Election Verifiability with ProVerif

Vincent Cheval
Inria
Paris, France
vincent.cheval@inria.fr

Véronique Cortier
Université de Lorraine, Inria, CNRS
Nancy, France
veronique.cortier@loria.fr

Alexandre Debant
Université de Lorraine, Inria, CNRS
Nancy, France
alexandre.debant@inria.fr

***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 [10] or Tamarin [25]. It is the case of major protocols like TLS1.3 [9] 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. Yet, 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 switch votes. It often focuses much of the attention since equivalence properties are typically harder to analyze than trace properties. In that respect, verifiability seems easier to prove. However, looking at the literature, it can first be noticed that it is not easy to define. Several terminologies have been developed. One is the motto: cast-as-intended, recorded-as-cast, tally-as-recorded. This intuitively says that the ballot of a voter should contain their intended vote, their ballot should be recorded without modification, and finally, the tally should make sure that it counts all the ballots. This terminology however fails to capture eligibility: only legitimate ballots should be counted, otherwise the system could be subject to ballot stuffing. Another standard terminology covers this case. It splits verifiability into three sub-properties:

- individual verifiability: a voter should be able to check that their ballot is in the ballot box. This sometimes includes cast-as-intended,
- universal verifiability: the result should correspond to the ballot box,
- eligibility verifiability: ballots should come from legitimate voters

However, these two sets of properties still miss some attacks! As pointed out by Ralf Küesters et al [24], a voting system should also prevent clash attacks, where two honest voters are (maliciously) given the same ballot, yielding the loss of one honest vote.

Hence, it is safer to define verifiability as a global property, as done in [16]. Intuitively, a voting system satisfies *E2E-verifiability* if the result of the election corresponds to:

- all the votes of honest voters that did verify their votes;
- a subset of the votes of honest voters that did not verify;
- k additional votes, where k is bounded by the number of dishonest voters.

This approach does not need to assume any structure on the voting systems like the existence of a board or a clear tally phase (sometimes invalid ballots are discarded during the tally, sometimes at an earlier stage). However, such a property is much more difficult to analyze for tools like Tamarin or ProVerif since it requires to *count* the votes. Therefore, previous works have proposed sufficient conditions, that imply E2E-verifiability and are easier to prove, either for ProVerif [15], or for Tamarin [6], [7]. Unsurpris-

This work was partly supported by the following grants: ANR Chair AI ASAP (ANR-20-CHIA-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 [6], 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.

$$\texttt{inj}-\mathsf{verified}(id, v) \Rightarrow \texttt{inj}-\mathsf{counted}(v)$$

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 [6], [7], [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 [11]: 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 [5] and Belenios [18], studied in previous approaches aiming at proving verifiability [6], [7], [15]. We also cover two complex, industrial-scale protocols, namely CHVote [22] and SwissPost [2]. 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 lemma 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 [10]. 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 $\mathsf{pk}(k)$ and the random r can be modeled with the term $\mathsf{aenc}(\mathsf{pk}(k), r, m)$ where $\mathsf{aenc}(\cdot,\cdot,\cdot)$ 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 $\mathsf{adec}(\cdot,\cdot)$ to represent the decryption algorithm and the rewriting rule: $\mathsf{adec}(k, \mathsf{aenc}(\mathsf{pk}(k), r, m)) \rightarrow m$. 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 [4]. Fresh names unknown to

$M, N, M_1, \ldots, M_k$ ::=	terms
$x \mid n \mid f(M_1, \ldots, M_k)$	
D::=	expressions
$M \mid h(D_1, \ldots, D_k)$	
ϕ::=	formula
$M = N \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi$	
P, Q::=	processes
0	nil
`out`$(c, M); P$	output
`in`$(c, x); P$	input
$P \mid Q$	parallel composition
$!P$	replication
`new` $n; P$	restriction
`let` $x = D$ `in` P	assignment
`if` ϕ `then` P `else` Q	conditional
`insert` $M; P$	table insertion
`get` $t(x_1, \ldots, x_k)$ `suchthat` ϕ `in` P `else` Q	table lookup
`event`$(E(M_1, \ldots, M_k)); P$	event

$x, x_1, \ldots, x_k$ are variables, n is an atomic data, f is a constructor symbol, h is a constructor or destructor symbol, t is a table name, and E 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 a entry satisfying a formula ϕ 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, $P \mid Q$ models that two processes are executed in parallel, and $!P$ represents an arbitrary number of processes P executed in parallel. ProVerif also contains a command `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 [10]. We note $\mathsf{Traces}(P)$ 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 [5] 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.*

```
let Voter(id,pkE) =
  in(c_pub,v);
  get voting_data(=id,(pseudo,c_auth)) in
  in(cell_voter(id), nb_vote); // Read on cell_voter(id)
  new r_ctxt;
  let ctxt = aenc(pkE,v,r_ctxt) in
  event voted(id,v);
  let b =(pseudo,ctxt) in // Create the ballot
  out(c_pub, b); out(c_auth, b);

  in(c_pub, is_verif);
  if is_verif then
    in(cell_tally(pseudo), =b); // Read and test equality
    event verified(id,v);
    out(cell_tally(pseudo), b)
  else out(cell_voter(id),nb_vote+1). // Increment cell_voter(id)
```

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 $cell_{\mathtt{voter}}(\mathsf{id}_{\mathtt{voter}})$ 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 $\mathsf{tr} = \mathsf{tr}_1. \ldots .\mathsf{tr}_n$, we say that tr executes `event`$(E(u_1, \ldots, u_l))$ at time $i \in \{1, \ldots, n\}$ if tr_i identifies a semantics step that executes `event`$(E(u_1, \ldots, u_l))$.
Correspondence queries are formulas of the form:

$$\bigwedge_{k=1}^{p} F_k(v_1,\dots,v_{l_k}) \Rightarrow \bigwedge_{i=1}^{m}\bigvee_{j=1}^{n_i} E_{i,j}(u_1^{i,j},\dots,u_{l_{i,j}}^{i,j})$$

Such a query is satisfied by a process P if for all traces $\mathsf{tr} \in \mathsf{Traces}(P)$, whenever there exists a substitution σ such that for all $k \in \{1,\dots,p\}$ the event $\mathtt{event}(F_k(v_1\dots,v_{l_k}))\sigma$ is executed in tr, then for any i, there exist a substitution σ' and j such that for all k, $F_k(v_1\dots,v_{l_k})\sigma' = F_k(v_1\dots,v_{l_k})\sigma$ and $\mathtt{event}(E_{i,j}(u_1^{i,j},\dots,u_{l_{i,j}}^{i,j}))\sigma'$ is executed in tr. Queries are extended as expected with test equalities $v = v'$, dis-equalities $v \neq v'$ and predicates $\mathtt{pred}(v_1,\dots,v_p)$.

Injective correspondence queries. To model finer grained properties, ProVerif allows to define *injective correspondence queries* whose purpose is to capture 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 [10].

Injective correspondence queries are formulas of the form:

$$\mathtt{inj}{-}F_0(v_1,\dots,v_{l_0}) \wedge \bigwedge_{k=1}^{p} F_k(v_1,\dots,v_{l_k}) \Rightarrow$$
$$\bigvee_{i=1}^{m}\left(\mathtt{inj}{-}E_{i,0}(u_1^{i,0},\dots,u_{l_{i,0}}^{i,0}) \wedge \bigwedge_{j=1}^{n_i} E_{i,j}(u_1^{i,j},\dots,u_{l_{i,j}}^{i,j})\right)$$

Given a trace $\mathsf{tr} = \mathsf{tr}_1.\dots.\mathsf{tr}_n$, we note $\mathcal{F}(\mathsf{tr})$ the set of indices α such that tr executes $F_0(v_1,\dots,v_{l_0})\sigma$ at time α for some σ. Similarly, we note $\mathcal{E}(\mathsf{tr})$ the set of indices β such that tr executes $E_{i,0}(u_1^{i,0},\dots,v_{l_{i,0}}^{i,0})\sigma$ at time β for some σ and i. An injective correspondence query as defined above is satisfied by a process P if for all trace $\mathsf{tr} \in \mathsf{Traces}(P)$, whenever there exists a substitution σ such that for all $k \in \{0,\dots,p\}$ the event $\mathtt{event}(F_k(v_1\dots,v_{l_k}))\sigma$ is executed in tr, then there exists i and a substitution σ' such that for all k, $F_k(v_1\dots,v_{l_k})\sigma' = F_k(v_1\dots,v_{l_k})\sigma$ and for all j, $\mathtt{event}(E_{i,j}(u_1^{i,j},\dots,u_{l_{i,j}}^{i,j}))\sigma'$ is executed in tr. Moreover, there exists an injective function $f : \mathcal{F}(\mathsf{tr}) \to \mathcal{E}(\mathsf{tr})$ such that if $F_0(v_1,\dots,v_{l_0})\sigma$ is executed in tr at time α then $E_{i,0}(u_1^{i,0},\dots,u_{l_{i,0}}^{i,j})\sigma'$ 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* $\mathsf{tr}_1 = \mathtt{event}(E_1(n)).\mathtt{event}(F(n)).\mathtt{event}(E_2(m)).\mathtt{event}(F(m))$, $\mathsf{tr}_2 = \mathtt{event}(E_1(n)).\mathtt{event}(E_2(n)).\mathtt{event}(F(n))$ *and* $\mathsf{tr}_3 = \mathtt{event}(E_1(n)).\mathtt{event}(F(n)).\mathtt{event}(F(n))$. *If we consider the injective query*

$$\rho = \mathtt{inj}{-}F(x) \Rightarrow \mathtt{inj}{-}E_1(x) \vee \mathtt{inj}{-}E_2(x)$$

then we have:

- tr_1 *satisfies* ρ. *We can define the injective function* f *such that* $f(2) = 1$ *and* $f(4) = 3$*;*
- tr_2 *satisfies* ρ. *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* $\mathcal{E}(\mathsf{tr}_2) = \{1,2\}$.
- tr_3 *does not satisfy* ρ. *There is no injective function* $f : \{2,3\} \to \{1\}$.

Temporal queries. Facts in queries may be annotated with temporal variables, yielding a temporal fact $F@i$. Given a trace $\mathsf{tr} = \mathsf{tr}_1.\dots.\mathsf{tr}_n$, the temporal fact $F@i$ is executed in tr w.r.t. a substitution σ if $\mathsf{tr}_{\sigma(i)} = F\sigma$. This allows e.g. to specify when an event should occur before another one.

Example 3. *The following query states that the event* $\mathsf{verified}$ *should occur at most once per voter.*

$$\mathsf{verified}(id,x)@i \wedge \mathsf{verified}(id,y)@j \Rightarrow i = j$$

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:

- $\mathsf{voted}(\mathsf{id}, v)$ 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.
- $\mathsf{verified}(\mathsf{id}, v)$ 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.
- $\mathsf{counted}(v)$ 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.
- finish tells that the tally procedure is over, no more $\mathsf{counted}$ event will be emitted.

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

- $\mathsf{hv}(\mathsf{id})$ means that the honest voter id will vote *and* verify their vote.
- $\mathsf{hnv}(\mathsf{id})$ means that the honest voter id will vote but *not* necessarily verify their vote.
- $\mathsf{corrupt}(\mathsf{id})$ 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 1, we first note that in the voter process, the events* $\mathsf{voted}(\mathsf{id}, v)$ *and* $\mathsf{verified}(\mathsf{id}, v)$

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:*

$$\mathsf{finish} \wedge \mathsf{hv}(\mathsf{id}) \Rightarrow \mathsf{verified}(\mathsf{id}, x).$$

This can easily be modeled in ProVerif, using a new feature, called restriction [11], 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 the last published ballot will be counted. This is modeled by the process `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, skiping voters 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 5. When the i-th ballot is processed by the tally process, the constant* OK *is sent on private channel $c_{end}(i)$. 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 $\mathtt{Voters} = \mathsf{HV} \cup \mathsf{HNV} \cup \mathsf{D}$ where:

- D is the set of corrupted voters, i.e., $\mathsf{D} = \{id \mid \mathsf{corrupt}(id) \in \mathsf{tr}\}$ is the set of dishonest voters.
- HV is the set of voters who verify, i.e., $\mathsf{HV} = \{id \mid \mathsf{hv}(id) \in \mathsf{tr} \text{ and } \mathsf{verified}(id, v) \in \mathsf{tr}\}$. We note V_{hv} the corresponding multiset of vote intentions:
$$V_{\mathsf{hv}} = \{v \mid \mathsf{verified}(id, v) \in \mathsf{tr} \text{ and } id \in \mathsf{HV}\}$$
- HNV is the set of voters who do not necessarily verify, i.e., $\mathsf{HNV} = \{id \mid \mathsf{hnv}(id) \in \mathsf{tr}\}$. We note V_{hnv} the corresponding multiset of vote intentions:
$$V_{\mathsf{hnv}} = \{v \mid \mathsf{voted}(id, v) \in \mathsf{tr} \text{ and } id \in \mathsf{HNV}\}$$

We note that for honest voters who do not verify, we consider all the votes that the voter has submitted. We will say that a system is secure as soon as if a ballot is counted for these voters then it corresponds to one of the submitted vote (not necessarily the last one). This definition is consistent with [7] and [15], [20] when no revote is permitted. In order to consider at most one vote per voter, we consider all the voted events associated to a voter.

$$V^{\mathsf{id}}_{\mathsf{hnv}} = \{(id, v) \mid \mathsf{voted}(id, v) \in \mathsf{tr} \text{ and } id \in \mathsf{HNV}\}$$

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

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

$$\mathsf{Result} = \{v \mid \mathsf{counted}(v) \in \mathsf{tr}\}$$

Definition 1. *A voting process P satisfies* E2E-verifiability *if, for any trace* tr *of P that contains the event* finish*, there exist $V'_{\mathsf{hnv}} \subseteq^{\mathsf{id}}_m V^{\mathsf{id}}_{\mathsf{hnv}}$ and V_d such that $|V_d| \leq |\mathsf{D}|$ and*

$$\mathsf{Result} = V_{\mathsf{hv}} \cup_m V'_{\mathsf{hnv}} \cup_m V_d$$

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 $V'_{\mathsf{hnv}} \subseteq^{\mathsf{id}}_m V^{\mathsf{id}}_{\mathsf{hnv}}$.
2) When no revote is allowed, we can equivalently write $V'_{\mathsf{hnv}} \subseteq^{\mathsf{id}}_m V^{\mathsf{id}}_{\mathsf{hnv}}$ and $V'_{\mathsf{hnv}} \subseteq_m V_{\mathsf{hnv}}$. 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 [7] 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.

```
let Voter_registration(id) =
  new c_auth;
  new pseudo; out(c_pub, pseudo);
  insert voting_data(id,(pseudo,c_auth));
  in(c_pub, voter_status);
  if voter_status = 0 then
    event corrupt(id);
    out(cell_BB(pseudo),empty) | out(c_pub,c_auth)
  else if voter_status = 1 then
    event hv(id);
    out(cell_BB(pseudo),empty)
  else if voter_status = 2 then
    event hnv(id);
    out(cell_BB(pseudo),empty).

let Close_voting_booth =
  in(c_pub, pseudo);
  in(cell_BB(pseudo),x_ballot);
  out(cell_tally(pseudo),x_ballot).
```

```
let Server =
  get voter_data(id,(pseudo,c_auth)) in
  in(c_auth, (=pseudo,x_ctxt));
  in(cell_BB(id), (x_old_psd,x_old_ballot));
  out(cell_BB(id), (pseudo,x_ctxt)).

let Tally(sk_EL) =
  in(c_pub, i);
  in(cell_tally(i),(x_psd,x_ctxt));
  if x_ctxt = empty then out(c_end(i), OK)
  else
    let x_res = adec(sk_EL,x_ctxt) in
    out(c_pub, x_res);
    event counted(x_res);
    out(c_end(i), OK).

let Finish =
  in(c_end(0), =OK);
  in(c_end(1), =OK);
  event finish.
```

Figure 2: ProVerif processes of Example 4

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 verified 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 $\mathsf{verified}(\mathsf{id}, v) \in \mathsf{tr}$ then $\mathsf{voted}(\mathsf{id}, v) \in \mathsf{tr}$. 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.

$$\mathsf{verified}(id, x)@i \wedge \mathsf{verified}(id, y)@j \Rightarrow i = j$$

A *voting process* is a process satisfying these 3 properties[1].

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 $V_{\mathsf{hv}} \cup_m V'_{\mathsf{hnv}} \cup_m V_d$.

query 1

$$\begin{aligned}\mathsf{finish} \wedge \mathtt{inj}{-}\mathsf{counted}(x) \quad \Rightarrow & \quad \mathtt{inj}{-}\mathsf{hv}(z) \wedge \mathsf{verified}(z, x) \\ \vee & \quad \mathtt{inj}{-}\mathsf{hnv}(z) \wedge \mathsf{voted}(z, x) \\ \vee & \quad \mathtt{inj}{-}\mathsf{corrupt}(z)\end{aligned}$$

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

$$\mathsf{finish} \wedge \mathtt{inj}{-}\mathsf{verified}(z, x) \Rightarrow \mathtt{inj}{-}\mathsf{counted}(x)$$

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.

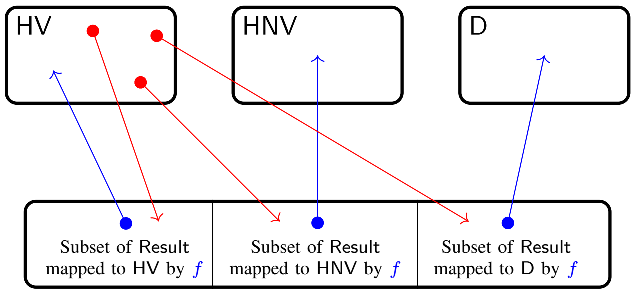
Lemma 1. *If a voting process P is E2E-verifiable then it satisfies* **query 1** *and* **query 2**.

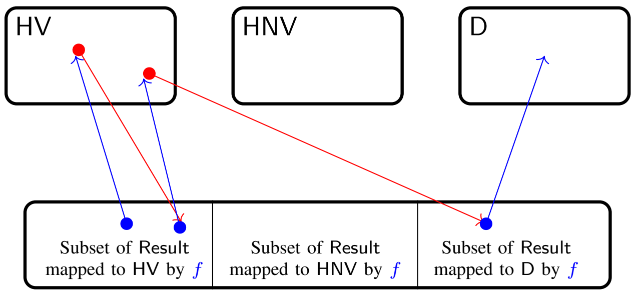
This direction is rather easy: the definition of E2E-verifiability allows to construct injective functions that satisfy **query 1** and **query 2**. The proof is given in appendix.

Interestingly, this direction was missing (and false) in previous approaches [6], [7], [15] that provided only *sufficient* conditions for E2E-verifiability, that were too strict in general. As mentioned in introduction, the study of [6] 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 B.

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 multiset 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


HV
HNV
D
Subset of Result mapped to HV by f
Subset of Result mapped to HNV by f
Subset of Result mapped to D by f
Figure 3: Two injective functions f and g.


HV
HNV
D
Subset of Result mapped to HV by f
Subset of Result mapped to HNV by f
Subset of Result mapped to D by f
Figure 4: Composition of f and g to compute h.

of an injective function f : Result $\rightarrow$ `Voters`. More precisely, the image of f is $\mathsf{Im}(f) = \mathsf{HV}' \cup \mathsf{HNV}' \cup \mathsf{D}'$ with $\mathsf{HV}' \subseteq \mathsf{HV}$, $\mathsf{HNV}' \subseteq \mathsf{HNV}$, and $\mathsf{D}' \subseteq \mathsf{D}$. If f were surjective over HV, we could easily conclude but unfortunately, f is not necessarily surjective over HV. From **query 2** and the assumption that $\mathsf{finish} \wedge \mathsf{hv}(\mathsf{id}) \Rightarrow \mathsf{verified}(\mathsf{id}, v)$, we deduce the existence of an injective function g : HV $\rightarrow$ Result. This is illustrated in Figure 3.

We use g to transform f into an injective function h : Result $\rightarrow$ `Voters` that is surjective over HV.

We define h by $h(x) =$

$$\begin{cases} g^{-1}(x) & \text{if } x \in g(\mathsf{HV}) \quad (i) \\ (f \circ g)^n \circ f(x) & \text{if } x \notin g(\mathsf{HV}) \text{ and } f(x) \in \mathsf{HV} \quad (ii) \\ & \text{where } n = \min\{i > 0 \mid (f \circ g)^i \circ f(x) \notin \mathsf{HV}\} \\ f(x) & \text{otherwise} \quad (iii) \end{cases}$$

Case (i) makes sure that h is surjective over HV. We need to show that h is well-defined. Intuitively, to associate a vote x to a voter, we can simply consider $f(x)$. But in order to ensure that h is injective, this does not work when $f(x) \in \mathsf{HV}$. Hence we try to escape from HV by composing $f \circ g$. We can prove that we eventually leave HV as HV is finite and f and g are injective, as illustrated in Figure 4.

We can then prove that h is injective and use h to prove E2E-verifiability. The full proof is given in appendix. □

5. A generic ProVerif voting framework

Thanks to our two verifiability queries, we can check E2E-verifiability using ProVerif. However, it requires to overcome different difficulties.

First, the model of the protocol must be accurate enough. In particular, it requires to faithfully model the tally process, which was typically abstracted away in most previous analyses. It needs to enforce that the tally phase processes *all* the ballots collected in the ballot-box during the voting phase of the protocol. As an immediate consequence, the model must precisely describe the voter registration phase too. To ease the modeling of a protocol, we propose a common framework which abstracts most of the technical details. The framework is described in Section 5.1.

Second, because of the accuracy of the model, it appeared that ProVerif fails to automatically prove our two security properties. Therefore, we extended our initial framework to guide ProVerif proofs. This includes techniques à la GSVerif [12] and generic lemmas that the framework should satisfy. These two elements make the ProVerif procedure precise enough to avoid over-approximations that would lead to false attacks. They also aim at improving termination. Details are given is Section 5.2.

5.1. Description of the framework

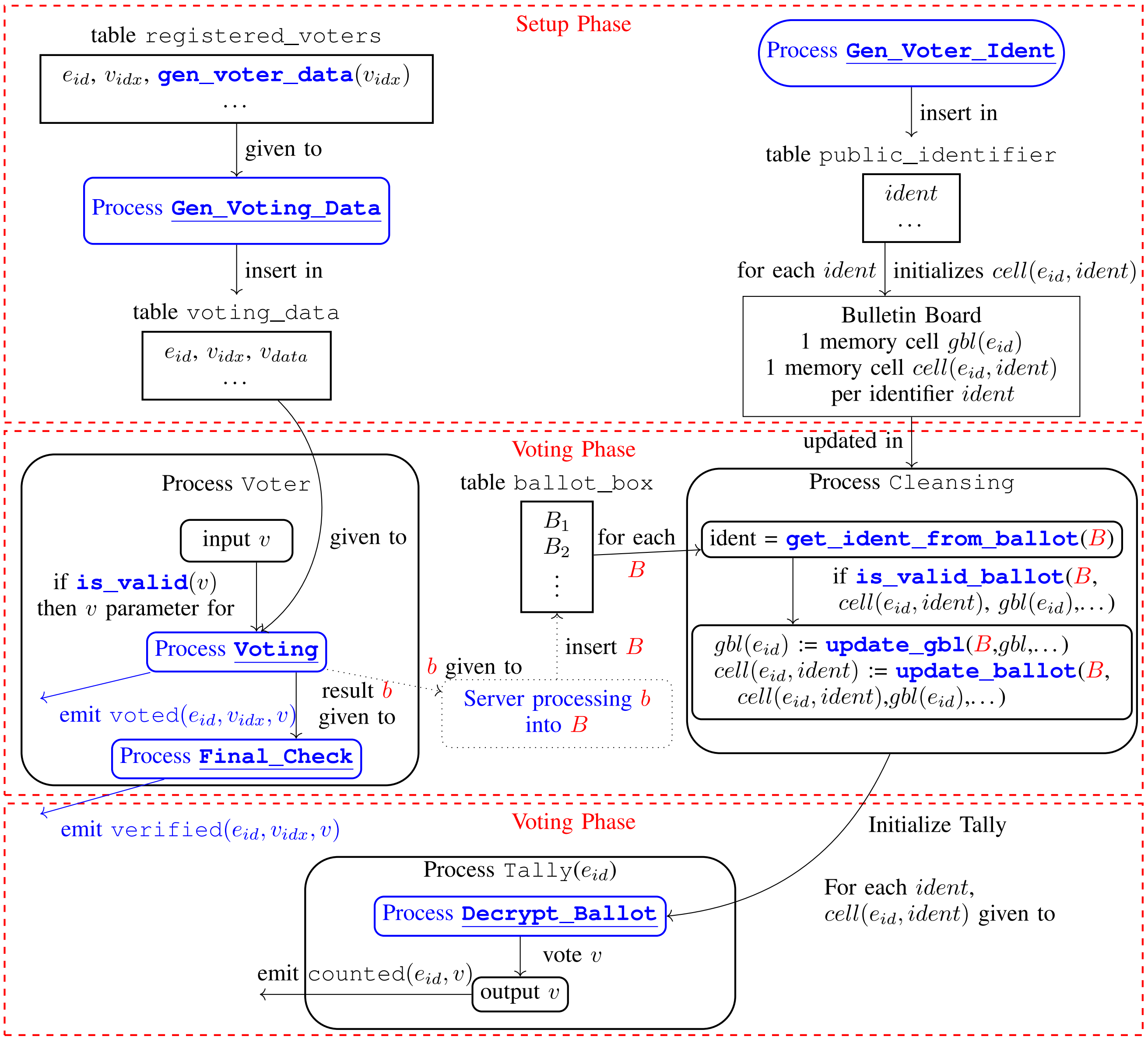
The framework defines two parts: *specific processes* and *generic processes*. The first one consists of the elements that describe the protocol under study. They must be defined for each protocol. The second one is fixed and defines how the specific elements are put together yielding the main model 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 **`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 **`gen_voter_data`** returns the initial knowledge (prior to the registration) corresponding to a given voter index. Finally, a process **`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 $\mathcal{V}$ is not part of this data and is instead defined by a macro **`is_valid`**(v) which returns *true* if and only if $v \in \mathcal{V}$.

Voting Phase. A process **`Voting`** describes how a voter votes. In general, this process should contain the voted event. Similarly, a process **`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.



Setup Phase
table registered_voters
e_{id}, v_{idx}, gen_voter_data(v_{idx})
…
given to
Process Gen_Voting_Data
insert in
table voting_data
e_{id}, v_{idx}, v_{data}
…
Process Gen_Voter_Ident
insert in
table public_identifier
ident
…
for each ident
initializes cell(e_{id}, ident)
Bulletin Board
1 memory cell gbl(e_{id})
1 memory cell cell(e_{id}, ident)
per identifier ident
updated in
Voting Phase
Process Voter
input v
given to
if is_valid(v)
then v parameter for
Process Voting
emit voted(e_{id}, v_{idx}, v)
result b
given to
Process Final_Check
table ballot_box
B_1
B_2
⋮
for each
B
insert B
b given to
Server processing b into B
Process Cleansing
ident = get_ident_from_ballot(B)
if is_valid_ballot(B, cell(e_{id}, ident), gbl(e_{id}),…)
gbl(e_{id}) := update_gbl(B,gbl,…)
cell(e_{id}, ident) := update_ballot(B, cell(e_{id}, ident),gbl(e_{id}),…)
emit verified(e_{id}, v_{idx}, v)
Voting Phase
Initialize Tally
Process Tally(e_{id})
Process Decrypt_Ballot
For each ident, cell(e_{id}, ident) given to
vote v
emit counted(e_{id}, v)
output v

in blue: parts to be defined specifically for each voting protocol

Figure 5: Overview of the framework.

Bulletin board. Two macros **update_gbl** and **update_ballot** describe how the (public or private) bulletin board evolves. More precisely, the bulletin board maintains a global memory cell $gbl(e_{id})$ as well as one memory cell $cell(ident)$ for each voter identifier $ident$ generated by **Gen_Voter_Ident**. Additionally, we assume a table ballot_box(e_{id}) that contains data (e.g. ballots) cast for publication onto the bulletin board in election e_{id}. Each time a valid entry (as defined by the Boolean macro **is_valid_ballot**) of this table is processed, the macro **update_gbl** is executed to update the value of the memory cell $gbl(e_{id})$. Moreover, after retrieving a voter identifier $ident$ from the entry using the macro **get_ident_from_ballot**, the macro **update_ballot** updates the memory cell $cell(ident)$.

Tally. The process **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 **FinalSystem** defines the scenarios under study. This process builds upon the 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-*

lios protocol described in Example 1 and Figure 2. Most of the macros and processes are very simple to define: `Decrypt_Ballot` *simply decrypts the ballot as expected,* `is_valid_ballot`(b) *checks that b is not an empty ballot,* `update_ballot` *returns the accepted ballot and* `update_gbl` *returns a dummy constant since there is no global data to be updated in the protocol. The precise definitions are given in Appendix C.*

The process `Gen_Voter_Ident` *takes as input the election identifier and a voter index, and outputs the a freshly generated pseudonym on a framework specific channel* `res_GVI`(e_id, v_idx)*, i.e.*

```
let Gen_Voter_Ident(e_id,v_idx) =
  new pseudo; out(res_GVI(e_id,v_idx),pseudo).
```

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 `gen_voter_data` *generates the authenticated channel* `c_auth` *which is forwarded to the voter by the process* `Gen_Voting_Data` *through the table* `voting_data`*. The macro also retrieves the pseudonym that was generated by* `Gen_Voter_Ident` *from the table* `public_identifier`*. Moreover, an event* `Link_Idx_Ident`$(e_{id}, v\_idx, pseudo)$ *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.*

```
let gen_voter_data(e_id,v_idx) =
  get public_identifier(=e_id,=v_idx,pseudo) in
  event Link_Idx_Ident(e_id,v_idx,pseudo);
  new c_auth;
  (pseudo,c_auth).

let Gen_Voting_Data(e_id,v_idx,voter_data) =
  insert voting_data(e_id,v_idx,voter_data).
```

Regarding the voting phase, the processes `Voting` *and* `Final_Check` *roughly correspond to the process* `Voter` *described in Example 1.* `Voting` *models how the voter forges a ballot, i.e.:*

```
let Voting(e_id,v_idx,nb_vote,v,voting_data) =
  get election_key(=e_id,_,pkE) in
  let (pseudo,c_auth) = voting_data in
  new r_ctxt;
  let ctxt = aenc(pkE,v,r_ctxt) in
  event voted(e_id,v_idx,v);
  let b = (pseudo,ctxt) in
  out(c_pub, b); out(c_auth, b);

  out(res_voting(e_id,v_idx,nb_vote),b).
```

This process exactly corresponds to the first part of the process `Voter` *up to the use of an election identifier and an output on the framework specific channel* `res_voting`(e_id, v_idx, nb_vote)*. We can note that* nb_vote *is a natural integer used to count how many times the voter voted. This information is particularly relevant to model protocols allowing re-votes.*

In our model, the voter verifies their last vote only. The check is thus modeled by the `Final_Check` *process which defines actions that the voter is expected to do once their last vote has been cast. In the context of Helios, this corresponds to the verification step in which the voter reads onto the bulletin board and checks that their ballot appears as expected. This process takes as argument the same elements as the voting process plus the last cast ballot.*

```
let Final_Check(e_id,v_idx,v,voting_data,b,...) =
  let (pseudo,_) = v_data in
  in(cell_tally(e_id,pseudo), =b);
  event verified(e_id,v_idx,v);
  out(cell_tally(e_id,pseudo), b).
```

Finally, we end the modeling of Helios by defining the scenarios under study. The process `FinalSystem` *models multiple elections, each using an encryption key chosen by the attacker (decryption authorities are supposed compromised). It calls the generic process* `System(e_id)` *defined by our framework and may also consider additional processes. In Helios, the process Server inserts in the table* `ballot_box` *the ballots it received on the authenticated channels shared with the voters. For more complex protocols, the Server may also model how to establish such an authenticated channel from the initial data.*

```
let Server(e_id) =
  get registered_voters(=e_id,v_idx,(pseudo,c_auth)) in
  in(c_auth,(=pseudo,x_ctxt));
  insert ballot_box(e_id,(pseudo,x_ctxt)).

let FinalSystem =
  ! new e_id;
    in(ch_pub,sk_EL);
    insert election_key(e_id,sk_EL,pk(sk_EL));
    System(e_id) | ! Server(e_id)
```

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.

Voter process. The role of a voter is modeled by a `Voter` process which mainly builds upon the protocol specific processes `Voting` and `Final_Check`. The process is very similar to the one presented in Example 1: it receives from the attacker the name of the candidate the voter is going to vote for. Then it executes the `Voting` process to cast a ballot and, if it is the last vote, executes the `Final_Check` process too. However, the reader can note that these actions are guarded by a communication on the private channel $cell\_voter(e\_id, v\_idx)$: 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:

```
let Voter(e_id) =
  in(c_pub,v);
  if is_valid(v) then
  get voting_data(e_id,v_idx,v_data) in
  in(cell_voter(e_id,v_idx),nb_vote);
  Voting(e_id,v_idx,nb_vote+1,v,v_data) |
  in(res_voting(e_id,v_idx),res_data);
  in(c_pub, is_last);
```

```
    if is_last then
      Final_Check(e_id,v_idx,v,v_data,res_data,nb_vote+1)
    else
      out(cell_voter(e_id,v_index),nb_vote+1).
```

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:

```
let Pub_Voter_Ident_Generation(e_id) =
  in(cell_ident(e_id),i);
  Gen_Voter_Ident(e_id,i+1) |
  in(res_GVI(e_id,i+1),ident);
  event Public_identifier(e_id,i+1,ident);
  insert public_identifier(e_id,i+1,ident);
  out(cell_ident(e_id),i+1) |
  out(cell(e_id,ident),empty_ballot).

let Stop_Pub_Voter_Ident_Generation(e_id) =
  in(cell_ident(e_id),i);
  event Nb_identifier(e_id,i);
  out(cell_tally(e_id),i).
```

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.

```
let Init_Tally(e_id) =
  get public_identifier(=e_id,i,ident) in
  in(cell(e_id,ident), x);
  out(cell_tally_last_vote(e_id,ident), x).

let Tally(e_id) =
  in(cell_tally(e_id),i);
  event Tally_Read(e_id,i)
  if i = 0 then event finish(e_id)
  else
    get public_identifier_id(=e_id,=i,ident) in
    in(cell_tally_last_vote(e_id,ident),x);
    if x = empty_ballot then out(cell_tally(e_id),i-1)
    else
      Decrypt_Ballot(e_id,i,ident,x) |
      in(res_decrypt(e_id,i),v);
      event Counted(e_id,v);
      event CountedExtended(e_id,v,i,ident);
      out(c_pub,v); out(cell_tally(e_id),i-1).
```

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 $CountedExtended(e\_id, v, i, ident)$ 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 $CounterInterval(c, i, j)$ 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 $CounterInterval(c, k)$ executed each time the value k is read on the cell c. We can then define the following axioms:

```
axiom c:channel, i,j,k:nat;
  event(CounterInterval(c,i,j))
    && event(CounterValue(c,k)) && i < k  ⟹  j <= k;
  event(CounterInterval(c,i,j))
    && event(CounterValue(c,k)) && k < j  ⟹  k <= i.
```

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

```
query x:bitstring, i,j:nat;
  event(E(x,i)) && event(F(x,j)) ⟹ i > j.

process
  out(cell, 0) | (
    ! in(cell, x); new n; event F(n,x);
    out(cell,x+1) | out(c,n)
  ) | (
    ! in(cell, x:nat); in(c, x_n:bitstring);
    event E(x_n,x); out(cell, x+1))
```

Figure 6: Toy example to illustrate ProVerif inaccuracy

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 $j \leq k$. 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;
  event(CounterInterval(c,i,j))
    && event(CounterInterval(c,k,l))
    && j <= l  ⟹  (j = l && i = k) || 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;
  event(TermFreshlyCreated(c,i,x)) && is_subterm(x,y)
    && 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 emphasize 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 simple 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;
  mess(cell_ident(e_id),i)  ⟹  is_nat(i);
  event Nb_identifier(e_id,i))
    && mess(cell_ident(e_id),j)  ⟹  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 protocol. 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;
  event(verified(e_id,v_idx,v))
    ⟹  event(Link_Idx_Ident(e_id,v_idx,ident)).
```

Induction lemma: Finally, our framework contains lemmas which specify useful induction invariants to make the proof. For instance, regarding the `Tally` process presented in Section 5.1, we can explicit the following invariant: 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 corresponds 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.

```
lemma e_id:election_id, i_1,i_2,v_idx:nat, v:vote,
  ident:voter_ident;
  event(verified(e_id,v_idx,v)) &&
  event(Link_Idx_Ident(e_id,v_idx,ident))  &&
  event(Public_identifier(e_id,i_2,ident)) &&
  event(Tally_Read(e_id,i_1))  ==>
    i_1 >= i_2
    || (event(CountedExtended(e_id,v,i_2,ident)) &&
       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 [2] 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 [7], 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 b_1, then a ballot b_2, and checks that b_2 is in the ballot box, a malicious server may re-cast the first ballot b_1. 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 [7] 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 [2] 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 protocols[2]. 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

2. The Swiss framework also considers a server. However, because it is considered untrustworthy, it is omitted in the security analysis.

years to prove both verifiability and privacy. These are complex models (about 500 lines of code), with ad-hoc equational theories. We imported the last available model [3] in our framework and proved E2E-verifiability immediately[3].

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 [8]. 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 and also on [1]. They could easily be proved in ProVerif using the latest development branch `improved_scope_lemma`[4] that is about to be merged in the branch `master` [5]. 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.

	Voter	**Registrar (setup)**	**Server (1 CCR/M)**	**E2E Verifiability**
Helios (toy ex.)	🙂	–	🙂	✓ 16s
Belenios tally	🙂	🙂 ☹	☹ 🙂	✓ 24s
Belenios last	🙂	🙂	🙂	✗ 5s
Belenios-counter last	🙂	🙂	🙂	✗ 8s
Belenios-hash last	🙂	🙂 ☹	☹ 🙂	✓ 62s
Swiss Post	🙂	🙂	🙂	✓ 58s
CHVote	🙂	🙂	🙂	✓ 17s

🙂 = 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
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 use 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

[1] https://bit.ly/e2e-verifiability.

[2] Swiss post voting specification - version 1.1.1. https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/tree/master/System, 2022.

[3] Symbolic analysis of the swiss post voting system. Swiss Post gitlab. https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/tree/master/Symbolic-models, 2022.

[4] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th Symposium on Principles of Programming Languages (POPL'01)*. ACM Press, 2001.

[5] B. Adida. Helios: Web-based Open-Audit Voting. In *USENIX*, 2008.

[6] S. Baloglu, S. Bursuc, S. Mauw, and J. Pang. Election Verifiability Revisited: Automated Security Proofs and Attacks on Helios and Belenios. In *34th IEEE Computer Security Foundations Symposium, (CSF 2021)*, pages 1–15, 2021.

[7] S. Baloglu, S. Bursuc, S. Mauw, and J. Pang. Provably Improving Election Verifiability in Belenios. In *6th International Joint Conference on Electronic Voting (E-Vote-ID 2021)*. Springer, 2021.

[8] D. Bernhard, V. Cortier, P. Gaudry, M. Turuani, and B. Warinschi. Verifiability analysis of chvote. Cryptology ePrint Archive, Report 2018/1052, 2018. https://eprint.iacr.org/2018/1052.

[9] K. Bhargavan, V. Cheval, and C. A. Wood. A symbolic analysis of privacy for TLS 1.3 with encrypted client hello. In H. Yin, A. Stavrou, C. Cremers, and E. Shi, editors, *Conference on Computer and Communications Security, (CCS'22)*, pages 365–379. ACM, 2022.

[10] 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, 2016.

[11] B. Blanchet, V. Cheval, and V. Cortier. Proverif with lemmas, induction, fast subsumption, and much more. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 69–86. IEEE, 2022.

[12] V. Cheval, V. Cortier, and M. Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in proverif. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 344–358. IEEE, 2018.

[13] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila. A Formal Security Analysis of the Signal Messaging Protocol. *J. Cryptol.*, 33(4):1914–1983, 2020.

[14] S. Confederation. Technical and administrative requirements for electronic vote casting. Annex to the FCh Ordinance of 13 December 2013 on Electronic Voting (OEV, SR 161.116), 2018.

[15] V. Cortier, A. Filipiak, and J. Lallemand. Beleniosvs: Secrecy and verifiability against a corrupted voting device. In *32nd IEEE Computer Security Foundations Symposium (CSF'19)*, 2019.

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

[17] V. Cortier, D. Galindo, and M. Turuani. A formal analysis of the neuchâtel e-voting protocol. In *3rd IEEE European Symposium on Security and Privacy (EuroSP'18)*, pages 430–442, London, UK, April 2018.

[18] V. Cortier, P. Gaudry, and S. Glondu. Belenios: a simple private and verifiable electronic voting system. In *Foundations of Security, Protocols, and Equational Reasoning*, pages 214–238. Springer, 2019.

[19] V. Cortier, P. Gaudry, and S. Glondu. Private communication, 2022.

[20] V. Cortier and J. Lallemand. Voting: You can't have privacy without individual verifiability. In *25th ACM SIGSAC Conf. on Computer and Communications Security (CCS'18)*, 2018.

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

[22] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis. CHVote system specification. Cryptology ePrint Archive, Report 2017/325, 2017.

[23] S. Kremer and R. Künnemann. Automated analysis of security protocols with global state. In *Symposium on Security and Privacy, S&P'14*. IEEE Computer Society, 2014.

[24] R. Küsters, T. Truderung, and A. Vogt. Clash Attacks on the Verifiability of E-Voting Systems. In *IEEE Symposium on Security and Privacy (S&P 2012)*, pages 395–409. IEEE Computer Society, 2012.

[25] S. Meier, B. Schmidt, C. Cremers, and D. Basin. The Tamarin Prover for the Symbolic Analysis of Security Protocols. In *25th International Conference on Computer Aided Verification (CAV'13)*, 2013.

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 $V'_{\mathsf{hnv}} \subseteq_m^{\mathsf{id}} V_{\mathsf{hnv}}^{\mathsf{id}}$ and V_d such that $|V_d| \le |\mathsf{D}|$ and

$$\mathsf{Result} = V_{\mathsf{hv}} \cup V'_{\mathsf{hnv}} \cup V_d$$

P satisfies query 1:
Let tr a trace of P that contains the event finish. We are going to define an injective function $f : \mathsf{Result} \to \mathtt{Voters}$ that satisfies **query 1**.

First, we define

$$f|_{V_{\mathsf{hv}}} : \begin{cases} V_{\mathsf{hv}} & \to \quad \mathsf{HV} \\ x & \mapsto \quad \mathsf{id} \text{ where } \mathsf{verified}(\mathsf{id}, x) \in \mathsf{tr} \\ & \qquad \text{is the event corresponding to } x \\ & \qquad \text{in the definition of } V_{\mathsf{hv}} \end{cases}$$

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

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

$$f|_{V'_{\mathsf{hnv}}} : \begin{cases} V'_{\mathsf{hnv}} & \to \quad \mathsf{HNV} \\ x & \mapsto \quad \mathsf{id} \text{ where } (\mathsf{id}, x) \in V^{\mathsf{id}} \end{cases}$$

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

Finally, because $|V_d| \le |\mathsf{D}|$ there exists an injective function $f|_{V_d} : V_d \to \mathsf{D}$.

We are now able to conclude that the trace tr satisfies **query 1** using the function $f = f|_{V_{\mathsf{hv}}} \cup f|_{V'_{\mathsf{hnv}}} \cup f|_{V_d}$. 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 $V_{\mathsf{hv}} \subseteq_m \mathsf{Result}$. We thus we immediate deduce that we can define an injective function $g : \mathsf{HV} \to \mathsf{Result}$ such that for all $\mathsf{verified}(\mathsf{id}, v) \in \mathsf{tr}$, $g(\mathsf{id}) = v$ and $\mathsf{counted}(v) \in \mathsf{tr}$. Therefore, tr satisfies **query 2**.

□

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 $value(x)$ the vote value v of x.

Recall that we have defined h by $h(x) =$

$$\begin{cases} g^{-1}(x) & \text{if } x \in g(\mathsf{HV}) & (i) \\ (f \circ g)^n \circ f(x) & \text{if } x \notin g(\mathsf{HV}) \text{ and } f(x) \in \mathsf{HV} & (ii) \\ & \text{where } n = \min\{i > 0 \mid (f \circ g)^i \circ f(x) \notin \mathsf{HV}\} & \\ f(x) & \text{otherwise} & (iii) \end{cases}$$

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

Justification of (ii). We need to show that $n = \min\{i > 0 \mid (f \circ g)^i \circ f(x) \notin \mathsf{HV}\}$ is well-defined.

Assume by contradiction that for all i, $(f \circ g)^i \circ f(x) \in \mathsf{HV}$. Then, since HV is a finite set, there exist $i < j$ such that $(f \circ g)^i \circ f(x) = (f \circ g)^j \circ f(x)$. Because f and g are injective, we know that $(f \circ g)^i$ is an injective function. We deduce that $f(x) = (f \circ g)^{j-i} \circ f(x)$. Applying once more that f is injective, we have $x = (g \circ f)^{j-i}(x)$. Thus $x \in g(\mathsf{HV})$ 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 $x, y \in \mathsf{Result}$, $x \neq y$ such that $h(x) = h(y)$.

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

We can therefore conclude that h is injective.

E2E-verifiability. Let $\mathtt{dom}_1$ be the (multi)set of x considered in case (i) and $\mathtt{dom}_2$ be the (multi)set of x considered in cases (ii) and (iii). By construction, $\mathsf{Im}(h|_{\mathtt{dom}_1}) = \mathsf{HV}$. Moreover, $\mathsf{Im}(h|_{\mathtt{dom}_2}) \subseteq \mathsf{Im}(f) \backslash \mathsf{HV}$. Hence $\mathsf{Im}(h|_{\mathtt{dom}_2}) = \mathsf{HNV}'' \cup \mathsf{D}''$ with $\mathsf{HNV}'' \subseteq \mathsf{HNV}'$ and $\mathsf{D}'' \subseteq \mathsf{D}'$. Hence $\mathsf{Im}(h) = \mathsf{HV} \cup \mathsf{HNV}'' \cup \mathsf{D}''$ with $\mathsf{HNV}'' \subseteq \mathsf{HNV}$ and $\mathsf{D}'' \subseteq \mathsf{D}$.

Let $R_1 = h^{-1}(\mathsf{HV})$, $R_2 = h^{-1}(\mathsf{HNV}'')$, and $V_d = h^{-1}(\mathsf{D}'')$. We have $|V_d| \leq |D|$.

We show first that $R_1 = V_{\mathsf{hv}}$:

$\boxed{\supseteq}$ Each $v \in V_{\mathsf{hv}}$ (with its occurrence) can be injectively mapped to $\mathsf{id} \in \mathsf{HV}$ such that $\mathsf{verified}(\mathsf{id}, v) \in \mathsf{tr}$. Thanks to the definitions of g, the multiset of these v is in correspondence with an equal multiset included in $g(\mathsf{HV})$. Since $g(\mathsf{HV}) = h^{-1}(\mathsf{HV})$. we deduce $V_{\mathsf{hv}} \subseteq_m R_1$.

$\boxed{\subseteq}$ Let $v \in R_1 = h^{-1}(\mathsf{HV})$. By definition of v, there exists $\mathsf{id} \in \mathsf{HV}$ such that $h(v) = \mathsf{id} = g^{-1}(v)$. By definition of g, $\mathsf{verified}(\mathsf{id}, \mathit{value}(v)) \in \mathsf{tr}$ and thus $v \in V_{\mathsf{hv}}$.

It remains to show that $R_2 \subseteq_m^{\mathsf{id}} V_{\mathsf{hnv}}^{\mathsf{id}}$. We already know that $V^{\mathsf{id}} \subseteq_m V_{\mathsf{hnv}}^{\mathsf{id}}$. Hence we simply need to show that $R_2 \subseteq_m \{v \mid (\mathsf{id}, v) \in V^{\mathsf{id}}\}$. Each $v \in R_2$ can be injectively mapped to an $\mathsf{id} \in \mathsf{HNV}''$ such that $h(v) = \mathsf{id}$. Let us show that $(\mathsf{id}, \mathit{value}(v)) \in V^{\mathsf{id}}$. By definition, $h(v) = \mathsf{id} = (f \circ g)^n \circ f(v)$ and $f(v) \in \mathsf{HV}$. Let us show by induction on $1 \leq i \leq n$ that $\mathit{value}((g \circ f)^i(v)) = \mathit{value}(v)$

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

Let $\tilde{v} = (g \circ f)^n(v)$. By definition, $h(v) = \mathsf{id} = f(\tilde{v})$ and we just showed $\mathit{value}(\tilde{v}) = \mathit{value}(v)$. By definition of f and since $\mathsf{id} \in \mathsf{HNV}$, we have $\mathsf{voted}(f(\tilde{v}), \mathit{value}(\tilde{v})) \in \mathsf{tr}$, that is $\mathsf{voted}(\mathsf{id}, \mathit{value}(v)) \in \mathsf{tr}$, hence $(\mathsf{id}, \mathit{value}(v)) \in V^{\mathsf{id}}$.

This allows us to conclude that $R_2 \subseteq_m^{\mathsf{id}} V_{\mathsf{hnv}}^{\mathsf{id}}$, which concludes the proof of E2E-verifiability.

□

Appendix B. Discussion on the sufficient conditions of [6]

We consider the scenario called $(\mathcal{V}_3, \mathcal{A}_3)$ in [6]. Under this scenario, Belenios does not satisfy the property called $\Phi^{\bullet}_{\mathsf{res}}$.

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 our 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 $\Phi^{\bullet}_{\mathsf{res}}$, hence one cannot conclude that Belenios satisfies $\mathsf{MS}^{\bullet}_{\mathsf{E2F}}$, the property closest to our notion of E2E-verifiability. The property $\Phi^{\bullet}_{\mathsf{res}}$ is defined as

$$\begin{aligned}&\mathsf{BBtally}(cr, b) \wedge b \neq \bot \wedge \mathsf{BBkey}(y) \Rightarrow \\ &\quad (\mathsf{Vote}(id, cr, v) \wedge v = \mathsf{open}(b, y)) \quad \vee \mathsf{Corr}(id, cr)\end{aligned}$$

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 cast a vote v, where v corresponds to the content of b.

[6] exhibits an attack trace against $\Phi^{\bullet}_{\mathsf{res}}$, where the (dishonest) registrar sends a credential cr_1 to an honest voter Alice and also uses this credential to cast a ballot $b_{\mathcal{A}}$ using the password of a dishonest voter. Then we obtain $\mathsf{BBtally}(cr_1, b_{\mathcal{A}})$ while cr_1 is not corrupted and no honest voter cast a vote corresponding to $b_{\mathcal{A}}$. Hence $\Phi^{\bullet}_{\mathsf{res}}$ 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 [6], this corresponds to a case where Alice would be prevented to vote since she can no longer use her credential cr_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 $\mathsf{MS}^{\bullet}_{\mathsf{E2F}}$ property is not equivalent to our definition (Definition 1) of E2E-verifiability. Instead of identifying voters by their id, $\mathsf{MS}^{\bullet}_{\mathsf{E2F}}$ 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.

```
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.

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