Towards formal verification and corrupted setup security for the SwissPost voting system

Sevdenur Baloglu, Sergiu Bursuc, Reynaldo Gil-Pons, Sjouke Mauw

University of Luxembourg

Abstract. The Swiss Post voting system is one of the most advanced cryptographic voting protocols deployed for political elections, offering end-to-end verifiability and vote privacy. It provides significant documentation and independent scrutiny reports. Still, we argue that two significant pillars of trust need to be further developed. One is formal verification accompanied by machine-checked proofs. The second is security in presence of a corrupt setup component. In this work, we propose formal specifications of a simplified version of the Swiss Post voting protocol and initial verification results with the Tamarin prover. We also propose a revised protocol design that mitigates risks from a corrupt setup, and a prototype implementation of necessary zero-knowledge proofs.

1 Introduction

The SwissPost voting system implements a cryptographic protocol aiming for strong security properties, including end-to-end verifiability and vote privacy. It is one of the most advanced voting systems aimed for national elections, featuring extensive documentation [1], continuous deployment and improvement [31], and security reports from independent researchers [22, 23, 25, 27, 28]. There have also been papers pointing out security weaknesses in various instantiations of the system, contributing to an overall strengthening of its security [15,24,26]. From a protocol perspective, the two main security properties that the SwissPost system aims to achieve are *vote privacy*, i.e. voter choices are private, and *end-to-end verifiability*, i.e. voters and auditors can obtain guarantees on the correct counting of voting choices. Furthermore, vote privacy should hold even if the network, voting servers and the bulletin board are corrupt. The guarantees expected for verifiability are even stronger: it should also hold even if, in addition to parties mentioned above, the voting platform of voters is corrupt.

The need for stronger security. SwissPost achieves these security properties through several interacting components: the *setup component* (SC); the *return codes and control components* ($\mathtt{CCR}_1, \mathtt{CCR}_2, \mathtt{CCR}_3, \mathtt{CCR}_4$); and the *voting server* (VS). The trust assumptions in this setting are that the setup component is honest, and that at least one out of the four control components is honest. The voting server mainly plays a forwarding and storage role, and can be corrupt. The setup component is therefore a single point of failure: simple attacks against privacy and verifiability properties are possible if the setup is corrupt.

This work was supported by the Luxembourg National Research Fund (FNR) under the grant agreement C22/IS/17238244/AVVA. This paper is an extended version of the short paper with the same name presented at the E-Vote-ID 2025 workshop.

This goes against the long sought goal of distributing trust in election systems, both for vote privacy [12, 30] and for election verifiability [9, 29]. For SwissPost, this is a special concern since it is already used or to be used in political elections in Switzerland [3]. The overall specification of the system, in particular that of the setup component, is quite complex and not formally verified. Furthermore, it is difficult to find the alignment between the specification and its software implementation, as discussed e.g. in one of the independent reports [25]. This leaves scope for attacks stemming in the specification, the implementation or the misalignment between the two. The setup component is a prime target in the overall architecture, and strengthening it is one of the main future goals mentioned in [1] (section Computational Proof).

The need for formal specifications and machine-checked proofs. The computational proof of security provided at [1] spans over four documents totalling over 400 pages, gathering specifications, security definitions and proofs (sections Computational proof, System Specification, Cryptographic Primitives and Verifier Specification from [1]). It is the best effort so far to fully specify and prove the security of an election system deployed for local or national elections. However, manual security proofs spanning over such length are error-prone and hard to verify independently. Furthermore, while the security definitions considered in [1] (section Computational Proof) are based on the standard approach of cryptographic games, it is hard to assess how they relate to state-of-the art definitions of e-voting security that have been developed for machine-checked cryptographic proofs [16, 21] or formal symbolic proofs [5, 6, 13, 20]. We need more abstract specifications that initially sacrifice some detail in order to make progress towards machine-checked or fully automated security proofs. Symbolic proofs for SwissPost in ProVerif are also provided at [1] (section Symbolic Models). They have some limitations that we aim to address with our Tamarin models. A general limitation of ProVerif is the lack of associative-commutative symbols and of a natural model of the Diffie-Hellman theory. Both play a crucial role in SwissPost. Furthermore, the verifiability proof at [1] is missing the "recorded-as-intended" property, called result integrity in our paper. This is an essential block in order to obtain a symbolic model of end-to-end verifiability that is sound with respect to more expressive mathematical models, as shown in [5, 17]. More generally, we aim towards an alignment of symbolic and cryptographic specifications.

Our contributions. We propose simplified specifications, a revised protocol design to achieve corrupt setup security, a prototype implementation of required zero-knowledge proofs for our revised design, and initial results on automated verification for two versions of the SwissPost protocol.

Specifications. We propose two protocol specifications - one symbolic and one cryptographic - for a simplified version of the SwissPost protocol. Our simplifications consist in, e.g., reducing the number of control components from four to two, removing unnecessary encryption layers, and focusing on at most two voting choices. We also remove the authentication process that allows voters to obtain their corresponding cryptographic voting keys from a start voting key, that is shorter and is included in the voting card: we hardcode the cryptographic key

directly in the voting card, thereby assuming that the voter authentication and the subsequent key download on the voting platform is secure. In the symbolic model, we need to make some further simplifications in order to perform automated analysis with Tamarin [8]. For example, we cannot perform multiplication of group elements. In some scenarios, e.g. when a return code is computed by combining several group elements, we can model the corresponding operation relying on the simpler theory of Diffie-Hellman exponentiation. In other cases this does not work, e.g. we don't yet capture ballots with multiple votes, since these are encoded as multiplied group elements. In Section 4 we discuss the potential impact of these simplifications.

Protocols. Based on our simplified description of SwissPost, we present our proposed protocol improvement, whose aim is to achieve privacy and verifiability when the setup component is corrupt. As also mentioned in [1] (section Computational Proof), it is not straightforward to cryptographically distribute the computations performed by the setup component, since they rely on block cipher-based primitives like hash functions and symmetric encryption. Even if distribution would be possible, the inherent design of the system, where the return code is reconstructed on a potentially corrupt server, allows its leakage and a subsequent privacy breach by combining it with data from a corrupt setup component. We therefore review the design of the protocol, and in particular the way of computing the return codes, while aiming to keep exactly the same voter experience and performance, not to hinder usability. The main idea is that the code can be recomputed directly on the voting platform, relying on a random mask to hide the final code from the server. The voting platform can reveal the correct return code by removing the mask from the term received in response from the voting server. When the voting platform is corrupt, it can compute a correct return code only if it encoded the correct vote in the ballot. When the setup component is corrupt and the voting platform is honest, a correct code can be computed only if the ballot constructed on the voting platform is correctly processed and recorded by election authorities. The cryptographic specifications for both protocols are discussed and presented in MSC diagrams in Section 2.

ZK proofs implementation. In the original SwissPost system, the setup component creates a list that contains hashes of return (pre-)codes in random order. This list is used at the time of voting to filter out, in a privacy-preserving way, any codes that do not respect the required format, i.e. corresponding to malformed votes. As shown in [24], it is essential to handle this check correctly in order to avoid attacks on verifiability, because otherwise e.g. multiple votes could be cast for targeted voters. If we assume that the setup component is corrupt, we cannot rely on this list anymore, since a correct lookup in the list would reveal the vote of the voter to the setup component (in collaboration with a corrupt server), which would compromise privacy. Instead, we have to directly prove the correctness of the vote encoded in the ballot. Specifically, we need to prove that one or more terms in the ballot are of the form $\mathtt{x}^{\mathtt{r}}$, for some (known and secret) random scalar $\mathtt{r}$ and $\mathtt{x} \in \{\mathtt{p}_1, \ldots, \mathtt{p}_n\}$ - a set of group elements representing eligible candidates. This can be done directly by combining a proof

of exponentiation and proof of inclusion in a set [11]. The challenge is to show that, for our case study, this implementation is practical and does not introduce significant overhead. We discuss our prototype implementation and results from experiments in Section 3.

Verification. We use Tamarin to automatically verify security for our symbolic specification, obtaining the expected results for verifiability, summarised in Table 2, confirming that our proposal can improve security when the setup is corrupt. We have also attempted to prove privacy, but we obtain attacks that cannot be confirmed, sometimes called false attacks in the Tamarin/ProVerif communities. We discuss formal verification results and challenges in Section 4.

2 Protocol specification and improvement

In Table 1 we present notation used in the paper. SwissPost relies on short human readable codes that we assume drawn from $\{0,1\}^\ell$. In symbolic or informal descriptions, we will use the symbol $\mathtt{H}$ to represent hash functions, sometimes implicitly assuming that the output of $\mathtt{H}$ is within the space of short codes. SwissPost relies on the following algebraic properties to combine terms contributed by various parties: $\mathsf{g}^{\mathsf{x}} \bullet \mathsf{g}^{\mathsf{y}} = \mathsf{g}^{\mathsf{x}+\mathsf{y}}$, $\mathsf{exp}(\mathtt{Enc}(\mathsf{x},\mathsf{k}),\mathsf{y}) = \mathtt{Enc}(\mathsf{x}^{\mathsf{y}},\mathsf{k})$ and $\mathtt{Enc}(\mathsf{x},\mathsf{k}) \circ \mathtt{Enc}(\mathsf{y},\mathsf{k}) = \mathtt{Enc}(\mathsf{x} \bullet \mathsf{y},\mathsf{k})$. Homomorphic encryption, modelled by the last two equations, is used only in the setup phase for an additional layer of security, but is not strictly necessary. We present this additional layer only in the cryptographic specification in supplementary material.

Table 1. Notation for cryptographic functions and domains.

Group of order $\mathtt{q}$ (candidates are encoded as elements of this group):	$\mathbb{G}$
Space for short codes:	$\{0,1\}^\ell$
Hash functions: $\mathtt{H}_1\colon \{0,1\}^* \mapsto \{0,1\}^*$, $\mathtt{H}_2\colon \{0,1\}^* \mapsto \mathbb{G}$, $\mathtt{H}_3\colon \{0,1\}^* \mapsto \{0,1\}^\ell$	
Generic hash function used in symbolic models:	$\mathtt{H}$
Public key encryption (ElGamal):	$(\mathtt{Enc}, \mathtt{Dec})$
Symmetric key encryption:	$(\mathtt{SEnc}, \mathtt{SDec})$
Product of group elements:	$\mathtt{x} \bullet \mathtt{y}$
Exponentiation of group element $\mathtt{x}$ to scalar $\mathtt{y}$:	$\mathtt{x}^{\mathtt{y}}$
Product of ElGamal ciphertexts:	$\mathtt{c}_1 \circ \mathtt{c}_2$
Exponentiation of ciphertext $\mathtt{c}$ to scalar $\mathtt{y}$:	$\mathsf{exp}(\mathtt{c},\mathtt{y})$

Protocol parties and trust assumptions. The SwissPost protocol is executed by the following parties:

- the *voting client* VC is used by voters to authenticate to the voting server, to create and cast ballots, and to verify return codes for ensuring their vote is correctly counted.
- the *setup component* SC is responsible for generating key material and voting cards that allow voters to authenticate, cast and verify votes.

– the *voting server* VS stores encrypted information obtained from the setup component that is downloaded and decrypted on the voting client after authentication; this allows the voter to cast and verify ballots, which are stored on the voting server for tallying if ballot verification is successful.
– the *return codes control components* $\mathtt{CCR}_1, \mathtt{CCR}_2$ participate in computing the return codes that are sent to the voter; they verify the ballot and ensure the computed return code corresponds to the encrypted vote.

For analysing security, we consider several scenarios under which a subset of these parties is assumed honest, and the other parties are assumed to be corrupt. In Table 2, we outline the main scenarios and whether we expect the corresponding properties of privacy and verifiability to hold. We use subscripts to make a distinction between SwissPost, which is the full SwissPost protocol as described in specifications online [1], and the two protocols that we consider in this paper: $\mathsf{SwissPost}_\mathbf{0}$ - simplified version of SwissPost used for specifications; and $\mathsf{SwissPost}_\mathbf{1}$ - our proposal for the case of a corrupt setup. In this paper we only make formal security claims about $\mathsf{SwissPost}_\mathbf{0}$ and $\mathsf{SwissPost}_\mathbf{1}$, but we expect that the same results apply to the full SwissPost protocol. The main trust assumption for $\mathsf{SwissPost}_\mathbf{0}$ is that the setup component and one of the control components is honest. In addition, for privacy to hold, the voting client should also be honest.

Table 2. Protocol parties and trust assumptions for SwissPost.

	VC	SC	$\mathtt{CCR}_1$	$\mathtt{CCR}_2$	VS	$\mathtt{ID}_c$	$\mathsf{SwissPost}_\mathbf{0}$		$\mathsf{SwissPost}_\mathbf{1}$	
$\mathcal{A}_1$:	H	H	H	C	C	C	✓	✓	✓	✓
$\mathcal{A}_2$:	C	H	H	C	C	C	✓	✗	✓	✗
$\mathcal{A}_3$:	H	C	H	C	C	C	✗	✗	✓	✓
corruption scenario††							ver	priv	ver	priv

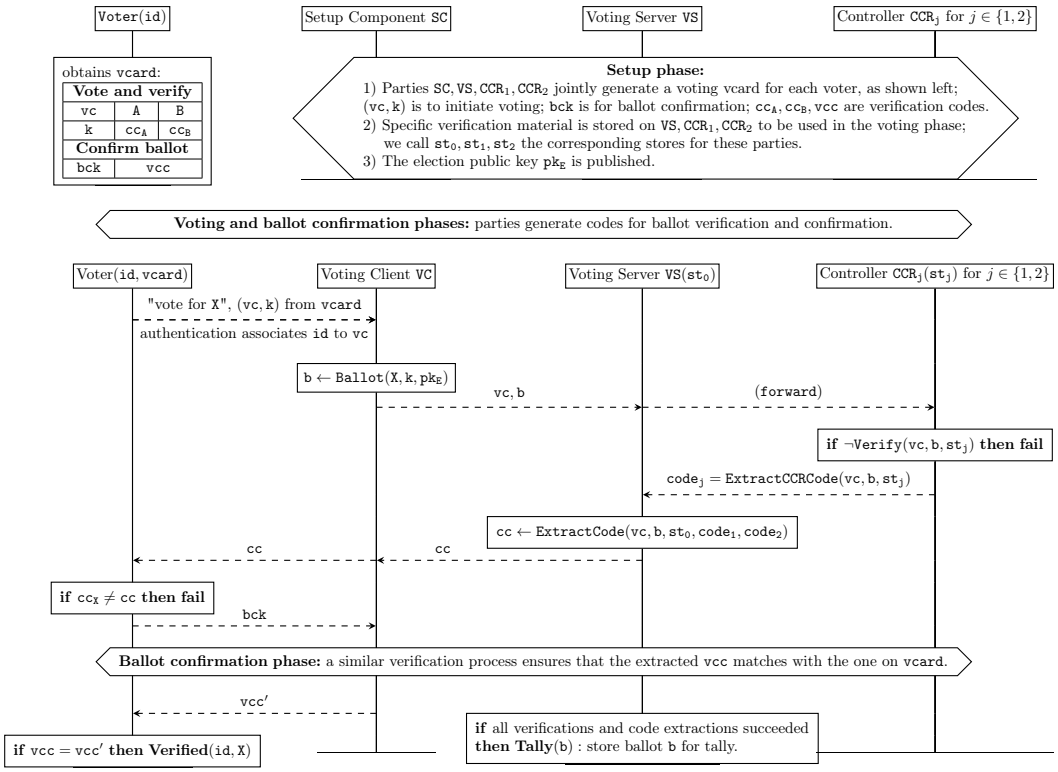
VC = Voting Client, SC = Setup Component, VS = Voting Server
$\mathtt{CCR}_1, \mathtt{CCR}_2$† = Return Codes Control Components, $\mathtt{ID}_c$ = subset of eligible voters
† : in SwissPost there are additional control components $\mathtt{CCR}_3$ and $\mathtt{CCR}_4$.
†† : Verifiability results are obtained with Tamarin; Privacy results are conjectured.

The voting card. Before voting begins, the voter obtains a voting card that follows the structure described at the top of Figure 1. For simplicity, we focus our description on two candidates. Furthermore, we use the same names, A and B, to represent these candidates as well as the corresponding group elements in the protocol implementation. The voting card contains:

– vc: the id that helps protocol parties retrieve associated data for the card.
– k: a secret key known only by the setup component and obtained by the VC from the voter; this is a simplification, since in reality k is too big to be written on the card: in SwissPost the vcard contains a so-called *start voting key* (svk) that is short and allows the VC to obtain the encrypted k from the server and decrypt it.

– $\mathtt{cc_A}, \mathtt{cc_B}$: return codes that correspond to A and B, respectively.
– bck: *ballot confirmation key* used by the voter to confirm that they received the expected return code, e.g. they received $\mathtt{cc_A}$ if they wanted to vote A.
– vcc: *vote confirmation code*, confirming the ballot is included in the tally.

Fig. 1. Illustration of the SwissPost protocol flow for two candidates A and B.



Protocol ceremony and generic description. In Figure 1 we present the generic structure of both $\mathsf{SwissPost_0}$ and $\mathsf{SwissPost_1}$ protocols. In the setup phase the parties generate key material and the voting cards sent to voters. In the voting phase, the voter enters the $(\mathtt{vc}, \mathtt{k})$ from the vcard and the desired vote on the voting client VC. The VC creates a corresponding ballot and posts it to the voting server. The CCR components verify the ballot and jointly provide information that allows the return code corresponding to the vote from the ballot to be reconstructed on the voting server. This return code is forwarded to the voting client and displayed to the voter, who should verify whether it matches the one displayed on the vcard. If it does, the voter enters bck, which allows parties to confirm that the voter agrees their vote is correctly encoded in the

ballot. Another code is extracted and forwarded to the voter. If this code matches vcc from the voting card, this confirms to the voter that their vote is included in the final tally.

SwissPost$_0$ specification. In Figure 2 we present how this general scheme is instantiated in SwissPost$_0$. The main idea is that the return codes $\mathtt{cc}_\mathtt{A}$ and resp. $\mathtt{cc}_\mathtt{B}$ are stored encrypted under keys $\mathtt{skcc}_\mathtt{A}$ and resp. $\mathtt{skcc}_\mathtt{B}$ that are derived by combining:

- information about the voting choice (A or B) to which that code corresponds; these are the terms $\mathtt{H}(\mathtt{A}^\mathtt{k})$ and $\mathtt{H}(\mathtt{B}^\mathtt{k})$ generated by the setup component.
- information about the id of the corresponding voting card; this is the vc term used as argument to the hash function.
- information contributed by the control components; these are the keys $\mathtt{k}_1, \mathtt{k}_2$ that are used as exponents.

All the keys k, $\mathtt{k}_1$, and $\mathtt{k}_2$ are distinct for each voter and there is a one-to-one correspondence between the set of voting cards and the set of eligible voters. The encrypted return codes are stored in a table denoted by CMT. During the voting stage, if the voter voted for $\mathtt{X} \in \{\mathtt{A}, \mathtt{B}\}$, the parties will jointly be able to recompute $\mathtt{skcc}_\mathtt{X}$, locate and decrypt the corresponding ciphertext from the table. In addition to the CMT table, the setup component also constructs a list containing hashes of pre-codes, denoted by $\mathtt{L}_{\mathtt{pCC}}$. The codes included in this list, of the form $\mathtt{H}(\mathtt{H}(\mathtt{X}^\mathtt{k}), \mathtt{vc})$ for $\mathtt{X} \in \{\mathtt{A}, \mathtt{B}\}$, allow the control components to verify that the vote included in the ballot is valid, as shown in the diagram for the voting phase. In addition to $\mathtt{X}^\mathtt{k}$, the ballot also contains an encryption of X along with a zero-knowledge proof that this is a correct encryption of X, i.e. that the plaintext inside the ciphertext equals the base of the exponentiation term. The zero-knowledge proof of ballot correctness is more complex in SwissPost than in SwissPost$_0$ because a ballot in SwissPost can contain (a product of) multiple votes. Finally, a similar process is used for generating confirmation codes at setup and for decrypting them when the voter performs ballot confirmation.

Problems when the setup is corrupt. Note that the return codes $\mathtt{cc}_\mathtt{X}$, and their associated votes, are known in advance by the setup component SC. The SC is not active during voting. The control components CCR collaborate to extract the corresponding decryption key from the ballot and decrypt $\mathtt{cc}_\mathtt{X}$. The crucial part is that the code is reconstructed in the clear on the voting server, which is important for verifiability, since parties need to agree the correct key is used for decryption. If the setup component is corrupt, this immediately breaks privacy: since the voting server is also assumed corrupt, the attacker can simply check which code is decrypted, the one for A or the one for B. Verifiability also easily breaks down, since the corrupt parties VS and SC can simply collude to send any code to the voter, without the need to involve the control components.

SwissPost$_1$ specification. The main idea to counter these problems in SwissPost$_1$ is to recompute the return code directly on the voting client, instead of the voting server. We also need to prevent the corrupt parties VS and SC to learn any information about the return code from the information revealed through voting and code extraction. For this, we change the structure of the

return code and the way in which data is aggregated. We also use a random exponent $\mathtt{r}$ instead of $\mathtt{k}$ in the ballot in order to hide the resulting code. Our proposal is described in Figure 3. The main differences are summarised in the table below:

	$\mathsf{SwissPost_0}$	$\mathsf{SwissPost_1}$
Code generation	$\mathtt{cc_X} \leftarrow\!\!\$\ \{0,1\}^\ell$	$\mathtt{cc_X} \leftarrow \mathtt{H}(\mathtt{H}(\mathtt{X}^{\mathtt{k_1}} \bullet \mathtt{X}^{\mathtt{k_2}})^{\mathtt{k}}, \mathtt{vc})$
Key generation	$\mathtt{skcc_X} \leftarrow \mathtt{H}(\mathtt{H}(\mathtt{X}^{\mathtt{k}})^{\mathtt{k_1}} \bullet \mathtt{H}(\mathtt{X}^{\mathtt{k}})^{\mathtt{k_2}}, \mathtt{vc})$	$\perp$ (not needed)
Ballot structure	$(\mathtt{e}, \mathtt{X}^{\mathtt{k}}, \Pi_0)$	$(\mathtt{e}, \mathtt{X}^{\mathtt{r}}, \Pi_1)$
Code extraction	decrypted on VS	computed on VC
Vote validity	hash list $\mathsf{L_{pCC}}$	zk proof

While in $\mathsf{SwissPost_0}$ the return code is generated randomly and encrypted under a computed key, in $\mathsf{SwissPost_1}$ the code itself is computed. In practice, the top layer hash function application would have to produce a human-readable code $\mathtt{cc_X}$ to be written on vcard. Note also that the order between the key $\mathtt{k}$ and the keys $\mathtt{k_1}, \mathtt{k_2}$ is switched in the computation of $\mathtt{cc_X}$ in $\mathsf{SwissPost_1}$. This is to allow the VC to apply $\mathtt{k}$ only after the pre-code information is returned and the masking randomness $\mathtt{r}$ is removed. When VC is honest, this design should prevent the corrupt parties from supplying a return code for any choice Y other than X, the voter's desired vote. For this, they would have to compute $\mathtt{Y}^{\mathtt{r}}$ from $\mathtt{X}^{\mathtt{r}}$, which we expect to be impossible under the *Subgroup Generated by Small Primes* cryptographic assumption considered in [1] (section Computational Proof). When VC is corrupt, if it cheats and encodes Y instead of X in the ballot, it would not be able to obtain the correct code $\mathtt{cc_X}$ from the information it gets in return. Lastly, we need to ensure that the vote encoded in the ballot lies within a valid set. The way we do it in $\mathsf{SwissPost_1}$ is significantly different from SwissPost and $\mathsf{SwissPost_0}$. Since we want to obtain privacy even when the setup component SC is corrupt, we cannot rely on the hash list $\mathsf{L_{pCC}}$, since SC knows the one-to-one correspondence between the elements of this list and the corresponding votes. Instead, we augment the zero-knowledge proof of ballot validity with a proof that the encoded vote lies within a valid set. In the next section, we present our construction and discuss its prototype implementation.

Assumptions and limitations. To obtain verifiability when SC is corrupt, we need to assume one of the two things: (1) the voting cards are correctly generated, or else (2) the return code components provide verifiable information that the exponentiation operations with $\mathtt{k_1}, \mathtt{k_2}$ are correctly performed with the key corresponding to each voter. These assumptions can be ensured with a combination of audits, zero-knowledge proofs or signatures - that we plan to explore in future work. Another limitation is that the randomness $\mathtt{r}$ is needed for recomputing the code for verification. This means that, for voters to verify their vote on a different machine than the one where they voted, we need an usable and secure way to transfer the randomness from one machine to another. Finally, if SC is corrupt, we cannot fully prevent ballot stuffing, since it knows all codes and can impersonate any voter. Formally, in this case, using the terminology from Section 4, we can obtain *result integrity* (ballots cannot be stuffed for honest voters that verified their votes), but not *strong result integrity*.

Fig. 2. Illustration of $\mathsf{SwissPost_0}$ for two candidates represented by group elements $\mathtt{A}$ and $\mathtt{B}$. For each voter $\mathtt{id}$, the id of the voting card $\mathtt{vc}$ and generated keys are distinct.

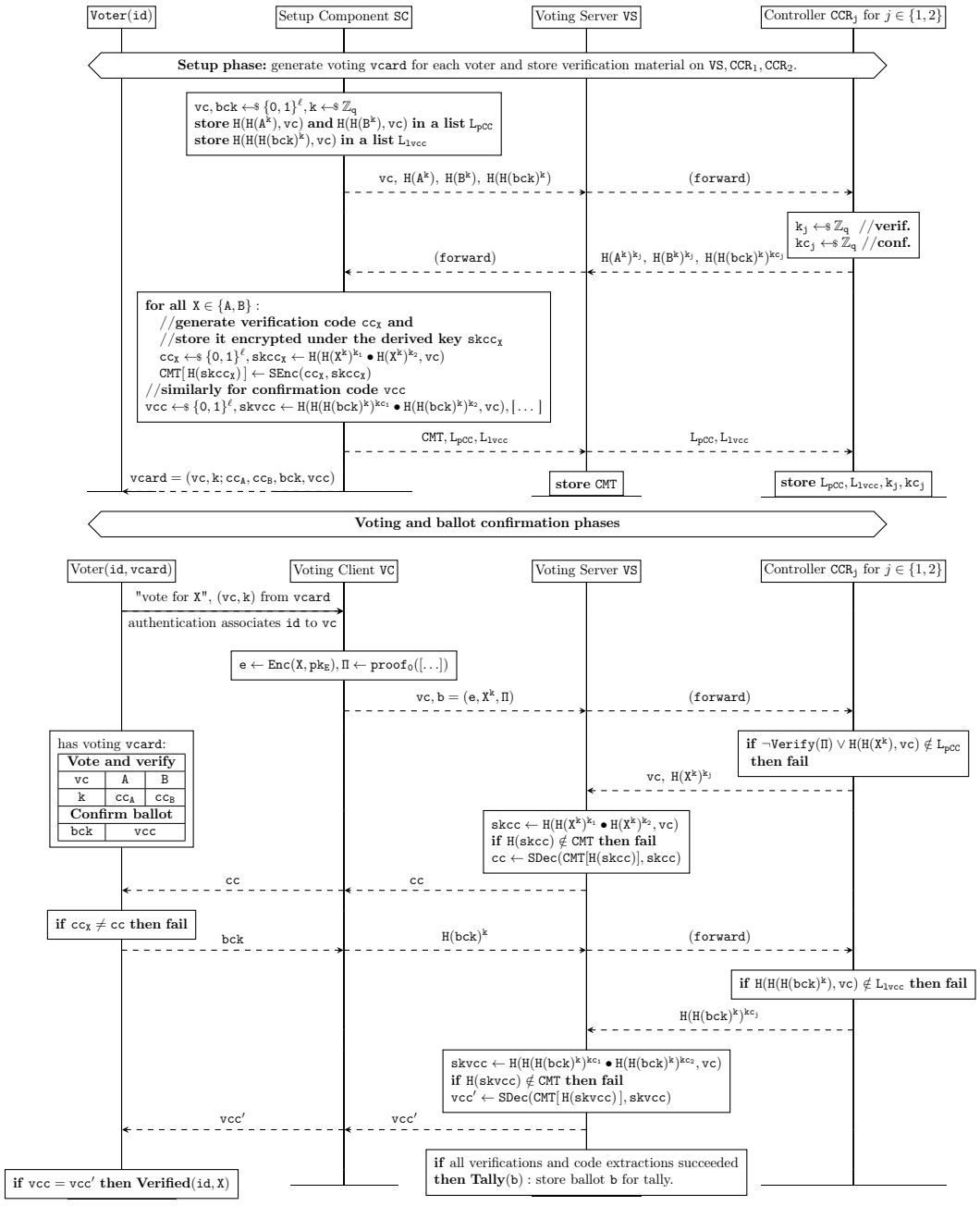
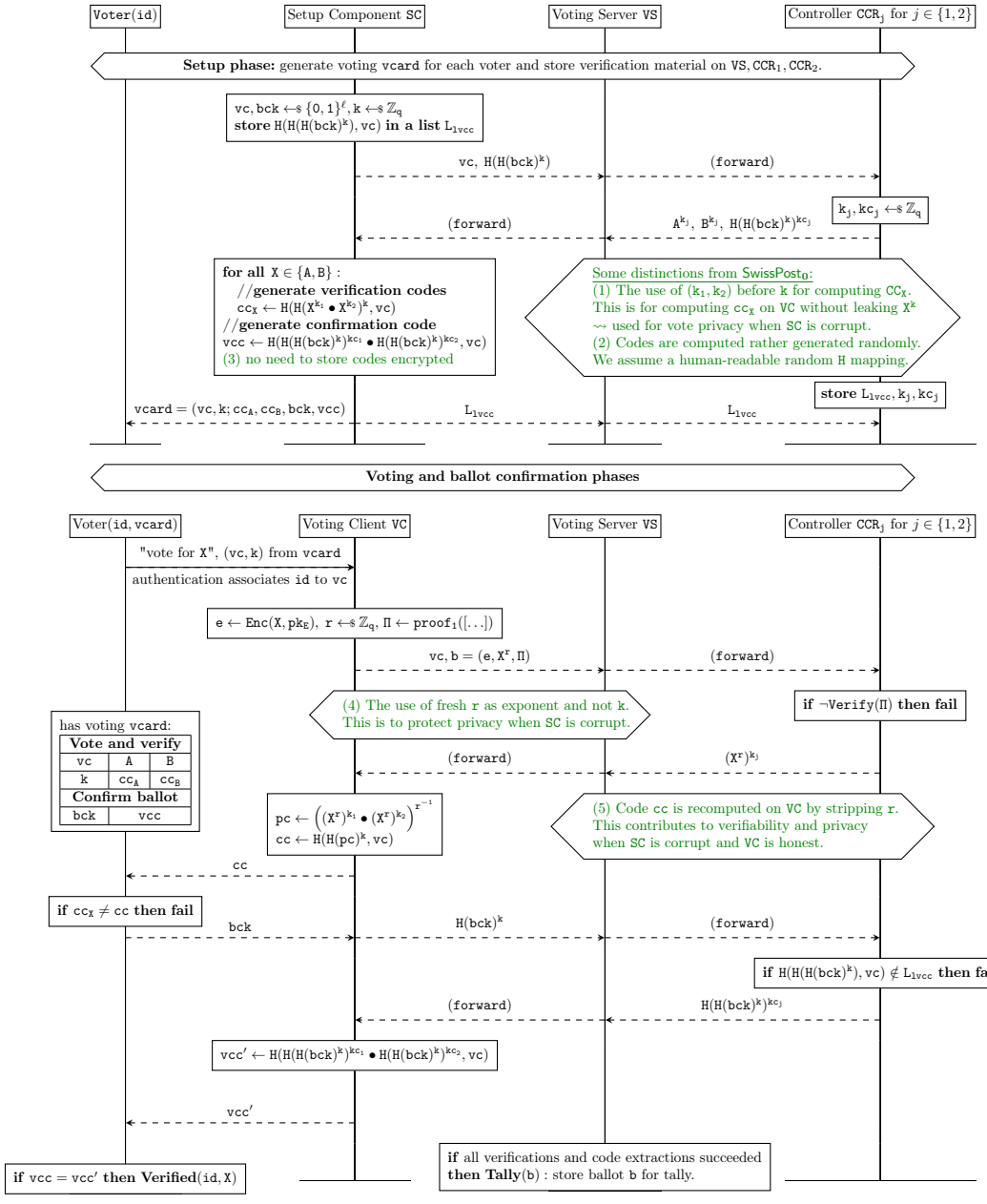


Fig. 3. Illustration of $\mathsf{SwissPost}_1$ for two candidates represented by group elements $\mathtt{A}$ and $\mathtt{B}$. For each voter $\mathtt{id}$, the id of the voting card $\mathtt{vc}$ and generated keys are distinct.



3 Zero-knowledge proofs

Fig. 4. Ballot construction for two votes $\mathtt{V_i}, \mathtt{V_j} \in \mathbb{G}$ in the original protocol and in ours.

`Procedure VC.Ballot`$(\mathtt{V_i}, \mathtt{V_j}, \mathtt{k}, \mathtt{pk_E})$ // SwissPost$_\mathbf{0}$

$\mathtt{e} \leftarrow \mathtt{Enc}(\mathtt{V_i} \bullet \mathtt{V_j}, \mathtt{pk_E})$, $\mathtt{pC_1} \leftarrow \mathtt{V_i^k}, \mathtt{pC_2} \leftarrow \mathtt{V_j^k}$, // $\Pi_{\mathtt{exp}}, \Pi_{\mathtt{eq}}$ are proofs that plaintext in $\mathtt{e}$
$(\mathtt{ex}, \Pi_{\mathtt{exp}}) \leftarrow \mathtt{exp}(\mathtt{e}, \mathtt{k})$, $\Pi_{\mathtt{eq}} \leftarrow \Pi\mathtt{EQ}(\mathtt{ex}, \mathtt{pC_1} \bullet \mathtt{pC_2})$, // is equal to the product of votes, i.e. the
$\Pi = (\mathtt{ex}, \Pi_{\mathtt{exp}}, \Pi_{\mathtt{eq}})$, $\mathtt{b} = (\mathtt{e}, \mathtt{pC_1}, \mathtt{pC_2}, \Pi)$, **return** $(\mathtt{b}, \bot)$ // exponentiation bases in $\mathtt{pC_1}, \mathtt{pC_2}$

`Procedure VC.Ballot`$(\mathtt{V_i}, \mathtt{V_j}, \mathtt{k}, \mathtt{pk_E})$ // SwissPost$_\mathbf{1}$, for allowed set of votes $\{\mathtt{V_1}, \ldots, \mathtt{V_n}\}$

$\mathtt{e} \leftarrow \mathtt{Enc}(\mathtt{V_i} \bullet \mathtt{V_j}, \mathtt{pk_E})$, $\mathtt{r} \leftarrow\!\!\$\ \mathbb{Z}_\mathtt{q}$, $\mathtt{rC_1} \leftarrow \mathtt{V_i^r}$, $\mathtt{rC_2} \leftarrow \mathtt{V_j^r}$, // use fresh $\mathtt{r}$ instead of $\mathtt{k}$
$(\mathtt{ex}, \Pi_{\mathtt{exp}}) \leftarrow \mathtt{exp}(\mathtt{e}, \mathtt{k})$, $\Pi_{\mathtt{eq}} \leftarrow \Pi\mathtt{EQ}(\mathtt{ex}, \mathtt{rC_1} \bullet \mathtt{rC_2})$, // same proofs as above for SwissPost$_\mathbf{0}$
$\Pi\mathtt{set} \leftarrow \Pi\mathtt{SET}(\mathtt{r}, \mathtt{g}^\mathtt{r}, (\mathtt{rC_1}, \mathtt{rC_2}), \{\mathtt{V_1}, \ldots, \mathtt{V_n}\})))$, // $\Pi\mathtt{set}$ is the new proof that bases of $\mathtt{rC_1}, \mathtt{rC_2}$
$\Pi = (\mathtt{ex}, \Pi_{\mathtt{exp}}, \Pi_{\mathtt{eq}}, \mathtt{g}^\mathtt{r}, \Pi\mathtt{set})$, $\mathtt{b} = (\mathtt{e}, \mathtt{rC_1}, \mathtt{rC_2}, \Pi)$, **return** $(\mathtt{b}, \mathtt{r})$ // are in the allowed set

For simplicity, in Section 2 we only presented the ballot structure for one voter choice. Our Tamarin models also only consider one choice. In reality, elections with SwissPost may allow multiple choice ballots, with multiple choices encoded as a product of corresponding group elements. In Figure 4, we show the difference between the ballot in the original protocol, and our modified version, for the case when two voter choices are allowed. As explained in Section 2, in addition to zk proofs used by SwissPost$_\mathbf{0}$, in SwissPost$_\mathbf{1}$ we need to provide a zero-knowledge proof to show that the votes encoded in the ballot lie within a valid set $\mathtt{S}$. In particular, for each vote represented by a term $\mathtt{rC_i}$, the voting client must prove that $\mathtt{rC_i} = \mathtt{V_i^r}$ for some private values $\mathtt{r}$ and $\mathtt{V_i} \in \{\mathtt{V_1}, \ldots, \mathtt{V_m}\} = \mathtt{S}$. This last step is essential, otherwise a corrupt voting client could exploit algebraic properties within the underlying group to cast more votes than allowed, as described in one of the attacks from [24]. To align with standard notation for sigma protocols [11], from here on we denote $\mathtt{V_i}$ by s_i, $\mathtt{r}$ by x, $\mathtt{rC_i}$ by $w_i\ (= s_i^x)$ and g^x by v.

We propose a construction that uses a combination of standard zero-knowledge protocols for exponentiation and for inclusion in a set. To define the problem more precisely, let $\mathbb{G}$ be a group of prime order q, $x \in \mathbb{Z}_q$ and $v, w_1, \ldots, w_n \in \mathbb{G}$ and $\mathtt{S} \subseteq \mathbb{G}$ with $n \leq |\mathtt{S}|$. We denote by $\Pi_{\mathtt{exp}} = \Pi\mathtt{SET}(x, v, (w_1, \ldots, w_n), \mathtt{S})$ a non-interactive zero-knowledge proof whose verification algorithm ensures the following relation:

- *Public statement:* $v, w_1, \ldots, w_n, \mathtt{S}$
- *Secret witness:* $x \in \mathbb{Z}_q, \{s_1, \ldots, s_n\} \subseteq \mathtt{S}$
- *Relation:* $v = g^x \wedge w_1 = s_1^x \wedge \ldots \wedge w_n = s_n^x$

The verifier can check that for some $s_1, \ldots, s_n \in \mathtt{S}$ the relation is true, but it doesn't know which elements of $\mathtt{S}$ they are. A particular case is when $|\mathtt{S}| = 1$,

Fig. 5. Non-interactive zero-knowledge proof protocol.

$\text{Commit}_S(x, s_i, v, w)$

assert $v = g^x \wedge w = s_i^x$;
for all $j \in \{1, \ldots, m\}$:
  $c_j \leftarrow\!\!\$\ \mathcal{C}; x'_j \leftarrow\!\!\$\ \mathbb{Z}_q$
  $v'_j \leftarrow g^{x'_j}$
  $z_j \leftarrow (x'_j + x \cdot c_j) \bmod q$
  $w'_j \leftarrow s_j^{z_j} \cdot w^{-c_j}$
endfor
$w'_i \leftarrow s_i^{x'_i}$
return $\big(v', w', x', z, c\big)$

$\text{GenPrf}_S(x, s_i, v, w, x', w', z, c, C)$

$c_i \leftarrow C$
$w'_i \leftarrow s_i^{x'_i}$
for all $j \in \{1, \ldots, m\} \setminus \{i\}$:
  $c_i \leftarrow c_i \oplus c_j$
endfor
$z_i \leftarrow (x'_i + x \cdot c_i) \bmod q$
return (z, c)

$\text{VrfyPrf}_S(v, w, v', w', z, c, C)$

for all $j \in \{1, \ldots, m\}$:
  if $g^{z_j} \neq v'_j \cdot v^{c_j} \vee s_j^{z_j} \neq w'_j \cdot w^{c_j}$:
    return `false`
  endif
endfor
if $\bigoplus_{j=1}^{m} c_j \neq C$:
  return `false`
endif
return `true`

$\texttt{ΠSET}(x, (s_i, s_j), v, (w_i, w_j), \mathtt{S})$

$\big(v'_i, w'_i, x'_i, z_i, c_i\big) \leftarrow \text{Commit}_S(x, s_i, v, w_i)$
$\big(v'_j, w'_j, x'_j, z_j, c_j\big) \leftarrow \text{Commit}_S(x, s_j, v, w_j)$
$C \leftarrow \mathtt{H}(v, (w_i, w_j), \big(v'_i, w'_i\big), \big(v'_j, w'_j\big))$
$(z_i, c_i) \leftarrow \text{GenPrf}_S(x, s_i, v, w_i, x'_i, w'_i, z_i, c_i, C)$
$(z_j, c_j) \leftarrow \text{GenPrf}_S(x, s_i, v, w_j, x'_j, w'_j, z_j, c_j, C)$
return $\big(\big(v'_i, w'_i, z_i, c_i\big), \big(v'_j, w'_j, z_j, c_j\big)\big)$

$\texttt{VerifySET}(\mathtt{v}, (\mathtt{w_1}, \mathtt{w_2}), (\mathtt{v'_1}, \mathtt{w'_1}, \mathtt{z_1}, \mathtt{c_1}), (\mathtt{v'_2}, \mathtt{w'_2}, \mathtt{z_2}, \mathtt{c_2}), \mathtt{S})$

$C \leftarrow \mathtt{H}(v, (w_1, w_2), \big(v'_1, w'_1\big), \big(v'_2, w'_2\big))$
$r1 \leftarrow \text{VrfyPrf}_S(v, w_1, v'_1, w'_1, z_1, c_1, C)$
$r2 \leftarrow \text{VrfyPrf}_S(v, w_2, v'_2, w'_2, z_2, c_2, C)$
return $r1 \wedge r2$

which gives the usual *proof-of-exponentiation* relation (s_1^x, g^x, s_1) where only the value of x is secret. Let $\mathtt{S} = (s_1, s_2, \ldots, s_m)$, and $\mathtt{H}$ be a random oracle with range $\mathcal{C}$ (this set must be closed under the xor operation, and of superpolynomial size on the security parameter). For the case $n = 2$ the non-interactive proof system $\Phi_S = (\texttt{ΠSET}, \texttt{VerifySET})$ for the relation

$\{(x, (s_i, s_j), v, w) \in \mathbb{Z}_q \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \colon v = g^x \wedge w = \big(s_i^x, s_j^x\big) \wedge s_i \in \mathtt{S}\}$ is as follows:

- on input $(x, (s_i, s_j), v, w, \mathtt{S})$, $\texttt{ΠSET}$ computes $\big(\big(v'_i, w'_i, z_i, c_i\big), \big(v'_j, w'_j, z_j, c_j\big)\big) \in \big(\mathbb{G}^m \times \mathbb{G}^m \times \mathbb{Z}_q^m \times \mathcal{C}^m\big)^2$.
- on input $(v, w, (v'_1, w'_1, z_1, c_1), (v'_2, w'_2, z_2, c_2), \mathtt{S})$, $\texttt{VerifySET}$ validates the proof.

The definition of the Non-interactive Zero-Knowledge protocol is in Figure 5. Our construction is based on standard OR and exponentiation proofs from [11]. For the full specification and proofs see the appendix.

Performance considerations. The zero-knowledge protocol above requires certainly more computation than the original protocol. That might imply that the protocol becomes impractical in the context of voting protocols. To test this

Table 3. Performance results of the zero-knowledge protocol. The global parameters for the tests were a random Sophie-Germain prime of 3072 bits for the group $\mathbb{G}$ and 100 candidates.

	Clock speed	Vote generation	Time to verify votes
Personal PC	3.6 GHz	2s per vote	3s for 5 votes with 8 cores
HPC	2.6 GHz	2s per vote	67s for 5000 votes with 128 cores

claim, we developed a proof-of-concept implementation of the non-interactive version of the protocol in Python, using **gmpy2**[1] to accelerate computation. Out proof of concept implementation code is in supplementary materials [2]. The results of the experiments are shown in Table 3. The voting process from the voter point of view should take less than a minute, while the response time from the server should take not more than a few seconds. Taking into account that our implementation is just a proof of concept, we believe the proposed zero-knowledge protocol performance does not affect the usability of the system.

4 Towards formal verification

In supplementary material [2], we present formal specifications for the protocols $\mathsf{SwissPost}_{\mathtt{0}}$ and $\mathsf{SwissPost}_{\mathtt{1}}$ from Section 2. Namely, we present

- formal cryptographic specifications, with complete specifications of algorithms for the setup, voting and verification phases.
- formal symbolic specifications in the form of Tamarin code.

Both the cryptographic and the symbolic specifications are aimed as first steps towards computer-aided fully verified proofs. In the cryptographic setting, they can be plugged into definitional frameworks developed for proofs with EasyCrypt [16, 21]. As assumed by these frameworks, we aimed for a clear separation of algorithms for voting, ballot casting, verification, etc. In the symbolic setting, we had to make some simplifications because automated verification tools like Tamarin or ProVerif currently cannot handle the group multiplication operation, namely the equation $\mathtt{g}^{\mathtt{x}} \bullet \mathtt{g}^{\mathtt{y}} = \mathtt{g}^{\mathtt{x}+\mathtt{y}}$. To handle this, we have taken advantage of the fact that, in our case study, $\mathtt{x}$ and $\mathtt{y}$ represent key parts that are contributed by the setup control components. Since one of these components is honest, we can have the corresponding key part, say $\mathtt{x}$, in the clear in our specification, and we can approximate the above equation with the equation $(\mathtt{g}^{\mathtt{x}})^{\mathtt{y}} = \mathtt{g}^{\mathtt{x}*\mathtt{y}}$. Proving the soundness of this abstraction, and more generally handling group operations in tools like Tamarin, remains as future work. Another limitation of Tamarin is that it currently does not allow mixing built-in theories, like Diffie-Hellman exponentiation, with user defined theories. As a consequence, we have not been able to naturally model the zero-knowledge proof verification with equations, and had to use pattern-matching instead. Finally, we have not yet modelled the

[1] `https://github.com/aleaxit/gmpy`

vote confirmation phase, which ensures that a ballot can be counted only for voters that have sucesfully verified and confirmed their votes.

Formal verification also requires specifications of security properties. We aim for verifiability and privacy for the corruption scenarios specified in Table 2. For verifiability, we rely on the definition from [5]. This definition results in a conjunction of four lemmas in our Tamarin files:

- *Individual verifiability:* correct vote is counted for voters verifying their votes;
- *Eligibility:* both verified and tallied votes correspond to registered voters;
- *Result integrity* (RI): the attacker can only cast votes for corrupt voters, or for voters that haven't verified their vote;
- *One ballot per voter:* at most one vote is tallied for each voter.

All these lemmas are verified in Tamarin according to results from Table 2. It is worth noting that RI also admits a stronger version, in which the attacker is not allowed to cast a ballot for honest voters even if they haven't voted or haven't verified their vote. In SwissPost, ballot confirmation requires voters to verify their vote, so we expect the gap between weak and strong RI to disappear for participating voters, or when the setup is honest, once we model the vote confirmation phase. However, when the setup is corrupt and voters choose not to complete the voting procedure, we cannot prevent ballot stuffing, since the adversary may execute the procedure in their place. We plan to analyse the relation among these notions and prove stronger versions of RI in future work.

For privacy, our Tamarin files currently rely on the classic definition from [19], that prescribes the indistinguishability of two experiments where two honest voters swap their votes. Unfortunately, even for the scenario $\mathcal{A}_1$ where most parties are honest, we find false attacks for $\mathsf{SwissPost}_\mathbf{0}$ and $\mathsf{SwissPost}_\mathbf{1}$: by inspecting the trace returned by Tamarin, we can confirm these are not real attacks on the protocols. One reason for the false attacks is the strong notion of equivalence considered by Tamarin, called dependency graph equivalence in [7]. The ProVerif tool also considers a strong notion of equivalence, called diff-equivalence [10], and techniques have been explored in order to bypass its limitations when handling privacy, in particular for e-voting [1,4,6,14,17,20]. We plan to explore these techniques in future work. Another challenge in formally proving privacy is that we need a new definition to handle the case of a corrupted setup. Indeed, in that case malicious parties could arbitrarily control the bulletin board. As shown in [18] in the setting of computational models, defining privacy in that case is challenging, and dedicated modelling techniques are needed to capture the influence and the observational power of the attacker. A recent work proposes such a technique in the symbolic model [6]. However, their approach still requires a certain integrity property to be proved for the bulletin board, which does not hold for SwissPost when the setup component is corrupt, since it could cast arbitrary ballots in the name of honest voters. Symbolic definitions of privacy need to be further refined for that case.

5 Conclusion and future work

We have presented a simplified version of the SwissPost voting protocol to facilitate research towards its machine-assisted verification and we proposed an alternative protocol design to improve security when the setup component is corrupt. In future work, we aim to address the limitations of our current formal verification results, as sketched in Section 4. The feasibility and usability of the alternative protocol design also needs to be further explored. We need to see what zk proof verification delays are acceptable in practice and if our numbers can't be improved. Also, care must be taken with respect to denial of service attacks, since we have dropped one of the (implicit) authentication factors in SwissPost, i.e. the ability of the server and election monitors to verify that the pre-codes in the ballot belong to an allowed list linked to eligible voters (there is an explicit authentication phase before being able to submit any ballot). In our updated version, the implicit eligibility check on the ballot can happen only in the confirmation phase. Perhaps a similar check could be added to the ballot casting phase, based on the start voting key that is already used in SwissPost for explicit voter authentication.

References

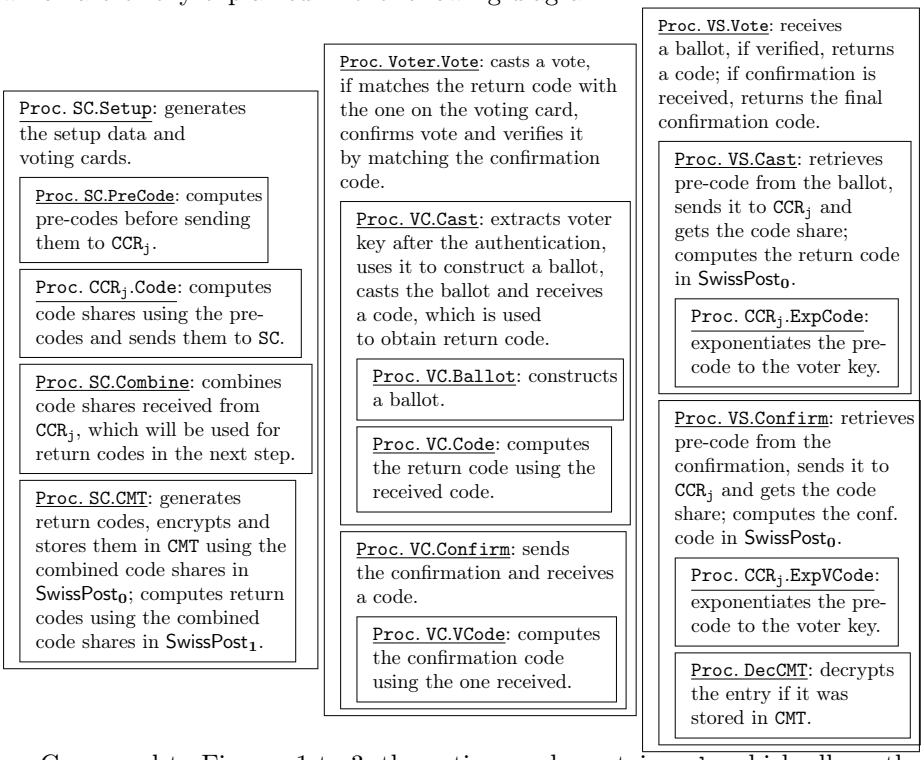
1. Swiss Post voting system. `https://gitlab.com/swisspost-evoting`.
2. Supplementary materials. `https://github.com/sergiubur/swisspost-evoting-security`.
3. Federal council authorises use of online voting in 2023 national council elections. `https://www.bk.admin.ch/bk/en/home/dokumentation/medienmitteilungen.msg-id-97361.html`.
4. David Baelde, Alexandre Debant, and Stéphanie Delaune. Proving unlinkability using ProVerif through desynchronised bi-processes. In *36th IEEE Computer Security Foundations Symposium, CSF'23*, pages 75–90. IEEE, 2023.
5. Sevdenur Baloglu, Sergiu Bursuc, Sjouke Mauw, and Jun Pang. Election verifiability in receipt-free voting protocols. In *36th IEEE Computer Security Foundations Symposium, CSF'23*, pages 59–74. IEEE, 2023.
6. Sevdenur Baloglu, Sergiu Bursuc, Sjouke Mauw, and Jun Pang. Formal verification and solutions for Estonian e-voting. In *19th ACM Asia Conference on Computer and Communications Security, ASIA CCS'24*. ACM, 2024.
7. David Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS'15*, pages 1144–1155, 2015.
8. David A. Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Tamarin: Verification of large-scale, real-world, cryptographic protocols. *IEEE Secur. Priv.*, 20(3):24–32, 2022.
9. Josh Benaloh. Simple verifiable elections. In *Electronic Voting Technology Workshop, EVT'06*. USENIX, 2006.
10. Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Found. Trends Priv. Secur.*, 1(1-2):1–135, 2016.
11. Dan Boneh and Victor Shoup. A Graduate Course in Applied Cryptography, 2023.
12. David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, feb 1981.

13. Vincent Cheval, Véronique Cortier, and Alexandre Debant. Election verifiability with ProVerif. In *36th IEEE Computer Security Foundations Symposium, CSF'23*, pages 43–58. IEEE, 2023.
14. Tom Chothia, Ben Smyth, and Christopher Staite. Automatically checking commitment protocols in ProVerif without false attacks. In *4th Int. Conf. on Principles of Security and Trust, POST'15*, volume 9036 of *LNCS*, pages 137–155. Springer, 2015.
15. Véronique Cortier, Alexandre Debant, and Pierrick Gaudry. A privacy attack on the Swiss Post e-voting system. In *Real World Crypto Symposium, RWC'22*. IACR, 2022.
16. Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, and Bogdan Warinschi. Machine-checked proofs for electronic voting: Privacy and verifiability for Belenios. In *31st IEEE Computer Security Foundations Symposium, CSF'18*, pages 298–312. IEEE, 2018.
17. Véronique Cortier, Alicia Filipiak, and Joseph Lallemand. BeleniosVS: Secrecy and Verifiability Against a Corrupted Voting Device. In *32nd IEEE Computer Security Foundations Symposium, CSF'19*, pages 367–381. IEEE, 2019.
18. Véronique Cortier, Joseph Lallemand, and Bogdan Warinschi. Fifty shades of ballot privacy: Privacy against a malicious board. In *33rd IEEE Computer Security Foundations Symposium, CSF'20*, pages 17–32. IEEE, 2020.
19. Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *J. Comput. Secur.*, 17(4):435–487, 2009.
20. Stéphanie Delaune and Joseph Lallemand. One vote is enough for analysing privacy. In *27th European Symposium on Research in Computer Security, ESORICS'22, Part I*, volume 13554 of *LNCS*, pages 173–194. Springer, 2022.
21. Constantin Catalin Dragan, François Dupressoir, Ehsan Estaji, Kristian Gjøsteen, Thomas Haines, Peter Y. A. Ryan, Peter B. Rønne, and Morten Rotvold Solberg. Machine-checked proofs of privacy against malicious boards for Selene & Co. *J. Comput. Secur.*, 31(5):469–499, 2023.
22. Bryan Alexander Ford. Auditing the Swiss Post e-voting system: An architectural perspective, 2022.
23. Rolf Haenni, Reto E Koenig, Philipp Locher, and Eric Dubuis. Examination of the Swiss Post internet voting system, 2023.
24. Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome. In *2020 IEEE Symposium on Security and Privacy, S&P'20*, pages 644–660. IEEE, 2020.
25. Thomas Haines, Olivier Pereira, and Vanessa Teague. Report on the Swiss Post e-voting system, 2022.
26. Thomas Haines, Olivier Pereira, and Vanessa Teague. Running the race: A Swiss voting story. In *International Joint Conference on Electronic Voting*, pages 53–69. Springer, 2022.
27. Christian Killer and Burkhard Stiller. The Swiss postal voting process and its system and security analysis. In *4th International Joint Conference on Electronic Voting, E-Vote-ID'19*, pages 134–149. Springer, 2019.
28. Cyrill Krähenbühl, Marc Wyss, Robin Burkhard, Joel Wanner, and Adrian Perrig. Swiss Post e-voting scope 4: Network security analysis, 2022.
29. Steve Kremer, Mark Ryan, and Ben Smyth. Election verifiability in electronic voting protocols. In *15th European Symposium on Research in Computer Security, ESORICS'10*, volume 6345 of *LNCS*, pages 389–404. Springer, 2010.
30. C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *ACM Conf. on Computer and Communications Security, CCS'01*, pages 116–125, 2001.

31. Oliver Spycher. Swiss online voting redesigned. E-Vote-ID 2023, 2023.

A More detailed specifications

See Figure 6 and Figure 7 below for more detailed cryptographic specifications of the two systems ($\mathsf{SwissPost_0}$ and $\mathsf{SwissPost_1}$) we consider. In these specifications, we specify the SwissPost voting protocol through three main procedures, representing the actions of the setup component SC, the voter and their voting client VC, and the voting server VS. Each main procedure contains subprocedures, which are briefly explained in the following diagram.



Compared to Figures 1 to 3, the voting cards contain $\mathtt{svk}$, which allows the VC to obtain the voting card identifier $\mathtt{vc}$ and the voting card key store $\mathtt{vc}_{\mathtt{KS}}$ once the voter is authenticated by the VS. The authentication is abstracted by the function: $(\mathtt{vc}, \mathtt{vc}_{\mathtt{KS}}, \mathtt{pk}_{\mathtt{E}}) \leftarrow \mathtt{Auth}(\mathtt{svk}, \mathtt{ea})$, where $\mathtt{ea}$ is the extended authentication factor, a piece of secret auxiliary data, known only to the voter and SC. After successful authentication, the key store $\mathtt{vc}_{\mathtt{KS}}$ is retrieved from VS and decrypted to extract the voter key $\mathtt{k}$.

The data table $\overline{\mathtt{x}}$ represents a tuple of entries $(\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{y})$, where $\mathtt{y}$ denotes either the number of eligible voters $\mathtt{n}_\mathtt{E}$ or the number of candidates $\mathtt{n}_\mathtt{C}$.

In the $\mathsf{SwissPost_0}$ algorithms, the encryption keys for the freshly generated return codes $\mathtt{cc}$ and the vote confirmation code $\mathtt{vcc}$ are derived during the setup phase, using a two-layer process. In the first layer, the setup component

$\mathtt{SC}$ computes the pre-codes: $\mathtt{pCC} = \mathtt{p}^{\mathtt{k}}$ and $\mathtt{CK} = \mathtt{H}_2(\mathtt{bck})^{\mathtt{k}}$, and encrypts them to obtain $\mathtt{c}_{\mathtt{pCC}}$ and $\mathtt{c}_{\mathtt{CK}}$. In the second layer, each $\mathtt{CCR}_{\mathtt{j}}$ (for $\mathtt{j} \in \{1, 2\}$) exponentiates the encrypted pre-codes using voter-specific keys $\mathtt{k}_{\mathtt{j}}$ and $\mathtt{kc}_{\mathtt{j}}$, yielding encrypted code shares $\mathtt{c}_{\mathtt{xpCC,j}}$ and $\mathtt{c}_{\mathtt{xCK,j}}$. These encrypted shares are returned to $\mathtt{SC}$, which decrypts them to recover the code shares, then combines them by multiplication to obtain: $\mathtt{pc} = \mathtt{H}_2(\mathtt{pCC})^{\mathtt{k}_1} \bullet \mathtt{H}_2(\mathtt{pCC})^{\mathtt{k}_2}$ and $\mathtt{lvcc} = \mathtt{H}_2(\mathtt{CK})^{\mathtt{kc}_1} \bullet \mathtt{H}_2(\mathtt{CK})^{\mathtt{kc}_2}$. These intermediate values are then used to derive the final encryption keys: $\mathtt{skcc} = \mathtt{H}_1(\mathtt{pc}, \mathtt{vc})$ and $\mathtt{skvcc} = \mathtt{H}_1(\mathtt{lvcc}, \mathtt{vc})$. These keys are used to encrypt the return codes $\mathtt{cc}$ and $\mathtt{vcc}$, which are stored in the Codes Mapping Table $\mathtt{CMT}$.

In the voting phase, the pre-code $\mathtt{pCC}$ corresponding to the voter's choice is extracted from the ballot and exponentiated to the voter key $\mathtt{k}_{\mathtt{j}}$ by each $\mathtt{CCR}_{\mathtt{j}}$, leading to code share $\mathtt{lcc}_{\mathtt{j}} = \mathtt{H}_2(\mathtt{pCC})^{\mathtt{k}_{\mathtt{j}}}$. The code shares are then combined by the voting server $\mathtt{VS}$ to obtain $\mathtt{pc}$, which enables to derive the encryption key $\mathtt{skcc}$ and decrypt the return code $\mathtt{cc}$ from the $\mathtt{CMT}$. The vote confirmation code $\mathtt{vcc}$ is extracted from the $\mathtt{CMT}$ in the same manner, after the confirmation message $\mathtt{CK}$ is received from the voter's $\mathtt{VC}$.

In addition to the table $\mathtt{CMT}$, the $\mathtt{SC}$ constructs two lists $\mathtt{L}_{\mathtt{pCC}}$ and $\mathtt{L}_{\mathtt{lvcc}}$ that allow protocol parties to check the validity of the vote and the confirmation message during the voting phase. The validity of the votes are checked through the pre-codes $\mathtt{pCC}$ since it is a part of the ballot. Therefore, the list $\mathtt{L}_{\mathtt{pCC}}$ contains hashes of pre-codes, i.e. $\mathtt{H}_1(\mathtt{H}_2(\mathtt{pCC}), \mathtt{vc})$. The validity of the confirmation message, on the other hand, are checked through the intermediate value $\mathtt{lvcc}$. Thus, the list $\mathtt{L}_{\mathtt{lvcc}}$ contains hashes of the partial intermediate values in the form of $\mathtt{H}_1(\mathtt{H}_1(\mathtt{H}_2(\mathtt{CK})^{\mathtt{kc}_1}, \mathtt{vc}), \mathtt{H}_1(\mathtt{H}_2(\mathtt{CK})^{\mathtt{kc}_2}, \mathtt{vc}), \mathtt{vc})$.

In contrast, the $\mathsf{SwissPost}_{\mathtt{1}}$ setup algorithm computes the return codes themselves, not their encryption keys, through a two-layer process. In the first layer, each $\mathtt{CCR}_{\mathtt{j}}$ (for $\mathtt{j} \in \{1, 2\}$) exponentiates the voting choices $\mathtt{p}$ and $\mathtt{H}_2(\mathtt{H}_2(\mathtt{bck})^{\mathtt{k}})$ using the voter-specific keys $\mathtt{k}_{\mathtt{j}}$ and $\mathtt{kc}_{\mathtt{j}}$, respectively. In the second layer, $\mathtt{SC}$ combines the code shares $\mathtt{p}^{\mathtt{k}_{\mathtt{j}}}$ and $\mathtt{H}_2(\mathtt{H}_2(\mathtt{bck})^{\mathtt{k}})^{\mathtt{kc}_{\mathtt{j}}}$ to obtain $\mathtt{pc} = \mathtt{p}^{\mathtt{k}_1} \bullet \mathtt{p}^{\mathtt{k}_2}$ and $\mathtt{lvcc} = \mathtt{H}_2(\mathtt{H}_2(\mathtt{bck})^{\mathtt{k}})^{\mathtt{kc}_1} \bullet \mathtt{H}_2(\mathtt{H}_2(\mathtt{bck})^{\mathtt{k}})^{\mathtt{kc}_2}$, and computes the return codes using the voter key $\mathtt{k}$ as follows: $\mathtt{cc} = \mathtt{H}_3(\mathtt{H}_2(\mathtt{pc})^{\mathtt{k}}, \mathtt{vc})$ and $\mathtt{vcc} = \mathtt{H}_3(\mathtt{lvcc}, \mathtt{vc})$. In the voting phase, the voting client $\mathtt{VC}$ masks the voting choice with a randomness $\mathtt{r}$. Therefore, the term $\mathtt{p}^{\mathtt{r}}$ will be extracted from the ballot and exponentiated by each $\mathtt{CCR}_{\mathtt{j}}$. The voting client will store the randomness $\mathtt{r}$ for the computation of the return code as follows: $\mathtt{cc} = \mathtt{H}_3(\mathtt{H}_2(\mathtt{pc})^{\mathtt{k}}, \mathtt{vc})$.

Fig. 6. Data structures and setup algorithms for the SwissPost voting protocol

Data tables

$\mathtt{ID}$: eligible voters $\{1, \ldots, \mathtt{n_E}\}$ with corresponding extended authentication factors $\{\mathtt{ea}_1, \ldots, \mathtt{ea}_{\mathtt{n_E}}\}$	$\mathtt{Cand}$: eligible candidates $\{1, \ldots, \mathtt{n_C}\}$ with corresponding primes $\{\mathtt{p}_1, \ldots, \mathtt{p}_{\mathtt{n_C}}\}$
$\overline{\mathtt{K}}$: public part of voter keys generated by $\mathtt{SC}$	$\overline{\mathtt{K}}_\mathtt{j}, \overline{\mathtt{Kc}}_\mathtt{j}$: public part of voter keys generated by $\mathtt{CCR}_\mathtt{j}$
$\mathtt{CMT}$: codes mapping table for return codes	$\mathtt{L_{pCC}}, \mathtt{L_{lvcc}}$: return codes allow lists

Setup algorithms ($\mathtt{CMT}, \mathtt{L_{pCC}}, \mathtt{L_{lvcc}}$ are initially empty; $\mathtt{Cand}$ and corresponding primes are public.)

Procedure $\mathtt{SC.Setup}(\mathtt{ID})$

for all $\mathtt{id} \in \mathtt{ID}$: // generate voting vcard
- $\mathtt{vc}, \mathtt{svk}, \mathtt{bck} \leftarrow\!\!\$\ \{0,1\}^\ell, \mathtt{k} \leftarrow\!\!\$\ \mathbb{Z}_\mathtt{q}, \mathtt{K} \leftarrow \mathtt{g}^\mathtt{k}$
- $\mathtt{vc_{KS}} \leftarrow \mathtt{SEnc}(\mathtt{k}, \mathtt{H}_1(\mathtt{svk}, \mathtt{vc}))$ // store encrypted voter key
- // compute pre-codes (encrypted in $\mathsf{SwissPost_0}$)
- $(\mathtt{c_{CK}}, \overline{\mathtt{c_{pCC}}}, \mathtt{L_{pCC}}) \leftarrow \mathtt{SC.PreCode}(\mathtt{vc}, \mathtt{k}, \mathtt{bck})$
- **for all** $\mathtt{j} \in \{1,2\}$: // receive voter keys and code shares
  - $(\mathtt{K}_\mathtt{j}, \mathtt{Kc}_\mathtt{j}, \overline{\mathtt{c_{xpCC,j}}}, \mathtt{c_{xCK,j}}, \Pi_{\mathtt{exp}_\mathtt{j}}) \leftarrow \mathtt{CCR}_\mathtt{j}.\mathtt{Code}(\mathtt{c_{CK}}, \overline{\mathtt{c_{pCC}}})$
- // combine code shares received from $\mathtt{CCR}_\mathtt{j}$
- $(\overline{\mathtt{pc}}, \mathtt{lvcc}, \mathtt{L_{lvcc}}) \leftarrow \mathtt{SC.Combine}(\mathtt{vc}, \overline{\mathtt{c_{xpCC,1}}}, \overline{\mathtt{c_{xpCC,2}}}, \mathtt{c_{xCK,1}}, \mathtt{c_{xCK,2}})$
- // compute return codes (encrypted & stored in $\mathtt{CMT}$ in $\mathsf{SwissPost_0}$)
- $(\overline{\mathtt{cc}}, \mathtt{vcc}, \mathtt{CMT}) \leftarrow \mathtt{SC.CMT}(\mathtt{vc}, \mathtt{k}, \overline{\mathtt{pc}}, \mathtt{lvcc})$
- $\mathtt{vcard} \leftarrow (\mathtt{svk}, (\mathtt{cc}_1, \ldots, \mathtt{cc}_{\mathtt{n_C}}), \mathtt{bck}, \mathtt{vcc})$

// $\mathtt{setup_data}$ contains data tables for voting phase

$\mathtt{setup_data} \leftarrow (\mathtt{pk_E}, \overline{\mathtt{K}}, \overline{\mathtt{K}}_1, \overline{\mathtt{K}}_2, \overline{\mathtt{Kc}}_1, \overline{\mathtt{Kc}}_2, \mathtt{CMT}, \mathtt{L_{pCC}}, \mathtt{L_{lvcc}})$

return $\mathtt{setup_data}$

// $\mathtt{SC}, \mathtt{CCR}_\mathtt{j}$ internal procedures in $\mathsf{SwissPost_1}$

Procedure $\mathtt{SC.PreCode}(\mathtt{vc}, \mathtt{k}, \mathtt{bck})$

$\mathtt{CK} \leftarrow \mathtt{H}_2(\mathtt{bck})^\mathtt{k}$, **return** $(\mathtt{H}_2(\mathtt{CK}), \bot, \bot)$

Procedure $\mathtt{CCR}_\mathtt{j}.\mathtt{Code}(\mathtt{c_{CK}}, \bot)$

// generate voter keys and compute code shares

$\mathtt{k}_\mathtt{j}, \mathtt{kc}_\mathtt{j} \leftarrow\!\!\$\ \mathbb{Z}_\mathtt{q}, \mathtt{K}_\mathtt{j} \leftarrow \mathtt{g}^{\mathtt{k}_\mathtt{j}}, \mathtt{Kc}_\mathtt{j} \leftarrow \mathtt{g}^{\mathtt{kc}_\mathtt{j}}$

for all $\mathtt{i} \in \mathtt{Cand}$: $\mathtt{c_{xpCC,j,i}} \leftarrow \mathtt{p_i}^{\mathtt{k}_\mathtt{j}}$

$\mathtt{c_{xCK,j}} \leftarrow \mathtt{c_{CK}}^{\mathtt{kc}_\mathtt{j}}$, **store** $(\mathtt{k}_\mathtt{j}, \mathtt{kc}_\mathtt{j})$

return $(\mathtt{K}_\mathtt{j}, \mathtt{Kc}_\mathtt{j}, \overline{\mathtt{c_{xpCC,j}}}, \mathtt{c_{xCK,j}}, \Pi_{\mathtt{exp}_\mathtt{j}})$

Procedure $\mathtt{SC.Combine}(\mathtt{vc}, \overline{\mathtt{c_{xpCC,1}}}, \overline{\mathtt{c_{xpCC,2}}}, \mathtt{c_{xCK,1}}, \mathtt{c_{xCK,2}})$

for all $\mathtt{i} \in \mathtt{Cand}$: $\mathtt{pc_i} \leftarrow \mathtt{c_{xpCC,1,i}} \bullet \mathtt{c_{xpCC,2,i}}$

for all $\mathtt{j} \in \{1,2\}$: $\mathtt{lvcc_j} \leftarrow \mathtt{c_{xCK,j}}$

$\mathtt{L_{lvcc}} \leftarrow \mathtt{L_{lvcc}} \cup \{\mathtt{H}_1(\mathtt{H}_1(\mathtt{lvcc}_1, \mathtt{vc}), \mathtt{H}_1(\mathtt{lvcc}_2, \mathtt{vc}), \mathtt{vc})\}$

return $(\overline{\mathtt{pc}}, (\mathtt{lvcc}_1 \bullet \mathtt{lvcc}_2), \mathtt{L_{lvcc}})$

Procedure $\mathtt{SC.CMT}(\mathtt{vc}, \mathtt{k}, \overline{\mathtt{pc}}, \mathtt{lvcc})$

for all $\mathtt{i} \in \mathtt{Cand}$: $\mathtt{cc_i} \leftarrow \mathtt{H}_3(\mathtt{H}_2(\mathtt{pc_i})^\mathtt{k}, \mathtt{vc})$

$\mathtt{vcc} \leftarrow \mathtt{H}_3(\mathtt{lvcc}, \mathtt{vc})$, **return** $(\overline{\mathtt{cc}}, \mathtt{vcc}, \bot)$

// $\mathtt{SC}, \mathtt{CCR}_\mathtt{j}$ internal procedures in $\mathsf{SwissPost_0}$

Procedure $\mathtt{SC.PreCode}(\mathtt{vc}, \mathtt{k}, \mathtt{bck})$

$\mathtt{CK} \leftarrow \mathtt{H}_2(\mathtt{bck})^\mathtt{k}$, $\mathtt{c_{CK}} \leftarrow \mathtt{Enc}(\mathtt{H}_2(\mathtt{CK}), \mathtt{pk_{SC}})$

for all $\mathtt{i} \in \mathtt{Cand}$: // compute encrypted pre-codes
- $\mathtt{pCC_i} \leftarrow \mathtt{p_i}^\mathtt{k}$, $\mathtt{c_{pCC,i}} \leftarrow \mathtt{Enc}(\mathtt{H}_2(\mathtt{pCC_i}), \mathtt{pk_{SC}})$
- $\mathtt{L_{pCC}} \leftarrow \mathtt{L_{pCC}} \cup \{\mathtt{H}_1(\mathtt{H}_2(\mathtt{pCC_i}), \mathtt{vc})\}$

return $(\mathtt{c_{CK}}, \overline{\mathtt{c_{pCC}}}, \mathtt{L_{pCC}})$

Procedure $\mathtt{CCR}_\mathtt{j}.\mathtt{Code}(\mathtt{c_{CK}}, \overline{\mathtt{c_{pCC}}})$

// generate voter keys and compute code shares

$\mathtt{k}_\mathtt{j}, \mathtt{kc}_\mathtt{j} \leftarrow\!\!\$\ \mathbb{Z}_\mathtt{q}, \mathtt{K}_\mathtt{j} \leftarrow \mathtt{g}^{\mathtt{k}_\mathtt{j}}, \mathtt{Kc}_\mathtt{j} \leftarrow \mathtt{g}^{\mathtt{kc}_\mathtt{j}}$

for all $\mathtt{i} \in \mathtt{Cand}$: $\mathtt{c_{xpCC,j,i}} \leftarrow \mathtt{exp}(\mathtt{c_{pCC,i}}, \mathtt{k}_\mathtt{j})$

$\mathtt{c_{xCK,j}} \leftarrow \mathtt{exp}(\mathtt{c_{CK}}, \mathtt{kc}_\mathtt{j})$, **store** $(\mathtt{k}_\mathtt{j}, \mathtt{kc}_\mathtt{j})$

return $(\mathtt{K}_\mathtt{j}, \mathtt{Kc}_\mathtt{j}, \overline{\mathtt{c_{xpCC,j}}}, \mathtt{c_{xCK,j}}, \Pi_{\mathtt{exp}_\mathtt{j}})$

Procedure $\mathtt{SC.Combine}(\mathtt{vc}, \overline{\mathtt{c_{xpCC,1}}}, \overline{\mathtt{c_{xpCC,2}}}, \mathtt{c_{xCK,1}}, \mathtt{c_{xCK,2}})$

for all $\mathtt{i} \in \mathtt{Cand}$: // decrypt and combine code shares
- $\mathtt{pc_i} \leftarrow \mathtt{Dec}(\mathtt{c_{xpCC,1,i}}, \mathtt{sk_{SC}}) \circ \mathtt{Dec}(\mathtt{c_{xpCC,2,i}}, \mathtt{sk_{SC}})$

for all $\mathtt{j} \in \{1,2\}$: $\mathtt{lvcc_j} \leftarrow \mathtt{Dec}(\mathtt{c_{xCK,j}}, \mathtt{sk_{SC}})$

$\mathtt{L_{lvcc}} \leftarrow \mathtt{L_{lvcc}} \cup \{\mathtt{H}_1(\mathtt{H}_1(\mathtt{lvcc}_1, \mathtt{vc}), \mathtt{H}_1(\mathtt{lvcc}_2, \mathtt{vc}), \mathtt{vc})\}$

return $(\overline{\mathtt{pc}}, (\mathtt{lvcc}_1 \bullet \mathtt{lvcc}_2), \mathtt{L_{lvcc}})$

Procedure $\mathtt{SC.CMT}(\mathtt{vc}, \mathtt{k}, \overline{\mathtt{pc}}, \mathtt{lvcc})$

for all $\mathtt{i} \in \mathtt{Cand}$: $\mathtt{cc_i} \leftarrow\!\!\$\ \{0,1\}^\ell$
- $\mathtt{skcc_i} \leftarrow \mathtt{H}_1(\mathtt{pc_i}, \mathtt{vc})$
- $\mathtt{CMT}[\mathtt{H}_1(\mathtt{skcc_i})] \leftarrow \mathtt{SEnc}(\mathtt{cc_i}, \mathtt{skcc_i})$

$\mathtt{vcc} \leftarrow\!\!\$\ \{0,1\}^\ell$, $\mathtt{skvcc} \leftarrow \mathtt{H}_1(\mathtt{lvcc}, \mathtt{vc})$

$\mathtt{CMT}[\mathtt{H}_1(\mathtt{skvcc})] \leftarrow \mathtt{SEnc}(\mathtt{vcc}, \mathtt{skvcc})$

return $(\overline{\mathtt{cc}}, \mathtt{vcc}, \mathtt{CMT})$

Fig. 7. Voting algorithms for the SwissPost voting protocol

Procedure $\texttt{Voter.Vote}(\texttt{id}, \texttt{ea}, \texttt{vcard})$

// $\texttt{vcard} = (\texttt{svk}, (\texttt{cc}_1, \ldots, \texttt{cc}_{n_C}), \texttt{bck}, \texttt{vcc})$ received from SC
$\texttt{cc} \leftarrow \texttt{VC.Cast}(\texttt{svk}, \texttt{ea}, \texttt{v})$ // cast vote and receive return code
if $\texttt{cc}_\texttt{v} \neq \texttt{cc}$ **then fail** // if the code matches then
$\texttt{vcc}' \leftarrow \texttt{VC.Confirm}(\texttt{bck})$ // confirm ballot and receive final code
if $\texttt{vcc} = \texttt{vcc}'$ **then** $\texttt{Verified}(\texttt{id}, \texttt{v})$ // successfully verified vote

Procedure $\texttt{VC.Cast}(\texttt{svk}, \texttt{ea}, \texttt{v})$

$(\texttt{vc}, \texttt{vc}_{KS}, \texttt{pk}_E) \leftarrow \texttt{Auth}(\texttt{svk}, \texttt{ea})$ // authentication
$\texttt{k} \leftarrow \texttt{SDec}(\texttt{vc}_{KS}, \texttt{H}_1(\texttt{svk}, \texttt{vc}))$ // extract voter key
$(\texttt{b}, \texttt{r}) \leftarrow \texttt{VC.Ballot}(\texttt{v}, \texttt{k}, \texttt{pk}_E)$ // construct ballot for v
$\texttt{Out}(\texttt{vc}, \texttt{b});\ \texttt{In}(\texttt{code});$ // cast ballot and receive code
$\texttt{cc} \leftarrow \texttt{VC.VCode}(\texttt{vc}, \texttt{v}, \texttt{k}, \texttt{r}, \texttt{code})$ // compute return code
store $(\texttt{k}, \texttt{r})$, **return** cc

Procedure $\texttt{VC.Confirm}(\texttt{bck})$

$\texttt{CK} \leftarrow \texttt{H}_2(\texttt{bck})^{\texttt{k}},\ \texttt{rCK} \leftarrow \texttt{H}_2(\texttt{bck})^{\texttt{r}}$ // $\texttt{rCK} \leftarrow \perp$ in $\mathsf{SwissPost}_0$
$\texttt{Out}(\texttt{vc}, (\texttt{CK}, \texttt{rCK}));\ \texttt{In}(\texttt{vcode});$ // send confirmation, receive code
$\texttt{vcc} \leftarrow \texttt{VC.VCode}(\texttt{vc}, \texttt{k}, \texttt{r}, \texttt{rCK}, \texttt{vcode})$ // compute final code
return $\texttt{vcc}'$

Procedure $\texttt{VC.Ballot}(\texttt{v}, \texttt{k}, \texttt{pk}_E)$ // $\mathsf{SwissPost}_0$

$\texttt{e} \leftarrow \texttt{Enc}(\texttt{p}_\texttt{v}, \texttt{pk}_E),\ \texttt{pCC} \leftarrow \texttt{p}_\texttt{v}^{\texttt{k}},\ (\texttt{ex}, \Pi_{exp}) \leftarrow \texttt{exp}(\texttt{e}, \texttt{k})$
$\Pi_{eq} \leftarrow \texttt{ΠEQ}(\texttt{ex}, \texttt{pCC}),\ \Pi = (\texttt{ex}, \Pi_{exp}, \Pi_{eq}),\ \texttt{b} = (\texttt{e}, \texttt{pCC}, \Pi)$
return $(\texttt{b}, \perp)$

Procedure $\texttt{VC.Code}(\texttt{vc}, \texttt{v}, \texttt{k}, \perp, \texttt{cc})$: **return** cc

Procedure $\texttt{VC.VCode}(\texttt{vc}, \texttt{k}, \perp, \perp, \texttt{vcc}')$: **return** $\texttt{vcc}'$

Procedure $\texttt{VC.Ballot}(\texttt{v}, \texttt{k}, \texttt{pk}_E)$ // $\mathsf{SwissPost}_1$

$\texttt{e} \leftarrow \texttt{Enc}(\texttt{p}_\texttt{v}, \texttt{pk}_E),\ \texttt{r} \leftarrow\$\ \mathbb{Z}_q,\ \texttt{rCC} \leftarrow \texttt{p}_\texttt{v}^{\texttt{r}},\ (\texttt{ex}, \Pi_{exp}) \leftarrow \texttt{exp}(\texttt{e}, \texttt{r})$
$\Pi_{eq} = \texttt{ΠEQ}(\texttt{ex}, \texttt{rCC}),\ \Pi'_{exp} \leftarrow \texttt{ΠEXP}(\texttt{rCC}, \texttt{g}^{\texttt{r}}, (\texttt{p}_1, \ldots, \texttt{p}_{n_C}))$
$\Pi = (\texttt{ex}, \Pi_{exp}, \Pi_{eq}, \texttt{g}^{\texttt{r}}, \Pi'_{exp}),\ \texttt{b} = (\texttt{e}, \texttt{rCC}, \Pi),$ **return** $(\texttt{b}, \texttt{r})$

Procedure $\texttt{VC.Code}(\texttt{vc}, \texttt{v}, \texttt{k}, \texttt{r}, (\texttt{lcc}_1, \texttt{lcc}_2, \Pi_{exp_1}, \Pi_{exp_2}))$

if $\forall \texttt{j} \in \{1, 2\}.\ \texttt{Verify}(\Pi_{exp_j}, \texttt{p}_\texttt{v}^{\texttt{r}}, \texttt{K}_\texttt{j}, \texttt{lcc}_\texttt{j})$ **then**
  $\texttt{pc} \leftarrow (\texttt{lcc}_1 \bullet \texttt{lcc}_2)^{\texttt{r}^{-1}},\ \texttt{cc} \leftarrow \texttt{H}_3(\texttt{H}_2(\texttt{pc})^{\texttt{k}}, \texttt{vc}),$ **return** cc

Procedure $\texttt{VC.VCode}(\texttt{vc}, \texttt{k}, \texttt{r}, \texttt{rCK}, (\texttt{rlvcc}_1, \texttt{rlvcc}_2, \Pi^{\texttt{r}}_{exp_1}, \Pi^{\texttt{r}}_{exp_2}))$

if $\forall \texttt{j} \in \{1, 2\}.\ \texttt{Verify}(\Pi^{\texttt{r}}_{exp_j}, \texttt{rCK}, \texttt{Kc}_\texttt{j}, \texttt{rlVVC}_\texttt{j})$ **then**
  $\texttt{vcc}' \leftarrow \texttt{H}_3(\texttt{H}_2((\texttt{rlvcc}_1 \bullet \texttt{rlvcc}_2)^{\texttt{r}^{-1}})^{\texttt{k}}, \texttt{vc}),$ **return** $\texttt{vcc}'$

Procedure $\texttt{VS.Vote}(\texttt{setup_data})$

// $\texttt{setup_data} = (\texttt{pk}_E, \overline{\texttt{K}}, \overline{\texttt{K}}_1, \overline{\texttt{K}}_2, \overline{\texttt{Kc}}_1, \overline{\texttt{Kc}}_2, \texttt{CMT}, \texttt{L}_{pCC}, \texttt{L}_{1vcc})$
$\texttt{In}(\texttt{vc}, \texttt{b});$ **if** $\texttt{Verify}(\texttt{b})$ **then** // receive ballot and verify
$\texttt{code} \leftarrow \texttt{VS.Cast}(\texttt{vc}, \texttt{b})$ // process ballot and compute code
$\texttt{Out}(\texttt{code});\ \texttt{In}(\texttt{vc}, (\texttt{CK}, \texttt{rCK}));$ // send return code
$\texttt{vcode} \leftarrow \texttt{VS.Confirm}(\texttt{vc}, (\texttt{CK}, \texttt{rCK}))$ // process confirmation
$\texttt{Out}(\texttt{vcode});\ \texttt{Tally}(\texttt{b})$ // send final code and tally ballot

Procedure $\texttt{VS.Cast}(\texttt{vc}, (\texttt{e}, \texttt{xCC}, \Pi))$

$(\texttt{lcc}_1, \Pi_{exp_1}) \leftarrow \texttt{CCR}_1\texttt{.ExpCode}(\texttt{xCC})$ // code share from $\texttt{CCR}_1$
$(\texttt{lcc}_2, \Pi_{exp_2}) \leftarrow \texttt{CCR}_2\texttt{.ExpCode}(\texttt{xCC})$ // code share from $\texttt{CCR}_2$
if $\texttt{lcc}_1 = \perp \vee \texttt{lcc}_2 = \perp$ **then return** $\perp$
return $(\texttt{lcc}_1, \texttt{lcc}_2, \Pi_{exp_1}, \Pi_{exp_2})$ // in $\mathsf{SwissPost}_1$

$\texttt{pc} \leftarrow \texttt{lcc}_1 \bullet \texttt{lcc}_2,\ \texttt{skcc} \leftarrow \texttt{H}_1(\texttt{pc}, \texttt{vc})$
$\texttt{cc} \leftarrow \texttt{DecCMT}(\texttt{skcc}),$ **return** cc // in $\mathsf{SwissPost}_0$

Procedure $\texttt{CCR}_\texttt{j}\texttt{.ExpCode}(\texttt{xCC})$ // $\texttt{k}_\texttt{j}$ stored at setup

if $\texttt{H}_1(\texttt{H}_2(\texttt{xCC}), \texttt{vc}) \in \texttt{L}_{pCC}$ **then** $\texttt{lcc}_\texttt{j} \leftarrow \texttt{H}_2(\texttt{xCC})^{\texttt{k}_\texttt{j}}$
else $\texttt{lcc}_\texttt{j}, \Pi_{exp_j} \leftarrow \perp,$ **return** $(\texttt{lcc}_\texttt{j}, \Pi_{exp_j})$

Procedure $\texttt{VS.Confirm}(\texttt{vc}, (\texttt{CK}, \perp))$

$(\texttt{lvcc}_1, \Pi_{exp_1}) \leftarrow \texttt{CCR}_1\texttt{.ExpVCode}(\texttt{CK})$ // share from $\texttt{CCR}_1$
$(\texttt{lvcc}_2, \Pi_{exp_2}) \leftarrow \texttt{CCR}_2\texttt{.ExpVCode}(\texttt{CK})$ // share from $\texttt{CCR}_2$
if $\texttt{H}_1(\texttt{H}_1(\texttt{lvcc}_1, \texttt{vc}), \texttt{H}_1(\texttt{lvcc}_2, \texttt{vc}), \texttt{vc}) \in \texttt{L}_{1vcc}$ **then**
  $\texttt{skvcc} \leftarrow \texttt{H}_1(\texttt{lvcc}_1 \bullet \texttt{lvcc}_2, \texttt{vc})$ // derive key
  $\texttt{vcc}' \leftarrow \texttt{DecCMT}(\texttt{skvcc}),$ **return** $\texttt{vcc}'$ **else return** $\perp$

Procedure $\texttt{CCR}_\texttt{j}\texttt{.ExpVCode}(\texttt{CK})$ // $\texttt{kc}_\texttt{j}$ stored at setup

$\texttt{lvcc}_\texttt{j} \leftarrow \texttt{H}_2(\texttt{CK})^{\texttt{kc}_\texttt{j}},$ **return** $(\texttt{lvcc}_\texttt{j}, \Pi_{exp_j})$

Procedure $\texttt{DecCMT}(\texttt{k})$

if $\texttt{H}_1(\texttt{k}) \in \texttt{CMT}$ **then return** $\texttt{SDec}(\texttt{CMT}[\texttt{H}_1(\texttt{k})], \texttt{k})$

Procedure $\texttt{CCR}_\texttt{j}\texttt{.ExpCode}(\texttt{xCC})$ // $\texttt{k}_\texttt{j}$ stored at setup

$\texttt{lcc}_\texttt{j} \leftarrow \texttt{xCC}^{\texttt{k}_\texttt{j}},$ **return** $(\texttt{lcc}_\texttt{j}, \Pi_{exp_j})$

Procedure $\texttt{VS.Confirm}(\texttt{vc}, (\texttt{CK}, \texttt{rCK}))$

$(\texttt{lvcc}_1, \texttt{rlvcc}_1, \Pi_{exp_1}, \Pi^{\texttt{r}}_{exp_1}) \leftarrow \texttt{CCR}_1\texttt{.ExpVCode}(\texttt{CK}, \texttt{rCK})$
$(\texttt{lvcc}_2, \texttt{rlvcc}_2, \Pi_{exp_2}, \Pi^{\texttt{r}}_{exp_2}) \leftarrow \texttt{CCR}_2\texttt{.ExpVCode}(\texttt{CK}, \texttt{rCK})$
if $\texttt{H}_1(\texttt{H}_1(\texttt{lvcc}_1, \texttt{vc}), \texttt{H}_1(\texttt{lvcc}_2, \texttt{vc}), \texttt{vc}) \in \texttt{L}_{1vcc}$ **then**
  return $(\texttt{rlvcc}_1, \texttt{rlvcc}_2, \Pi^{\texttt{r}}_{exp_1}, \Pi^{\texttt{r}}_{exp_2})$ **else return** $\perp$

Procedure $\texttt{CCR}_\texttt{j}\texttt{.ExpVCode}(\texttt{CK}, \texttt{rCK})$ // $\texttt{kc}_\texttt{j}$ stored

$\texttt{lvcc}_\texttt{j} \leftarrow \texttt{H}_2(\texttt{CK})^{\texttt{kc}_\texttt{j}},\ \texttt{rlVCC} \leftarrow \texttt{rCK}^{\texttt{kc}_\texttt{j}}$
return $(\texttt{lvcc}_\texttt{j}, \texttt{rlVCC}_\texttt{j}, \Pi_{exp_j}, \Pi^{\texttt{r}}_{exp_j})$

B Zero-knowledge proof

All references below refer to [11]. For simplicity our proof assumes that the voter votes for a single candidate. Using the standard result for combining zero-knowledge proofs with the AND-proof construction (Theorem 19.18 page 787), we deduce that the protocol is secure with respect to several votes as well. Next we show the proof steps necessary for proving the zero-knowledge protocol and its non-interactive version in Figure 8 secure:

- Combine the Chaum-Pedersen zero-knowledge protocol (Section 19.5.2 page 777) and the OR-proof construction (Section 19.7.2 page 788) to define a Sigma protocol.
- Prove that it provides completeness, special soundness (Definition 19.4 page 773), unpredictable commitments (Definition 19.7 page 785) and is Special HVZK (Definition 19.5 page 774).
  Completeness: an interaction between an honest verifier and an honest prover is always successful.
  Special Soundness: Given the statement (v, w) and two accepting transcripts with distinct challenges $((v', w'), C', (z', c'))$ and $((v', w'), C'', (z'', c''))$ the witness x can be computed as follows. Let j such that $c'_j \neq c''_j$, which clearly exists as $C' \neq C''$. Then $c'_j - c''_j$ is not zero modulo q and:

$$\begin{aligned} & g^{z'_j} = v'_j \cdot v^{c'_j} \wedge g^{z''_j} = v'_j \cdot v^{c''_j} \\ \implies & g^{z'_j - z''_j} = v^{c'_j - c''_j} \\ \implies & g^{(z'_j - z''_j)\cdot(c'_j - c''_j)^{-1}} = v = g^x \\ \implies & x = (z'_j - z''_j) \cdot (c'_j - c''_j)^{-1} \end{aligned}$$

  From $s_j^{z'_j} = w'_j \cdot w^{c'_j}$ and $s_j^{z''_j} = w'_j \cdot w^{c''_j}$ we obtain:

$$w = s_j^{(z'_j - z''_j)\cdot(c'_j - c''_j)^{-1}} = s_j^x$$

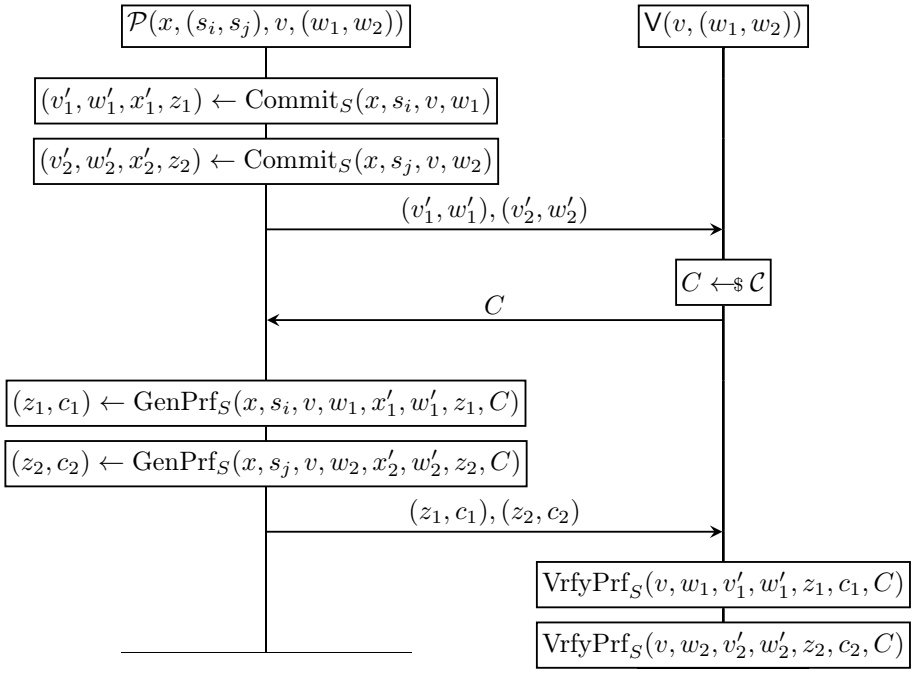
  as was needed.
  Comment: From this proof we can deduce that generating two accepting transcripts with the same commitment in which the vectors c' and c'' differ in more than one element should be a hard problem.
  Unpredictable commitments: commitments are unpredictable because they contain the values $v'_j = g^{x'_j}$ which are unpredictable given that x'_j is selected at random modulo q.
  Special Honest Verifier Zero Knowledge: Given a challenge C and a statement (v, w), an accepting transcript can be generated without knowledge of x as follows:
  - Generate c_i at random such that $\bigoplus_{i=1}^{m} c_i = C$
  - Generate z_j at random from $\mathbb{Z}_q$
  - Compute $w'_j = s_j^{z_j} \cdot w^{-c_j}$

- Compute $v'_j = g^{z_j} \cdot v^{-c_j}$

Notice this is a valid transcript as all equations checked in `VrfyPrf` are valid by construction, and is indistinguishable from an honestly generated transcript.

– Apply Theorem 20.1 page 824 to obtain soundness, then Theorem 20.2 page 834 to obtain non-interactive soundness and finally Theorem 20.3 page 836 to prove security of the non-interactive version.

Fig. 8. Zero-knowledge proof protocol, interactive and non-interactive versions.



$\mathrm{Commit}_S(x,s_i,v,w)$

assert $v = g^x \wedge w = s_i^x$;
for all $j \in \{1,\ldots,m\}$:
  $c_j \leftarrow\!\!\$\ \mathcal{C}; x'_j \leftarrow\!\!\$\ \mathbb{Z}_q$
  $v'_j \leftarrow g^{x'_j}$
  $z_j \leftarrow (x'_j + x \cdot c_j) \bmod q$
  $w'_j \leftarrow s_j^{z_j} \cdot w^{-c_j}$
endfor
$w'_i \leftarrow s_i^{x'_i}$
return (v', w', x', z, c)

$\mathrm{GenPrf}_S(x,s_i,v,w,x',w',z,c,C)$

$c_i \leftarrow C$
$w'_i \leftarrow s_i^{x'_i}$
for all $j \in \{1,\ldots,m\} \setminus \{i\}$:
  $c_i \leftarrow c_i \oplus c_j$
endfor
$z_i \leftarrow (x'_i + x \cdot c_i) \bmod q$
return (z, c)

$\mathrm{VrfyPrf}_S(v,w,v',w',z,c,C)$

for all $j \in \{1,\ldots,m\}$:
  if $g^{z_j} \neq v'_j \cdot v^{c_j} \vee s_j^{z_j} \neq w'_j \cdot w^{c_j}$:
    return `false`
  endif
endfor
if $\bigoplus_{j=1}^{m} c_j \neq C$:
  return `false`
endif
return `true`

$\Pi\mathtt{SET}(x,(s_i,s_j),v,(w_i,w_j),\mathtt{S})$

$(v'_i,w'_i,x'_i,z_i,c_i) \leftarrow \mathrm{Commit}_S(x,s_i,v,w_i)$
$(v'_j,w'_j,x'_j,z_j,c_j) \leftarrow \mathrm{Commit}_S(x,s_j,v,w_j)$
$C \leftarrow \mathtt{H}(v,(w_i,w_j),(v'_i,w'_i),(v'_j,w'_j))$
$(z_i,c_i) \leftarrow \mathrm{GenPrf}_S(x,s_i,v,w_i,x'_i,w'_i,z_i,c_i,C)$
$(z_j,c_j) \leftarrow \mathrm{GenPrf}_S(x,s_i,v,w_j,x'_j,w'_j,z_j,c_j,C)$
return $((v'_i,w'_i,z_i,c_i),(v'_j,w'_j,z_j,c_j))$

$\mathtt{VerifySET}(\mathtt{v},(\mathtt{w_1},\mathtt{w_2}),(\mathtt{v'_1},\mathtt{w'_1},\mathtt{z_1},\mathtt{c_1}),(\mathtt{v'_2},\mathtt{w'_2},\mathtt{z_2},\mathtt{c_2}),\mathtt{S})$

$C \leftarrow \mathtt{H}(v,(w_1,w_2),(v'_1,w'_1),(v'_2,w'_2))$
$r1 \leftarrow \mathrm{VrfyPrf}_S(v,w_1,v'_1,w'_1,z_1,c_1,C)$
$r2 \leftarrow \mathrm{VrfyPrf}_S(v,w_2,v'_2,w'_2,z_2,c_2,C)$
return $r1 \wedge r2$