cast, the tags corresponding to the chosen options are added into the vote, and therefore the output of $\mathsf{CreateVote}(\mathtt{VC}_{\mathtt{id}}, \{(v_1 - \mathtt{tag}_1), \ldots, (v_t - \mathtt{tag}_t\}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}})$ is a vote $\mathtt{V} = (\alpha, \beta, \gamma)$, where $\alpha \leftarrow c$, $\beta \leftarrow \{(\mathtt{pCC}^{\mathtt{id}}_{\ell} - \mathtt{tag}_{\ell})\}^{t}_{\ell=1}$, $\gamma \leftarrow (\tilde{c}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \pi_{\mathtt{sch}}, \pi_{\mathtt{exp}}, \pi_{\mathtt{pleq}})$. That is, for each partial Choice Code inside the vote, the tag for list or candidate corresponding to the original voting option is attached.

During the execution of the ProcessVote algorithm, an additional validation is done in order to ensure that the structure of the vote fulfills the election rules according to Ω.

In order to ensure that the tags are related to the voting options in the vote, each tag has been coupled with the original voting option at the time of generating the Codes Mapping Table during configuration, and in the same way the tags inside the vote are coupled with the corresponding partial Choice Codes during the voting phase, in order to look for a match in the voter's Codes Mapping Table: the long Choice Codes are computed as $\mathtt{CC}^{\mathtt{id}}_i = f_{\mathtt{C}_{\mathtt{sk}}}(v_i^{\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}} | \mathtt{tag}_i)$ both in the Register and in the CreateCC algorithms. This way, tags and voting options are cryptographically joined using a collission-resistant function.

As a final comment, we have to point out that this functionality does not affect to the voter privacy given that the only information leaked from the tags is that the vote has a correct structure.

References

[1] Adida, B., Neff, C.A.: Ballot casting assurance. In: Wallach, D.S., Rivest, R.L. (eds.) 2006 USENIX/ACCURATE Electronic Voting Technology Workshop, EVT'06, Vancouver, BC, Canada, August 1, 2006. USENIX Association (2006)

[2] Allepuz, J.P., Castelló, S.G.: Internet voting system with cast as intended verification. In: Kiayias, A., Lipmaa, H. (eds.) VOTE-ID. Lecture Notes in Computer Science, vol. 7187, pp. 36–52. Springer (2011)

[3] Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7237, pp. 263–280. Springer (2012), http://dx.doi.org/10.1007/978-3-642-29011-4_17

[4] Bellare, M.: New proofs for NMAC and HMAC: Security without collision-resistance. Cryptology ePrint Archive, Report 2006/043 (2006)

[5] Bellare, M., Kilian, J., Rogaway, P.: The security of the cipher block chaining message authentication code. J. Comput. Syst. Sci. 61(3), 362–399 (2000)

[6] Bellare, M., Rogaway, P.: PSS: Provably secure encoding method for digital signatures. Submission to IEEE P1363, August 1998 (1998), http://grouper.ieee.org/groups/1363/P1363a/contributions/pss-submission.pdf

[7] Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Proceedings of the 1st ACM Conference on Computer and Communications Security. pp. 62–73. CCS '93, ACM, New York, NY, USA (1993), http://doi.acm.org/10.1145/168588.168596

[8] Bellare, M., Rogaway, P.: The exact security of digital signatures-how to sign with RSA and Rabin. In: Proceedings of the 15th Annual International Conference on Theory and Application of Cryptographic Techniques. pp. 399–416. EUROCRYPT'96, Springer-Verlag, Berlin, Heidelberg (1996)

[9] Bernhard, D., Pereira, O., Warinschi, B.: On necessary and sufficient conditions for private ballot submission. Cryptology ePrint Archive, Report 2012/236 (2012)

[10] Bernhard, D., Cortier, V., Galindo, D., Pereira, O., Warinschi, B.: SoK: A comprehensive analysis of game-based ballot privacy definitions. In: IEEE Symposium on Security and Privacy 2015. IEEE Computer Society (5 2015)

[11] Chaum, D., Pedersen, T.P.: Wallet databases with observers. In: Brickell, E.F. (ed.) Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings. Lecture Notes in Computer Science, vol. 740, pp. 89–105. Springer (1992)

[12] Coron, J.: Optimal security proofs for PSS and other signature schemes. In: Knudsen, L.R. (ed.) Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2332, pp. 272–287. Springer (2002), http://dx.doi.org/10.1007/3-540-46035-7_18

[13] Cortier, V., Galindo, D., Glondu, S., Izabachène, M.: Election verifiability for Helios under weaker trust assumptions. In: Kutylowski, M., Vaidya, J. (eds.) Computer Security - ESORICS 2014 Proceedings, Part II. Lecture Notes in Computer Science, vol. 8713, pp. 327–344. Springer (2014)

[14] Damgård, I.: Commitment schemes and zero-knowledge protocols. In: Damgård, I. (ed.) Lectures on Data Security, Lecture Notes in Computer Science, vol. 1561, pp. 63–86. Springer Berlin Heidelberg (1999), http://dx.doi.org/10.1007/3-540-48969-X_3

[15] Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 263, pp. 186–194. Springer (1986)

[16] Gamal, T.E.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Blakley, G.R., Chaum, D. (eds.) CRYPTO. Lecture Notes in Computer Science, vol. 196, pp. 10–18. Springer (1984)

[17] Gerlach, J., Gasser, U.: Three case studies from Switzerland: E-voting (2009)

[18] Gjøsteen, K.: Analysis of an internet voting protocol. Cryptology ePrint Archive, Report 2010/380 (2010)

[19] Gjosteen, K.: The Norwegian internet voting protocol. Cryptology ePrint Archive, Report 2013/473 (2013)

[20] IETF: RFC 8018. PKCS#5: Password-Based Cryptography Specification Version 2.1 (2017)

[21] Maurer, U.: Unifying zero-knowledge proofs of knowledge. In: Preneel, B. (ed.) Advances in Cryptology - AfricaCrypt 2009. Lecture Notes in Computer Science, Springer-Verlag (Jun 2009)

[22] McGrew, D.A., Viega, J.: Flexible and efficient message authentication in hardware and software (2003)

[23] NIST SP 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf` (2007)

[24] Puigalli, J., Guasch, S.: Cast-as-intended verification in Norway. In: Electronic Voting. pp. 49–63 (2012)

[25] RSA Laboratories: PKCS#1: RSA cryptography standard version 2.1 (2002)

[26] Santis, A.D., Persiano, G.: Zero-knowledge proofs of knowledge without interaction (extended abstract). In: FOCS. pp. 427–436. IEEE Computer Society (1992)

[27] Schnorr, C.: Efficient identification and signatures for smart cards. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings. Lecture Notes in Computer Science, vol. 435, pp. 239–252. Springer (1989)