Election Verifiability with ProVerif
Vincent Cheval, Véronique Cortier, Alexandre Debant

To cite this version:
Vincent Cheval, Véronique Cortier, Alexandre Debant. Election Verifiability with ProVerif. CSF 2023 - 36th IEEE Computer Security Foundations Symposium, Jul 2023, Dubrovnik, Croatia. hal-04177268v2

HAL Id: hal-04177268
https://inria.hal.science/hal-04177268v2
Submitted on 5 Dec 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
Abstract—Electronic voting systems should guarantee (at least) vote privacy and verifiability. Formally proving these two properties is challenging. Indeed, vote privacy is typically expressed as an equivalence property, hard to analyze for automatic tools, while verifiability requires to count the number of votes, to guarantee that all honest votes are properly tallied.

We provide a full characterization of E2E-verifiability in terms of two simple properties, that are shown to be both sufficient and necessary. In contrast, previous approaches proposed sufficient conditions only. These two properties can easily be expressed in a formal tool like ProVerif but remain hard to prove automatically. Therefore, we provide a generic election framework, together with a library of lemmas, for the (automatic) proof of E2E-verifiability. We successfully apply our framework to several protocols of the literature that include two complex, industrial-scale voting protocols, namely Swiss Post and CHVote, designed for the Swiss context.

1. Introduction

It is now a standard practice to analyze security protocols with formal methods during their design for example with tools like ProVerif [9] or Tamarin [25]. It is the case of major protocols like TLS1.3 [8] or Signal [13]. Voting protocols form a particular class of protocols that combines several difficulties: complex cryptographic primitives that yield complex equational theories, a large number of actors (e.g. server, registrar, voting client, tally authorities) and therefore a large number of threat scenarios, and non standard security properties. Thus voting protocols push formal tools at their limits. However, it remains crucial to conduct a formal analysis to avoid undetected flaws and it is actually even a legal requirement in Switzerland [14].

Voting systems should guarantee two main security properties: vote secrecy (no one should know how I voted) and verifiability (my vote should be properly counted). Depending on the election context, other properties may be desired such as coercion-resistance or accountability. Vote secrecy is expressed as an equivalence property: an attacker should not distinguish between two scenarios where voters

This work was partly supported by the following grants: ANR Chair AI ASAP (ANR-20-CHEA-0024) and ANR France 2030 project SVP (ANR-22-PECY-0006).
ingly, these sufficient conditions are close to the notion of individual, universal, eligibility verifiability, and no-clash. These approaches however suffer from several drawbacks:

- These sufficient conditions may be too strong. For example, in [5], they found that Belenios does not satisfy one of their properties, with no corresponding attack against E2E-verifiability.
- These conditions are specific to some architecture and assume e.g. a bulletin board, registrars, and voting credentials. [15] even needs to define two sets of sufficient conditions, depending on whether the Registrar or the Voting Server is honest.
- These conditions are stated on ballots “about to be counted”. The tally part is typically excluded from the models and hence not verified.

Our contributions. Our first contribution is to characterize E2E-verifiability with two simple injective properties. The first one says that any honest verified vote should (injectively) be counted in the result of the election.

\[ \text{inj-verified}(id, v) \Rightarrow \text{inj-counted}(v) \]

The other property controls the other direction and makes sure that any counted vote come from a legitimate voter (honest or dishonest). We show that these two properties not only suffice to prove E2E-verifiability but are also necessary, yielding a complete characterization of E2E-verifiability. This result holds for arbitrary protocols and arbitrary cryptographic primitives. The only assumption is that the result of the election is the (multi)set of votes, which covers most elections. Interestingly, these two properties are directly expressible in ProVerif, hence we chose ProVerif to build our framework, although a similar approach could probably be developed for Tamarin, after some preliminary work like the one performed for handling global states [23].

However, our properties only hold once the tally is over. Indeed, even if a voter successfully verifies during the voting phase, their vote is (obviously) not yet counted. Moreover, the model needs to reflect the fact that all valid ballots are eventually processed. Most existing models (e.g [5], [6], [17], [21]) are instead very permissive and let the tally stop at any time. This, in passing, show that they only check some partial properties w.r.t. verifiability. Instead, we need here to faithfully model the fact that ballots are typically stored in a stack and the tally is over when all ballots have been processed, yielding an empty stack. Combined with the verification of injective properties, this may cause all kind of issues to verification tools, as ProVerif experts may have already guessed.

To overcome these issues, we use the whole machinery recently introduced in ProVerif [10]: use of counters, lemmas, axioms, and restrictions. Our second contribution is a generic and re-usable framework for proving E2E-verifiability. Namely, we propose a reference model for a mini-voting scheme, with a library of corresponding lemmas and axioms. This generic models needs then to be populated with the behavior of the considered protocol. For example, the setup process needs to define the public and private data generated for each voter, possibly introducing new actors. Similarly, the voting process defines the voter’s behavior and returns in the end the ballot that should be cast. We applied our framework to Helios [4] and Belenios [18], studied in previous approaches aiming at proving verifiability [5], [6], [13]. We also cover two complex, industrial-scale protocols, namely CHVote [22] and SwissPost [1]. These two protocols involve complex primitives (e.g. oblivious transfer for CHVote) with ad-hoc encoding as equational theories, and their architecture differs a lot from Helios and Belenios. In particular, the server is split into 4 independent components and there is no public bulletin board. Voters do not check the presence of their ballots but instead receive return codes from their voting device, checked against a code sheet. It is important to note that we did not write any of these ProVerif models. For our 4 case studies, we filled our framework (designed once and for all) with the existing models. We got them from public repositories presenting security analyses of these protocols and just fit them with a few adjustments. In particular, because our two properties cover E2E-verifiability, we needed to more faithfully model the registration and the tally phase. ProVerif was able to automatically prove our two injective properties, hence E2E-verifiability, thanks to our main theorem. No manual work outside of the framework was required except for one variant of Belenios that involves a chain of hashes. Due to the intrinsic complexity of this protocol, a couple of customized lemmas have been added. In all cases, the ProVerif verification was fast (around 1 minute or below).

2. ProVerif

We provide an informal introduction of the syntax and the semantics of the ProVerif tool. A comprehensive description is available in [9]. An excerpt of the syntax of ProVerif is displayed in Figure 1.

2.1. Messages

ProVerif relies on a symbolic model in which messages are represented by terms, formed of symbolic functions and atomic data. Atomic data represent nonces, keys, or any constant. Function symbols model cryptographic primitives such as encryption, signature, or hash function. For example, the encryption of the message \( m \) with the public key \( \text{pk}(k) \) and the random \( r \) can be modeled with the term \( \text{aenc}(\text{pk}(k), r, m) \) where \( \text{aenc}(\cdot, \cdot, \cdot) \) is a symbol of function modeling any encryption scheme. Relations between terms are modeled by rewriting rules. For instance, to model that a participant is able to decrypt a message as soon as he knows the corresponding private key, we define a destructor symbol \( \text{adec}(\cdot, \cdot) \) to represent the decryption algorithm and the rewriting rule: \( \text{adec}(k, \text{aenc}(\text{pk}(k), r, m)) \rightarrow m \). Similarly, destructor symbols and rewriting rules can be defined to model the verification algorithms of signatures or ZK proofs.

2.2. Protocols

Protocols are modeled through a process algebra similar to the applied pi-calculus [3]. Fresh names unknown to
M, N, M_1, \ldots, M_k \ ::= \ \text{terms} \n\ \ \ \ \ \ x \ | \ n \ | \ f(M_1, \ldots, M_k) \\
D ::= \ \text{expressions} \\
M \ | \ h(D_1, \ldots, D_k) \\
\phi ::= \ \text{formula} \\
M = N \ | \ \phi_1 \land \phi_2 \ | \ \phi_1 \lor \phi_2 \ | \ \neg \phi \\
P, Q ::= \ \text{processes} \\
0 \ | \ \text{out}(c, M); P \ | \ \text{in}(c, x); P \ | \ P ; Q \ | \ P ! Q \\
\text{new } n \ | \ P \ | \ \text{restriction} \\
\text{let } x = D \ \text{in } P \ | \ \text{assignment} \\
\text{if } \phi \ \text{then } P \ \text{else } Q \ | \ \text{conditional} \\
\text{insert } M; P \ | \ \text{table insertion} \\
\text{get } t(x_1, \ldots, x_k) \ \text{such that } \phi \ \text{in } P \ | \ P ! Q \ | \ \text{table lookup} \\
\text{event } (E(M_1, \ldots, M_k)); P \ | \ P \ \text{event} \\
x, x_1, \ldots, x_k \ \text{are variables}, n \ \text{is an atomic data}, f \ \text{is a constructor symbol}, h \ \text{is a constructor or destructor symbol}, t \ \text{is a table name}, \ \text{and } E \ \text{is an event symbol.}

Figure 1: Syntax of ProVerif (excerpt).

the attacker are generated with the command new n. Tests can be evaluated using conditional declarations of the form let x = v in P: the process P in which x is replaced by v is executed if v reduces to a term without destructor symbols when applying rewriting rules. Communications are modeled using the usual in(c, x) and out(c', u) commands, modeling respectively, an input on channel c and the output of message u on channel c'. It is important to note that an input can be filled with any message known by the attacker when c is public. Agent states are defined with table in which processes can insert elements using the insert command and then test whether there exists an entry satisfying a formula \( \phi \) using the get command. If so, the process P in which \( x_1, \ldots, x_k \) are instantiated according to the considered table entry is executed, and Q otherwise. Finally, P | Q models that two processes are executed in parallel, and P!Q represents an arbitrary number of processes P executed in parallel. ProVerif also contains a command \text{event} \( (E(u_1, \ldots, u_l)) \) where \( E(\cdot) \) is an event function symbol and \( u_1, \ldots, u_l \) are messages. These events are annotations whose semantics does not interfere with the execution of the protocol.

The operational semantics is defined e.g. in [9]. We note Traces(P) the set of execution traces of the process P, i.e. a sequence \( P \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \) where \( \rightarrow \) denotes a step in the operational semantics. We note \( \rightarrow^{*} \) the transitive closure of \( \rightarrow \).

**Example 1.** We consider the voting protocol Helios [4] as a running example. In this protocol, when a voter wishes to vote for v, they authenticate themselves to the server and submit an encrypted ballot pseudo, aenc(pk_E, r, v) using the public key pk_E of the election and a pseudonym pseudo. The pseudonym is optional in Helios but we consider here a version with pseudonym as it allows to distinguish between the id of the voter and their cryptographic data. Voters can check right before the tally that their ballot belongs to the (public) final bulletin board. The corresponding process, associated to a voter id is as follows.

Helios assumes a secure authenticated channel between the voter and the server. We model this as a channel created during the registration phase and shared between voter and server using a table voting_data. Both can read in this data base to retrieve the corresponding channel. The attacker should be able to read data sent on the authenticated channel (since it is not assumed to be a private channel), and thus all the outputs on the private channel are preceded by an output of the same data on the public channel c_pub.

The second part of the voter process models the fact that the voter may verify that their vote appears on the bulletin board. More precisely, we model that the voter verifies their last ballot only, that is, the one that should be counted. To do so, we use a private channel cell_voter(id) that is read at the beginning of the process (hence consuming the channel since communication are synchronous) and is only released if the voter does not verify, offering them the possibility to vote again. The content of this channel also records how many ballots the voter has cast so far relying on the variable nb_vote which is incremented at the end of each session.

The voting process is annotated with events, whose role will be explained in the next sections.

**2.3. Security properties**

ProVerif allows to verify two kinds of properties: correspondence queries and observational equivalence. In the e-voting context, queries are used to express verifiability while observational equivalence is used to model privacy.

**Correspondence queries.** Given a trace \( tr = tr_1 \ldots tr_n \), we say that tr executes \( \text{event}(E(u_1, \ldots, u_l)) \) at time \( i \in \{1, \ldots, n\} \) if \( tr_i \) identifies a semantics step that executes \( \text{event}(E(u_1, \ldots, u_l)) \). Correspondence queries are formulas of the form:
Then we have:

\[
\bigwedge_{k=1}^p F_k(v_1, \ldots, v_n) \Rightarrow \bigwedge_{i=1}^m \bigvee_{j=1}^{n_i} E_{i,j}(u_{i,j}^{i,0}, \ldots, u_{i,j}^{i,\alpha_i})
\]

Such a query is satisfied by a process \( P \) if for all traces \( tr \in \text{Traces}(P) \), whenever there exists a substitution \( \sigma \) such that for all \( k \in \{1, \ldots, p\} \) the event \( \text{event}(F_k(v_1, \ldots, v_n)) \sigma \) is executed in \( tr \), then for any \( i \), there exist a substitution \( \sigma' \) and \( j \) such that for all \( k \), \( F_k(v_1, \ldots, v_n) \sigma' = F_k(v_1, \ldots, v_n) \sigma \) and \( \text{event}(E_{i,j}(u_{i,j}^{i,0}, \ldots, u_{i,j}^{i,\alpha_i})) \sigma' \) is executed in \( tr \). Queries are extended as expected with test equalities \( v = v' \), dis-equivalences \( v \neq v' \) and predicates \( \text{pred}(v_1, \ldots, v_p) \).

**Injective correspondence queries.** To model finer grained properties, ProVerif allows to define **injective correspondence queries** whose purpose is to establish a one-to-one relationship between events. For sake of simplicity, we define here the semantics of the class of injective queries we need to establish our main result. A comprehensive semantics for injective queries is available in [9].

Injective correspondence queries are formulas of the form:

\[
inj - F_0(v_1, \ldots, v_n) \wedge \bigwedge_{k=1}^p F_k(v_1, \ldots, v_n) \Rightarrow m \bigvee_{i=1}^m \left( inj - E_{i,0}(u_{i,0}^{i,0}, \ldots, u_{i,\alpha_i}^{i,0}) \land \bigwedge_{j=1}^{n_i} E_{i,j}(u_{i,j}^{i,0}, \ldots, u_{i,j}^{i,\alpha_i}) \right)
\]

Given a trace \( tr = tr_1, \ldots, tr_n \), we note \( F(tr) \) the set of indices \( \alpha \) such that \( tr \) executes \( F_0(v_1, \ldots, v_n) \sigma \) at time \( \alpha \) for some \( \sigma \). Similarly, we note \( E(tr) \) the set of indices \( \beta \) such that \( tr \) executes \( E_{i,0}(u_{i,0}^{i,0}, \ldots, u_{i,\alpha_i}^{i,0}) \sigma \) at time \( \beta \) for some \( \sigma \) and \( i \). An injective correspondence query as defined above is satisfied by a process \( P \) if for all trace \( tr \in \text{Traces}(P) \), whenever there exists a substitution \( \sigma \) such that for all \( k \in \{0, \ldots, p\} \) the event \( \text{event}(F_k(v_1, \ldots, v_n)) \sigma \) is executed in \( tr \), then there exists \( i \) and a substitution \( \sigma' \) such that for all \( k \), \( F_k(v_1, \ldots, v_n) \sigma' = F_k(v_1, \ldots, v_n) \sigma \) and for all \( j \), \( \text{event}(E_{i,j}(u_{i,j}^{i,0}, \ldots, u_{i,j}^{i,\alpha_i})) \sigma' \) is executed in \( tr \). Moreover, there exists an injective function \( f : F(tr) \rightarrow E(tr) \) such that if \( F_0(v_1, \ldots, v_n) \sigma \) is executed in \( tr \) at time \( \alpha \) then \( E_{i,0}(u_{i,0}^{i,0}, \ldots, u_{i,\alpha_i}^{i,0}) \sigma' \) is executed in \( tr \) at time \( f(\alpha) \).

Informally, a protocol \( P \) satisfies an injective query if it satisfies the non-injective counterpart, and if there is a one-to-one mapping between injective events used to satisfy the query.

**Example 2.** Let \( E_1(\cdot), E_2(\cdot) \) and \( F(\cdot) \) be three event function symbols of arity 1. Let \( tr_1, tr_2, \) and \( tr_3 \) be three traces such that \( tr_1 = \text{event}(E_1(n)).\text{event}(F(n)).\text{event}(E_2(m)).\text{event}(F(m)) \), \( tr_2 = \text{event}(E_1(n)).\text{event}(E_2(n)).\text{event}(F(n)) \) and \( tr_3 = \text{event}(E_1(n)).\text{event}(F(n)) \). If we consider the injective query

\[
\rho = inj - F(x) \Rightarrow inj - E_1(x) \lor inj - E_2(x)
\]

then we have:

- \( tr_1 \) satisfies \( \rho \). We can define the injective function \( f \) such that \( f(2) = 1 \) and \( f(4) = 3 \);
- \( tr_2 \) satisfies \( \rho \). We can define the injective function \( f_1 \) such that \( f_1(3) = 2 \) or the injective function \( f_2 \) such that \( f_2(3) = 1 \). We note that, contrary to item 1, the functions \( f_1 \) and \( f_2 \) are not surjective over \( E(tr_2) = \{1, 2\} \);
- \( tr_3 \) does not satisfy \( \rho \). There is no injective function \( f : \{2, 3\} \rightarrow \{1\} \).

**Temporal queries.** Facts in queries may be annotated with temporal variables, yielding a temporal fact \( F@i \). Given a trace \( tr = tr_1, \ldots, tr_n \), the temporal fact \( F@i \) is executed in \( tr \) w.r.t. a substitution \( \sigma \) if \( tr(\alpha) = F \sigma \). This allows e.g. to specify when an event should occur before another one.

**Example 3.** The following query states that the event verified should occur at most once per voter.

\[
\text{verified}(id, x)@i \land \text{verified}(id, y)@j \Rightarrow i = j
\]

3. Definition of E2E verifiability

E2E verifiability intuitively guarantees that the result of the election reflects the votes of the voters. We state this property using events that record when a voter votes or verifies and when a vote is counted.

3.1. Events

A voting process is a process that contains several special events:

- \( \text{verified}(id, v) \) represents the fact that voter id believe they have voted for \( v \). In any voting protocol, the voter knows for who they have tried to vote for.
- \( \text{verified}(id, v) \) represents the fact that voter id has voted for \( v \) and successfully performed the verification test, that is optional in many cases. The event should be executed at most once for each voter, typically for the final vote when revoting is allowed.
- \( \text{counted}(v) \) tells that a vote has been counted for \( v \). We consider elections where the multiset of votes is displayed at the end of the election.
- \( \text{finish} \) tells that the tally procedure is over, no more counted event will be emitted.

Moreover, we assume that any voter id occurs exactly once in one of these three events:

- \( \text{hv}(id) \) means that the honest voter id will vote and verify their vote.
- \( \text{hvn}(id) \) means that the honest voter id will vote but not necessarily verify their vote.
- \( \text{corrupt}(id) \) records that voter id is corrupted and the attacker typically has access to their credential.

Our characterization of E2E-verifiability will hold independently on how these events are placed but of course, E2E verifiability will only be meaningful when these events are properly placed (like in any model). We illustrate this on our running example.

**Example 4.** Continuing Example 7 we first note that in the voter process, the events \( \text{voted}(id, v) \) and \( \text{verified}(id, v) \)
occur as expected. The remaining processes representing Helios are displayed in Figure 2.

The process Voter_registration(id) creates an authenticated channel and a public pseudonym for any voter and assigns the voter to corrupt, hv, or hnv, at the attacker’s will, with corresponding events. As expected, the channel is given to the adversary when the voter is corrupted. In order to model that voters in hv will verify, we restrict the analysis to traces satisfying the property:

\[ \text{finish} \land \text{hv}(id) \Rightarrow \text{verified}(id, x). \]

This can easily be modeled in ProVerif, using a new feature, called restriction \([10]\), that allows to discard traces that do not satisfy the property stated in the restriction.

The public bulletin board is modelled by private channels cell_BB(id). There is one per eligible voter. This allows each voter to easily find their vote on the board. But we make sure to let the adversary reads the board and also writes on it when the server is corrupted.

The Voting Server is modelled as expected. It receives ballots and publishes them on the public board, that is on the channel cell_BB(id) corresponding to the voter.

The content of the channel cell_BB(id) is initialized at empty at registration and may then change several times when the voter revotes. At some point, the voter stops and starts counting ballots, skipping voters that did not vote.

The process \( \text{Close_voting_booth}() \) that transfers the content of cell_BB(id) to a new channel cell_tally(id). This process does not release cell_BB(id) hence it can no longer be used.

Finally, the process Tally(skE) decrypts all received ballots, skipping id that did not vote.

How to place the event finish? Placing the finish event is a challenging task. Indeed, this event must ensure that all the ballots have been counted. In this first example, we describe a solution when considering a bounded number of voters. A more general approach for an unbounded number is presented in Section 4. When the \( i \)-th ballot is processed by the tally process, the constant OK is sent on private channel cend(i). We then consider the process Finish that gathers all these confirmations (a finite predefined number) before executing finish. The process Finish is an example for two voters.

3.2. Definition

Given two multisets \( S_1 \) and \( S_2 \) we denote \( S_1 \subseteq_m S_2 \) their inclusion and \( S_1 \cup_m S_2 \) their union. The subscript \( m \) may be omitted when it is clear from the context that the relation is applied to multisets.

Given a trace \( tr \) of a voting process \( P \), we define the set of involved voters as \( \text{Voters} = \text{HV} \cup \text{HNV} \cup D \) where:

- \( D \) is the set of corrupted voters, i.e., \( D = \{ id \mid \text{corrupt}(id) \in tr \} \) is the set of dishonest voters.
- \( \text{HV} \) is the set of voters who verify, i.e., \( \text{HV} = \{ id \mid \text{hv}(id) \in tr \land \text{verified}(id, v) \in tr \} \). We note \( V_{hv} \) the corresponding multiset of vote intentions:

\[ V_{hv} = \{ v \mid \text{verified}(id, v) \in tr \land id \in \text{HV} \} \]

- \( \text{HNV} \) is the set of voters who do not necessarily verify, i.e., \( \text{HNV} = \{ id \mid \text{hnv}(id) \in tr \} \). We note \( V_{hnv} \) the corresponding multiset of vote intentions:

\[ V_{hnv} = \{ v \mid \text{voted}(id, v) \in tr \land id \in \text{HNV} \} \]

We note that for honest voters who do not verify, we consider all the votes that the voter has submitted. We will see that a system is secure as soon as if a ballot is counted for these voters then it corresponds to one of the submitted votes (not necessarily the last one). This definition is consistent with \([6]\) and \([15], [20] \) when no revote is permitted. In order to consider at most one vote per voter, we consider all the votes events associated to a voter.

\[ V_{id}^\text{hv} = \{ (id, v) \mid \text{voted}(id, v) \in tr \land id \in \text{HNV} \} \]

We say that \( V \subseteq_m V_{hnv}^\text{id} \) if \( V \) is a selection of at most one vote per voter, that is, if there exists a multiset \( V_{\text{id}}^\text{hv} \) such that \( V = \{ v \mid (id, v) \in V_{\text{id}}^\text{hv} \} \) and \( V_{\text{id}}^\text{hv} \subseteq_m V_{\text{id}}^\text{nnv} \), and two elements \( (id, v), (id', v') \) of \( V_{\text{id}}^\text{nnv} \) have different ids, i.e., \( id \neq id' \).

The result of the election is the multiset of counted votes:

\[ \text{Result} = \{ v \mid \text{counted}(v) \in tr \} \]

**Definition 1.** A voting process \( P \) satisfies E2E-verifiability if, for any trace \( tr \) of \( P \) that contains the event finish, there exist \( V_{hnv}^\prime \subseteq_m V_{hnv}^\text{id} \) and \( V_{d} \) such that \( |V_{d}| \leq |D| \) and

\[ \text{Result} = V_{hv} \cup_m V_{hnv}^\prime \cup_m V_{d} \]

For sake of understandability, this definition deserves some comments:

1) As stated, E2E verifiability provides strong guarantees to honest voters who decide to abstain. Indeed, this definition guarantees that an attacker is not able to cast a ballot on behalf of such a voter. Any ballot recorded in the name of an honest voter must have been intentionnally cast by this voter; this is the purpose of the multiset \( V_{hv}^\prime \).

2) When no revote is allowed, we can equivalently write \( V_{hnv}^\prime \subseteq_m V_{hnv}^\text{id} \) and \( V_{hnv}^\prime \subseteq_m V_{hnv} \). In this case, we retrieve the original definition of \([15], [20] \).

3) When revote is allowed and in case the voted event occurs each time a voter votes, the definition is rather weak: not only ballots may be dropped for voters who do not verify but the adversary can actually pick any of the ballots a voter has voted. This corresponds to the definition proposed in \([6]\) and the one actually satisfied in practice. Indeed, in most existing protocols, if Alice does not verify, the adversary may simply let her vote normally until the vote of his choice, and then drop all future ballots.

4) In case voted events occur only when a voter votes for the last time, then E2E-verifiability captures a stronger property where voters who do not verify are guaranteed that their last ballot is either counted or dropped, but no previous ballots can be counted. Interestingly, our results hold independently on how voted events are placed. Hence both properties can be considered using our simpler, equivalent queries.
4. Characterization of E2E verifiability

We can equivalently translate E2E verifiability into two simple injective properties, more amenable to verification.

4.1. Assumptions

The previously mentioned events can actually be placed arbitrarily, this will not impact our main theorem. Of course, E2E-verifiability may not hold when the events are not properly placed, for example if

\[ \text{query 1} \]

is placed before the voter is done checking their ballot. However, the equivalence between E2E-verifiability and the two properties will remain. We only need a few assumptions, that we list here.

- We assume the sets HV, HNV and D to be pairwise disjoint. This should hold in any reasonable model and can be easily checked in ProVerif.
- We assume that whenever verified(id, v) ∈ tr then voted(id, v) ∈ tr. This is easy to ensure and check.
- Finally, the verified event should be executed at most once per voter. Note that the E2E-verifiability property makes sense only in this case anyway. As seen in Example 3, this property can easily be written in ProVerif using a temporal query.

\[ \text{verified}(id, x)@i \land \text{verified}(id, y)@j \Rightarrow i = j \]

A voting process is a process satisfying these 3 properties.

4.2. ProVerif queries

We can characterize E2E-verifiability by two simple injective queries that intuitively check the double inclusion of the two sets Result and \( V_{HV} \cup \cup V_{HNV} \cup \cup V_{D} \).

**query 1**

\[ \text{finish} \land \text{inj}^{-}\text{counted}(x) \Rightarrow \text{inj}^{-}\text{verified}(z, x) \lor \text{inj}^{-}\text{hvn}(z) \land \text{voted}(z, x) \lor \text{inj}^{-}\text{corrupt}(z) \]

1. Note that, as mentioned in Section 5.2.2 in our framework all these protocol assumptions are automatically checked by ProVerif before starting the security analysis.

**query 2**

\[ \text{finish} \land \text{inj}^{-}\text{verified}(z, x) \Rightarrow \text{inj}^{-}\text{counted}(x) \]

Intuitively, **query 1** ensures that any counted vote comes from a legitimate voter, honest or dishonest. Conversely, **query 2** makes sure that any verified vote is counted.

**Theorem 1.** A voting process \( P \) is E2E-verifiable if and only if it satisfies **query 1** and **query 2**.

We prove Theorem 1 by showing that a trace satisfies the E2E-verifiability property if and only if it satisfies **query 1** and **query 2**. Not only this allows us to conclude but interestingly, this shows that our result is independent of the exact execution semantics of a process. In particular, our result is likely to apply in other contexts like the framework of Tamarin [25].

We prove separately the two implications in Lemma 1 and Lemma 2.

**Lemma 1.** If a voting process \( P \) is E2E-verifiable then it satisfies **query 1** and **query 2**.

This direction is rather easy; the definition of E2E-verifiability allows to construct injective functions that satisfy **query 1** and **query 2**. The proof is given in appendix.

Interestingly, this direction was missing (and false) in previous approaches [5], [6], [15] that provided only sufficient conditions for E2E-verifiability, that were too strict in general. As mentioned in introduction, the study of [5] shows that the Belenios protocol does not satisfy one of their conditions, while no attack against verifiability is exhibited. We further comment this false negative in appendix.

**Lemma 2.** If a voting process \( P \) satisfies **query 1** and **query 2** then it is E2E-verifiable.

**Proof sketch.** This direction requires more care. Instead of representing Result as a multisets of votes \( v \), we represent it as a set of pairs \( x = (v, i) \) where \( v \) is a vote and \( i \) is its occurrence index, i.e. \( x \) is the \( i \)-th vote for \( v \) in the trace under study. Thus, from **query 1**, we deduce the existence
5.1. Description of the framework

The framework defines two parts: specific processes and generic processes. The first one consists of the elements that describe the protocol under study. They must be defined for each protocol. The second one is fixed and defines how the specific elements are put together yielding the main model that is finally analyzed. For readability, we write in bold and blue the processes and macros specific to the protocol and in orange the generic processes. An overview of the framework is given in Figure 5.

5.1.1. Specific processes and macros

The framework builds upon 12 processes that describe the protocol under study. We distinguish standard processes and macros that do not contain input/output commands and return a term. These are processes defined with the keyword letfun in the ProVerif syntax.

Setup. In the model, each voter is assigned a natural number, called voter index, denoted \( V_{idx} \). Given an index, a process \texttt{Gen\_Voter\_Ident} describes how to generate a corresponding public identifier. Typically, it may correspond to the voter index itself for simple protocols or another public credential (e.g. a public signing key). A macro \texttt{gen\_voter\_data} returns the initial knowledge (prior to the registration) corresponding to a given voter index. Finally, a process \texttt{Gen\_Voting\_Data} describes how a voter receives their voting data. It takes as input, an election identifier, a voter index, and their corresponding initial knowledge and inserts in a table voting\_data the voting data received by the voter in order to vote. Note that the set of candidates \( V \) is not part of this data and is instead defined by a macro \texttt{is\_valid}(v) which returns \texttt{true} if and only if \( v \in V \).

Voting Phase. A process \texttt{Voting} describes how a voter votes. In general, this process should contain the voted event. Similarly, a process \texttt{Final\_Check} describes what the voter is expected to do after having cast their last ballot. It typically models what a voter should do to verify their vote, and thus contain the verified event.
**Bulletin board.** Two macros \texttt{update\_gbl} and \texttt{update\_ballot} describe how the (public or private) bulletin board evolves. More precisely, the bulletin board maintains a global memory cell \texttt{gbl}(e\_id) as well as one memory cell \texttt{cell}(ident) for each voter identifier \texttt{ident} generated by \texttt{Gen\_Voter\_Ident}. Additionally, we assume a table \texttt{ballot\_box}(e\_id) that contains data (e.g. ballots) cast for publication onto the bulletin board in election \texttt{e\_id}. Each time a valid entry (as defined by the Boolean macro \texttt{is\_valid\_ballot}) of this table is processed, the macro \texttt{update\_gbl} is executed to update the value of the memory cell \texttt{gbl}(e\_id). Moreover, after retrieving a voter identifier \texttt{ident} from the entry using the macro \texttt{get\_ident\_from\_ballot}, the macro \texttt{update\_ballot} updates the memory cell \texttt{cell}(ident).

**Tally.** The process \texttt{Decrypt\_Ballot} describes how ballots are decrypted during the tally phase. For the sake of simplicity, our framework allows to model protocols based on mixnets for tally only.

**Main process.** Finally, the main process \texttt{FinalSystem} defines the scenarios under study. This process builds upon the \texttt{System(e\_id)} process provided by the framework to model an election. For sake of generality, it is possible to define other protocol specific processes to faithfully model the peculiarities of each protocol.

**Example 5.** We illustrate our framework with the He-
The process \texttt{Gen Voter Ident} takes as input the election identifier and a voter index, and outputs a freshly generated pseudonym on a framework specific channel \(\text{res GVI(e_id,v_idx)}\), i.e.
\begin{verbatim}
1 let Gen_Voter_Ident(e_id,v_idx) = 
2 new pseud; out(\text{res GVI(e_id,v_idx)},pseud).
\end{verbatim}

In our toy example, the voter registration is abstracted. We consider that the authenticated channel used by the voter to cast a ballot is part of their initial knowledge. Therefore, the macro \texttt{gen voter data} generates the authenticated channel \(\text{c_auth}\) which is forwarded to the voter by the process \texttt{Gen Voting Data} through the table voting data. The macro also retrieves the pseudonym that was generated by \texttt{Gen Voter Ident} from the table public identifier. Moreover, an event \texttt{Link Idx Ident(e_id,v_idx,pseud)} is executed to store the relationship between the voter index and its corresponding identifier. This event will be useful to help ProVerif proofs as explained in Section 5.2.

Regarding the voting phase, the processes \texttt{Voting} and \texttt{Final Check} roughly correspond to the process \texttt{Voter} described in Example 7. \texttt{Voting} models how the voter forges a ballot, i.e.: 
\begin{verbatim}
1 let gen voter data(e_id,v_idx) = 
2 get public_identifier(=e_id,v_idx,pseud) in 
3 event Link Idx Ident(e_id,v_idx,pseud); 
4 new c_auth; 
5 (pseud,c_auth). 
6 let Gen Voting Data(e_id,v_idx,voter_data) = 
7 insert voting data(e_id,v_idx,voter_data).
\end{verbatim}

5.1.2. Generic processes

In order to put together the protocol specific macros and processes, the framework defines different generic processes. Because the understanding of them is not a requirement to use the framework, we just present an excerpt of these processes.

\textbf{Voter process.} The role of a voter is modeled by a \texttt{Voter} process which mainly builds upon the protocol specific processes \texttt{Voting} and \texttt{Final Check}. The process is very similar to the one presented in Example 7: it receives from the attacker the name of the candidate the voter is going to vote for. Then it executes the \texttt{Voting} process to cast a ballot and, if it is the last vote, executes the \texttt{Final Check} process too. However, the reader can note that these actions are guarded by a communication on the private channel \texttt{cell voter(e_id,v_idx)}: an input is executed before executing the voting process and an output is executed right after if the vote is not the last. Based on the semantics of communications on private channels, this allows to prevent concurrent voter sessions executions. Hence, it allows to count (by increasing the variable \(nb\_vote\)) the number of (re)votes by voters. Concretely, the process is as follows:
\begin{verbatim}
1 let Voter(e_id) = 
2 in[\text{c_pub,v}]; 
3 if is valid(v) then 
4 get voting data(e_id,v_idx,v_data) in 
5 in[\text{cell voter(e_id,v_idx),nb_vote}]; 
6 Voting(e_id,v_idx,nb_vote\+1,v,v_data) | 
7 in[\text{res voting(e_id,v_idx),res_date}]; 
8 in[\text{c_pub, is_last}];
\end{verbatim}
Tally process. In the framework, the tally is closely related to the generation of the voter identifiers. Indeed, in order to guarantee that the finish\((e_id)\) event is executed once all the ballots have been tallied, we proceed as follows: during the setup of the election, the framework counts the number of voter identifiers that are generated. Then, the tally executes the event \(finish(e_id)\) once one (possibly empty) ballot is tallied for each of them. Technically, it relies on the use of a private channel \(cell\_ident(e_id)\) that contains a counter which is incremented each time a new identifier is generated (process \(Pub\_Voter\_Ident\_Generation\)). At some point, this generation of identifiers is stopped (process \(Stop\_Pub\_Voter\_Ident\_Generation\)) and the current value on the channel is transferred to another private channel \(cell\_tally(e_id)\) whose value will be decremented by 1 each time a ballot is tallied. Concretely, these two first processes are as follows:

\[
\begin{align*}
\text{let } & \text{Pub\_Voter\_Ident\_Generation}(e_id) = \\
\text{in} & (cell\_ident(e_id), i); \\
\text{Gen\_Voter\_Ident}(e_id, i+1) | \\
\text{in} & (res\_GVI(e_id, i+1), ident); \\
\text{event} & \text{Public\_identifier}(e_id, i+1, ident); \\
\text{insert} & \text{public\_identifier}(e_id, i+1, ident); \\
\text{out} & (cell\_ident(e_id), i+1) | \\
\text{out} & (cell\_ident(e_id), ident, empty\_ballot).
\end{align*}
\]

\[
\begin{align*}
\text{let } & \text{Stop\_Pub\_Voter\_Ident\_Generation}(e_id) = \\
\text{in} & (cell\_ident(e_id), i); \\
\text{event} & \text{Nb\_identifier}(e_id, i); \\
\text{out} & (cell\_tally(e_id), i).
\end{align*}
\]

We can note that the voter identifier associated to each counter value is recorded with the event \(Public\_identifier(e_id, i+1, ident)\). Finally, the number of generated public identifiers is stored in the event \(Nb\_identifier(e_id, i)\) (see Section 5.2 for usage).

We can now model the tally through the two following processes: \(Init\_Tally\) collects the last ballot for each voter identifier and stores them in specific private channels read by \(Tally\) to compute the tally.

\[
\begin{align*}
\text{let } & \text{Init\_Tally}(e_id) = \\
\text{get} & \text{public\_identifier}(e_id, i, ident) \text{ in} \\
\text{in} & (cell\_tally(e_id), i, ident, x); \\
\text{out} & (cell\_tally(e_id), i, ident, x). \\
\end{align*}
\]

\[
\begin{align*}
\text{let } & \text{Tally}(e_id) = \\
\text{in} & (cell\_tally(e_id), i); \\
\text{event} & \text{Tally\_Read}(e_id, i); \\
\text{if} & i = 0 \text{ then} \text{finish}(e_id) \\
\text{else} & \text{get} \text{public\_identifier}(e_id, i, ident) \text{ in} \\
\text{in} & (cell\_tally\_last\_vote(e_id, i), x); \\
\text{if} & x = \text{empty\_ballot} \text{ then} \text{out} (cell\_tally(e_id), i-1) \\
\text{else} & \text{Decrypt\_Ballot}(e_id, i, ident, x) | \\
\text{in} & \text{res\_decrypt}(e_id, i, v); \\
\text{event} & \text{Counted}(e_id, v); \\
\text{event} & \text{Counted\_Extended}(e_id, v, i, ident); \\
\text{out} & (c\_pub, v); \\
\text{out} & (cell\_tally(e_id), i-1).
\end{align*}
\]

The \(Tally\) process is very similar to the process described in Figure 2. It relies on the cell \(cell\_tally(e_id)\) to count the number of tallied ballots and execute the \(finish(e_id)\) event at the right place. Note that the events \(Tally\_Read(e_id, i)\) and \(Counted\_Extended(e_id, v, i, ident)\) are new annotations that will be useful to help ProVerif proofs. Their usage is commented in the next section.

5.2. Improving ProVerif accuracy

Unsurprisingly, our first experiments have shown that ProVerif was not able to prove the security of even simple protocols when using this complex framework. To overcome this limitation of the tool, we extended our framework with well chosen axioms and lemmas. All of them are protocol independent and can thus be reused to analyze a new protocol without any modification.

5.2.1. GSVerif-like annotations

The first difficulty for ProVerif lies in the manipulation of cells and counters. Indeed, some of ProVerif proof strategies lead to dramatic over-approximations in this context. In 2018, Cheval et al. proposed the tool GSVerif [12] to overcome these limitations. Based on events, it adds formally proven correct axioms to improve ProVerif accuracy.

Unfortunately, our framework is too complex and GSVerif cannot be used to automatically annotate the processes and add the relevant axioms. Therefore, we decided to manually place those events and gather in the ProVerif library gsverif_library.pvl all the GSVerif-like axioms that are being used. Interestingly, because of the complexity of our framework, we needed to extend GSVerif approach by defining new events and related axioms, as described below.

Counter intervals: Let’s consider a cell \((i.e. a channel used as a cell)\) that manipulates an increasing counter, but where steps may be greater than 1, i.e. the counter may increase by \(n > 1\). This may typically happen when manipulating cells containing global data and cells containing data specific to an identifier. In our generic framework, both rely on linked counters: if the counter associated to the global data always increases by 1, the one of the specific identifier may increase by greater steps (it is not updated for each accepted ballot).

In this case, it is important to reflect that intermediate values cannot appear in the cell. To do so, we define a new event \(Counter\_Interval(c, i, j)\) executed each time the counter of the cell \(c\), whose current value is \(i\), is incremented to \(j (j > i)\). We also consider a new event \(Counter\_Interval(c, k)\) executed each time the value \(k\) is read on the cell \(c\). We can then define the following axioms:

\[
\begin{align*}
\text{axiom } & \text{ct=channel}, i, j, k, n: \text{nat}; \\
\text{event} & \text{Counter\_Interval(c, i, j)}; \\
\text{&&} & \text{event}(\text{Counter\_Value(c, k)}) \& \& i < k \implies j < k; \\
\text{event} & \text{Counter\_Interval(c, i, j)}; \\
\text{&&} & \text{event}(\text{Counter\_Value(c, k)}) \& \& k < j \implies k < i.
\end{align*}
\]

We can easily convince ourselves of the validity of the axiom in two steps. First, we can check that we placed such events only when the underlying channel is a cell in the sense of GSVerif, hence we know from [12] that the
value of the counter increases strictly. Then, since events
CounterInterval(c, i, j) are placed in case the values of
the counter is skipped from i to j, if the value k of the
counter is such that i < k then we must have j ≤ k. The
same argument holds for the second axiom.

Similarly, we can define the following axiom which states
that the intervals cannot overlap:

```
axiom c:channel, i,j,k,l:nat;
1 event (CounterInterval(c, i, j))
2 && event (CounterInterval(c, k, l))
3 && j <= l  ⇒  \( j = l \land i = k \) \lor \( j = k \).
```

Its correctness follows directly from the fact that values
inside an interval are skipped.

Of course, these are excerpts of the axioms that can
be defined following the same approach. A complete
description of them is provided in the GSVerif library
accompanying the framework.

**Term freshness:** An other source of inaccuracy lies in
the fact that ProVerif is not able to detect that a message
must be fresh enough to be input at some points of the
protocol. Figure 6 presents a minimal process and a simple
query that ProVerif cannot prove satisfied. To overcome this
limitation, we define the following events and axiom: the
event TermFreshlyCreated(c, i, x) is executed when the
message x is created (that contains a nonce) and the current
value of the cell c is i, the event TermReceived(c, j, y) is
executed each time the message y is received and the current
value of the cell c is j. The following axiom is defined:

```
axiom c:channel, i,j:nat, x,y:bitstring;
1 event (TermFreshlyCreated(c, i, x)) \&\& is_subterm(x, y)
2 \&\& event (TermReceived(c, j, y))  ⇒  i < j.
```

where is_subterm(x, y) is a predicate evaluated to true
when x is a subterm of y. These events and axiom are
generic and can be used anywhere in the process to em-
phasize the need of freshness. The correctness of the axiom
follows from the fact that the nonce used in x cannot appear
in a term y before its creation.

Our axioms are only informally justified. As future work,
we plan to characterize more generally in the GSVerif setting
where the corresponding events can be placed and then
prove the axioms to be correct (which typically follows
easily from the characterization). Our axioms could thus be
added in a future version of the tool.

### 5.2.2. Prove intermediate lemmas

In addition to axioms, ProVerif supports the definition of
lemmas, i.e. queries that will be proved before the analysis
of the main query and then internally used to guide the
analysis. In our framework, we provide several generic
lemmas that should be satisfied by our generic processes
or the protocol under study. These are gathered in a unique
ProVerif library, properties.pvl. We provide here an
overview of these lemmas.

**Simple lemmas:** First, lemmas are used to emphasize sim-
ple properties such as: the value of the counter in the cell
cell_pub_id(e_id) is a natural number and smaller than the
value recorded in the event Number_Voter_Id(e_id, i). This
is formally defined by the following lemma:

```
lemma e_id:election_id, x:bitstring, i,j:nat;
1 mess(cell_ident(e_id),i)  ⇒  is_nat(i);
2 event Nb_identifier(e_id,i))
3 \&\& mess(cell_ident(e_id,i))  ⇒  j <= i.
```

In our framework, different simple events like this one
are defined. For instance, this lemma holds for the cell
cell_tally(e_id) too.

**Assumption lemmas:** As presented in Section 4.1, a few
assumptions on the protocol under study must be verified
to apply Theorem 1. They are thus defined as lemmas in our
framework and automatically proved satisfied by ProVerif
before starting the analysis of the main queries.

Moreover, even if our framework has been designed to
be as generic as possible, we found that an extra protocol
assumption was needed to make ProVerif prove: all the
verified events are executed by legitimate voters. Hopefully,
this assumption should be satisfied by any reasonable proto-
col. Concretely, the following lemma is defined and proved
by ProVerif before starting the security analysis:

```
lemma e_id:election_id, ident:voter_ident, v_idx:nat;
1 mess(cell_ident(e_id),i)  ⇒  is_nat(i);
2 event verified(e_id,v_idx)
3 \&\& mess(cell_ident(e_id,i))  ⇒  j <= i.
```

**Induction lemma:** Finally, our framework contains le-
mas which specify useful induction invariants to make
the proof. For instance, regarding the Tally process pre-
sented in Section 5.1 we can explicit the following in-
variant: given an index \( i_1 \) and its corresponding identifier
ident, if ident has been associated to a voter, v_idx, who
is expected to verify, then either the tally process has not
reached the ballot corresponding to ident yet (i.e.
\( i_1 \geq i_2 \)), or there is a Counted event which corre-
sponds to the intended vote of the voter. One can note
that the event CountedExtended is used to strengthen
the property. Moreover, the events Link_Idx_Ident and
Public_identifier link the voter index to an index
manipulated by the tally process. Indeed, even if both
are associated to the same public voter identifier ident
these associations are defined at two different steps of the
protocol: Link_Idx_Ident should be executed during
the registration the voter, while Public_identifier
is executed in the process Pub_Voter_Ident_Generation, i.e.
during the initialization of the bulletin board. The framework

```
does not guarantee that a public voter identifier is bound to the same (voter) index in both cases.

1 lemma e_idx:selection_id, i_1,i_2,v_idx:inat, v:vote, ident:voter_ident;  
2 event{voter_ident, ident}  
3 event{verified(e_id,v_idx,v)} &&  
4 event{link_idx_ident(e_id,v_idx,ident)} &&  
5 event{public_identifier(e_id,i_2,ident)} &&  
6 event{tally_read(e_id,i_1)}  
7 i_1 >= i_2  
8 || (event{counted_extended(e_id,v,i_2,ident)} &&  
9 event{counted(e_id,v)})

6. Case studies

We apply our framework to prove verifiability of several voting protocols. The first one is our running example, Helios, in a simple form. We then study the security of Belenios, a protocol that inherits a lot of structure from Helios. To show the applicability of our framework, we consider two complex protocols, namely the Swiss-Post voting protocol [1] and the CHVote protocol [22], developed to comply with the demanding Swiss regulations. We started from existing ProVerif models, as written by their respective authors, and showed that they can be directly imported into our framework in order to prove E2E-verifiability.

In these previous models, only individual verifiability was proven, with external justifications (on paper). Since we now prove E2E-verifiability, we had to model the tally process for each of these protocols (with some abstraction regarding the cryptographic primitives), yielding a more complete model. All these protocols could be automatically proved with ProVerif using our framework, without any modification. Only one variant of Belenios required to manually write specific lemmas, due to the recursive nature of this protocol that makes use of chains of hashes.

6.1. Helios

We first studied the Helios protocol as presented in running example. As a proof of concept, we decided to propose a simple model. As a consequence, we do not model the audit mechanism and assume an honest voting device instead. Thanks to our framework, we prove the security of Helios as expected: Helios ensures E2E-verifiability as soon as the server is honest. Otherwise, a malicious server could impersonate any voter and vote on their behalf which defeats E2E-verifiability.

6.2. Belenios

Belenios is an improvement of Helios in which voters are provided with a private signing key used to authenticate the senders of the ballots and created by a registrar. This new feature allows to obtain E2E-verifiability when either the registrar or the server is honest; the trust is shared between those two agents.

An implicit assumption in this analysis (same for Helios) is that the voter verifies the presence of their ballot onto the bulletin board after the election is over. This is what we call Belenios tally. As suggested in [6], we also explore a more realistic assumption: voters verify their last ballot during the voting phase, i.e. just after they cast their vote. We call this Belenios last. In this context, Belenios (as Helios) fails to ensure E2E-verifiability. Indeed, if a voter first casts a ballot b1, then a ballot b2, and checks that b2 is in the ballot box, a malicious server may re-cast the first ballot b1. We thus consider a variant of Belenios, called Belenios-hash last, where ballots are chained: each ballot contains the hash of a preceding ballot. This solution was suggested in [6] and has been further simplified by Belenios authors [19] for efficiency reasons. Interestingly, the first simplification by Belenios authors was a version with counters, called Belenios-counter last, that our approach proved to be flawed.

We applied our framework to prove the E2E-verifiability of the different versions of Belenios. We confirmed the expected security guarantees and the applicability of our framework. Only Belenios-hash required to write manually three main lemmas. The most technical one intuitively says that whenever two ballots from the same voter appear on the bulletin board then the hash of the first submitted ballot must be contained in the second ballot. The need of this lemma is due to the intrinsic difficulty of the analysis by ProVerif of a chain of hashes and is orthogonal to our framework.

6.3. Swiss Post protocol

The Swiss Post protocol [1] is developed for more than 5 years and was initially designed by Scytl. It is planned to be deployed in several cantons in 2023. The protocol relies on a setup component, whose purpose is to generate and broadcast the voting material to voters. Its role can be compared with the role of the registrar in the Belenios protocol. It assumes four control components (CCR and CCM) to process ballots during the voting phase. One of them is honest to ensure verifiability (and vote secrecy). Their role is similar to the role of the server in Helios or Belenios protocol. One challenge of the voting protocols designed for the Swiss context is that they need to offer a cast-as-intended mechanism based on return codes. Before the voting phase, each voter receives a voting sheet with some authentication material as well as return codes, one for each voting choice. During the voting phase, a voter selects their choice(s) for each question, using their voting device. They then receive a code for each selection and check that it matches the code on their voting sheet. Only once they agree with all the received codes, they send their final confirmation code. This protects against a malicious voting device that may try to modify their vote. Formally, E2E-verifiability has to hold even if the voting device is compromised. Of course, the voting server should not learn how a voter voted. Hence the return codes are computed by the four control components so that the vote remains hidden, involving non trivial cryptographic primitives.

In order to comply with the demanding Swiss regulation [14], ProVerif models have been developed over the
years to prove both verifiability and privacy. These are complex models (about 500 lines of code), with ad-hoc equal-tional theories. We imported the last available model [2] in our framework and proved E2E-verifiability immediately.

6.4. CHVote

Another protocol, CHVote [22], is designed since 2017 to also comply with the Swiss voting requirements. Therefore, it also includes a cast-as-intended mechanism with return codes. In CHVote, the control components jointly compute the return codes during the voting phase using oblivious transfer. CHVote has been supported by the Canton of Geneva during several years but is currently not considered for deployment for budget reasons. However, in order to comply with the obligation to provide computational and symbolic proofs, ProVerif models have been developed [7]. We imported these models in our framework and we again proved E2E-verifiability immediately.

Our findings are summarized in Table 1. All models are available as supplementary material in [11]. They could easily be proved in ProVerif using the latest development branch improved_scope_lemma that is about to be merged in the branch master. For all the voting protocols, our two queries could automatically be proved by ProVerif, thanks to our library of lemmas and axioms, with the exception of Belenios-hash last that required a couple of manual lemmas, as explained above. The assumptions for our characterization of E2E-verifiability have been written once for all in our library of lemmas and are easily proved by ProVerif, as expected. The benchmark have been obtained on a laptop Intel quad-core i7, 2.3 GHz, 32Go RAM.

<table>
<thead>
<tr>
<th>Voter</th>
<th>Registrar (setup)</th>
<th>Server (1 CCR/M)</th>
<th>E2E Verifiability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helios (toy ex.)</td>
<td></td>
<td></td>
<td>✓ 16s</td>
</tr>
<tr>
<td>Belenios tally</td>
<td></td>
<td></td>
<td>✓ 24s</td>
</tr>
<tr>
<td>Belenios last</td>
<td></td>
<td></td>
<td>× 5s</td>
</tr>
<tr>
<td>Belenios-counter last</td>
<td></td>
<td></td>
<td>× 8s</td>
</tr>
<tr>
<td>Belenios-hash last</td>
<td></td>
<td></td>
<td>✓ 62s</td>
</tr>
<tr>
<td>Swiss Post</td>
<td></td>
<td></td>
<td>✓ 58s</td>
</tr>
<tr>
<td>CHVote</td>
<td></td>
<td></td>
<td>✓ 17s</td>
</tr>
</tbody>
</table>

= honest                      = dishonest

**TABLE 1:** Results of the security analyses.

3. In this model, the voting device is dishonest, hence not modeled.
4. [https://gitlab.inria.fr/bblanche/proverif/-/tree/improved_scope_lemma](https://gitlab.inria.fr/bblanche/proverif/-/tree/improved_scope_lemma)
5. According to personal communications with the developers.

7. Conclusion

We designed a ProVerif framework to analyze the E2E verifiability of voting protocols based on our characterization of verifiability with two simple injective queries. This framework embeds lemmas to let ProVerif automatically check that the protocol satisfies the main assumption of the characterization. Two libraries of generic lemmas and axioms, designed in the spirit of the GSVerif tool, are included in the framework to improve ProVerif accuracy. Finally, this framework has been successfully applied to several protocols of the literature. It has been filled using existing ProVerif models of these protocols.

Our characterization holds for counting functions that are equivalent to the multiset of votes, which already covers most of the elections. Interestingly, for alternative counting functions (e.g. Condorcet, Single Transferable Vote, d’Hondt method), our two queries still imply E2E-verifiability, hence they can be used to prove E2E-verifiability with ProVerif. However, the converse implication does not hold: these two queries may be too strong in general. Writing tight queries in the general case is left as future work.

We have written our generic framework with the goal of being as general as possible, hoping that other voting systems will be cast in our framework in the future, in order to prove verifiability. It would be interesting to also be able to prove vote secrecy with the same framework. The tally phase needs to be reworked to model the fact that ballots are mixed before been decrypted. Moreover, since vote secrecy is modeled as an equivalence property, this will certainly create new challenges for ProVerif proofs.

**References**


Appendix A. 

Proof of the main theorem

We provide the detailed proofs of Lemma 1 and Lemma 2

Lemma 1. If a voting process P is E2E-verifiable then it satisfies query 1 and query 2.

Proof. Because P is E2E verifiable, we know that for all traces tr of P that contains the event finish, there exist \(V_{hv}^\prime \subseteq m V_{hv}^id\) and \(V_d\) such that \(|V_d| \leq |D|\) and

\[
V_{hv}^\prime \cup V_{hv}^id \cup V_d
\]

\(P\) satisfies query 1:
Let tr a trace of \(P\) that contains the event finish. We are going to define an injective function \(f : Result \rightarrow Voters\) that satisfies query 1. We need to show that \(f|_{V_{hv}^id}\) is a bijective function.

First, we define

\[
f|_{V_{hv}^id} : \begin{cases} V_{hv}^id \rightarrow HV \\
                      x \mapsto \text{id where verified}(id, x) \in \text{tr}
\end{cases}
\]

Because the events \(\text{verified}(id, \cdot)\) are executed at most once for each voter \(id\), we know that \(f|_{V_{hv}^id}\) is a bijective function.

Then, because \(V_{hv}^id \subseteq m V_{hv}^id\) we know that there exists a multiset \(V^id\) such that \(V_{hv}^id = \{v \mid (id, v) \in V^id\}\) and \(V^id \subseteq m V_{hv}^id\) and for all \((id, v), (id', v')\) elements in \(V^id\), we have \(id \neq id'\). We define

\[
f|_{V_{hv}^id} : \begin{cases} V_{hv}^id \rightarrow HNV \\
                      x \mapsto \text{id where } (id, x) \in V^id
\end{cases}
\]

By definition of \(V^id\) there is no \((id, v), (id', v') \in V^id\), such that \(id = id'\). Hence, we immediately conclude that \(f|_{V_{hv}^id}\) is an injective function.

Finally, because \(|V_d| \leq |D|\) there exists an injective function \(f|_{V_d} : V_d \rightarrow D\).

We are now able to conclude that the trace \(tr\) satisfies query 1 using the function \(f = f|_{V_{hv}^id} \cup f|_{V_{hv}^id} \cup f|_{V_d}\). Indeed, the multisets \(HV, HNV, D\) are disjoint by definition and thus \(f\) is injective by construction.

\(P\) satisfies query 2:
Let tr be a trace of \(P\) such that \(tr\) contains the event finish. Because \(P\) is E2E verifiable, we have that \(V_{hv} \subseteq m Result\). We thus we immediately deduce that we can define an injective function \(g : HV \rightarrow Result\) such that for all \(\text{verified}(id, v) \in tr\), \(g(id) = v\) and \(\text{counted}(v) \in tr\). Therefore, \(tr\) satisfies query 2.

\(\square\)

Lemma 2. If a voting process \(P\) satisfies query 1 and query 2 then it is E2E-verifiable.

Proof. Recall that we see Result as a set of pairs \(x = (v, i)\) where \(v\) is the value of the vote and \(i\) is its index of occurrence in Result. We will note \(\text{value}(x)\) the vote value \(v\) of \(x\).

Recall that we have defined \(h\) by \(h(x) = \begin{cases} 
g^{-1}(x) & \text{if } x \in g(HV) \\
(f \circ g)^n \circ f(x) & \text{if } x \not\in g(HV) \text{ and } f(x) \in HV \\
f(x) & \text{otherwise}
\end{cases}
\)

Case (i) makes sure that \(h\) is surjective over HV.

Justification of (ii). We need to show that \(n = \min\{i > 0 \mid (f \circ g)^i \circ f(x) \not\in HV\}\) is well-defined.

Assume by contradiction that for all \(i, (f \circ g)^i \circ f(x) \in HV\). Then, since HV is a finite set, there exist \(i < j\) such

\[
\begin{align*}
\text{tr} & = \text{tr}_1 \cup \text{tr}_2 \\
\text{tr}_1 & = \{(id, v) \mid \text{verified}(id, v) \in \text{tr} \land \text{counted}(v) \in \text{tr} \land v \in HV\} \\
\text{tr}_2 & = \{(id, v) \mid \text{verified}(id, v) \in \text{tr} \land \text{counted}(v) \in \text{tr} \land v \not\in HV\}
\end{align*}
\]

\(\square\)
that \((f \circ g)^{i} \circ f(x) = (f \circ g)^{i} \circ f(x)\). Because \(f\) and \(g\) are injective, we know that \((f \circ g)^{i}\) is an injective function. We deduce that \(f(x) = (f \circ g)^{i} \circ f(x)\). Applying once more that \(f\) is injective, we have \(x = (g \circ f)^{i} \circ x\). Thus \(x \in g(HV)\) and has already been considered in case (i), contradiction. We conclude that \(n\) is well-defined.

\(h\) is injective. Let now show that \(h\) is injective. Assume by contradiction that this is not the case. Then there exist \(x, y \in \text{Result}\), \(x \neq y\) such that \(h(x) = h(y)\).

- either \(x\) is considered in (i) and \(y\) is considered in (ii).
- We have \(h(x) = g^{-1}(x) \in HV\) and \(h(y) = (f \circ g)^{n} \circ f(y) \notin HV\) by construction. Hence, contradiction.
- or \(x\) is considered in (i) and \(y\) is considered in (iii).
- Again, \(h(x) = g^{-1}(x) \in HV\) and \(h(y) = (f \circ g)(y) \notin HV\), contradiction.
- or \(x\) is considered in (ii) and \(y\) is considered in (iii).
- We have \(h(x) = (f \circ g)^{n} \circ f(x)\) and \(h(y) = f(y)\) with \(y \notin g(HV)\). By construction, \(n > 0\). Hence \(h(x) = f \circ g \circ f(x)\). Since \(f\) is injective, we deduce that \(g \circ f \circ y \notin g(HV)\).
- \(x\) and \(y\) cannot be both considered in (i) nor in (iii) since \(f\) and \(g\) are injective.
- the last case is when \(x\) and \(y\) are considered in (ii), that is \(h(x) = (f \circ g)^{i} \circ f(x)\) and \(h(y) = (f \circ g)^{j} \circ f(y)\) and \(x, y \notin g(HV)\). We can assume \(i < j\). Since \((f \circ g)^{j}\) is injective, we deduce \((f \circ g)^{j} \circ x = (f \circ g)^{j} \circ y\) and thus \((g \circ f)^{j}(y) = x \in (g \circ f)^{i}(y)\). Thus \(x \in g(HV)\), contradiction.

We can therefore conclude that \(h\) is injective.

**E2E-verifiability.** Let \(\text{dom}_{1}\) be the (multi)set of \(x\) considered in case (i) and \(\text{dom}_{2}\) be the (multi)set of \(x\) considered in cases (ii) and (iii). By construction, \(\text{Im}(h|_{\text{dom}_{1}}) = HV\). Moreover, \(\text{Im}(h|_{\text{dom}_{2}}) \subseteq \text{Im}(f) \setminus HV\). Hence \(\text{Im}(h|_{\text{dom}_{2}}) = HV'' \cup D''\). Hence \(\text{Im}(h) = HV'' \cup D''\). Let \(R_{1} = h^{-1}(HV)\), \(R_{2} = h^{-1}(HV'')\), and \(V_{d} = h^{-1}(D'')\). We have \(|V_{d}| \leq |D|\).

We show first that \(R_{1} = V_{hv}\):

\(\text{Each } v \in V_{hv} (\text{with its occurrence}) \text{ can be injectively mapped to } id \in HV \text{ such that } \text{verified}(id, v) \in \text{tr}. \)

Thanks to the definitions of \(g\), the multiset of this \(v\) is in correspondence with an equal multiset included in \(g(HV)\). Since \(g(HV) = h^{-1}(HV)\), we deduce \(V_{hv} \subseteq R_{1}\).

Let \(v \in R_{1} = h^{-1}(HV)\). By definition of \(v\), there exists \(id \in HV\) such that \(h(v) = id = g^{-1}(v)\). By definition of \(g\), \(\text{verified}(id, value(v)) \in \text{tr}\) and thus \(v \in V_{hv}\).

It remains to show that \(R_{2} \subseteq V_{hv}\). We already know that \(V_{hv} \subseteq R_{1}\). Hence we simply need to show that \(R_{2} \subseteq \{v \mid (id, v) \in V''\}\). Each \(v \in R_{2}\) can be injectively mapped to an \(id \in HV''\) such that \(h(v) = id\). Let us show that \((id, value(v)) \in V''\). By definition, \(h(v) = id = (f \circ g)^{n} \circ f(v)\) and \(f(v) \in HV\). Let us show by induction on \(1 \leq i \leq n\) that \(value((g \circ f)^{i}(v)) = value(v)\)

- case \(i = 1\). By definition of \(f\), we have \(\text{verified}(f(v), value(v)) \in \text{tr}\). Let \(v' = g(f(v))\). By definition of \(g\) and since \(\text{verified}(f(v), value(v)) \in \text{tr}\), we have \(value(v) = value(v')\) hence \(value(v) = value((g \circ f)^{1}(v))\).
- Let \(1 < i \leq n\). By minimality of \(n\), we have that \((f \circ g)^{i} \circ f(v) \in HV\). By minimality of \(n\), \((f \circ g)^{i} \circ f(v) = f(v')\) where \(v' = (g \circ f)^{i}(v)\). By induction hypothesis, we have \(value(v') = value(v')\).

Since \(f(v') \in HV\), by definition of \(f\), \(\text{verified}(f(v'), value(v')) \in \text{tr}\). Let \(v'' = g(f(v'))\). By definition of \(g\) and since \(\text{verified}(f(v'), value(v')) \in \text{tr}\), we have \(value(v'') = value(v')\) = value(v). Hence \(value((g \circ f)^{i}(v)) = value(v)\).

Let \(\tilde{v} = (g \circ f)^{n}(v)\). By definition, \(h(v) = id = f(\tilde{v})\) and we just showed \(\text{verified}(\tilde{v}) = value(\tilde{v})\). By definition of \(f\) and since \(id \in HNV\), we have \(\text{verified}(\tilde{v}) \in \text{tr}\). Hence \(id, value(\tilde{v}) \in V'\).

This allows us to conclude that \(R_{2} \subseteq V_{hv}\), which concludes the proof of E2E-verifiability.

\[\square\]

**Appendix B. Discussion on the sufficient conditions of [5]**

We consider the scenario called \((\mathcal{V}_3, \mathcal{A}_3)\) in [5]. Under this scenario, Belenios does not satisfy the property called \(\Phi_{\text{res}}^{*}\).

Intuitively, \(\mathcal{A}_3\) is the case where the decryption trustees, some voters, and the registrar are dishonest. The server is honest. \(\mathcal{V}_3\) corresponds to the scenario where voters check that their ballots belong to the final board, the one that is tallied. In that case, Belenios does satisfy E2E-verifiability, as proven in their files (see Belenios-tally). Indeed, voters who verify are ensured that their ballots are ready to be tallied, by definition. Moreover, since the server is honest, at most \(k\) votes can be added by the adversary, where \(k\) is the number of corrupted voters.

However, Belenios does not satisfy \(\Phi_{\text{res}}^{*}\), hence one cannot conclude that Belenios satisfies MS2E2F, the property closest to our notion of E2E-verifiability. The property \(\Phi_{\text{res}}^{*}\) is defined as

\[\text{BBTally}(cr, b) \land b \neq \perp \land \text{BBkey}(y) \Rightarrow (\text{Vote}(id, cr, v) \lor v = \text{open}(b, y)) \lor \text{Corr}(id, cr)\]

This intuitively says that, whenever a ballot \(b\) is published on the bulletin board with credential \(cr\) then

- either \(cr\) is a credential corresponding to a corrupted voter
- or there is a (honest) voter \(id\) with credential \(cr\) that casts a vote \(v\), where \(v\) corresponds to the content of \(b\).

[5] exhibits an attack trace against \(\Phi_{\text{res}}^{*}\), where the (dishonest) registrar sends a credential \(cr_{1}\) to an honest voter Alice and also uses this credential to cast a ballot \(b_{A}\) using the password of a dishonest voter. Then we obtain \(\text{BBTally}(cr_{1}, b_{A})\) while \(cr_{1}\) is not corrupted and no honest voter cast a vote corresponding to \(b_{A}\). Hence \(\Phi_{\text{res}}^{*}\) is violated. However, this is not an attack against E2E-verifiability since
the result of the election would then consist of one dishonest vote, which is fine since the attack assumes at least one dishonest voter. As noticed by the authors of [5], this corresponds to a case where Alice would be prevented to vote since she can no longer use her credential \(crt_1\). Such a denial of service attack can also happen if the adversary simply stops the communication channel. In any case, Alice would notice an unexpected behavior if she attempts to vote.

More generally, this inconsistency in the results is due to the fact the \(MS_{E2F}\) property is not equivalent to our definition of E2E-verifiability. Instead of identifying voters by their \(id\), \(MS_{E2F}\) identifies voters by their \(cred\). Unfortunately, two different voters may receive the same credential from a malicious registrar. Hence, identifying voters based on their credential is ambiguous.

**Appendix C.**

**Casting Helios in our framework**

Continuing Example 5, we define the missing macros and processes.

For sake of generality, the macros update_ballot and update_gbl take as input the following data: \(e_id\) is the election identifier, \(cur\_data\) is the current data associated to the voter \(ident\), \(cur\_gbl\_data\) is the current global data, and \(b\) the ballot being processed. Moreover, they take as input the whole context of the bulletin board, i.e. \(nb\_vote\) the total number of ballots processed so far, \(i\_ident\) the number of accepted ballots for the given \(ident\), and \(i\_gbl\) the total number of ballots accepted when the last ballot corresponding to \(ident\) was processed. All these elements may be useful to decide whether a ballot is valid and define how the data is updated.

```plaintext
update_ballot(e_id, nb_vote, cur_gbl_data, 
              i_ident, i_gbl, cur_data, b) = b.

update_gbl(e_id, nb_vote, cur_gbl_data, 
           i_ident, i_gbl, cur_data, b) = empty.

is_valid_ballot(e_id, nb_vote, cur_gbl_data, 
                i_ident, i_gbl, cur_data, b) = b <> empty.
```

Finally, the macro \(is\_valid(v)\) always returns \(true\) and the macro get_ident_from_ballot simply returns the index of the voter associated to the ballot.

```plaintext
get_ident_from_ballot((ident, ctxt)) = ident.
```