A formal analysis of the Neuchâtel e-voting protocol

Véronique Cortier
Université de Lorraine, CNRS, Inria, LORIA, France
Email: cortier@loria.fr

David Galindo
University of Birmingham Birmingham, UK
Email: D.Galindo@cs.bham.ac.uk

Mathieu Turuani
Université de Lorraine, CNRS, Inria, LORIA, France
Email: turuani@loria.fr

***Abstract*—Remote electronic voting is used in several countries for legally binding elections. Unlike academic voting protocols, these systems are not always documented and their security is rarely analysed rigorously.**

In this paper, we study a voting system that has been used for electing political representatives and in citizen-driven referenda in the Swiss canton of Neuchâtel. We design a detailed model of the protocol in ProVerif for both privacy and verifiability properties. Our analysis mostly confirms the security of the underlying protocol: we show that the Neuchâtel protocol guarantees ballot privacy, even against a corrupted server; it also ensures cast-as-intended and recorded-as-cast verifiability, even if the voter's device is compromised. To our knowledge, this is the first time a full-fledged automatic symbolic analysis of an e-voting system used for politically-binding elections has been realized.

1. Introduction

Remote electronic voting (or Internet voting) allows voters to vote from their home or when they are travelling. It is also seen as a means to get feedback from citizens on a more regular basis. Therefore, Internet voting has been used in legally binding elections in several countries, including e.g. Estonia [22], Australia [10], France [32] or Switzerland [20]. Of course, designing and implementing a secure electronic voting system is a difficult task and many attacks or weaknesses have been discovered on deployed systems (see e.g. [16], [29], [35], [37]). Voting protocols should offer some basic security guarantees, such as ballot privacy (no one knows how I voted) as well as verifiability (voters can check the voting process) with as few trust assumptions as possible.

On an academic level, several Internet voting protocols have been developed and some of them offer a prototype implementation or even a voting platform. This is for example the case of Helios [1], Civitas [12], Belenios [13], or Select [28]. These protocols are well documented and typically come with a proof of their security, at least w.r.t. privacy, in a symbolic or a cryptographic model (see e.g. [4], [13], [24], [28]). On the other hand, industrial scale protocols are being deployed and now also aim at offering some verifiability properties. For instance some systems offer voters to check their vote. Some examples are: the Estonian [23] protocol, where voters can check their vote during a short period of time; the so-called Norwegian protocol [21], that does so using return codes; or the New South Wales iVote protocol [10]. The systems deployed in Norway and New South Wales have been (co)-developed by Scytl, a company specialized in e-voting solutions. Each of those systems has been adapted to suit the requirements and needs of each jurisdiction.

Contributions. In this paper, we analyse the next generation of the Norwegian protocol [20], [21], that has been deployed and is being used in the Swiss cantons of Neuchâtel and Fribourg [33]. Our main contribution is a thorough analysis of the Neuchâtel protocol (as specified in [20], [34], [36]) in ProVerif. The tool ProVerif [6], [7] is a state-of-the-art tool for the formal analysis of security protocols. We are able to prove *ballot privacy* (modelled as an equivalence property) as well as cast-as-intended and recorded-as-cast *verifiability* (modelled as a reachability property).

It should be noted that the Neuchâtel protocol, as well as most industrial-scale protocols, is not fully verifiable according to the academic tradition, since the content of the ballot box is not publicly disclosed. Instead, the protocol aims at providing *cast-as-intended* verifiability: if the voting server registers a ballot in the name of a voter then the ballot contains the vote *intended* by the voter, even if the voting device is corrupted and tries to cast a vote for another voting option. Cast-as-intended is achieved through return codes: after casting a vote, a voter receives a code and checks (on her voting card) that it matches the code displayed next to her intended choice. The Neuchâtel protocol additionally guarantees *recorded-as-cast* verifiability: if a voter completes the voting process then she is guaranteed that her ballot, as built by her voting device, has reached the voting server. These two verifiability properties hold under the assumption that the voting server is not compromised. Note that cast-as-intended verifiability is not offered by academic systems such as Civitas [12] and Belenios [13]. Often when it is offered, quite a burden is placed on the voter: this is the case for Helios (cast-or-audit mechanism [1], [26]), where the voter needs to use two voting devices that are not simultaneously compromised. Namely, the device used for auditing the vote needs to be different from the device

used to cast the vote. On the other hand, Civitas, Helios, and Belenios offer universal verifiability: anyone can check that the result corresponds to the ballots on the bulletin board, which is not the case for the Neuchâtel protocol, where ballots are not public.

In order to prove our security claims, we present a detailed model of the protocol, that includes for example the authentication phase after which the voter retrieves her voting credentials (i.e. by opening a password-protected key-store). Whereas such an initialization phase must be present in any real-world e-voting system, it is omitted in virtually every security analysis. Even worse, such authentication mechanism is typically not specified by the protocols in the academic literature. This is not necessarily surprising, and it might stem from the fact that academic research artifacts often do not get used in practice and thus do not need to be described as a fully detailed system. Additionally, we capture in our model elections where voters can select k options among n voting options (while systems in the literature are often analysed in the case of elections where voters selects 1 option among n or even 1 among 2).

We chose to perform an automated security analysis using ProVerif (instead of a manual proof) precisely to be able to model as many details as possible. We also believe that our model could serve as a basis for further studies of other voting protocols, as it is easier to adapt a symbolic model than a manual proof.

One difficulty we had to face resides in the fact that ProVerif does not handle well protocols with global states. This is of particularly critical importance in the case of the Neuchâtel protocol, since it becomes insecure (w.r.t. cast-as-intended) as soon as revoting is allowed. It is therefore crucial to model the fact that each voter votes "at most once", and to do it in such a way that ProVerif can still handle the resulting model and provide a proof. Again, we believe that the techniques developed here to circumvent this issue are likely to be found useful elsewhere.

Our analysis mostly confirms the security of the protocol: we prove in ProVerif ballot privacy against a dishonest voting server as well as cast-as-intended and recorded-as-cast verifiability against a dishonest voting device, for an *unbounded number* of voting options and of voters. However, while modelling the protocol, we also discovered small variations thereof, which could realistically come up when implementing the protocol in a real scenario, that would render the protocol insecure in practice.

Related Work. The study most closely related to this paper is the analysis of the Norwegian protocol [17], that solely studies ballot privacy. The Norwegian protocol is an ancestor of the Neuchâtel protocol. The goal of the analysis in [17] was to provide a modeling as precise as possible of the protocol's underlying primitives (with associativity and commutativity properties) and prove ballot privacy by hand, which was accomplished by developing some general lemmas regarding equivalence. A brief analysis was also performed in ProVerif but in a quite abstract model. Reusing the previous model was deemed to be not possible, as the Neuchâtel protocol has evolved quite significantly and the resulting equational theories for the atomic primitives are different.

Earlier research proposed the first symbolic models of electronic voting protocols. This includes a model of JCJ [3] and Helios [16], [18]. These models solely study privacy properties and consider a simpler scenario where voters select one candidate among a finite number of options. These protocols also allow revoting, which significantly simplifies the analysis in ProVerif (cf. the discussion above on dealing with a global state).

Computational proofs of privacy and/or some verifiability properties have been provided for Helios [4], [15], Civitas [24], Select [28], Belenios [13] and [25] for example. Cryptographic models are more accurate w.r.t. the underlying primitives and consider a more powerful attacker, which may for example exploit algebraic properties of the primitives. Most of these proofs are done by hand, with the exception of [15], that provides a mechanized proof of ballot privacy for Helios-like protocols. Given their complexity, these proofs focus on the core of the protocol, abstracting away many details, including detailed analysis of the high-level interactions between the different parties of the protocol (e.g. computational proofs would typically assume a secure channel between the voter and the ballot box, without being explicit nor studying how this is done). The intrinsic complexity of computational proofs of elaborated protocols, as those used in e-voting systems, make those proofs even more error-prone and thus harder to verify.

2. Overview of the Neuchâtel's system

The Neuchâtel voting protocol involves four main participants: the Voter (V), that casts a vote with the help of a Voting Device (VD); the Voting Server (S), that interacts with the voter's device to store the voter's ballot in a database, and next it computes return codes that need to be approved by the voter V; finally the Tallying Authority (T), that computes the result of the election by tallying the ballots database built by the server.

For the sake of clarity, we summarize the Neuchâtel protocol in the case of a selection of $k = 2$ choices among n voting options $1, \ldots, n$. Our analysis accounts however for several values of k other than $k = 2$. A synthetic view of the protocol is provided in Figure 1.

Tallying Authority (T). Creates an ElGamal asymmetric encryption key pair (pk_b, sk_b). The public encryption key pk_b is communicated to the Voting Server. Once the election is closed, the authority T computes and outputs the result of the election as follows. From each ballot to be counted, it extracts the ElGamal encryption of the vote. Next, T applies a mixnet to the resulting ciphertext list, decrypts every entry with the corresponding election decryption key sk_b and computes a zero-knowledge proof of correct decryption. The result is simply the multiset of the decrypted votes.

Voter (V). The voter V associated to id enters into her voting device VD her password pwd_{id} and her preferred

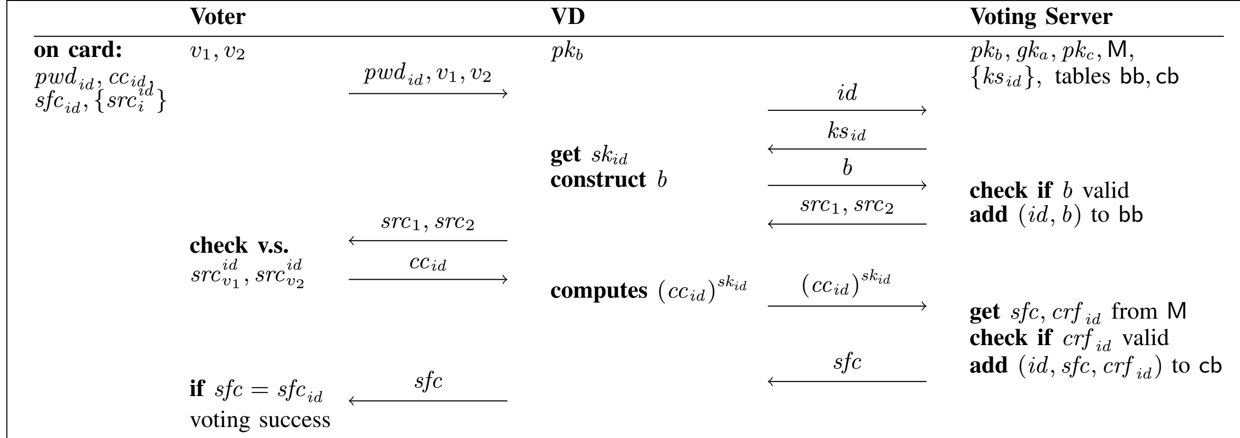

Figure 1. Overview of Neuchâtel Voting Protocol

voting choices v_1, v_2. At some point, VD displays short return codes src_1, src_2 to the voter. The voter retrieves the codes $src_{v_1}^{id}$ and $src_{v_2}^{id}$ linked to v_1, v_2 from her voting card. If the displayed and the retrieved code sets coincide, the voter confirms her votes by entering the confirmation code cc_{id}. Finally, VD displays a short finalization code sfc, which should be equal to the code sfc_{id} on the voter's card. In this case, V is ensured that her ballot contains her intended voting options and that it has been accepted by the voting server S.

Voting Device (VD). The device VD requests a keystore ks_{id} associated to id, from which it retrieves the voters' key pair (pk_{id}, sk_{id}) using the password pwd_{id}. Next, the device computes a ballot $b = (\mathsf{ctxt}, (v_1)^{sk_{id}}, (v_2)^{sk_{id}}, pk_{id}, \mathsf{P})$ that is received by the Voting Server. The ballot consists of several parts:

- $\mathsf{ctxt} = aenc\big(pk_b, \phi(v_1, v_2), r\big)$ is an encryption of voting choices v_1, v_2 with key pk_b and random r using an ElGamal encryption algorithm $aenc$, and a bijective compacting function ϕ that maps any list of integers to a single integer (this relies on the uniqueness of prime factors decomposition).
- so-called partial return codes $(v_1)^{sk_{id}}, (v_2)^{sk_{id}}$, built as exponentiations, allow the Voting Server to compute the short return codes $src_{v_1}^{id}, src_{v_2}^{id}$ that appear in the voter's voting card;
- the remaining components serve to guarantee consistency between ciphertext ctxt and the partial return codes $(v_1)^{sk_{id}}, (v_2)^{sk_{id}}$, by using a zero-knowledge proof P.

If the ballot is accepted by the server S, the voting device VD receives short return codes src_1, src_2 from the server and displays them to the voter V. Next, on input the finalization code $ccid$ entered by the voter, VD computes and sends $(cc_{id})^{sk_{id}}$ to the server. Finally, the server sends back a short finalization code sfc, to be displayed to the voter.

Voting Server (S). It interacts with a voter V with identifier id through her voting device VD as follows. Firstly, the server receives the voter's identifier id from VD and replies with the corresponding keystore ks_{id}. [1] Next it receives from VD a ballot $b = (\mathsf{ctxt}, (v_1)^{sk_{id}}, (v_2)^{sk_{id}}, pk_{id}, \mathsf{P})$, and checks that it is a valid ballot. In particular, it verifies the zero-knowledge proof P and checks that the voter V did not vote already. If valid, the ballot is stored in a database bb. From the partial return codes $(v_1)^{sk_{id}}$ and $(v_2)^{sk_{id}}$, the server can compute the values rc_1 and rc_2 through a keyed pseudo-random function, and retrieve their corresponding short return codes src_1, src_1 by looking into a table M. These short codes are sent to VD. Next, if the voter is satisfied with the return codes, S receives the value $(cc_{id})^{sk_{id}}$, from which it can compute fc (same way as rc) and retrieve its short code sfc by looking again into M. The server S also retrieves a validity proof crf_{id} from M, that tells whether the retrieved code sfc is valid or not. If all the tests pass, S adds the confirmation values sfc and crf_{id} to the database (cb here), to keep track that the ballot b was successfully confirmed, and sends sfc to VD.

3. Framework

In the coming sections we present a ProVerif model of the Neuchâtel protocol. A detailed presentation of the syntax and semantics of ProVerif can be found in [7]. For the sake of readability, we give next an overview of the ProVerif model, focusing on the parts that are more relevant to our model. Namely, we provide the syntax and semantics of processes in ProVerif, as well as the definitions of correspondence and equivalence properties. Notations and definitions are mainly borrowed from [7].

3.1. Syntax

We assume a set $\mathcal{V}$ of variables, a set $\mathcal{N}$ of names, a set $\mathcal{T}$ of types. By default in ProVerif, types include $channel$ for

1. Actually, the credential retrieval through the keystore also includes a challenge-response phase where the voter proves that she has successfully opened her keystore. We abstracted away this phase as it is not used in the rest of the protocol.

$M, N, M_1, \ldots, M_k$ ::= terms
  $x \mid n \mid f(M_1, \ldots, M_k)$ where $x \in \mathcal{V}$, $n \in \mathcal{N}$, and $f \in \mathcal{C}$

D::= expressions
  $M \mid h(D_1, \ldots, D_k)$ where $h \in \mathcal{C} \cup \mathcal{D}$

ϕ::= formula
  $M = N \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi$

P, Q::= processes
  0 nil
  $\mathsf{out}(N, M); P$ output
  $\mathsf{in}(N, x : T); P$ input
  $P \mid Q$ parallel composition
  $!P$ replication
  $\mathsf{new}\ a : T; P$ restriction
  $\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ assignment
  $\mathsf{if}\ \phi\ \mathsf{then}\ P\ \mathsf{else}\ Q$ conditional
  $\mathsf{event}(M); P$ event

Figure 2. Syntax of the core language of ProVerif.

channel's names, and *bitstrings* for bitstrings (also written any). The syntax for *terms*, *expressions*, and *processes* is displayed in Figure 2.

Terms and expressions. Symbols for functions are split into two sets of constructors $\mathcal{C}$ and destructors $\mathcal{D}$ respectively. Terms are built over names, variables and constructors and represent actual messages sent over the network, while expressions may also contain destructors representing cryptographic computations that extract or rebuild data. Function symbols are given with their types: $g(T_1, \ldots, T_n) : T$ means that the function g takes n arguments as input of types respectively $T_1, \ldots, T_n$ and returns a result of type T. A substitution is a mapping from variables to terms, denoted $\{U_1/x_1, \ldots, U_n/x_n\}$. The application of a substitution σ to a term U, denoted $U\sigma$, is obtained by replacing variables by the corresponding terms and is defined as usual. We only consider well-typed substitutions.

The evaluation of an expression is defined through rewrite rules. Specifically, each destructor d is associated with a rewrite rule of the form $d(U_1, \ldots, U_n) \rightarrow U$, over terms. Then the evaluation of an expression is recursively defined as follows:

- $g(D_1, \ldots, D_n)$ *evaluates* to U, which is denoted by $g(D_1, \ldots, D_n) \Downarrow U$, if $\forall i,\ D_i \Downarrow U_i$, and g is a constructor ($g \in \mathcal{C}$) and $U = g(U_1, \ldots, U_n)$; or g is a destructor ($g \in \mathcal{D}$) and there exists a substitution σ such that $U_i = U'_i\sigma$, $U = U'\sigma$, where $g(U'_1, \ldots, U'_n) \rightarrow U'$ is the rewrite rule associated to g.
- $g(D_1, \ldots, D_n)$ *evaluates* to fail, which is denoted by $g(D_1, \ldots, D_n) \Downarrow \mathsf{fail}$, otherwise.

The evaluation $[\![\phi]\!]$ of a formula ϕ is defined by $[\![M = N]\!] = \top$ if $M = N$ syntactically, or $[\![M = N]\!] = \bot$ otherwise, and is then extended to $\wedge, \vee, \neg$ as expected.

Example 1. *To model the simple theory of encryption and concatenation, we consider a type symkey for symmetric keys and the sets of constructors and destructors with their associated rewrite rules as follows:*

$$\begin{aligned}
\mathcal{C}_{\mathsf{basic}} = \{ \quad & \mathit{pair}(\mathit{any}, \mathit{any}) : \mathit{any}, \\
& \mathit{enc}(\mathit{symkey}, \mathit{any}) : \mathit{any} \quad \} \\
\mathcal{D}_{\mathsf{basic}} = \{ \quad & \mathit{proj}1(\mathit{any}) : \mathit{any},\ \mathit{proj}2(\mathit{any}) : \mathit{any}, \\
& \mathit{dec}(\mathit{symkey}, \mathit{any}) : \mathit{any} \quad \} \\
\mathcal{R}_{\mathsf{basic}} = \{ \quad & \mathit{proj}1(\mathit{pair}(x, y)) \rightarrow x, \\
& \mathit{proj}2(\mathit{pair}(x, y)) \rightarrow y, \quad \} \\
& \mathit{dec}(y, \mathit{enc}(y, x)) \rightarrow x
\end{aligned}$$

We often write (m_1, m_2) *instead of* $\mathit{pair}(m_1, m_2)$ *and* $(m_1, m_2, \ldots, m_k)$ *stands for* $(m_1, (m_2, (\ldots, m_k)))$.

Processes. Figure 2 provides a convenient abstract language for describing protocols (formally modeled as processes). The output of a message M on channel N is represented by $\mathsf{out}(N, M); P$ while $\mathsf{in}(N, x : T); P$ represents an input on channel N, stored in variable x. Process $P \mid Q$ models the parallel composition of P and Q, while $!P$ represents P replicated an arbitrary number of times. $\mathsf{new}\ a : T; P$ generates a fresh name of type T and behaves like P. $\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$ evaluates D and behaves like P unless the evaluation fails, in which case it behaves like Q. The if case is similar. $\mathsf{event}(M); P$ is used to specify security property: the process emits an *event* (not observable by an attacker) to reflect that fact that it reaches some specific state, with some values, stored in M.

The set of free names of a process P is denoted $\mathit{fn}(P)$, and the set of it's free variables by $\mathit{fv}(P)$. A *closed* process is a process with no free variables. Following ProVerif's handy notations, we may write $\mathsf{in}(c, = x).P$ instead of $\mathsf{in}(c, y : T).\mathsf{if}\ x = y\ \mathsf{then}\ P$, where T is the type of x. Similarly, we may write $\mathsf{in}(c, (x : T, y : T')).P$ instead of $\mathsf{in}(c, z : \mathit{any}).\mathsf{let}\ x : T = \mathit{proj}1(z)\ \mathsf{in}\ \mathsf{let}\ y : T' = \mathit{proj}2(z)\ \mathsf{in}\ P$.

Example 2. *In the Neuchâtel voting protocol, the Voter interacts with a Voting Device (e.g. her computer or cell phone) to cast her vote. Initially, the voter receives a voting card with her personal data for the election, including a password* pwd *(used to derive the voter's key and id), one short return code* src *for each candidate in the list, a confirmation code* cc_{id} *(sent if the received return codes are valid), and a short finalization code* sfc_{id} *(that should correspond to the server's last acknowledgement message). For simplicity, we model an election where voters have to select two options. We model a voter that votes for two options* v_1, v_2*, with corresponding return codes* $\mathit{src}_1, \mathit{src}_2$ *(read from the voting card). The corresponding process is defined in Figure 3. It communicates on channel* c *with the voting device. It includes two events that witness some important states of the voter. They will be used later to*

$$
\begin{array}{ll}
\mathit{Voter}(c, \mathit{pwd}, v_1, v_2, \mathit{src}_1, \mathit{src}_2, \mathit{cc}_{id}, \mathit{sfc}_{id}) := & \\
\quad \mathsf{out}(c, (\mathit{pwd}, v_1, v_2)); & \text{(* Sends password \& choices. *)} \\
\quad \mathsf{in}(c, (\mathit{src}'_1 : \mathit{any}, \mathit{src}'_2 : \mathit{any})); & \text{(* Gets the return codes. *)} \\
\quad \mathsf{if}\ (\mathit{src}'_1 = \mathit{src}_1 \wedge \mathit{src}'_2 = \mathit{src}_2)\ \vee\ (\mathit{src}'_1 = \mathit{src}_2 \wedge \mathit{src}'_2 = \mathit{src}_1)\ \mathsf{then} & \\
\quad\quad \mathsf{event}(\mathit{confirmed}(\mathit{pwd}, v_1, v_2)); & \text{(* Checks ok; Reaches '}\mathit{confirmed}\text{'. *)} \\
\quad\quad \mathsf{out}(c, \mathit{cc}_{id}); & \text{(* Confirms the vote. *)} \\
\quad\quad \mathsf{in}(c, =\mathit{sfc}_{id}); & \text{(* Final confirmation. *)} \\
\quad\quad \mathsf{event}(\mathit{happy}(\mathit{pwd}, v_1, v_2)). & \text{(* All ok; Reaches '}\mathit{happy}\text{'. *)}
\end{array}
$$

Figure 3. The Voter Process

$$
\begin{array}{l}
E,\mathcal{P} \cup \{0\} \rightarrow E,\mathcal{P} \\
E,\mathcal{P} \cup \{P \| Q\} \rightarrow E,\mathcal{P} \cup \{P, Q\} \\
E,\mathcal{P} \cup \{!P\} \rightarrow E,\mathcal{P} \cup \{P, !P\} \\
(\mathcal{N}_{pub}, \mathcal{N}_{priv}),\mathcal{P} \cup \{\mathsf{new}\ a : T; P\} \rightarrow (\mathcal{N}_{pub}, \mathcal{N}_{priv} \cup \\
\quad \{a'\}),\ \mathcal{P} \cup \{P[a'/a]\} \quad \text{where } a' \notin \mathcal{N}_{pub} \cup \mathcal{N}_{priv} \\
E,\mathcal{P} \cup \{\mathsf{out}(N, M); Q, \mathsf{in}(N, x); P\} \rightarrow \\
\quad E,\mathcal{P} \cup \{Q, P[M/x]\} \\
E,\mathcal{P} \cup \{\mathsf{let}\ x = D\ \mathsf{in}\ P\} \rightarrow E,\mathcal{P} \cup \{P[M/x]\} \\
\qquad\qquad\qquad\qquad \text{if } D \Downarrow M \text{ and } M \neq \mathsf{fail} \\
E,\mathcal{P} \cup \{\mathsf{if}\ \phi\ \mathsf{then} P\} \rightarrow E,\mathcal{P} \cup \{P\} \qquad \text{if } [\![\phi]\!] = \top \\
E,\mathcal{P} \cup \{\mathsf{event}(M); P\} \rightarrow E,\mathcal{P} \cup \{P\}
\end{array}
$$

Figure 4. Transitions between configurations, without types for clarity

.

formally state security properties (see Section 5), and are defined by:

$$
\begin{array}{rl}
\mathcal{C}_{\mathsf{voter}} = \{ & \mathit{confirmed}(\mathit{password}, \mathit{int}, \mathit{int}) : \mathit{any}, \\
& \mathit{happy}(\mathit{password}, \mathit{int}, \mathit{int}) : \mathit{any} \quad \}
\end{array}
$$

3.2. Semantics

A *configuration* $E, \mathcal{P}$ is given by a multiset $\mathcal{P}$ of processes, representing the current state of the processes, and a set $E = (\mathcal{N}_{pub}, \mathcal{N}_{priv})$ representing respectively the public and private names used so far. The semantics of processes is defined through a reduction relation $\rightarrow$ between configuration, defined in Figure 4. A *trace* is a sequence of reductions between configurations $E_0, \mathcal{P}_0 \rightarrow \cdots \rightarrow E_n, \mathcal{P}_n$. We say that a trace $E_0, \mathcal{P}_0 \rightarrow^* E', \mathcal{P}'$ *executes* an event M if it contains a reduction $E, \mathcal{P} \cup \{\mathsf{event}(M); P\} \rightarrow E, \mathcal{P} \cup \{P\}$ for some $E, \mathcal{P}, P$.

3.3. Properties

As usual, we assume that protocols are executed in an untrusted network, meaning that communications over a public network are fully controlled by an attacker who may eavesdrop, intercept, or send messages. This is easily modeled by executing a protocol $\mathcal{P}_0$ in parallel with an arbitrary process Q. Formally, we assume given a set of *public constructors*, subset of the constructors. An *adversarial process* w.r.t. to a set of names $\mathcal{N}_{pub}$ is a process Q such that $\mathit{fn}(Q) \subset \mathcal{N}_{pub}$ and Q uses only public constructors (and destructors). In what follows, all constructors are public, unless otherwise specified.

3.3.1. Correspondence. Many security properties can be stated as “if Alice reaches some state (e.g. finishes her session) then Bob must have engaged a conversation with her”. This is for example the case of many variants of agreement properties [30]. In the context of voting, such correspondence properties can be used to express verifiability. For example, we may wish to state that whenever Alice thinks she has voted for v then there is indeed a ballot registered on her name that corresponds to v.

ProVerif allows to specify *correspondence* properties between events.

Definition 1. *A closed process P_0 satisfies the correspondence*

$$
\mathsf{event}(M) \rightsquigarrow \bigwedge_{i=1}^{m} \bigvee_{j=1}^{l_i} \mathsf{event}(M_{ij})
$$

where the $M_{i,j}$ do not contain names, if for any (adversarial) closed process Q such that $\mathit{fn}(Q) \subset \mathit{fn}(P_0)$, for any trace tr of $P_0 \mid Q$, for any substitution σ, if tr executes the event $M\sigma$, then for any i, there exists j such that tr executes event $M_{ij}\sigma'$.

Examples can be found in Section 5.

3.3.2. Equivalence. Equivalence properties are crucial to express vote privacy. Observational equivalence of two processes P and Q models the fact that an adversary cannot distinguish between the two processes. Slightly more precisely, whenever P may emit on some channel c (interacting with an adversarial process R), then Q can emit on c as well. For readability, we summarize here the definition of equivalence from [7]. We write $\mathcal{C} \downarrow_N$ when a configuration $\mathcal{C} = E, \mathcal{P}$ with $E = (\mathcal{N}_{pub}, \mathcal{N}_{priv})$ can output on some channel N, i.e. if there exists $\mathsf{out}(N, M); P \in \mathcal{P}$ such that $\mathit{fn}(N) \in \mathcal{N}_{pub}$. Also, an adversarial context $C[_]$ is a process of the form $\mathsf{new}\ n : \mathit{any}; _ \mid Q$ where $\mathit{fv}(Q) = \emptyset$ and all functional symbols in Q are public, with _ being a 'hole' expected to be filled by a configuration $\mathcal{C} = (\mathcal{N}_{pub}, \mathcal{N}_{priv}), \mathcal{P}$. Therefore, and assuming that $\mathcal{N}_{priv} \cap \mathit{fn}(Q) = \emptyset$, the application of one to the other is defined by :

$$
\begin{array}{rcl}
C[\mathcal{C}] & = & (\mathcal{N}'_{pub}, \mathcal{N}'_{priv}), \mathcal{P} \cup \{Q\} \\
\text{with } \mathcal{N}'_{pub} & = & (\mathcal{N}_{pub} \cup \mathit{fn}(Q)) \backslash \{n\}
\end{array}
$$

and $\mathcal{N}'_{priv} = \mathcal{N}_{priv} \cup \{n\}$

From this, the definition of observational equivalence follows :

Definition 2. *The* observational equivalence *between configurations, denoted by $\approx$, is the largest symmetric relation such that $\mathcal{C} \approx \mathcal{C}'$ implies :*

- *if $\mathcal{C} \downarrow_N$ then $\exists \mathcal{C}'_1$ s.t. $\mathcal{C}' \to^* \mathcal{C}'_1$ and $\mathcal{C}'_1 \downarrow_N$;*
- *if $\mathcal{C} \to \mathcal{C}_1$, then $\exists \mathcal{C}'_1$ s.t. $\mathcal{C}' \to^* \mathcal{C}'_1$ and $\mathcal{C}_1 \approx \mathcal{C}'_1$;*
- *$C[\mathcal{C}] \approx C[\mathcal{C}']$, for any adversarial context $C[_]$.*

4. Formal model of the Neuchâtel's voting system

We present here the main parts of our formal model. For simplicity, we present a model for elections with $k = 2$ choices to be made amongst n voting options. In our automated analysis, we generate the model corresponding to any given particular value k automatically, and then run ProVerif on several values of k (up to $k = 4$ due to ProVerif time out).

4.1. Standard primitives

The Neuchâtel protocol makes use of the standard primitives: symmetric and asymmetric encryption, signatures, hashes and concatenation. We consider the corresponding types $agentId$, int, $ekey$, $epkey$, $skey$, $spkey$, $symkey$. The constructors and the associated rewrite rules are defined as follows.

$$\begin{aligned}\mathcal{C}_{\mathsf{stand}} = \{ \quad & sk_e(agentId) : ekey,\\ & pub_e(ekey) : epkey,\\ & aenc(epkey, any, int) : any,\\ & enc(symkey, any) : any,\\ & pair(any, any) : any,\\ & sk_s(agentId) : skey,\\ & pub_s(skey) : spkey,\\ & sign(skey, any) : any \quad \}\end{aligned}$$

$$\begin{aligned}\mathcal{R}_{\mathsf{stand}} = \{ \quad & adec(k, aenc(pub_e(k), m, r)) \to m\\ & dec(k, enc(k, m)) \to m\\ & proj1(pair(a, b)) \to a\\ & proj2(pair(a, b)) \to b\\ & verify(pub_s(k), m, sign(k, m)) \to \mathsf{ok} \quad \}\end{aligned}$$

where all function symbols are public, except sk_e and sk_s that are private. The set of destructors $\mathcal{D}_{\mathsf{stand}}$ can be inferred easily. The term $sk_\alpha(id)$ represents the private key of user id w.r.t. scheme α, where $\alpha = e$ stands for asymmetric encryption while $\alpha = s$ stands for signature. The rewrite rules are the standard ones for these primitives. For example $adec(k, aenc(pub_e(k), m, r)) \to m$ models the fact that the plaintext of an (asymmetric) encryption can be retrieved by decrypting with the corresponding private key.

4.2. Voting device

A voter id is provided with a password pwd and codes (return codes as well as a confirmation code). When she connects to the voting server through her voting device, she first needs to retrieve her personal private key $sk_e(\delta_{Id}(pwd))$. The identifier id of the voter is actually derived from her password, that is $id = \delta_{Id}(pwd)$. This key is stored in a keystore (on the server's side), encrypted with a key that can be derived from the password: $\delta_{Key}(pwd)$. We therefore introduce the following theory to model the key store.

$$\begin{aligned}\mathcal{C}_{ks} = \{ \quad & \delta_{Id}(password) : agentId,\\ & \delta_{Key}(password) : symkey,\\ & c_{ekey}(ekey) : any \quad \}\\ \mathcal{R}_{ks} = \{ \quad & c_{any}(c_{ekey}(k)) \to k \quad \}\end{aligned}$$

The corresponding set of destructors $\mathcal{D}_{ks}$ can be inferred easily. The functions c_{any} and c_{ekey} are auxiliary functions that convert private keys to bitstring and conversely.

We denote

$$ks(pwd) := enc(\delta_{Key}(pwd), c_{ekey}(sk_e(\delta_{Id}(pwd))))$$

the encrypted value stored in the key store for voter $\delta_{Id}(pwd)$.

As explained in Section 2, the voting device builds a ballot as follows. It encrypts the choices v_1, v_2 of the voters; builds pre-return codes for each choice $prc(sk_e, v_1)$, $prc(sk_e, v_2)$ using the private key of the voter; and proves that the return codes correspond to the encrypted votes, through a zero-knowledge proof. This is modeled as follows.

$$\begin{aligned}\mathcal{C}_{\mathsf{zk}} = \{ & zkp(epkey, epkey, any, int, int, int, ekey) : any,\\ & prc(ekey, int) : int \quad \}\end{aligned}$$

$$\begin{aligned}\mathcal{R}_{\mathsf{zk}} = \{ & verif_{zkp}(pk_b, pk, e, p_1, p_2,\\ & \quad zkp(pk_b, pk, e, p_1, p_2, \ r, sk_{id})) \to \mathsf{ok} \quad \}\\ & \text{with } pk = pub_e(sk_{id}),\ e = aenc(pk_b, v, r),\\ & \quad p_1 = prc(sk_{id}, v_1),\ p_2 = prc(sk_{id}, v_2)\end{aligned}$$

with $v = (v_1, v_2)$. Note that here, our model abstracts some properties of the primitives. For example, the two choices v_1, v_2 of the voters are not encrypted as a list but are "compacted" in a single integer $\Phi(v_1, v_2)$, which is actually commutative: $\Phi(v_1, v_2) = \Phi(v_2, v_1)$. Similarly, the zero-knowledge proof compacts the pre-return codes. Since ProVerif cannot handle associative and commutative properties, we abstract away these properties, assuming a slightly stronger proof system.

We can now provide the Device process, which is shown in Figure 5.

4.3. Voting server

When the voting server is contacted by some voter id (through her voting device), the server first needs to retrieve the personal keystore associated to id. Only valid ids, of the

form $id = \delta_{Id}(pwd)$, are registered. The server also recovers (for later use) a signature $sign(sk_c, sfc(gk_a, id))$ of the short finalization code $sfc(gk_a, id)$ that should be rebuilt and sent at the end by the server (defined in $\mathcal{C}_{rc}$ below). The signing key sk_c is a fixed long term key of the setup authorities and gk_a is the global audit key of the server. We model this data retrieval by considering the following rewrite rule:

$$\mathcal{R}_{\mathsf{retrieve}} = \{ \ Get(\delta_{Id}(pwd), gk_a) \rightarrow (ks(pwd), sign(sk_c, sfc(gk_a, id))) \ \}$$

The server needs to compute a long return code, i.e. $f(gk_a, prc_1)$, from the partial return code prc_1 sent by the voting device. This long return code is too long to be human readable and is therefore associated (in a table) to a short return code. This table also provides a correspondence for long and short finalization codes. This is modeled through the following theory:

$$\mathcal{C}_{rc} = \{ \begin{array}{ll} src(symkey, agentId, int) & : any \\ sfc(symkey, agentId) & : any \\ f(symkey, int) & : symkey \\ cc(agentId) & : int \end{array} \}$$

$$\mathcal{R}_{rc} = \{ \begin{array}{l} read_{RC}(f(gk_a, prc(sk_e(id), j))) \\ \quad \rightarrow src(gk_a, id, j) \\ read_{FC}(f(gk_a, prc(sk_e(id), cc(id)))) \\ \quad \rightarrow sfc(gk_a, id) \end{array} \}$$

with cc, src and sfc private function symbols. Actually, short return codes are stored encrypted using the long return code as a symmetric key. For the sake of clarity, we omit this part here but it is reflected in our ProVerif's model.

Validity of pre-return codes. A careful server must also check that the pre-return codes it receives belong to a known list of *valid codes*, meaning that the votes corresponding to those codes are in the range of valid votes for the election. While omitted here for readability, this test is reflected in our ProVerif's model by improving the destructor $verif_{zkp}$. This destructor will return ok only if the given pre-return codes can be matched against a pattern in which only valid pre-return codes belong to. This avoids the (impossible) storage of an infinite list of codes.

No revote. One of the main challenging tasks when modeling the voting server is the fact that it accepts at most one request from each voter. Note that the protocol is insecure otherwise. Indeed, in case revoting was allowed, a malicious voting device could first vote as queried by the voter and display the (correct) return codes and then revotes for the candidate of its choice (discarding the corresponding return codes). Therefore, it is crucial to model accurately that no voter can vote twice.

A first approach (as described in [20]) is to use a table that stores whether a voter already voted or not. Then, intuitively, the code of the server is (informally) as follows:

if $id \notin$ Table then proceed and add id to Table
else stop

However, since the voting device may process several requests at the same time, checking whether $id \notin$ Table is actually insufficient. Indeed, if two requests from the same voter reach the server at the same time, they would both pass the test $id \notin$ Table and both ballots would be accepted. We therefore need a clean lock mechanism, for which there is a lot of implementation support.

However, when it comes to modeling this lock mechanism in ProVerif, we have two options. The first option is to encode the lock mechanism directly in ProVerif, for example using private channels, that can be used as "tokens". This approach presents two drawbacks. Firstly, this encoding would necessarily be ad-hoc. And in principle there would be no assurance that the protocol is secure if the lock mechanism is implemented in another way (and of course, real lock mechanisms will never use private channels). Secondly, it is also known that ProVerif, due to its internal behavior, cannot properly handle private channels when used as tokens or, more generally, cannot handle properly events that happen "at most once".

Instead, we take a different point of view. We model a voting server that does *not* prevent revoting and adds blindly new ballots. Then, instead of asking ProVerif whether some property ϕ holds, we query a property of the form

$\phi \vee$ two ballots have been accepted for the same voter

If ProVerif proves this query, it guarantees that for any execution trace where no 2 ballots are accepted by the server, then ϕ holds. It is then up to the implementation to ensure that any execution is such that no 2 ballots for the same voter are accepted by the server.

This yields a more flexible result: for any realization of the protocol that further ensures that "no two ballots from the same voters are accepted", then the realization satisfies ϕ, no matter how the "no revote policy" is actually implemented. We believe that this approach is of independent interest and could re-used in other contexts when modeling e.g. lock mechanisms in ProVerif.

Formally, the server simply issues events built from $\mathcal{C}_{\mathsf{BB}} = \{InsertBB(agentId, any) : any\}$ to record the fact that a ballot has been added to the box. The process corresponding to the voting server is then defined in Figure 6. These events are used later to specify security properties as just discussed.

4.4. Tally process

Once the voting process is over, the tally phase can start. The tally authorities check the validity of each ballot (same checks as the server); mix the ciphertexts containing the votes; decrypt the mixed ciphertexts and publish the election result. In our analysis, we consider that either the tally authorities are corrupted (for cast-as-intended and recorded-as-cast properties) or that the overall tally process is honest (for ballot privacy). As we shall see in Section 5 (security properties), we only need to consider two honest voters, as well as arbitrary many dishonest voters.

Since privacy is ensured as soon as the two honest ballots are mixed, we model a tally process that mixes the two honest ballots only. The mixnet is modeled in a standard way, by sending the ballots over a private channel called `mix` concurrently, thus without fixing the order, and reading them back from that same channel. Formally, we define two processes: TallyH, as shown in Figure 7 for the honest voters (whose ballots are mixed); and TallyD as shown in Figure 8 for any dishonest voter (that can be executed arbitrarily). The process TallyD simply decrypts any (valid) ballot provided the corresponding id is not an honest voter, that is, is neither id_a nor id_b.

Note that our model of tally is very abstract: it takes as input the encrypted ballots and returns the votes in clear, in an arbitrary order. In practice, such behaviour is typically implemented using several mixers, that re-encrypt ballots at each intermediary step.

5. Security properties

The Neuchâtel protocol is designed to achieve cast-as-intended verifiability: even if the voter's device is corrupted, the ballot registered in the name of a voter corresponds to the vote *intended* by *that* voter. This offers a strong protection against attackers-controlled personal computers and smartphones (e.g. through malware). The voting server shall be trusted for this step. Conversely, the Neuchâtel's protocol also guarantees vote privacy against a dishonest voting server. Note however that the protocol is not universally verifiable: the content of the ballot box is not public and therefore voters cannot check that the result corresponds to the received ballots. We define next the security properties proved in ProVerif.

5.1. Verifiability properties

The protocol ensures cast-as-intended and recorded-as-cast: if a voter successfully completes the voting procedure, she is guaranteed that her ballot has been property recorded by the voting server.

Cast-as-Intended. The Neuchâtel's protocol provides *Cast-as-Intended* verifiability: if the server registers a ballot for some voter id then this ballot contains the votes *intended* by the voter. This can be formalized by the following correspondence property. Remember that the identity id of a voter is derived from her password ($id = \delta_{Id}(pwd)$).

$$\begin{aligned}&\mathsf{event}(\mathit{HasVoted}(pk_b, \delta_{Id}(pwd_a), e)) \Rightarrow\\ &\exists\ v_1, v_2, j_1, j_2, r,\\ &\quad \mathsf{event}(\mathit{confirmed}(pwd_a, v_1, v_2))\\ &\quad \wedge\ e = \mathit{aenc}(pk_b, (j_1, j_2), r)\\ &\quad \wedge\ ((j_1 = v_1 \wedge j_2 = v_2)\ \vee\ (j_1 = v_2 \wedge j_2 = v_1))\end{aligned} \tag{1}$$

Intuitively, the above reads as follows: if the server issues an event $\mathit{HasVoted}(pk_b, \delta_{Id}(pwd_a), e)$, meaning that he accepted a ballot containing an encryption e, then the voter with password pwd_a must have had cast a vote (v_1, v_2) that corresponds to e. Note that we cannot exclude the case where a malicious device swaps the vote (that is, casts (v_2, v_1) instead of (v_1, v_2)) and then swaps the received return codes. This is captured in the property above by allowing the two cases (the option order has no impact on the way votes are counted).

Unfortunately, ProVerif fails to prove this property. Indeed, cast-as-intended cannot be guaranteed as soon as the server may answer two requests from the same voter, as the attacker would then get two sets of return codes and could show the wrong one. This is explicitly forbidden by the Neuchâtel's protocol: the server does not answer to revote queries. However, ProVerif over-approximates the behaviors and takes into account the case where the server would answer twice (yielding "cannot be proved").

Instead, we consider the following correspondence property.

$$\begin{aligned}&\mathsf{event}(\mathit{HasVoted}(pk_b, \delta_{Id}(pwd_a), e)) \Rightarrow\\ &\exists\ v_1, v_2, j_1, j_2, j_3, j_4, j_5, j_6, r, r_1, r_2,\\ &\quad \mathsf{event}(\mathit{confirmed}(pwd_a, v_1, v_2))\\ &\quad \wedge\ e = \mathit{aenc}(pk_b, (j_1, j_2), r)\\ &\quad \wedge\ \mathsf{event}(\mathit{InsertBB}(\delta_{Id}(pwd_a), e))\\ &\quad \wedge\ \mathsf{event}(\mathit{InsertBB}(\delta_{Id}(pwd_a), \mathit{aenc}(pk_b, (j_3, j_4), r_1)))\\ &\quad \wedge\ \mathsf{event}(\mathit{InsertBB}(\delta_{Id}(pwd_a), \mathit{aenc}(pk_b, (j_5, j_6), r_2)))\\ &\quad \wedge\ (j_3 = v_1 \vee j_4 = v_1)\ \wedge\ (j_5 = v_2 \vee j_6 = v_2)\end{aligned} \tag{2}$$

This property states that if the server accepts a ballot containing an encryption e then the voter corresponding to pwd_a must have cast a vote (v_1, v_2) such that the encryption e has been inserted in the ballot box, on behalf of $\delta_{Id}(pwd_a)$. Moreover, there must have been two (not necessarily distinct) insertions in the ballot box: one with (j_3, j_4) and one with (j_5, j_6), such that v_1 is equal to either j_3 or j_4 and v_2 is equal to either j_5 or j_6.

Why is this useful? Because we know that the protocol's implementation further guarantees that there is at most one insertion for each voter. Combined with Property 2, this implies $j_1 = j_3 = j_5 \wedge j_2 = j_4 = j_6$, hence the desired Property 1 (since $v_1 \neq v_2$).

More formally, for any trace $\mathtt{tr}$ of a process, if $\mathtt{tr}$ satisfies (2) and is such that there no distinct insertion for the same voter, that is:

$$\begin{aligned}&\mathsf{event}(\mathit{InsertBB}(\delta_{Id}(pwd_a), \mathit{aenc}(pk_b, (j_3, j_4), r_1)))\\ \wedge&\mathsf{event}(\mathit{InsertBB}(\delta_{Id}(pwd_a), \mathit{aenc}(pk_b, (j_5, j_6), r_2)))\\ &\Rightarrow (j_3 = j_5) \wedge (j_4 = j_6) \wedge (r_1 = r_2)\end{aligned}$$

then $\mathtt{tr}$ satisfies (1). The proof is immediate.

The first interesting feature of this encoding is of course that it circumvents the issue that ProVerif over-approximates the no-revote policy. The second feature is that Property 1 is proved *independently* of the exact implementation of the no-revote policy. Assuming that the implementation guarantees that there is at most one insertion for each voter, then the protocol achieves cast-as-intended, no matter how this is implemented in practice.

Note that our encoding strongly relies on the fact that voters select two distinct options ($v_1 \neq v_2$). There are attacks otherwise, as explained in Section 6.2. In our model,

```
Device(c_1, c_2, pk_b) :=                    (* channels c_1 with Voter and c_2 with Server. *)
  in(c_1, (pwd : password, v_1 : int, v_2 : int));          (* Get password & choices.   *)
  out(c_2, δ_Id(pwd));                                      (* Ask for the keystore,      *)
  in(c_2, ks : any);                                        (* ... receive keystore,      *)
  let sk_id : ekey = c_any(dec(δ_Key(pwd), ks)) in          (* ... retrieve the key.      *)
  new r : int; let e = aenc(pk_b, (v_1, v_2), r) in         (* Encrypt voter's choices.   *)
  let p = zkp(pk_b, pub_e(sk_id), e, prc(sk_id, v_1), prc(sk_id, v_2), r, sk_id) in
  out(c_2, (e, prc(sk_id, v_1), prc(sk_id, v_1), pub_e(sk_id), p));   (* Sends the ballot. *)
  in(c_2, (src_1 : any, src_2 : any));                      (* Get the short return codes *)
  out(c_1, (src_1, src_2));                                 (* ... transmit codes.        *)
  in(c_1, cc_id : int);                                     (* Get confirmation code      *)
  out(c_2, prc(sk_id, cc_id));                              (* ... transmit it.           *)
  in(c_2, sfc_id : any);                                    (* Get short finalization code *)
  out(c_1, sfc_id).                                         (* ... transmit it.           *)
```

Figure 5. The Device Process

```
Server(c : channel, pk_b : epkey, gk_a : symkey, pk_c : spkey, c_t : channel) :=
  in(c, id : agentId);                                      (* New voting requests.       *)
  let (ks : any, crf : any) = Get(id, gk_a) in              (* Recovers the keystore,     *)
  out(c, ks);                                               (* ... and transmits it.      *)
  in(c, b : any);                                           (* Waits for a ballot.        *)
  let (e : any, prc_1 : int, prc_2 : int, =pk_e(id), p : any) = b in      (* Parse it.   *)
  if verif_zkp(pk_b, pk_e(id), e, prc_1, prc_2, p) then     (* Checks the proof.          *)
    event(InsertBB(id, e));                                 (* Table addition.            *)
    let src_1 = read_RC(f(gk_a, prc_1)) in                  (* Gets the short return codes. *)
    let src_2 = read_RC(f(gk_a, prc_2)) in
    out(c, (src_1, src_2));                                 (* Sends them to the Device.  *)
    ! in(c, cm : int);                                      (* Waits for confirmation.    *)
      let sfc_id : any = read_FC(f(gk_a, cm)) in            (* Gets the finalization code. *)
      if verify(pk_c, sfc_id, crf) then                     (* Checks the signature.      *)
      event(HasVoted(pk_b, id, e);                          (* Vote approval.             *)
      out(c, sfc_id); out(c_t, (id, b, sfc_id, crf)).       (* Confirms; Feeds the Tally. *)
```

Figure 6. The Server Process

```
TallyH(c_t : channel, sk_b : ekey, pk_c : spkey, id_a : agentId, id_b : agentId) :=
  in(c_t, (= id_a, b_a : any, sfc_a : any, crf_a : any));
  in(c_t, (= id_b, b_b : any, sfc_b : any, crf_b : any));
  let (e_a : any, prc_a^1 : int, prc_a^2 : int, = pk_e(id_a), p_a : any) = b_a in
  let (e_b : any, prc_b^1 : int, prc_b^2 : int, = pk_e(id_b), p_b : any) = b_b in
  if   verif_zkp(pub_e(sk_b), pk_e(id_a), e_a, prc_a^1, prc_a^2, p_a) ∧ verify(pk_c, sfc_a, crf_a)
  ∧    verif_zkp(pub_e(sk_b), pk_e(id_b), e_b, prc_b^1, prc_b^2, p_b) ∧ verify(pk_c, sfc_b, crf_b) then
    out(mix, e_a) | out(mix, e_b) |
    ( in(mix, e'_a : any); in(mix, e'_b : any); out(c, (adec(sk_b, e'_a), adec(sk_b, e'_b))) ).
```

Figure 7. The Tally Process – Honest version

```
TallyD(c_t : channel, sk_b : ekey, pk_c : spkey, id_a : agentId, id_b : agentId) :=
  in(c_t, (id, b : any, sfc : any, crf : any));
  let (e : any, prc^1 : int, prc^2 : int, = pk_e(id), p : any) = b in
  if   verif_zkp(pub_e(sk_b), pk_e(id), e, prc^1, prc^2, p) ∧ verify(pk_c, sfc, crf)
  ∧    id ≠ id_a ∧ id ≠ id_b then
    out(c, adec(sk_b, e)).
```

Figure 8. The Tally Process – Dishonest version

we only consider voters that select distinct options, as instructed, and therefore $v_1 \neq v_2$ trivially holds.

Recorded-as-cast. The protocol further guarantees that if a voter completes the voting process then she is ensured that her vote has been recorded by the server. This property can be formally stated as follows.

$$\begin{array}{l} \mathsf{event}(\mathit{happy}(\mathit{pwd}_a, v_1, v_2)) \Rightarrow \\ \exists\ j_1, j_2, r, \\ \quad \mathsf{event}(\mathit{HasVoted}(\mathit{pk}_b, \delta_{Id}(\mathit{pwd}_a), e)) \\ \quad \wedge\ e = \mathit{aenc}(\mathit{pk}_b, (j_1, j_2), r) \\ \quad \wedge\ (j_1 = v_1 \wedge j_2 = v_2)\ \vee\ (j_1 = v_2 \wedge j_2 = v_1) \end{array} \tag{3}$$

In our ProVerif model, we further show that the ballot registered by the server is well-formed and will therefore be accepted at the tally phase. Similarly to cast-as-intended, this property cannot be proved in ProVerif. So instead, we prove an amended property which implies the desired property as soon as the implementation guarantees that there is at most one ballot insertion per voter.

$$\begin{array}{l} \mathsf{event}(\mathit{happy}(\mathit{pwd}_a, v_1, v_2)) \Rightarrow \\ \exists\ j_1, j_2, j_3, j_4, j_5, j_6, r, r_1, r_2, \\ \quad \mathsf{event}(\mathit{HasVoted}(\mathit{pk}_b, \delta_{Id}(\mathit{pwd}_a), e)) \\ \quad \wedge\ e = \mathit{aenc}(\mathit{pk}_b, (j_1, j_2), r) \\ \quad \wedge\ \mathsf{event}(\mathit{InsertBB}(\delta_{Id}(\mathit{pwd}_a), e)) \\ \quad \wedge\ \mathsf{event}(\mathit{InsertBB}(\delta_{Id}(\mathit{pwd}_a), \mathit{aenc}(\mathit{pk}_b, (j_3, j_4), r_1))) \\ \quad \wedge\ \mathsf{event}(\mathit{InsertBB}(\delta_{Id}(\mathit{pwd}_a), \mathit{aenc}(\mathit{pk}_b, (j_5, j_6), r_2))) \\ \quad \wedge\ (j_3 = v_1 \vee j_4 = v_1)\ \wedge\ (j_5 = v_2 \vee j_6 = v_2) \end{array} \tag{4}$$

Trust Assumptions. We prove cast-as-intended (Property 2) and recorded-as-cast (Property 4) even if the voting device and the tally process are corrupted. We assume however the voting server to be honest. Formally, we consider the following process:

$$\begin{array}{l} \mathit{Init};\ !\mathit{Server}(c, \mathit{pk}_b, \mathit{gk}_a, \mathit{sk}_c, c)\ \mid\ !\mathit{Corr} \\ \quad \mid\ \mathit{Voter}(c, \mathit{pwd}_a, v_1, v_2, \mathit{src}(\mathit{gk}_a, \mathit{id}_a, v_1), \\ \qquad\qquad \mathit{src}(\mathit{gk}_a, \mathit{id}_a, v_2), \mathit{cc}(\mathit{id}_a), \mathit{sfc}(\mathit{gk}_a, \mathit{id}_a)) \end{array}$$

where *Init* is an initialization process: it broadcasts $\mathit{id}_a = \delta_{Id}(\mathit{pwd}_a)$ and $\mathit{pk}_e(\mathit{id}_a)$ on some public channel c, generates the elections keys sk_b, gk_a and sk_c, and publishes $\mathit{pk}_b = \mathit{pub}_e(\mathit{sk}_b)$, $\mathit{pub}_s(\mathit{sk}_c)$, and sk_b. The process $!\mathit{Corr}$ models an arbitrary number of dishonest voters: each generates a password pwd_i for some voter i and broadcasts all the corresponding (public and private) data.

Note that we do not need to include the tally process (or Voting Device) since it is assumed to be dishonest. Instead, we simply provide the election key (sk_b) to the adversary: cast-as-intended and recorded-as-cast properties are guaranteed even if the decryption key is lost.

5.2. Privacy

Intuitively, a voting protocol guarantees *ballot privacy* if an attacker cannot learn any information about how a voter voted. In symbolic models, this is typically formalized as follows [18], [27]:

$$V_A(0) \mid V_B(1)\ \approx\ V_A(1) \mid V_B(0)$$

An attacker should not be able to distinguish the case where Alice votes 0 and Bob votes 1 from the case where the votes are swapped.

We show that the Neuchâtel's protocol ensures ballot privacy, even if the voting server and all but two voters (and their voting devices) are corrupted. Formally, we consider a setup process I similar to *Init*, but with two honest voters A, B with passwords $\mathit{pwd}_A, \mathit{pwd}_B$, and a corrupted *Server* process that leaks its private data $\mathit{gk}_a, \mathit{sk}_c$. The election key sk_b (generated during the setup and distributed to election authorities) is assumed to be secret. We call $V_A(a, b)$ the process for voter A that votes for a and b.

$$\begin{array}{l} V_A(a, b)\ := \mathit{Device}(c_A, c, \mathit{pk}_b) \\ \quad \mid\ \mathit{Voter}(c_A, \mathit{pwd}_A, a, b, \mathit{src}(\mathit{gk}_a, \mathit{id}_A, a), \\ \qquad\qquad \mathit{src}(\mathit{gk}_a, \mathit{id}_A, b), \mathit{cc}(\mathit{id}_A), \mathit{sfc}(\mathit{gk}_a, \mathit{id}_A)) \end{array}$$

Then we wish to prove (in ProVerif) that:

$$\begin{array}{rl} & I \mid V_A(a, b) \mid V_B(c, d) \mid !\mathit{Corr} \mid T \\ \approx & I \mid V_A(c, d) \mid V_B(a, b) \mid !\mathit{Corr} \mid T \end{array} \tag{5}$$

$$\begin{array}{l} \text{with}\ \ T\ := \mathit{TallyH}(c, \mathit{sk}_b, \mathit{pk}_c, \mathit{id}_A, \mathit{id}_B) \\ \qquad\qquad \mid !\mathit{TallyD}(c, \mathit{sk}_b, \mathit{pk}_c, \mathit{id}_A, \mathit{id}_B) \end{array}$$

ProVerif cannot prove arbitrary equivalences. Instead, it proves diff-equivalence of pairs of processes that only differ in treatment of terms, which is a *stronger* notion of equivalence [8]. More formally, ProVerif considers bi-processes P that may contain bi-terms $\mathit{choice}(t_1, t_2)$ instead of pure terms. Then ProVerif proves equivalence of $\mathit{proj}_1(P)$ and $\mathit{proj}_2(P)$ where $\mathit{proj}_i(P)$ is obtained from P by replacing any occurrence of a bi-term $\mathit{choice}(t_1, t_2)$ by t_i. We refer the reader to [8] for a detailed and formal definition of bi-processes and diff-equivalence.

So proving the equivalence $V_A(0) \mid V_B(1) \approx V_A(1) \mid V_B(0)$ amounts to considering the process

$$V_A(\mathit{choice}(0, 1)) \mid V_B(\mathit{choice}(1, 0))$$

However, applying directly this transformation to the equivalence (5) yields a process that ProVerif cannot prove. Instead, we need to further transform it by also swapping the output of the tally. This is a usual technique, as devised e.g. in [9]. Formally, we replace the $\mathsf{out}(\mathtt{mix}, e_a) \mid \mathsf{out}(\mathtt{mix}, e_b)$ part in the *TallyH* process by:

$$\mathsf{out}(\mathtt{mix}, \mathit{choice}(e_a, e_b)) \mid \mathsf{out}(\mathtt{mix}, \mathit{choice}(e_b, e_a))$$

Our mixing here is very weak since it only swaps the two honest votes. Proving trace equivalence for this weak mixing actually enforces trace equivalence for a more general mixing: any execution trace from the left must be matched with a trace where the two honest votes are swapped, which ensures *a fortiori* that it can be matched with a trace from a more general mixer.

	Voting Device	Server	Tally
Cast-as-Intended	D	H	D
Recorded-as-Cast	D	H	D
Ballot Privacy	H	D	H

TABLE 1. PROPERTIES AND TRUST ASSUMPTIONS. D STANDS FOR DISHONEST WHILE H STANDS FOR HONEST.

Number of option's choices	1	2	3	4	5
Cast-as-Intended	< 1s	< 1s	2s	8m	time out > 48h
Recorded-as-Cast	< 1s	< 1s	3s	20m	time out > 48h
Ballot Privacy	14s	49m	time out > 48h	time out > 48h	time out > 48h

TABLE 2. SECURITY ANALYSIS IN PROVERIF.

Given that the resulting processes are equivalent (since $P \mid Q \approx Q \mid P$), we deduce the desired equivalence (Property 5). Note that since the server is assumed to be dishonest, we do not need to model it, hence we do not need to model the no-revote policy.

6. Results and lessons learned

Previous symbol models of electronic voting protocols [3], [16], [17], [18] consider a simple scenario where the voter selects one candidate among finitely many options. In this study and for the sake of clarity, we have presented the Neuchâtel's protocol for the particular case of an election where $k = 2$ options among n options need to be selected. In our ProVerif model we have considered an arbitrary number of options n, an arbitrary number of voters m, and several values for the number of selections k. To be able to cope with an arbitrary number of selections, we would need to handle lists of arbitrary size (representing the selection of a voter). While there are some preliminary results for protocols with lists [2], [11], [31], none of them can be applied to our symbolic model for the Neuchâtel protocol since they do not cover equivalence properties and do not offer any tool support. This is why we consider several fixed values for k.

The security properties together with the corresponding trust assumptions are summarized in Table 1, while the experiments are presented in Table 2.

6.1. Results

We run ProVerif version 1.94 on a Xeon E5-2687W v3 @ 3.10GHz. We were able to analyse cast-as-intended and individual verifiability up to $k = 4$ and ballot privacy up to $k = 2$. The detailed analysis times are reported in Table 2. The models in ProVerif can be found in [14]. As explained in Section 5.2, ballot privacy is expressed as an equivalence property of the form:

$$V_A(a,b) \mid V_B(c,d) \;\approx\; V_A(c,d) \mid V_B(a,b)$$

for $k = 2$, where a, b, c, d are constants. This implicitly means that A and B vote for distinct options. So in the case $k = 2$ we further prove privacy when the two honest voters were respectively voting (a,b), (b,c), or (a,b), (c,b), or (a,b), (b,a), to check that no attack appears when A and B share one or two options. ProVerif proves these cases in exactly the same time than the case where the four options are pairwise distinct.

6.2. Lessons learned

Our analysis mostly confirms that the Neuchâtel protocol is secure w.r.t. both privacy and verifiability properties. However, while modeling the protocol we discovered that flaws may occur in case of small but realistic deviations of the protocol. We reported these subtleties to the the company that designed the system who confirmed to be aware of them and that they have been taken care of for the actual implementation.

Exactly k choices. In case of an election where voters select k options out of n, with $k \geq 2$, voters (and authorities) should be aware (and properly instructed) that voters should select *exactly* k options (and not less). Otherwise, a dishonest voting device may use the remaining "unused" choices for other unexpected voting options (and discard their return codes).

No duplicate. Still in case of an election where voters select k options out of n, voters should not be offered the possibility to vote twice for the same option (for example, the election rules may allow voters to choose twice the same candidate). Indeed, the protocol would then be vulnerable to an attack where the intruder uses the duplicated choice (say Alice votes twice for a, that is, she votes a, a) to make her vote for a, b (and manually duplicate the return code corresponding to a, to make Alice happy).

Blank vote. It becomes particularly tricky for elections that allow voters to abstain (that is, vote "blank"). In that case, k different blank voting options must be provided to voters. Those blank voting options shall have different individual return codes, and voters shall be advised that they need to check a return code for each blank option. In other words, if Alice wishes to abstain in an election where voters can select k options out of n, then she must receive (and check) k distinct return codes, corresponding to k blank voting options. Of course, in case the election includes several questions (e.g., several sub-elections) then these blank options have to be specific to each question. This may be difficult to understand for voters.

Synchronization. As pointed before, the protocol is no longer secure w.r.t. the cast-as-intended property as soon as the voting server answers two different requests from the same voters. (Note that this is explicitly forbidden by the Neuchâtel

protocol since revoting is not allowed). Therefore, the voting server must implement some form of thread synchronization to guarantee that two different ballots will never be accepted for the same id, even if none of them has yet been confirmed by the voter. This should be enforced even when voting servers are duplicated for efficiency reasons. In particular, the use of tables as described in [20] is insufficient and further requires a proper lock mechanism.

7. Discussion and conclusion

We provide an automated proof of an e-voting protocol in use for politically-binding elections in the Swiss cantons of Neuchâtel and Fribourg. Our analysis confirms the security of the protocol: it ensures cast-as-intended and recorded-as-cast against a dishonest voting device (assuming an honest voting server) and it guarantees privacy against a dishonest voting server (assuming an honest voting device). The protocol does not aim at universal verifiability (the fact that anyone can check that the result corresponds to the cast ballots) nor coercion-resistance. Previous analysis of other e-voting protocols (eg [3], [16], [17], [18]) left several parts of the protocol undefined because they had not been deployed for actual elections. Our ProVerif model covers the authentication phase, voters' cryptographic keys derivation from passwords (as voters cannot be asked to copy long strings), no revote, as well as elections where voters may select several options. One particular challenging aspect of the Neuchâtel protocol is the fact that it forbids re-voting since it would be insecure otherwise. Due to ProVerif's over-approximations, we had to propose several ideas in order to still obtain automated proofs of all the desired properties.

As future work, we plan to explore how we could extend ProVerif in order to cope with protocols where some events happen only once. This is the case as soon as a protocol embeds some lock mechanisms or uses of counters. Similarly, our encoding works for correspondence properties only. To cover equivalence properties such as privacy, we would need to first modify the ProVerif tool, in order to be able to specify that executions have to be in equivalence, *except* when something deemed impossible has occurred.

At the time we started this case study, ProVerif was the only tool that supported flexible equational theories as well as correspondence and equivalence properties, for an unbounded number of sessions. Now, the Tamarin tool [19] also covers a wide range of primitives and the same class of properties. It would be interesting to translate our model into Tamarin in order to compare the two tools and understand which one is the best suited in the context of voting. In particular, Tamarin does not suffer from the same approximation issues than ProVerif w.r.t. global states but would probably require some user interactions.

It is worth noticing that our analysis applies to the *specification* of the Neuchâtel protocol, not its implementation. For the voting device, this is not necessarily an issue, at least for the verifiability properties: cast-as-intended and recorded-as-cast are guaranteed even if the voting device runs an arbitrary code. However, the voting server is assumed to be honest and both privacy and verifiability rely on the fact that the voting server behaves as specified. Checking whether the code matches its specification is clearly beyond the scope of this study. One direction would be to re-implement the voting server (at least its core part) in some language suitable for proofs, like F^*, as it has been done for TLS [5].

Acknowledgments

This work has been partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement No 645865-SPOOC).

References

[1] B. Adida. Helios: Web-based open-audit voting. In P. C. van Oorschot, editor, *17th USENIX Security Symposium*, pages 335–348. USENIX Association, 2008.

[2] M. Arnaud, V. Cortier, and S. Delaune. Deciding security for protocols with recursive tests. In *23rd International Conference on Automated Deduction (CADE'11)*, Lecture Notes in Artificial Intelligence, pages 49–63, Wrocław, Poland, 2011. Springer.

[3] M. Backes, C. Hritcu, and M. Maffe. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *IEEE 21st Computer Security Foundations Symposium (CSF'08)*, pages 195–209, 2008.

[4] D. Bernhard, O. Pereira, and B. Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In *Advances in Cryptology (ASIACRYPT'12)*, volume 7658 of *Lecture Notes in Computer Science*, pages 626–643. Springer, 2012.

[5] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security & Privacy (S&P'13)*, 2013.

[6] B. Blanchet. Automatic verification of security protocols in the symbolic model: The verifier ProVerif. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 54–87. Springer, 2014. prosecco.gforge.inria.fr/personal/bblanche/proverif.

[7] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, Oct. 2016.

[8] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340, 2005.

[9] B. Blanchet and B. Smyth. Automated reasoning for equivalences in the applied pi calculus with barriers. In *IEEE 29th Computer Security Foundations Symposium (CSF'16)*, 2016.

[10] I. Brightwell, J. Cucurull, D. Galindo, and S. Guasch. An overview of the ivote 2015 voting system. Avaialable through https://www.elections.nsw.gov.au, 2015.

[11] N. Chridi, M. Turuani, and M. Rusinowitch. Decidable analysis for a class of cryptographic group protocols with unbounded lists. In *IEEE 22nd Computer Security Foundations Symposium (CSF'09)*, pages 277–289, 2009.

[12] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. In *2008 IEEE Symposium on Security and Privacy (S&P 2008)*, pages 354–368. IEEE Computer Society, 2008.

[13] V. Cortier, D. Galindo, S. Glondu, and M. Izabachene. Election Verifiability for Helios under Weaker Trust Assumptions. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*, LNCS. Springer, 2014.

[14] V. Cortier, D. Galindo, and M. Turuani. ProVerif model for the Neuchâtel e-voting protocol. http://homepages.loria.fr/MTuruani/Neuchatel-Analysis/. Experiments.

[15] V. Cortier, B. Schmidt, C. C. Dragan, P.-Y. Strub, F. Dupressoir, and B. Warinschi. Machine-checked proofs of privacy for electronic voting protocols. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'17)*, pages 993—-1008. IEEE Computer Society Press, 2017.

[16] V. Cortier and B. Smyth. Attacking and fixing Helios: An analysis of ballot secrecy. *Journal of Computer Security*, 21(1):89–148, 2013.

[17] V. Cortier and C. Wiedling. A formal analysis of the Norwegian e-voting protocol. *Journal of Computer Security*, 25(15777):21–57, 2017.

[18] S. Delaune, S. Kremer, and M. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[19] J. Dreier, C. Duménil, S. Kremer, and R. Sasse. Beyond subterm-convergent equational theories in automated verification of stateful protocols. In *6th International Conference on Principles of Security and Trust (POST'17)*, volume 10204 of *Lecture Notes in Computer Science*, pages 117–140. Springer, 2017.

[20] D. Galindo, S. Guasch, and J. Puiggali. 2015 Neuchâtel's Cast-as-Intended Verification Mechanism. In *5th International Conference, (VoteID 2015)*, pages 3–18, 2015.

[21] K. Gjøsteen. The Norwegian internet voting protocol. Cryptology ePrint Archive, Report 2013/473, 2013. http://eprint.iacr.org/2013/473.

[22] S. Heiberg, P. Laud, and J. Willemson. The application of i-voting for Estonian parliamentary elections of 2011. In *VoteID 2011*, volume 7187 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2012.

[23] S. Heiberg, T. Martens, P. Vinkel, and J. Willemson. Improving the verifiability of the Estonian internet voting scheme. In *E-Vote-ID 2016*, volume 10141 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2017.

[24] A. Juels, D. Catalano, and M. Jakobsson. Coercion-resistant electronic elections. In *Proceedings of Workshop on Privacy in the Eletronic Society (WPES'05)*, pages 61–70. ACM Press, 2005.

[25] A. Kiayias, T. Zacharias, and B. Zhang. DEMOS-2: scalable E2E verifiable elections without random oracles. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 352–363. ACM, 2015.

[26] A. Kiayias, T. Zacharias, and B. Zhang. Ceremonies for end-to-end verifiable elections. In *Public-Key Cryptography - PKC 2017*, volume 10175, pages 305–334. Springer, 2017.

[27] S. Kremer and M. Ryan. Analysis of an electronic voting protocol in the applied pi-calculus. In *14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 186–200, Edinburgh, U.K., 2005. Springer.

[28] R. Küsters, J. Müller, E. Scapin, and T. Truderung. sElect: A lightweight verifiable remote voting system. In *IEEE 29th Computer Security Foundations Symposium (CSF'16)*, pages 341–354, 2016.

[29] R. Küsters, T. Truderung, and A. Vogt. Clash Attacks on the Verifiability of E-Voting Systems. In *33rd IEEE Symposium on Security and Privacy (S&P'12)*, pages 395–409. IEEE Computer Society, 2012.

[30] G. Lowe. A hierarchy of authentication specifications. In *10th Computer Security Foundations Workshop (CSFW'97)*. IEEE Computer Society Press, 1997.

[31] M. Paiola and B. Blanchet. Verification of security protocols with lists: From length one to unbounded length. *Journal of Computer Security (JCS)*, pages 781–816, 2013.

[32] T. Pinault and P. Courtade. E-voting at expatriates' mps elections in France. In *5th International Conference on Electronic Voting 2012, (EVOTE'12)*, volume 205 of *LNI*, pages 189–195. GI, 2012.

[33] Swiss. Post. Swiss post's e-voting solution. https://www.post.ch/en/business/a-z-of-subjects/industry-solutions/swiss-post-e-voting.

[34] Scytl. *Swiss On-line Voting Protocol*, 2016. Manuscript.

[35] D. Springall, T. Finkenauer, Z. Durumeric, J. Kitcat, H. Hursti, M. MacAlpine, and J. A. Halderman. Security Analysis of the Estonian Internet Voting System. In *ACM Conference on Computer and Communications Security (CCS'14)*, pages 703–715. ACM, 2014.

[36] Swiss Post. *Individual Verifiability: Swiss Post E-Voting Protocol Explained*, November 2017. https://www.post.ch/-/media/post/evoting/dokumente/swiss-post-online-voting-protocol-explained.pdf.

[37] S. Wolchok, E. Wustrow, D. Isabel, and J. A. Halderman. Attacking the Washington, D.C. Internet voting system. In *16th Intl. Conference on Financial Cryptography and Data Security (FC'12)*, 2012.