Swiss Online Voting Protocol

R&D

Scytl Secure Electronic Voting, Spain

Abstract. This document describes Scytl's Swiss electronic voting protocol, which is used in the Swiss Post Online Voting platform. The document first presents the protocol at a high level and with a generic set of participants, providing a syntax and a formal description that can be useful for other return code-based voting protocols, as well as for performing a formal analysis of the security of such schemes. The document provides details on the implementation and usability layers, finally providing an informal security analysis focused on the cast-as-intended verification mechanism.

Keywords: electronic voting protocols, binding election, cast-as-intended verifiability, malicious voting client, return 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 [16]. The first binding trials were held in 2004. Nowadays 14 cantons offer the electronic voting channel to their electors, which until recently has been restricted to be used by up to 10% of the eligible voters. In 2011 the Federal Council of Switzerland started a task force for studying the security issues of electronic voting. As a result, the Federal Council published, in 2013, a report with the requirements for extending the use of the electronic voting systems to a larger part of the electorate. This framework [12], which became binding in January 2014, provides requirements of functionality, security, verifiability and testing/certification which allow the electronic voting systems to be extended to 30%, 50% or up to 100% of the electorate. More specifically, while current electronic voting systems may be allowed to be used for up to 30% of the electorate provided that they fulfil a certain set of functional and security

requirements, systems to be used for up to 100% of the electorate are required to additionally provide verifiability features. Although the modality of electronic voting (DRE, remote, ...) is not specified in the report, it refers to electronic voting systems where the vote is cast electronically. In this paper, we will talk specifically of remote electronic voting systems.

Verifiability in remote electronic voting is traditionally divided in three types, which are related to the phase of the voting process which is verified [5]. 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.

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 cast-as-intended verification, and systems for up to 100% of the electorate are required to additionally provide methods for recorded-as-cast and counted-as-recorded verification, while also enforcing the separation of duties on operations impacting the privacy, integrity and verifiability of the system.

Our contribution. In this paper we present a protocol which provides cast-as-intended verification, according to the requirements of the Federal Council 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. The protocol is an evolution of the so-called *Norwegian voting protocol* [17, 18, 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 [6], from which the Norwegian scheme was initially derived.

The paper is structured as follows: the related work and the main contributions are detailed in Section 2. The syntax and a formal description of the solution are provided in Section 3. The building blocks and the instantiation of the protocol implemented for the Swiss Post Online Voting platform are presented in Sections 4 and 5. Then, some details on the usability and verifiability aspects are provided in Section 6. Finally, an informal analysis on the security

aspects of the protocol is provided in Section 7 and the conclusions are shown in Section 8.

2 Related work

There have been several proposals of cast-as-intended verification schemes during the last decade. In [10], Benaloh presents the *Immediate decryption* scheme, where the voter's device encrypts a vote and the voter is allowed to challenge the encryption generated. In case they choose to challenge it, the device reveals the randomness which was used to perform the encryption of the voting options. Using this randomness, the voter can check that the encrypted vote was constructed correctly. After the audit, the voting options are encrypted again with fresh randomness prior to casting the vote, so that the voter cannot use the randomness provided for audit as a proof to a third party of how they voted. However, this approach presents several drawbacks, such as usability (this randomness is a rather large string, cumbersome to be typed by a voter), and the fact that it does not allow for simple verification (i.e. verification must be done using a secondary computing device, under the assumption that at least one of the two devices is not compromised). This approach is used by the Helios voting system [2–4] and in the Wombat system [25]. A similar approach is used in VoteBox [26], by disclosing audited votes in the poll station local network in order to allow them to be verified.

A different approach consists on using the so-called return codes, which are targeted against malicious voting clients while enjoying some degree of usability [6, 17, 20, 21, 24]. In these proposals the voter selects their voting options and the voting client sends an encrypted vote to the remote voting servers, where return codes are calculated from the encrypted vote and sent back to the voter for verification. Voters possess a *verification card* where return codes (pre-computed in a configuration phase) are shown against matching voting options, and verification can be made by rather simple visual inspection. The current proposals assume that the voter can cast multiple votes. If the return codes do not match the selected voting options, then voters can cast another vote that invalidates the previous one (typically, this would happen if the voting client is malicious and encrypts voting options independently of the voter). However, some countries do not allow voters to cast multiple votes (such as France or Switzerland [12, 23]), so it is important to provide a proposal for these cases. Still, multiple voting is also used as a countermeasure for vote selling and voter coercion in such schemes, so they have to be taken into account when single voting is used[1].

One possible solution to support single vote casting is to add a *confirmation phase* to validate the vote after checking the return codes. In the first phase, the vote is encrypted and sent to the voting server, which calculates the return codes, stores the vote and communicates the return codes to the voter. In the second phase, the voter, after inspection of the return codes, sends a *confirmation code*

[1] For example, the risk of voter coercion or vote selling in Switzerland is assumed to be affordable given the fact that many voters already use the postal channel.

to the voting server, that stores it together with the ballot as a proof that the vote has been confirmed by the voter. Only votes with a valid voter confirmation code will be taken into account during the tally phase.

The return codes are computed from the probabilistic encryption of voting options, but at the same time they have to be deterministic: during the voting phase, the values computed by the server-side from an encrypted vote have to match those computed during the verification card generation phase (which happens at election configuration time) for the same set of voting options. Therefore, the randomness from the voting options encryption has to be removed for computing the return codes, which poses a serious risk on the vote secrecy. This was solved in the Norwegian voting system [17, 18, 24] by splitting the generation of the return codes in two independent entities: a ballot box server and a code generation server, which were assumed not to collude. To prevent these components from colluding and compromising the election privacy [17,18], both components were located in independent locations and managed by different companies. However, this approach is not always feasible to implement (the economic and organizational cost of setting up two different and independent environments are high).

In contrast, in the Puiggali-Guasch [6] scheme one of the previous independent entities is embedded in the voting client application. In their proposal there is no need of two separate components at the server-side of the voting platform, although then vote casting becomes computationally more expensive for the voting client (2,5 times more exponentiations than in the Norwegian protocol are required approximately). This is important considering the fact that the cryptographic operations done at the voting application level are often performed using web technologies such as Java Applets or Javascript, so the performance is slowed down when compared to a C/assembly implementation that uses lower level instructions. Naturally, ballot construction and casting needs to be executed in an acceptable time-frame to prevent voter disenfranchisement.

In this paper, we present a modification of the protocol [24] which, while very similar to [6] in the sense that it *moves* the operations of one of the server-side entities to the voting client application, reduces dramatically the number of operations to be performed at the voting client application (as it will be shown in Section 5, the cost of encryption and of proofs computation does not depend on the number of options anymore). Moreover, we add a confirmation phase in order to support single vote casting, so that it fulfils the requirements of the Swiss Federal Council [12].

3 Single voting with return codes

We start by presenting a syntax for a voting scheme with return codes, which will be later used to describe the protocol[2]. We build on existing definitions of

[2] As usual, the terms "scheme" and "protocol" can be read interchangeably without much loss of precision.

single-pass voting schemes, such as [11], and enrich them by adding a second interaction of the voter with the system, in order to confirm a cast vote.

3.1 Syntax

The scheme has the following participants: *Election Authorities*, who are in charge of setting up the election, computing the tally and publishing the results; *Voters*, who participate in the election by choosing their preferred options; *Registrars*, who are responsible for providing to the voters all the information they need to vote and, in particular, the return codes that provide the cast-as-intended integrity property; the *Voting Server*, which receives, processes and stores the ballots cast by eligible voters in the Ballot Box, and may as well publish some information; the *Voting Device*, which is in charge of casting a ballot given the voting options selected by the voter; the *Code Generator*, which is in charge of generating return codes from the ballots cast by the voting device. Finally, *Auditors*, who are responsible for verifying the integrity of the procedures run in the counting phase.

We assume that non-cryptographic election specifications such as the sets of administrators and voter identities are fixed in advance. Furthermore we assume a counting function $\rho : (V \cup \{\perp\})^n \to R$ is given, where Vs is the set of voting options, $\perp$ denotes abstention, n is the number of voters and R is the set of results. Voters may use credentials in order to be able to cast their ballots. However, how the voters obtain and use such credentials is out of the scope of this presentation.

There exists a public bulletin board $\mathtt{PBB}$ to which every algorithm in the voting scheme has read-only access to. As is common in the literature, some authorized parties have writing append-only access to it.

The voting scheme is characterized by the following protocols/algorithms:

- $\mathsf{Setup}(1^\lambda)$ is an interactive protocol run by the election authorities. On input a security parameter 1^λ, it generates and outputs an election public key pk_e and an election private key sk_e. In addition, it generates a global code generation public/private key pair (pk_c, sk_c), a signing public/private key pair (pk_s, sk_s), and the set of values which will represent the voting options: $V = \{v_1, \ldots, v_k\}$. The public keys pk_e, pk_c and pk_s, and the set of voting options V, are implicit inputs to the remaining algorithms.
- $\mathsf{Register}(\mathtt{id}, sk_c, sk_s)$ is an interactive protocol run by the registrars. It takes as input a voter identity $\mathtt{id}$, the global code generation private key sk_c and the signing private key sk_s. It outputs a voter's code generation public/private key pair $(pk_{\mathtt{id}}, sk_{\mathtt{id}})$, a set of voter return codes linked to voting options $\{v_i, \mathtt{RC}_i^{\mathtt{id}}\}_{i=1}^k$, a voter confirmation value $\mathtt{CV}^{\mathtt{id}}$, a voter finalization value $\mathtt{FC}^{\mathtt{id}}$ and a validity proof for such finalization code, $\Pi_{\mathtt{FC}^{\mathtt{id}}}$. Additionally, the registrars publish a set of reference values $\{\mathtt{RF}_i^{\mathtt{id}}\}_{i=1}^k$ that are linked to the codes $\{\mathtt{RC}_i^{\mathtt{id}}\}_{i=1}^k$. We sometimes refer to the set $\left\{\{v_i, \mathtt{RC}_i^{\mathtt{id}}\}_{i=1}^k, \mathtt{CV}^{\mathtt{id}}, \mathtt{FC}^{\mathtt{id}}\right\}$ as the *Verification Card*.

- $\mathsf{Vote}(\mathtt{id}, sk_{\mathtt{id}}, \{v_{j_1}, \ldots, v_{j_t}\})$ is a probabilistic algorithm run at the voting device. It receives as input a set of values $\{v_{j_1}, \ldots, v_{j_t}\}$, the voter identifier $\mathtt{id} \in \mathrm{ID}$ and the voter's code generation private key $sk_{\mathtt{id}}$; outputs a ballot b.
- $\mathsf{ProcessBallot}(\mathtt{BB}, b, \mathtt{id}, pk_{\mathtt{id}})$ is run by the voting server. It receives as input a ballot box $\mathtt{BB}$, a ballot b, an identity $\mathtt{id}$ and a voter's code generation public key $pk_{\mathtt{id}}$. It outputs 1 in case of success, 0 otherwise.
- $\mathsf{RCGen}(b, \mathtt{id}, sk_c)$ is an algorithm run by the code generator. On input a ballot b, the voter identifier $\mathtt{id}$ and the global code generation private key sk_c, it outputs an (unordered) set of return codes $\{\overline{\mathtt{RC}^{\mathtt{id}}}\}$ if the operation is successful, or $\perp$ in case of error/rejection. Typically this algorithm will look-up at $\mathtt{PBB}$ to check the list of legitimate reference values $\left\{\{\mathtt{RF}_i^{\mathtt{id}}\}_{i=1}^{k}\right\}_{\mathtt{id}\in\mathrm{ID}}$.
- $\mathsf{RCVerif}(\{v_{j_1}, \ldots, v_{j_t}\}, \{\overline{\mathtt{RC}^{\mathtt{id}}}\}, \{v_i, \mathtt{RC}_i^{\mathtt{id}}\}_{i=1}^{k})$ is an algorithm run by the voter. On input a set of voting options $\{v_{j_1}, \ldots, v_{j_t}\}$, a set of return codes $\{\overline{\mathtt{RC}^{\mathtt{id}}}\}$ and a voting card $\{v_i, \mathtt{RC}_i^{\mathtt{id}}\}_{i=1}^{k}$, it outputs 1 if $\{\mathtt{RC}_{j_i}^{\mathtt{id}}\}_{i=1}^{t} = \{\overline{\mathtt{RC}^{\mathtt{id}}}\}$ as sets, 0 otherwise.
- $\mathsf{Confirm}(\mathtt{CV}^{\mathtt{id}}, \mathtt{id}, sk_{\mathtt{id}})$ is a simple algorithm run by the voting device. On input a voter confirmation value $\mathtt{CV}^{\mathtt{id}}$, the voter identity $\mathtt{id}$, and the voter code generation private key $sk_{\mathtt{id}}$, it outputs a confirmation message $\mathtt{CM}^{\mathtt{id}}$.
- $\mathsf{FCGen}(\mathtt{CM}^{\mathtt{id}}, \mathtt{id}, sk_c, \Pi_{\mathtt{FC}^{\mathtt{id}}})$ is an algorithm run by the code generator. It receives as input a confirmation message $\mathtt{CM}^{\mathtt{id}}$, a voter identity $\mathtt{id}$, the global code generation private key sk_c and the proof $\Pi_{\mathtt{FC}^{\mathtt{id}}}$. It outputs a finalization code $\overline{\mathtt{FC}^{\mathtt{id}}}$ if the operation is successful, or $\perp$ in case of error/rejection.
- $\mathsf{Tally}(\mathtt{BB}, sk_e, \{\Pi_{\mathtt{FC}^{\mathtt{id}}}\}_{\mathtt{id}\in\mathrm{ID}})$ is an interactive protocol run by the election authorities. It takes as input the ballot box $\mathtt{BB}$, the election private key sk_e and the set of validity proofs $\{\Pi_{\mathtt{FC}^{\mathtt{id}}}\}_{\mathtt{id}\in\mathrm{ID}}$. It outputs a result $r \in R$ and a proof π of the tally correctness, or $\perp$.
- $\mathsf{Verify}(\mathtt{PBB}, \mathtt{BB})$ is an interactive protocol run by the auditors/election observers. It takes as input the bulletin board $\mathtt{PBB}$ and the ballot box $\mathtt{BB}$. The output is 1 if their verification of the counting process succeeds, 0 otherwise.

3.2 Workflow

Configuration phase: Election authorities define the set $\mathtt{ID}$ of voter identities participating in the election and run the Setup algorithm. They publish the election public key pk_e, the global code generation public key pk_c, the set of voter identities $\mathtt{ID}$, the signing public key pk_s and the set of voting options V in the bulletin board. They provide the global code generation private key sk_c to both the registrars and the code generator. Finally the signing private key sk_s is provided to the registrars.

Registration phase: Voters register to participate in the election. To register, a voter first provides their identity $id \in \mathtt{ID}$ to the registrars, who run the $\mathsf{Register}$ algorithm. The outputs $(pk_{\mathtt{id}}, sk_{\mathtt{id}})$, $\{v_i, \mathtt{RC}_i^{\mathtt{id}}\}_{i=1}^{k}$, $\mathtt{CV}^{\mathtt{id}}$, and $\mathtt{FC}^{\mathtt{id}}$ are provided to the voter, while the voter's code generation public key $pk_{\mathtt{id}}$, the proof $\Pi_{\mathtt{FC}^{\mathtt{id}}}$ and the reference values $\{\mathtt{RF}_i^{\mathtt{id}}\}_{i=1}^{k}$ are published in the bulletin board $\mathtt{PBB}$.

Voting phase: This phase consists of several steps:

1. The voter authenticates through the voting device to the voting server. If the authentication is successful, the values $\mathtt{id}, pk_{\mathtt{id}}$ are stored in the voting device. The voter chooses a set of voting options $\{v_{j_1}, \ldots, v_{j_t}\} \in V$ and enters them into the voting device as her choices for the election, together with the private key $sk_{\mathtt{id}}$[3]. The voting device then runs the Vote protocol and produces a ballot b. The ballot b and the voter identity $\mathtt{id}$ are sent to the voting server.
2. Upon reception of $(b, \mathtt{id})$, the voting server calls the ProcessBallot algorithm. In case the result of the execution is 1, the ballot box $\mathtt{BB}$ is updated with the ballot b and the voter identity $\mathtt{id}$, with the state *ballot received*. Otherwise, the voting device is notified of the error.
3. The code generator is notified of the new update in $\mathtt{BB}$ and executes the RCGen algorithm with the newly arrived ballot. In case the operation is successful, a set of return codes $\{\overline{\mathtt{RC}^{\mathtt{id}}}\}$ is generated and sent to the voting server, which updates the status of the ballot in the $\mathtt{BB}$ to *return code generated*, and forwards the return codes to the voting device. In case the operation is not successful the voting device is notified accordingly.
4. The voting device shows the voter the set of generated return codes $\{\overline{\mathtt{RC}^{\mathtt{id}}}\}$. The voter is then asked to confirm the ballot cast by providing the confirmation value $\mathtt{CV}^{\mathtt{id}}$ to the voting device, which they will do **only** in case the RCVerif algorithm accepts. The voting device then runs Confirm and outputs a confirmation message $\mathtt{CM}^{\mathtt{id}}$, which is sent to the voting server together with the voter identity $\mathtt{id}$.
5. The voting server forwards the confirmation message $\mathtt{CM}^{\mathtt{id}}$ to the code generator, which executes the FCGen algorithm. If the operation is successful, the resulting finalization code $\overline{\mathtt{FC}^{\mathtt{id}}}$ is sent back to the voting sever, which stores it together with the ballot, updates the ballot status to *confirmed* and forwards $\overline{\mathtt{FC}^{\mathtt{id}}}$ to the voting device. In case the operation is not successful, the voter is notified accordingly.
6. Finally, the voter checks whether the displayed finalization code $\overline{\mathtt{FC}^{\mathtt{id}}}$ matches the value $\mathtt{FC}^{\mathtt{id}}$ received during registration. In case of a successful verification, the received finalization code serves the voter as a confirmation of the correct submission and confirmation of their vote. Otherwise, they complain to the election administrators, and might need to cast their vote using a different channel (i.e. at a polling station).

Counting phase: The election authorities run the interactive protocol Tally on $\mathtt{BB}$, obtaining and publishing in the bulletin board the result r and the proof π, or set $r = \perp$ in case of error. The auditors run the Verify protocol. 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.

[3] How this key is provided to the voting device in the specific implementation is explained in Section 6.1

3.3 Trust Assumptions

The following conditions are assumed in order to provide cast-as-intended verification and voter privacy with the proposed protocol:

For **cast-as-intended verifiability**, it is assumed that the following entities, as pairs, are not simultaneously malicious: the voting device and (1) the code generator, (2) the registrar, or (3) the voting server; (4) the code generator and the registrar.

For **privacy**, the following conditions are necessary: (1) the voting device is not compromised; (2) the election authorities are honest; (3) the verification card contents are only known to the voter.

4 Building blocks

Encryption scheme. Our protocol uses the ElGamal asymmetric encryption scheme [15], which consists of three algorithms: key generation, encryption and decryption ($\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}$). The key generation algorithm KGen takes on 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, sk), where $pk \in \mathbb{G}$ such that $pk = g^{sk} \mod p$ and $sk \in \mathbb{Z}_q$. On input $m \in \mathbb{G}$ and the public key pk, the Enc algorithm chooses a random $r \in \mathbb{Z}_q$ and computes $(c_1, c_2) = (g^r, pk^r \cdot m)$. The Dec algorithm receives (c_1, c_2) and the private key sk and outputs $m = c_2/c_1^{sk}$.

Voting options. The voting options $V = \{v_1, \ldots, v_k\}$ are chosen as small bit-length primes belonging to the group $\mathbb{G}$. A vote is encoded as the product of voting options chosen by the voter (prior to encryption), and the individual voting options are recovered via factorisation after decryption. 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 not larger than p.

Pseudo-random function family. A function family is a map $F : K \times D \to R$, where K is the set of keys, D is the domain and R is the range. A pseudo-random 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. We use the HMAC algorithm as a PRF [8], parametrized by the key K.

Signature scheme. A signature scheme is defined by three probabilistic algorithms $\mathsf{SignKeyGen}, \mathsf{Sign}, \mathsf{SignVerify}$, that stand for key generation, signature generation and signature verification. Our protocol uses the RSA signature algorithm with the hash variant (or *RSA Full Domain Hash signature scheme (RSA-FDH)* [9]), therefore the key generation algorithm $\mathsf{SignKeyGen}$ outputs a pair of keys (pks, sks), for which $pks = \{pk_{\mathtt{rsa}}, N_{\mathtt{rsa}}\}$, $N_{\mathtt{rsa}} = p \cdot q$ where p and q are two distinct primes, and $sks = sk_{\mathtt{rsa}}$. The signature algorithm Sign takes as input a message m, which is not restricted to a specific space, and the private key sks, and outputs $\sigma = H(m)^{sk_{\mathtt{rsa}}} \mod N_{\mathtt{rsa}}$, where H denotes a hash function. The signature verification algorithm $\mathsf{SignVerify}$ takes as input the public

key pks, the message m and the signature σ, and checks that $H(m) = \sigma^{pk_{\mathtt{rsa}}} \bmod N_{\mathtt{rsa}}$. It outputs 1 if successful, 0 otherwise.

Non-interactive zero-knowledge proofs of knowledge. We use

$$\mathsf{EqDL}_G(g_1, \ldots, g_n, h_1, \ldots, h_n),$$

a generalization of the NIZK proof system [13], to prove in zero-knowledge that $\log_{g_1} h_1 = \log_{g_2} h_2 = \ldots = \log_{g_n} h_n$ for $g_1, \ldots, g_n, h_1, \ldots, h_n \in \mathbb{G}$ (with proof builder ProveEq and proof verifier VerifyEq); and the NIZK proof system [27] $\mathsf{PrDL}_G(g, h)$ to prove in zero-knowledge the knowledge of $\log_g h$ for $g, h \in \mathbb{G}$ (with proof builder ProveDL and proof verifier VerifyDL). G is a hash function mapping strings to $\mathbb{Z}_q$.

We additionally use the DecP NIZKPK scheme for proving the correct decryption of a ciphertext c. This proof is also based on the Chaum-Pedersen protocol [13], and a detailed description can be found in [14]. We denote the proving algorithm as ProveDec, which receives a ciphertext c, a message m (which corresponds to the decryption of c), and a private key sk, and outputs a proof of correct decryption π_{dec}; the verification algorithm is denoted as VerifyDec, and receives as input a ciphertext c, a message m (which corresponds to the decryption of c), and a proof of correct decryption π_{dec}, and outputs 1 in case the validation is successful, 0 otherwise.

Mixnet. We model a verifiable mixnet with two algorithms: Mix shuffles and transforms a set of input ciphertexts, and additionally generates proofs π_{mix} of correct mixing (shuffling and transformation). MixVerify takes as input the ciphertexts which were the input of Mix, the output shuffled and transformed ciphertexts, and the proof π_{mix} of correct mixing, and outputs 1 in case the verification is successful, 0 otherwise. In this implementation, we use the verifiable mixnet proposed by Stephanie Bayer and Jens Groth [7]. This mixnet has been proven by their authors to be sound, meaning that MixVerify will only output 1 given a correct execution of Mix, and zero-knowledge in the standard model, or in the random oracle model in case of using the Fiat-Shamir heuristic for making the proofs non-interactive.

5 A protocol for cast-as-intended verification with single voting

The protocol implementation consists of the following algorithms:

- Setup(1^λ): the algorithm chooses a group $\mathbb{G}$ and runs KGen to generate an encryption key pair (pk, sk). As discussed before, the voting options $V = \{v_1, \ldots, v_k\}$ are chosen as small bit-length primes belonging to the group $\mathbb{G}$. The algorithm then generates a random K to choose a pseudorandom function $f_K \in F$, and chooses hash functions H, G. The election public key is $pk_e = (pk, \mathbb{G}, H, G)$, and the election private key is sk_e, where $sk_e = sk$ if there is only one trustee; alternatively sk_e consists of the shares of sk if

there are several trustees (for instance, by using [22]). Finally, the global code generation key pair is set to $pk_c = \perp$, $sk_c = K$, and $\mathsf{SignKeyGen}$ is run and the result is set to be the signing key pair (pk_s, sk_s)[4].

- $\mathsf{Register}(\mathtt{id}, sk_c, sk_s)$: the algorithm runs KGen with input $\mathbb{G}$ to generate a keypair (pk, sk) which is set to be the voter public/private code generation[5] key pair $(pk_{\mathtt{id}}, sk_{\mathtt{id}}) \in \mathbb{G} \times \mathbb{Z}_q$. Then it generates the voter confirmation value $\mathtt{CV}^{\mathtt{id}}$ by selecting a random element from $\mathbb{G}$. For each voting option $v_i \in V$ it computes the corresponding return code $\mathtt{RC}_i^{\mathtt{id}} = f_{sk_c}(v_i^{sk_{\mathtt{id}}})$, and computes the finalization value $\mathtt{FC}^{\mathtt{id}} = f_{sk_c}((\mathtt{CV}^{\mathtt{id}})^{sk_{\mathtt{id}}})$. The validity proof for the finalization code $\Pi_{\mathtt{FC}^{\mathtt{id}}}$ is computed by running $\mathsf{Sign}(\mathtt{FC}^{\mathtt{id}}, sk_s)$. Finally, the set of reference values $\{\mathtt{RF}_i^{\mathtt{id}}\}_{i=1}^{k}$ is generated by computing $\mathtt{RF}_i^{\mathtt{id}} = H(\mathtt{RC}_i^{\mathtt{id}})$ for each return code.
- $\mathsf{Vote}(\mathtt{id}, sk_{\mathtt{id}}, \{v_{j_1}, \ldots, v_{j_t}\})$: the algorithm receives the voting options selected by the voter as input, sets $v = \prod_{l=1}^{t} v_{j_l}$ and encrypts them, obtaining $(c_1, c_2) = \mathsf{Enc}(pk, v)$. The algorithm then makes a partial computation of the return codes corresponding to such voting options using the voter private key $sk_{\mathtt{id}}$: $(v_{j_1}^{sk_{\mathtt{id}}}, \ldots, v_{j_t}^{sk_{\mathtt{id}}})$[6]. Finally, it also computes $(c_1^{sk_{\mathtt{id}}}, c_2^{sk_{\mathtt{id}}})$. The following NIZK proofs are computed to prove the correct computation of these values:
  - A proof $\pi_{enc} \leftarrow \mathsf{ProveDL}(g, c_1)$, which proves knowledge of the randomness used for computing the encryption of v.

  Two proofs to demonstrate that the voting options in the ciphertext (c_1, c_2) and the voting options used to for the partial computation of return codes are the same:
  - A proof $\pi_{exp} \leftarrow \mathsf{ProveEq}(g, c_1, c_2, pk_{\mathtt{id}}, c_1^{sk_{\mathtt{id}}}, c_2^{sk_{\mathtt{id}}})$ which proves that $(c_1^{sk_{\mathtt{id}}}, c_2^{sk_{\mathtt{id}}})$ are computed by raising the ciphertext (c_1, c_2) to the voter's code generation private key $sk_{\mathtt{id}}$ corresponding to the public key $pk_{\mathtt{id}}$.
  - A proof $\pi_{prod} \leftarrow \mathsf{ProveEq}\left(g, pk, c_1^{sk_{\mathtt{id}}}, c_2^{sk_{\mathtt{id}}} \cdot (v_{j_1}^{sk_{\mathtt{id}}}, \ldots, v_{j_t}^{sk_{\mathtt{id}}})^{-1}\right)$ which proves that the ciphertext $(c_1^{sk_{\mathtt{id}}}, c_2^{sk_{\mathtt{id}}})$ is the encryption of the product $(v_{j_1}^{sk_{\mathtt{id}}} \cdots v_{j_t}^{sk_{\mathtt{id}}})$ under the election public key pk_e.

  The result of the above computations is a ballot b consisting of

$$b = \left(\mathtt{id}, (c_1, c_2), (v_{j_1}^{sk_{\mathtt{id}}}, \ldots, v_{j_t}^{sk_{\mathtt{id}}}), (c_1^{sk_{\mathtt{id}}}, c_2^{sk_{\mathtt{id}}}), pk_{\mathtt{id}}, \pi_{enc}, \pi_{exp}, \pi_{prod}\right).$$

[4] Note that sk_c is not considered to be divided in shares in this protocol. This is due to the fact that the secrets for computing the return codes (sk_c and $sk_{\mathtt{id}}$) belong to two different entities that are assumed not to collude for providing vote secrecy. However, distributing sk_c might be considered to weaken the trust assumptions.

[5] Notice that this is formally an encryption key pair, but it is being used here differently.

[6] As explained in Section 2, return codes have to be computed between two entities which are assumed not to collude, in order to ensure vote secrecy. In this implementation, the voting device computes a partial computation in the Vote algorithm, while the voting server computes the final values in the ProcessBallot algorithm.

– $\mathsf{ProcessBallot}(\mathtt{BB}, b)$: in the first place, the algorithm checks if there already exists a ballot for the voter identity $\mathtt{id}$ in the ballot box $\mathtt{BB}$; if this is the case, it outputs 0. Otherwise, it continues by validating the NIZK proofs $\pi_{enc}, \pi_{exp}, \pi_{prod}$. In case all the validations are successful, 1 is returned.

– $\mathsf{RCGen}(b, \mathtt{id}, sk_c)$: the algorithm computes the set of return codes contained in ballot b as follows:

  - Computes the final return code value $\overline{\mathtt{RC}^{\mathtt{id}}_{jl}} = f_{sk_c}(v^{sk_{\mathtt{id}}}_{jl})$ for each of the partially computed return codes $(v^{sk_{\mathtt{id}}}_{j1}, \ldots, v^{sk_{\mathtt{id}}}_{jt})$ from b.
  - Checks that $\left\{\overline{\mathtt{RC}^{\mathtt{id}}_{j1}}, \ldots, \overline{\mathtt{RC}^{\mathtt{id}}_{jt}}\right\}$ is a subset of $\{\mathtt{RF}^{\mathtt{id}}_i\}_{i=1}^{k}$. In a positive case, the set of return codes $\{\overline{\mathtt{RC}^{\mathtt{id}}}\} = \left\{\overline{\mathtt{RC}^{\mathtt{id}}_{j1}}, \ldots, \overline{\mathtt{RC}^{\mathtt{id}}_{jt}}\right\}$ is output. In a negative case, $\perp$ is returned.

– $\mathsf{Confirm}(\mathtt{CV}^{\mathtt{id}}, \mathtt{id}, sk_{\mathtt{id}})$: the algorithm computes $\mathtt{CM}^{\mathtt{id}} = (\mathtt{CV}^{\mathtt{id}})^{sk_{\mathtt{id}}}$.

– $\mathsf{FCGen}(\mathtt{CM}^{\mathtt{id}}, \mathtt{id}, sk_c, \Pi_{\mathtt{FC}^{\mathtt{id}}})$: runs $\mathsf{SignVerify}(pk_s, \overline{\mathtt{FC}^{\mathtt{id}}}, \Pi_{\mathtt{FC}^{\mathtt{id}}})$, where $\overline{\mathtt{FC}^{\mathtt{id}}} = f_K(\mathtt{CM}^{\mathtt{id}})$. $\overline{\mathtt{FC}^{\mathtt{id}}}$ is returned if the signature verification is successful, $\perp$ otherwise.

– $\mathsf{Tally}(\mathtt{BB}, sk_e, \{\Pi_{\mathtt{FC}^{\mathtt{id}}}\}_{\mathtt{id}\in\mathtt{ID}})$: it runs $\mathsf{ProcessBallot}$ for all the ballots in the bulletin board which have a finalization code $\overline{\mathtt{FC}^{\mathtt{id}}}$ stored. Then for those which passed the verification it runs $\mathsf{SignVerify}(pk_s, \overline{\mathtt{FC}^{\mathtt{id}}}, \Pi_{\mathtt{FC}^{\mathtt{id}}})$, discarding those for which this validation failed. Ciphertexts c are extracted from the remaining ballots and passed as input to the mixnet, which runs the Mix algorithm. The resulting list of mixed ciphertexts $\{C_m\}$ is decrypted: for each ciphertext $c_z \in \{C_m\}$, $\mathsf{Dec}(c, sk_e)$ is run to obtain v_z (in case sk_e was divided in shares, a secret reconstruction algorithm [22] is used to recover the private key previous to decryption). Then the $\mathsf{ProveDec}$ algorithm is run with inputs the statement (c, v_z) and the witness sk_e. The cleartext v_z is factorized to recover from $v_z = v_1^{\beta_1} \cdots v_k^{\beta_k}$ the factors v_i such that $\beta_i = 1$. The small primes representing the voting options v_i are tested to belong to V. Otherwise, the whole factorized vote is discarded. Finally, the result r composed of the values v_i recovered from each vote is provided as the output, together with the proof π which is composed by the proofs of correct mixing and decryption: $\pi = (\pi_{mix}, \{C_m\}, \pi_{dec})$.

– $\mathsf{Verify}(\mathtt{PBB}, \mathtt{BB})$ performs the same validations than Tally: runs $\mathsf{ProcessBallot}$ for all the ballots in the ballot box which have a which have a finalization code $\overline{\mathtt{FC}^{\mathtt{id}}}$ stored. Then for those which passed the verification it runs $\mathsf{SignVerify}(pk_s, \overline{\mathtt{FC}^{\mathtt{id}}}, \Pi_{\mathtt{FC}^{\mathtt{id}}})$, discarding those for which this validation is not successful. It extracts the ciphertexts c from the ballots which have passed the previous validations and composes the list $\{C\}$. Then it parses π as $(\pi_{mix}, \{C_m\}, \pi_{dec})$ and verifies that the mixing was correct by running $\mathsf{MixVerify}(C, C_m, \pi_{mix})$. Finally it checks that the decryption of each ciphertext was correct by running $\mathsf{VerifyDec}$ from the NIZKPK scheme, using as input the statement $(c_z, \prod(\{v_i\}))$, for all the ciphertexts $c_z \in \{C_m\}$ and the proof $\pi_l \in \pi_{dec}$, where $\prod(\{v_i\})$ denotes the product of all the voting options

v_i in the z-th entry of r (belonging to the same ballot). The output is the result of these validations.

6 Usability layer

The protocol described in the previous sections may pose significant usability problems to the voters. In order to cast a vote, the voter is asked to type in the voting device a series of secret values from their voting card, such as the confirmation value $\mathtt{CM}^{\mathtt{id}}$ and the private key $sk_{\mathtt{id}}$. In order to confirm their vote, the voter is asked to compare the return codes $\overline{\mathtt{RC}^{\mathtt{id}}}$ shown by the voting device with those in their verification card. The same applies to the finalization code $\overline{\mathtt{FC}^{\mathtt{id}}}$.

The problem is that, according to current cryptographic key length recommendations [1], the aforementioned values have a length of 256 or 2048 bits, depending on whether they are the output of a symmetric or an asymmetric key cryptographic operation. To be more concrete, in case a Base32 encoding is used to represent such values, this implies 52 and 410 characters, respectively. It is clearly not realistic to ask a voter to perform such task.

Therefore, an additional layer for improving usability is required on top of the protocol from Section 5. This layer allows to reduce the length of the values in the verification card, and to provide the voter's code generation key to the voting device in a way that is transparent to the voter.

6.1 Private key provision

Details about the authentication layer have been deliberately omitted in previous sections, for the sake of clarity. Authentication consists on a username and password which are derived using a password-based key derivation function (PBKDF) from a secret value generated during registration and printed onto the voter's verification card. 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 voter's code generation key pair $(pk_{\mathtt{id}}, sk_{\mathtt{id}})$ in the following way: the voting device sends the username ($\mathtt{id}$ in the protocol) to the voting server, which checks that it is in the electoral roll (in the list $\mathtt{ID}$ in the protocol) and then sends back to the voting device a keystore with the corresponding private keys, together with a challenge. The voting device uses the password to open the keystore and uses the private keys it recovers to answer the challenge. The server then issues an authentication token which grants that the voter has successfully passed this phase.

6.2 Short Return Codes

The usability layer is in charge of generating short values $\{\mathtt{sRC}^{\mathtt{id}}\}_{i=1}^{k}, \mathtt{sFC}^{\mathtt{id}}$, that are printed in the verification card. One key ingredient of this layer is the length

of such values, which actually represents a neat trade-off between usability and security: the longer they are, the harder it is to guess them by a corrupted voting device, but the harder is to use them by the voter. Specifically, in the Swiss Post Online Voting platform they are of 4 and 8 numeric digits respectively[7].

Additionally, the registrar secretly generates a table which relates each code $\mathtt{sRC}_i^{\mathtt{id}}$ or $\mathtt{sFC}^{\mathtt{id}}$ to the corresponding long codes $\mathtt{RC}_i^{\mathtt{id}}$ or $\mathtt{FC}^{\mathtt{id}}$. We call this table the *mapping table*, and *mapping* to each one of the correspondences. During the voting phase, the code generator uses this table to obtain the corresponding short codes. The mapping table is designed to be an injective function from codes $\{\mathtt{RC}^{\mathtt{id}}\}_{i=1}^{k}, \mathtt{FC}^{\mathtt{id}}$ to short codes $\{\mathtt{sRC}^{\mathtt{id}}\}_{i=1}^{k}, \mathtt{sFC}^{\mathtt{id}}$.

Our implementation of the mapping table contains one entry for each (long) return code $\mathtt{RC}_i^{\mathtt{id}}$ of the form: $[H(\mathtt{RC}_i^{\mathtt{id}}), E_{\mathtt{RC}_i^{\mathtt{id}}}(\mathtt{sRC}_i^{\mathtt{id}})]$, where H denotes a hash function, and $E_k(m)$ denotes the encryption of the message m with a symmetric encryption algorithm[8] and a secret key k.

An additional entry is computed in the same way with the (long) finalization code $\mathtt{FC}^{\mathtt{id}}$ and the short finalization code $\mathtt{sFC}^{\mathtt{id}}$.

7 (Informal) Security Analysis

The protocol is focused on preventing a corrupt voting client from changing the voter intention without being detected, while maintaining the privacy of such voter in front of a malicious voting server/code generator. In this section we informally discuss how these security properties are fulfilled given the trust assumptions presented in Section 3.3.

7.1 Cast-as-Intended Verifiability

The voting device can try to modify the voter's intention without detection in two ways: (i) by showing to the voter return codes which do not correspond to the maliciously modified contents of the vote (but that correspond with those of the voter's choices); (ii) by confirming a vote without the participation of the voter.

For the first attack, the voting device could try to send a ballot where the encrypted options do not correspond to the partial computation of return codes. However, in that case the proofs π_{exp}, π_{prod} would not be successfully verified by the voting server. The collaboration of the code generator is needed to generate the return codes. However, the only way the code generator receives a ballot cast by the voting device is that the voting server verifies the proofs first. Therefore, the code generator and the voting device cannot collaborate in case of a honest voting server and the only strategy the voting device can follow is to guess the

[7] Since the voting device has only one chance to show the values to the voter, a brute force attack succeeds with probability at most 10^{-4t} in changing the value of t voting choices without detection.

[8] The SHA-256 hash and the AES-128 symmetric encryption algorithms are used in the current implementation.

return codes the voter expects. A brute force attack cannot be done in this case, since the voter will detect consecutive attempts of displaying wrong return codes.

A possibility for the second attack is that the voting device generates a fake cofirmation message, so that the code generator computes a fake finalization value. Even in case the code generator does not verify the proof of validity of this finalization value (because it colludes with the voting device), the election authorities or the auditors would detect that it is fake at the counting or audit phases, so that the vote would not be counted. The alternative is that the voting device guesses a valid confirmation message. In order to limit the possibility of a brute force attack, the voting server allows a limited number of retries.

7.2 Privacy

Privacy in electronic voting is understood as the property of maintaining the intention of a voter unknown. Besides recovering the voter selections or the encryption randomness from the voting device (which we assume that cannot happen because for privacy the voting device is assumed to be honest), there are two ways to attack the voter privacy in this scheme.

The first one is to target the voting options encryption. This can be done by brute forcing the encryption, by decrypting the votes without shuffling them (so that they could be connected to the voter's identities), or by recovering the shuffling permutation. However, according to the assumptions previously detailed and using a strong encryption algorithm, none of these attacks are feasible.

The second attack is to target the return code generation mechanism. The ballot cast by the voter includes some partial computations of the return codes, which consist on the voter selections raised to some exponent known by the voting device. As far as it does not reveal such secret exponent, neither the voting server nor the code generator (even in coalition) can compute back the voter's original voting options (see [17] for the analysis). Given that the relation between return codes and voting options is only known to the voter, neither the voting server nor the code generator (or any third party who could have access to them) can use the generated return codes to infer which are the choices selected by the voter.

Finally, a voter cannot copy a vote of another voter and cast it as it was theirs, so that they receive return codes matching those in their voting card. In order to do that they need to compute the exponentiation the original selections to an exponent they know (while not knowing the selections themselves). Otherwise, they would get return codes that they would not be able to understand (because they would belong to another verification card).

8 Conclusions

In this paper, we have presented the Swiss electronic voting protocol to be used in the Swiss Post Online Voting platform, and specifically we have focused on

the cast-as-intended verification in the case of single voting. This mechanism improves on the performance and infrastructure requirements of previous proposals using return codes. Besides a syntax (that could be useful to design other return code-based voting protocols) and a formal description of the scheme, we have provided various details on the implementation of this mechanism for the online voting platform. These details include techniques applied to improve the usability of the system without breaking vote privacy. Finally, an informal security analysis has been provided.

References

1. Crytographic key length recommendation. `http://www.keylength.com` (2015)
2. Adida, B.: Helios: Web-based open-audit voting. In: van Oorschot, P.C. (ed.) USENIX Security Symposium. pp. 335–348. USENIX Association (2008)
3. Adida, B., de Marneffe, O., Pereira, O.: Helios voting system, `http://heliosvoting.org`
4. Adida, B., de Marneffe, O., Pereira, O., Quisquater, J.J.: Electing a university president using open-audit voting: Analysis of real-world use of Helios. In: Proceedings of the 2009 conference on Electronic voting technology/workshop on trustworthy elections (2009)
5. 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)
6. 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)
7. 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`
8. Bellare, M.: New proofs for NMAC and HMAC: Security without collision-resistance. Cryptology ePrint Archive, Report 2006/043 (2006)
9. 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)
10. Benaloh, J.: Simple verifiable elections. In: Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006. pp. 5–5. EVT'06, USENIX Association, Berkeley, CA, USA (2006)
11. Bernhard, D., Pereira, O., Warinschi, B.: On necessary and sufficient conditions for private ballot submission. Cryptology ePrint Archive, Report 2012/236 (2012)
12. Chancellery, S.F.: Explications relatives à l'ordonnance de la chancellerie fédérale sur le vote électronique (OVotE). Available at `http://www.bk.admin.ch/themen/pore/evoting/07979` (2013)
13. Chaum, D., Pedersen, T.P.: Wallet databases with observers. In: Brickell, E.F. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 740, pp. 89–105. Springer (1992)

14. Cramer, R., Gennaro, R., Schoenmakers, B.: A secure and optimally efficient multi-authority election scheme. In: Fumy, W. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 1233, pp. 103–118. Springer (1997)
15. 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)
16. Gerlach, J., Gasser, U.: Three case studies from Switzerland: E-voting (2009)
17. Gjøsteen, K.: Analysis of an internet voting protocol. Cryptology ePrint Archive, Report 2010/380 (2010)
18. Gjosteen, K.: The Norwegian internet voting protocol. Cryptology ePrint Archive, Report 2013/473 (2013)
19. Kripp, M.J., Volkamer, M., Grimm, R. (eds.): 5th International Conference on Electronic Voting 2012, (EVOTE 2012), Co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC, July 11-14, 2012, Castle Hofen, Bregenz, Austria, LNI, vol. 205. GI (2012)
20. Lipmaa, H.: Two simple code-verification voting protocols. Cryptology ePrint Archive, Report 2011/317 (2011)
21. Malkhi, D., Margo, O., Pavlov, E.: E-voting without 'cryptography'. In: Blaze, M. (ed.) Financial Cryptography. Lecture Notes in Computer Science, vol. 2357, pp. 1–15. Springer (2002)
22. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO. pp. 129–140 (1991)
23. Pinault, T., Courtade, P.: E-voting at expatriates' MPs elections in France. In: Kripp et al. [19], pp. 189–195
24. Puigalli, J., Guasch, S.: Cast-as-intended verification in Norway. In: Kripp et al. [19], pp. 49–63
25. Rosen, A., Ta-shma, A., Riva, B., Ben-Nun, J.: Wombat voting, `http://www.wombat-voting.com/`
26. Sandler, D., Derr, K., Wallach, D.S.: Votebox: A tamper-evident, verifiable electronic voting system. In: van Oorschot, P.C. (ed.) USENIX Security Symposium. pp. 349–364. USENIX Association (2008)
27. Schnorr, C.: Efficient signature generation by smart cards. J. Cryptology 4(3), 161–174 (1991)