# Simultaneously Proving Privacy and Verifiability: A ProVerif Framework for Internet Voting

Vincent Cheval*, Veronique Cortier†, Alexandre Debant† and Florian Moser†
* University of Oxford, Oxford, United Kingdom
vincent.cheval@cs.ox.ac.uk
† Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
{veronique.cortier,alexandre.debant,florian.moser}@loria.fr

***Abstract*—Electronic voting aims to guarantee the same security properties than traditional paper-based voting, namely vote privacy and verifiability. While the formal verification of security protocols is now a mature field with several powerful tools such as ProVerif or Tamarin, verifying voting protocols still pushes existing tools at their limits.**

**We present in this work a proof framework for ProVerif that models an extensible election setting with an unbounded number of concurrent elections, voters and votes. We also develop a library of lemmas to aid the automatic verification. Crucially, the framework allows to reuse the same protocol model for both privacy and verifiability proofs. We apply the framework to several protocols of the literature and industry, showing the flexibility and applicability of our framework.**

***Index Terms*—privacy, verifiability, internet voting, formal verification, ProVerif**

## I. Introduction

Electronic voting aims to guarantee the same security properties than traditional paper-based voting, namely vote privacy and verifiability. Vote privacy ensures that no one learns how voters voted, while verifiability guarantees that the result corresponds to the votes of (legitimate) voters, without any manipulation. The design of electronic voting systems is error-prone with flaws discovered on notorious protocols such as Helios [1], [2], Belenios [2], the Estonian system [3], [4], and Swiss Post [5]. Therefore, as for security protocols in general, it is now expected to formally verify voting systems in order to obtain higher security assurance. Actually, formal and cryptographic proofs are even required by law for voting systems in the Swiss context [6].

Mature tools such as ProVerif [7] and Tamarin [8] have been successfully applied to analyze industrial scale protocols such as TLS [9]–[11], 5G [12] or Signal [13]. However, voting protocols push existing tools at their limits. Indeed, vote privacy is expressed as an equivalence property [14]. Intuitively, an attacker should not be able to distinguish when Alice is voting 1 and Bob is voting 0 from the converse scenario where Bob is voting 1 and Alice 0. Such equivalence properties are notoriously harder to verify than correspondence properties [15], [16]. Verifiability may seem simpler since it is modeled as a correspondence property, but it requires to *count* the votes, in order to ensure that the counted votes exactly correspond to the votes cast by voters.

In 2023, a framework [17] has been introduced to ease verifiability proofs in ProVerif. It offers pre-optimized building blocks common to voting systems such as a bulletin board or a tally process. It implements revote, and an unbounded number of elections and voters. It includes 40+ generic lemmas and axioms to aid with termination, building upon this rather recent capability of ProVerif [18]. The framework has been successfully applied to several existing protocol models such as the Swiss Post protocol and has been used in subsequent works to prove verifiability of novel protocols [19], [20].

However, no such support exists for privacy proofs. Actually, the complexity of the verifiability framework, which notably includes various counters, renders privacy proofs even harder. Worse, the verifiability framework is incompatible with vote privacy due to a simplification. Indeed, the pre-built tally process opens the ballots in the order in which they were received. So if Alice votes first, her vote will be published first, which completely breaks vote privacy. Therefore, to analyze a voting system, it is currently necessary to develop two models. One model within the verifiability framework, and a second one, ad-hoc, for privacy. Developing two models does not only cause extra efforts but also increases the risk of modelling mistakes and too simplistic assumptions. Even when the verifiability framework is not used, it is typical to develop two models, since the two security properties (vote privacy and verifiability) are very different. For verifiability, it is tempting to simplify the tally phase that is mainly there to protect privacy. Some models even omit the encryption entirely, resorting to casting the votes in clear [21], [22]. Worse, privacy models often do not include the verification steps (e.g., [23]) made by the voters, while the verification could potentially break privacy, too.

*Our contributions.* We provide a unified framework to prove both vote privacy and verifiability based on a single model of the protocol. Our framework is backward compatible with the initial framework [17] in order to be able to reuse existing models. Our first step consisted in enriching the tally phase in order to model a variety of privacy-preserving tally processes. In particular, we wanted to be able to model both corrupted and

This work was supported by funding from the French National Research Agency under the France 2030 programme with the reference ANR-22-PECY-0006 and by the European Research Council (ERC) under the European Union's Horizon Europe research and innovation programme through the ERC Synergy Grant VePaSS (Grant agreement No. 101224640).

honest parties during the tally and several types of mixnets, for example onion decryption mixnets or pure shuffling followed by threshold decryption. We note that this is already richer than many existing models where the tally is often abstracted away, even for privacy. Then, the main issue was to guide ProVerif in order to prove vote privacy. This needs to be expressed as an equivalence property, but due to its internal proof strategy, ProVerif has issues in proving even simple equivalences of the form:

$$out(a) \mid out(b) \approx out(b) \mid out(a)$$

where outputs are made in parallel (concurrently) but are written in order in the models. This issue is due to the fact that ProVerif tries to match the first part of the left-hand side with the first part of the right-hand side instead of trying to consider also the second part. Therefore, we need to guide ProVerif by (soundly) swapping the relevant parts. Secondly, the core reason why a voting protocol preserves vote privacy is that the ballots of Alice and Bob are encrypted as long as the adversary can track them, and they are decrypted only after destroying this link (e.g., by cryptographically shuffling the ballots). We therefore need to follow the journey of a ballot, from the voter to the final decryption, in order let ProVerif reason that vote privacy is indeed preserved. To do this, we introduce a special lemma that intuitively proves an invariant: ballots either come from Alice or Bob and their contents are swapped, or they come from any other voter. A priori, this lemma needs to be customized for each protocol, depending on the format of a ballot.

In order to save users the difficulty of writing customized lemmas, we equip the framework with a generic lemma, that simply depends on a (user-defined) function that explains how to extract a vote from a ballot. While our lemma is simple, it could unfortunately not be expressed as a ProVerif lemma as introduced in [18]. Indeed, lemmas can only contain constructor functions (such as encryption) and not destructor functions (such as decryption). We therefore extend ProVerif to lemmas *with* destructors. Intuitively, given a lemma with destructors, we compute (automatically) a set of equivalent lemmas without destructors, taking into account when the "destruction" may fail. This contribution is of independent interest since lemmas with destructors can be used in other contexts, outside voting.

To aid ProVerif termination, we also develop the privacy counterpart of generic lemmas, generalizing to equivalence properties some of the lemmas proposed in the verifiability framework. We also identify many lemmas that are no longer useful for privacy, and only increase the complexity of the proof or even cause non-termination.

We applied our novel framework to all previous protocols considered in the initial version of the framework, as well as to protocols subsequently using it [19], [20]. We could prove privacy and verifiability in all cases but 2: CHVote [24] and Vote&Check [19]. For CHVote, privacy-related aspects such as encrypting the votes have not yet been modeled, and indeed, privacy issues have been discovered recently [25]. The Vote&Check highlights a limitation of our framework inherited from its initial version: the individual verifiability check performed by the voter is made before the tally phase. As Vote&Check does not encrypt the votes, and therefore the ballots before and after the tally are equivalent, the verifiability framework can still be applied. For privacy, this trick however no longer works (and in general, does not work for other tracking-codes based protocols), and enriching the framework would require to break the backward compatibility that we wanted to maintain.

## II. Internet Voting

We introduce here the terminology of electronic voting and we define our running example.

### A. Agents

A voting system typically involves several actors.

During the *Setup phase*, the **Election authorities** are in charge of defining the list of eligible voters. The **Talliers** compute the public key of the election, such that the decryption key is shared among them, often in a threshold manner. In some systems such as JCJ [26] or Belenios [27], a **Registrar** is in charge of generating and distributing voting credentials to the voters.

During the *Voting phase*, the **Voter** casts typically using a **Voting device** that encrypts their vote to form the voter ballot. Sometimes the voter and their voting device are considered as a single entity, in particular when the voting device is assumed to be honest. The voter interacts with a voting **Server** that typically authenticates the voter and validates the ballot. The ballot is then published on the **Bulletin Board**, an authenticated append-only storage that can be read by anyone.

During the *Tally phase*, the **Talliers** jointly compute the result of the election, using their decryption keys.

### B. A simple voting system

We sketch a simple voting system, inspired from Helios [28], which will serve as the running example in this work. We summarize the voting phase in Figure 1.

*Setup phase.* The talliers jointly compute the public key of the election, formed by the product of the public keys of the talliers: each tallier has a share of the decryption key and is needed for decryption.

*Voting phase.* The voter forms the ballot by encrypting their vote using the public key of the election, and attaching a zero-knowledge proof that binds the encryption to the voter identity. The ballot is then sent to the voting server over an authenticated channel (e.g., the voter signs the ballot using a national eID scheme). Afterward, the voter verifies that the voting server has indeed published that exact ballot on the public bulletin board.

The voting server checks for each received ballot whether it is authenticated by an eligible voter, and that the zero-knowledge proof is valid. If both checks succeed, then the ballot is published on the public bulletin board, else it is discarded.

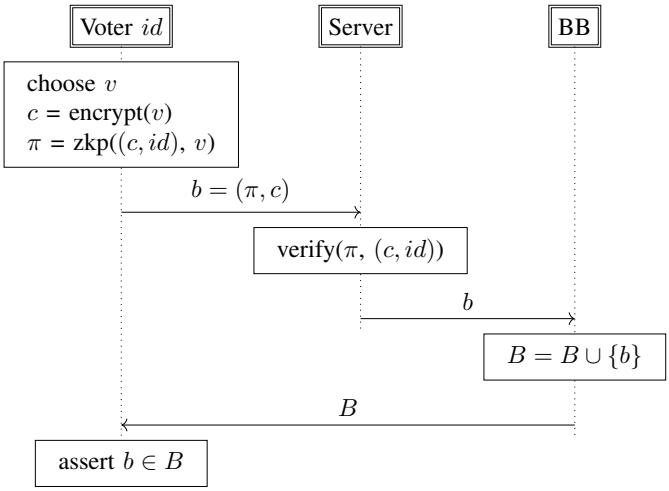



Fig. 1. The voting phase of our running example. The voter forms their ballot $b$, and sends it to the server. The server verifies the zero-knowledge proof, and then publishes the ballot on the bulletin board. Lastly, the voter verifies that their ballot has indeed been stored on the bulletin board.

*Tally phase.* A chain of mix servers — each operated by a tallier — shuffle and partially decrypt the ballots. The first mix server starts with the list of ballots as published on the public bulletin board and shuffles it. It then partially decrypts each ballot using its share of the election private key, and generates zero-knowledge proofs that this has been done correctly. Each mix server one after the other performs this process, taking as input (and verifying) the output of the previous mix servers in the chain. The last mix server publishes its output, which, after verification by the other mix servers, is attested to be the election result.

Such a chain of mix and decrypt servers is used for example in the Swiss system [29]. Another common option is to first shuffle the ballots using several mixers, and then decrypt the shuffled ballots. Other techniques exist, such as homomorphic aggregation, but they are not the main focus of this work which targets mixnet-based tallies.

### C. Properties

Voting systems should enjoy a variety of security properties depending on the context. In this work, we focus on the two core properties, namely vote privacy and verifiability.

Vote privacy guarantees that the vote of a voter is kept secret. This is typically expressed as an indistinguishability property [30], [31]: an adversary should not be able to distinguish the case where Alice votes 0 and Bob votes 1, from the converse scenario where Alice votes 1 and Bob votes 0.

$$Alice(0) \mid Bob(1) \approx Alice(1) \mid Bob(0)$$

As explained in introduction, verifiability ensures that the result corresponds to the votes of the voters and is often split into subproperties: cast-as-intended, recorded-as-cast, tallied-as-recorded, and eligibility. We adopt here an end-to-end verifiability property [32] that captures all these subproperties and avoids missing some properties [2], [33] such as clash attacks [34]. Intuitively, no matter how a voting system is designed, it should ensure that the result of the election is formed of:

- all the votes of the honest voters that performed the verification checks
- a subset of the votes of honest (but sloppy) voters that did not perform the verification checks. When no verification is performed, an attacker can typically always drop the ballot, but it should not be able to *modify* them.
- some additional valid votes, whose number does not exceed the number of dishonest voters. The attacker should not be able to add more votes than the number of voters it controls. This intuitively ensures that votes only come from legitimate voters.

Going back to our simple protocol, the voting server needs to be trusted for verifiability since it is in charge of authenticating the voters. It is possible to remove this assumption e.g., by assuming an external Registrar that is in charge of generating voting credentials such as in Belenios [27]. For vote privacy, voters need to trust the voting client and one of the several mixservers.

Verifiability and vote privacy are difficult to analyze for tools such as ProVerif or Tamarin. Therefore, a framework has been developed for ProVerif [17] which supports proving verifiability properties. Unfortunately, this framework does not cover privacy.

## III. Verifiability framework

In this section, we provide an overview of the framework proposed in [17] for proving verifiability properties with ProVerif.

### A. ProVerif

ProVerif has been widely used to analyse the security of protocols such as TLS [9], Signal [13], and also voting protocols [21]. Cryptographic primitives are abstracted by functions, that only capture the functional behavior of the primitives. For example, encryption and decryption can be represented respectively by the functions `encrypt` and `decrypt`, and a *reduction rule* specifies that the decryption of an encryption yields the plaintext as expected.

```
1  reduc forall elec, m, r;
2     decrypt(sk(elec),encrypt(pk(elec),m,r)) = m.
```

Here, we consider an election key pair `sk(elec)` and `pk(elec)`, where `sk` is a (private) function that derives the secret key of the election from the election identifier. Private functions cannot be used by the adversary. More complex rules can be considered, for example to model shared decryption.

Then a protocol is modelled through *processes*, written in an abstract programming language. For example, to model how the (honest) voter casts their ballot, we define the process `Vote`. It uses the election public key `pk(elec)` to encrypt the vote `v` into the ciphertext `c`. Then, it creates a zero-knowledge proof `p` which binds `c` to the casting voter. The ballot `b` = (`c`, `p`) is sent over the private channel `ch_auth`

to the voting server. We additionally publish b over the public channel ch_pub to model an authenticated channel.[1]

```
let Vote(elec: elec_id, id: voter_id, v: vote) =
  (* form ballot *)
  new r: random;
  let c = encrypt(pk(elec), v, r) in
  let p = zkp_encrypt(pk(elec), (c, id), (v, r)) in
  let b = (c, p) in

  (* cast ballot *)
  event Voted(elec, id, v);
  out(ch_pub, b); out(ch_auth(id), b)
```

The event Voted(elec, id, v) is used to specify security properties. For example, it will allow to state that any tallied vote $v$ should come from a voter that indeed cast $v$ (or from a dishonest voter).

The adversary fully controls the network: they can read and intercept any message sent over a public channel, and they can send any message that it can build from their knowledge and using (public) functions.

*Properties.* ProVerif allows to specify correspondence and equivalence queries. Correspondence properties typically allow to say that whenever some event $A(n, \ldots)$ has occurred in the execution of the protocol, then another event $B(m, \ldots)$ must have occurred before, with $m = n$.

In particular, it has been shown [17] that *end-to-end verifiability* can be characterized by the two following correspondence properties. The first one ensures that the result only contains votes from honest voters and some malicious ones.

```
query elec, v, id;
  event(Finish(elec)) && inj-event(Counted(elec,v)) ==>
     inj-event(HV(elec,id)) && event(Verified(elec,id,v))
  || inj-event(HNV(elec,id)) && event(Voted(elec,id,v))
  || inj-event(Corrupt(elec,id))
```

More precisely, it states that whenever a vote v in an election elec is counted Counted(elec, v) , and the tally of that election is finished, then this vote must come from:

- either an honest voter (HV) that verified their vote v
- or an honest but non verifying voter (HNV) that voted v
- or a corrupted voter

The second property guarantees that *all* honest votes are counted, for voters who did verify.

```
query elec, v , id;
  event(Finish(elec)) && inj-event(Verified(elec,id,v))
  ==> inj-event(Counted(elec,v))
```

These properties are *injective* (with inj-event) to specify that any Counted event can be injectively mapped to an honest or a corrupted voter.

An equivalence property $P \approx Q$ expresses that the adversary cannot distinguish between two scenarios modeled as two processes $P$ and $Q$. The $\approx$ relation is formally defined as the observational equivalence [35].

In particular, we consider here the standard notion of vote privacy, introduced in [14]. This definition considers two particular voters $Alice$ and $Bob$ (modeled as processes) and assumes that the rest of the system is modeled as a process $S[_]$ with a "hole" that expects the two voters.

**Definition 1** ( [14]). *A voting protocol modeled as processes $S$, $Alice$, and $Bob$ satisfies* vote-privacy *if*

$$S[Alice(a)|Bob(b)] \approx S[Alice(b)|Bob(a)]$$

*for all possible votes $a$ and $b$.*

Intuitively, this definition says that the vote privacy of Alice is preserved as soon as there is another honest voter Bob that could vote like her, while all other voters may be dishonest. A caveat of this definition is that it always admits a trivial attack: an attacker may block the ballot of Bob and wait for the result, hence observing Alice's vote. Therefore, the system $S$ typically encodes that the ballots of Alice and Bob are received before proceeding to the tally. In other words, this definition says that the vote privacy of Alice is preserved as soon as there is another honest voter Bob that could vote like her, assuming the attacker did not block the ballot of neither Alice nor Bob.

In ProVerif, equivalence properties are typically written between processes that only differ by terms using the diff operator of arity 2. It means that the process obtained by considering the left parts of diff should be equivalent to the process obtained by considering the right parts of diff. For example, $S[Alice(a)|Bob(b)] \approx S[Alice(b)|Bob(a)]$ is written as

```
S[Alice(diff[a,b]) | Bob(diff[b,a])]
```

Actually, ProVerif proves a stronger notion of equivalence than the usual observational equivalence. Namely, it proves *diff-equivalence*, that shows that each move of a subprocess on the left part can be matched with strictly the same sub-process on the right. This has been shown to imply observational equivalence [15], but it is often too strong, as we will see later.

### B. Framework overview

The framework proposed in [17] provides a flexible skeleton to integrate internet voting protocols, including a pre-built public bulletin board. Further, it defines generic lemmas and axioms which aid towards a precise and terminating proof.

To model a concrete protocol, the framework supports a total of 11 user-defined functions and processes for custom functionality. Further, it provides framework-defined channels and tables for storage. By these means, the provided skeleton internet voting system can be heavily customized, including the registration of voters, the casting, and verification of ballots, the storage of these ballots on the bulletin board, as well as finally their decryption during the tally.

To demonstrate how to use the framework, we model the protocol introduced in Section II. We present only the most important user-defined processes, and place the rest in Section A. For visual clarity, this section differentiates framework-defined channels, tables and processes with a gray background .

The user-defined process corresponding to the voter process is the one presented in the previous section III-A. We just

[1]ProVerif only distinguishes between private channels, which the attacker can neither read (confidentiality) nor write (authenticity), and public channels, where the adversary can both read and write. To model the authenticated channel for our running example, we send the ballot over the private channel (authentication), and then publish it on the public channel (no confidentiality).

need to inform the framework about the result of the Vote process, by sending the ballot b on the framework-defined channel res_vote. This is done by adding the following line at the end of the process:

```
13    out(res_vote(elec, id), b)
```

To model how the election server stores the ballot, we define a process Server. It receives the ballot b sent over the authenticated channel ch_auth. Then, it verifies the zero-knowledge proof, and if it is valid, adds b to the framework-defined table ballot_box.

```
13  let Server(elec: elec_id) =
14    get identifiers(=elec, id: voter_id) in
15    in(ch_auth(id), (c, p));
16    if verify_zkp_encrypt(p, (c, id)) then (
17      insert ballot_box(elec, (id, (c, p)))
18    )
```

The public bulletin board is modelled by private channels of the form cell_BB(elec, i), with one per eligible voter i and election elec. To model decryption by the trustees, we consider a Decrypt process. Given some ballot b = (c, p), the election private key sk(elec) is used to decrypt the ciphertext c into the plaintext vote v. This v is then sent over the framework-defined channel res_decrypt.

```
19  let Decrypt(elec: elec_id, id: voter_id, b: ballot) =
20    let (c, _) = b in
21    let v = decrypt(sk(elec), c) in
22    out(res_decrypt(elec, id), v)
```

Finally, we can launch the full voting system by starting the System process provided by the Framework that takes as input the user-defined processes (see the other user-defined processes, trivial in our running example, in Section A). In parallel, we launch other processes modelled outside the framework (e.g., in our example the Server process).

```
23  expand Framework(Vote, Decrypt, System)
24
25  process
26    ! new elec: elec_id; (* bang to repeat (unbounded) *)
27    System(elec) | !Server(elec)
```

The System process encapsulates the functionality of the framework, and launches a variety of internal processes. Under the hood, the framework boots multiple internal processes, e.g., for the registration, for the honest voter who votes and then verifies, or for the tally. These processes then call the user-defined processes at appropriate places with appropriate arguments, effectively combining the generic framework functionality with the protocol-specific user-defined functionality.

As an example, we give an intuition for the Tally process.

```
1   let Tally(elec: elec_id) =
2     ! in(cell_tally(elec), i: nat);
3     if i > 0 then (
4       in(cell_BB(elec, i), b: ballot);
5
6       Decrypt(elec, i, b) |
7       in(res_decrypt(elec, i), v: vote);
8
9       event Counted(v);
10      out(ch_pub, v);
11      out(cell_tally(elec), i-1)
12    )
```

First, the process receives i on the cell_tally channel. At the very end of Tally, over this same channel $i - 1$ is sent out, modelling a decreasing counter. Its initial message is the number of registered voters in the corresponding election, sent out by a preceeding process of the framework.

For each voter i, the process retrieves the corresponding ballot from the bulletin board channel cell_BB. This ballot is decrypted by invoking the user-defined Decrypt process, which responds with the decrypted vote v on the channel res_decrypt. Tally then marks v as counted and publishes it to the public channel ch_pub.

*Limitations.* The verifiability framework cannot be reused as such to prove privacy. Indeed, the tally counts votes one-after-the-other. This is a sound simplification in the context of verifiability since the order in which the ballots are published does not matter. However, this simplification trivially breaks privacy: the adversary can easily attribute the plain votes to their respective voters by considering the order in which they are published. This is one important issue that we needed to overcome to adapt this framework to cover *both* verifiability and privacy.

## IV. Extending the framework to support privacy

To extend the framework towards vote privacy, we need a tally process that allows to model the privacy measures (e.g., mixing). We also need to model the vote privacy property. Then, the second main task is to provide techniques to *prove* privacy. We first start with a simple tally and then show how to model complex tally procedures with several mixers, where only some of them are honest.

### A. Specifying vote privacy

We implement the privacy definition into the framework by enforcing that voter 1 and 2 are always honest, and that they always vote diff[voteA, voteB] and diff[voteB, voteA], respectively. This means that we ask the adversary to distinguish between the case where voter 1 chooses voteA and voter 2 chooses voteB from the converse case where voter 1 chooses voteB and voter 2 chooses voteA. Other voters and vote-choices are left as in the verifiability framework, that is, the adversary can freely decide which voters are honest and what they vote for. In particular, our voting framework supports an arbitrary number of honest and dishonest voters and votes. As in the verifiability framework, we support revotes. Revotes allow a voter to override a previously cast vote, e.g., to correct a wrongly cast vote, or a vote cast under coercion. The honest voters 1 and 2, as all other voters, may revote arbitrarily.

One of our primary targets is to be able to use the same model for both verifiability and privacy and to make minimal modifications to the verifiability framework in terms of protocol model. Therefore, we limit our modifications to the introduction of two functions and one process that are defined differently depending on whether privacy or verifiability is considered. Namely, we introduce two functions map_status and map_vote. The first function is called

by the framework when the adversary decides on whether a given voter is honest or not, and the second when the adversary decides on the vote to be cast for a given voter. By defining these functions differently for privacy and verifiability, we can implement the respective definition without changing any other part of the corresponding framework processes.

For privacy, `map_status` enforces that the voters 1 and 2 always have their status set to `honest`. Further, `map_vote` enforces that the voters 1 and 2 choose their vote according to the privacy definition. For all other voters, the functions return the attacker chosen status or vote, respectively.

```
letfun map_status(id: voter_id, attacker_s: nat) =
  if id = 1 || id = 2 then honest
  else attacker_s

letfun map_vote(id: voter_id, attacker_v: vote) =
  if id = 1 then diff[voteA, voteB]
  else if id = 2 then diff[voteB, voteA]
  else attacker_v
```

For verifiability, the attacker chooses the values freely.

```
letfun map_status(id: voter_id, attacker_s: nat) =
  attacker_s

letfun map_vote(id: voter_id, attacker_v: vote) =
  attacker_v
```

Besides these two functions, the only other difference between the privacy and verifiability model is a new process `HBallots`. For privacy, this process enforces that the honest votes of voters 1 and 2 reach the bulletin board, which is necessary for vote privacy. For verifiability, the trivial implementation is again sufficient. We elaborate on this process in the next section.

### B. Simple tally model

The original framework publishes tallied votes one-after-the-other, which trivially breaks privacy. We need to model that votes are shuffled before being published, which is usually done by publishing votes in parallel. This also maps directly to real-world systems, which typically announce the voting result as a set of (aggregated) votes. Syntactically, this is easy to implement into the framework. The `Tally` process, instead of decrypting and publishing each vote `v` sequentially for each voter `i`, can do this in parallel.

```
(* extract from the Tally processus *)
out(ch_pub, v) | (* semicolon replaced by pipe *)
out(cell_tally(elec), i-1)
```

Vote privacy is trivially broken if the adversary can block votes: it may block all but the vote of Alice, and then learn the vote of Alice from the result. Hence, vote privacy implicitly assumes that the ballots of voters 1 and 2 are always counted. To make this explicit in our model, we introduce a private channel `res_vote_ballot` that

- records the ballots formed by voters by adding `out(res_vote_ballot(elec, id), b)` in the user-defined `Vote` process, and
- magically passes this information to a new process `HBallots` that ensures that the ballots of voters 1 and 2 are transmitted to the tally.

The process `HBallots` is defined as follows.

```
let HBallots(elec: elec_id) =
  (* get the honest ballots *)
  in(res_vote_ballot(elec, 1), b1: ballot);
  in(res_vote_ballot(elec, 2), b2: ballot);

  (* ensure ballots are on the bulletin board *)
  in(cell_BB(elec, i1), =b1));
  in(cell_BB(elec, i2), =b2));

  (* record the two indices *)
  insert honest_voters_index(elec, i1, i2);

  (* forward all ballots to the tally *)
  ! get identifiers(=elec, i) in
  in(cell_BB(elec, i), b: ballot);
  out(cell_BB_checked(elec, i), b)
```

It fetches the ballots cast by voters 1 and 2 on channel `res_vote_ballot` and guarantees that the two ballots from voters 1 and 2 are on the bulletin board (`cell_BB`). Then, it forwards all ballots from the bulletin board to `cell_BB_checked`. Further, we adapt the `Tally` process to read from `cell_BB_checked` instead of `cell_BB`. For verifiability, `HBallots` is trivial, i.e. simply forwards all ballots from `cell_BB` to `cell_BB_checked`.

```
let Tally(elec: elec_id) =
  ! in(cell_tally(elec), i: nat);
  if i > 0 then (
    in(cell_BB_checked(elec, i), b: ballot);

    Decrypt(elec, i, b) |
    in(res_decrypt(elec, i), v: vote);

    event Counted(v);
    out(ch_pub, v) |
    out(cell_tally(elec), i-1)
  )
```

This concludes the extension to privacy for simple tally procedures. Further, as it is fully backward compatible, the models written for the verifiability framework directly translate. More complex internet voting protocols however employ a so-called *verifiable shuffle* to anonymize the votes, which unfortunately interferes with our current tally modeling. We show in the next section how to further extend our framework to complex tallies.

### C. Modelling a verifiable decryption shuffle

Verifiable shuffles typically take as input a list of ciphertext, shuffle it, and then re-randomize or (partially) decrypt each individual element. This allows to break the link between input and output elements. The shuffles are made verifiable by producing zero-knowledge proofs that the plaintext of the elements have not been altered. This is not only important for correctness of the shuffle, but also for its privacy: For example, in an unverified shuffle, an adversary might remove all but one ciphertext, trivially exposing its plaintext after decryption.

A verifiable shuffle is typically executed by multiple components one-after-the-other, as depicted in Figure 2. This allows to distribute the trust between the components: as long as one component is honest, the shuffle effectively anonymizes the elements. Intuitively, privacy and verifiability are ensured despite the existence of dishonest shufflers through three steps:

1) the dishonest shufflers that play first may only return a list of ciphertexts whose plaintext correspond the initial

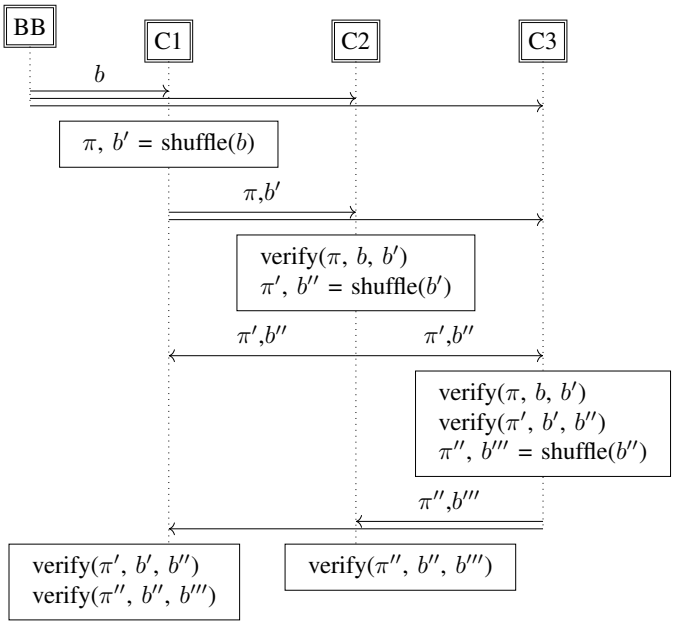



Fig. 2. Verifiable shuffle with three components C. They receive the list of ballots $b$ from the bulletin board BB. Each then performs a verifiable shuffle which rerandomizes or partial decrypts each ballot, and changes their order in the list, while producing a proof $\pi$ that this has been done correctly. Each component verifies the execution of the other components.

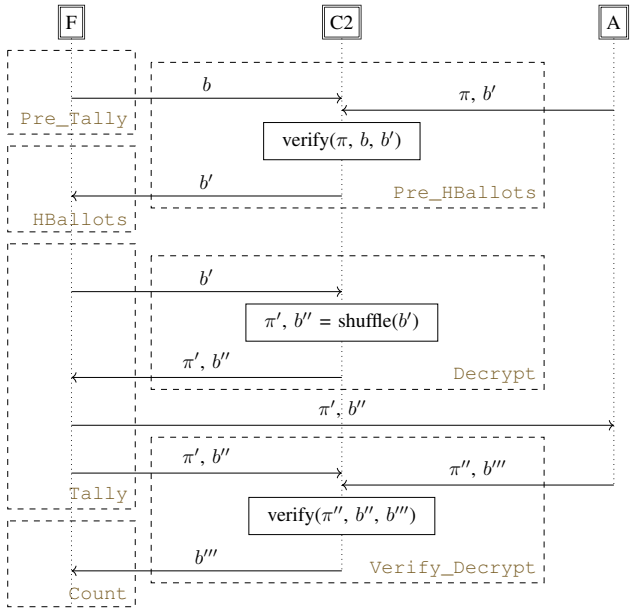



Fig. 3. Modelling of a verifiable shuffle with honest component C2, visualizing the interactions with the framework F and the adversary A (which notably runs the untrusted components C1 and C3). C2 performs the same actions as in Figure 2, while we visualize how these actions are split between the different processes. `Pre_HBallots`, `Decrypt` and `Verify_Decrypt` are user-defined processes, while the others are provided by the framework. Newly introduced for privacy are the upper three processes, which together allow to implement the privacy definition and handle the swap-related challenges.

list of ciphertexts, thanks to zero-knowledge proofs. This preserves verifiability and also privacy since the votes of all voters (including 1 and 2) are kept.

2) at least one honest shuffler honestly permutes the ciphertexts, which prevents the adversary from tracking the votes and preserves privacy.
3) finally, the dishonest shufflers that play last may not change the set of plaintexts, again thanks to the zero-knowledge proofs, which preserves verifiability.

From a modelling perspective, the shuffles performed by the adversary during Steps 1 and 3 strongly interferes with our current tally. Indeed, the honest ballots are (legitimately) modified by the dishonest shufflers. However, the framework needs to be able to continue to track the individual ballots that are part of the tally. This is essential not only to prove the correspondence queries required for verifiability, but also to rule out false privacy attacks that could arise from losing track of honest voters' ballot.

To tackle the modelling of the verifiable shuffles, we split the mixing and tally procedure as illustrated in Figure 3.

First, we introduce a new framework process `Pre_Tally`. It reads all ballots from the bulletin board, and then launches for each ballot the new user-defined `Pre_HBallots` process.

```
let Pre_Tally(elec:elec_id) =
  ! get identifiers(=elec, i) in
  in(cell_BB(elec, i), b: ballot);

  Pre_HBallots(elec, i, b)
```

The `Pre_HBallots` process is in charge of reading the ballots produced by the dishonest shufflers. Using the zero-knowledge proofs, this processes attributes the permuted ballots back to their respective index on the new channel `res_pre_hballots`. Then, the `HBallots` only forwards these ballots if the honest ballots have indeed been part of the tally. Concretely, we can replace the last lines of the `HBallots` process with:

```
(* forward all ballots to the tally *)
! get identifiers(=elec, i) in
in(res_pre_hballots(elec, i), b: ballot);
out(cell_BB_checked(elec, i), b)
```

Then, as before, the `Tally` continues on the ballots it receives over `cell_BB_checked`. The decryption or rerandomizing step performed by the honest shuffler, as well as the shuffling step, is implemented in the user-defined `Decrypt` process. The `Decrypt` process no longer computes directly the result in clear but computes instead some partial decryption, output on channel `res_pre_decrypt`.

We now differentiate between the `Tally` process, in charge of the honest contributions of the tally process and a new `Count` process in charge of announcing the result, possibly getting some data from an untrusted source. This is for example the case when the final shufflers are dishonest.

The `Tally` process now reads the ballot box, launches the `Decrypt` process, retrieves its result and stores it into a table `log_i_c`. When all the ballots are processed, i.e., when the decreasing loop on channel `cell_tally(elec)` reached 0, all the stored information in the table `log_i_c` is published and passed to the final `Verify_Decrypt` process which will determine the result from the adversarial contributions and write it on `res_decrypt`. These actions are performed

in parallel to avoid assuming a specific order of execution between them.

```
let Tally(elec: elec_id) =
  ! in(cell_tally(elec), i: nat);
  if i = 0 then (
    ! get log_i_c(elec,i,c) in
      ( out(ch_pub,c) | Verify_Decrypt(elec, i, c) )
  ) else (
    in(cell_BB_checked(elec, i), b: ballot);

    Decrypt(elec, i, b) |

    in(res_pre_decrypt(elec, i), c:bitstring);
    insert log_i_c(elec,i,c);
    out(cell_tally(elec), i-1)
  )
```

Finally, the `Count` framework process announces the result by reading it on `res_decrypt` and raising `Counted(elec,v)` events.

```
let Count(elec: elec_id) =
  ! in(cell_result(elec), i:nat);
  if i = 0 then event Finish(elec)
  else (
    in(res_decrypt(elec, i), v:vote);
    event Counted(elec, v);
    out(cell_result(elec), i-1)
  )
```

The channel `cell_result` is a private channel that allows to emulate a loop on all voters, similarly to the `cell_tally` channel.

We illustrate this on an example.

### D. Example

We continue our running example Helios model, assuming a verifiable shuffle with three components, the middle one being honest. The first (dishonest) shuffler may provide an arbitrary list of ciphertexts as output, provided that it can also provide a valid zero-knowledge proof of correct mixing. We assume here that the list formed of the ballots that appear on the ballot box has been signed by an authority with verification key `vk_auth`. This guarantees that all the shufflers agree on the same initial list.

```
let Pre_HBallots(elec: elec_id, i: nat, b: ballot)
  let (c, _) = b in
    out(ch_pub, c);
    in(ch_pub, l0, l1, zkp, s);
    if verify_sign(l0, s, vk_auth) &&
       valid_mix(l0, l1, zkp) &&
       c = get_list(l0, i) then (
        let c' = get_list(l1, i) in
        out(res_pre_hballots(elec, i), c')
    )
```

where the function `get_list(l, i)` returns the `i`-th element of the list `l`, and the functions `verify_sign` and `valid_mix` check respectively the validity of the signature `s` and of the zero-knowledge proof of correct mixing `zkp`. Since we consider an arbitrary number of voters, defining such a zero-knowledge proof of correct mixing would require to check equality of the plaintexts of lists of ballots of arbitrary size. This seems difficult to encode without AC (associative and commutative) symbols, that are currently out of scope of ProVerif.

Instead, we hardcode that there is a bijection between the ballots of the bulletin board and the ones returned by the first dishonest shuffler. Indeed, the zero-knowledge proof of correct mixing [36]–[38] ensures that the content of the ciphertexts has not been modified: the multiset of votes contained in the initial list $l_0$ is equal to the multiset of votes contained in the list $l_1$ returned by the dishonest shuffler. Hence, we know that there exists such a bijection between the ciphertexts of $l_0$ and of $l_1$ such that the corresponding plaintexts are equal. So we consider instead the following `Pre_HBallots` process.

```
let Pre_HBallots(elec: elec_id, i: nat, b: ballot)
  let (c, ) = b in
  out(ch_pub, c);

  (* associate output of first shuffler *)
  in(ch_pub, (c': ciphertext_t));
  if recognize_partial_decrypt(c, c') then (
    out(res_pre_hballots(elec, i), wrap_m_ballot(c'))
  )
```

The function `recognize_partial_decrypt` magically associates each input ciphertext `c` with the output ciphertext `c'` by decrypting `c` and attributing `c'` to the cell initially corresponding to `c`. This enforces that ballot content are not modified and that the framework can track ballots.

Then the second (honest) shuffler mixes the ballots (in `HBallots` by sending all ballots in parallel), decrypts and returns the return to the `Tally` that will forward the result to the attacker.

```
let Decrypt(elec: elec_id, i: nat, b: m_ballot)
  let wrap_m_ballot(c') = b in

  (* perform decryption as second shuffler *)
  let c'' = decrypt(sk(elec, 2), c') in
  out(res_decrypt(elec,i), c'').
```

The third shuffler is the last one and it returns the votes in clear. The second shuffler waits for its answer and checks the zero-knowledge proof. We again do not model this proof and assume instead that there is a bijection between the input ciphertexts and output cleartext, yielding the following `Verify_Decrypt` process.

```
let Verify_Decrypt(elec: elec_id, i: nat, c'':bitstring)
  (* associate output of third shuffler *)
  in(ch_pub, v: vote);
  if recognize_partial_decrypt(c'', v) then (
    out(res_count(elec, i), v)
  )
```

We can similarly consider the case where the honest shuffler plays first or last. For these cases, the only difference is that the invocation of `recognize_partial_decrypt` moves as expected. Concretely, if the honest shuffle plays first, both function invocations move after its decryption; and the inverse if the honest shuffle plays last. We provide implementations thereof as part of this work.

### E. Limitations

Our framework can cover a variety of mixnet-based tally operations but does not model homomorphic tally. Indeed, to express an homomorphic tally, it would be necessary to specify an equational theory that expresses that the multiplication of two encryptions yields the encryption of the addition of the respective plaintexts:

```
equation forall pke, m1, m2, r1, r2;
  encrypt(pke, m1, r1) * encrypt(pke, m1, r2)
  = encrypt(pke, m1+m2, r1+r2)
```

where $*$ and $+$ are two AC symbols. Such an equation is however out of the scope of ProVerif at the moment. More generally, our framework cannot cover complex tally systems such as tally-hiding computations [39], [40].

Moreover, as mentioned in Section IV-D, we cannot fully capture the zero-knowledge proof of correct mixing, since this would also require AC symbols.

## V. How to prove privacy?

Attempting a proof over the model as proposed is not yet possible (even with a simple tally), due to both a lack of precision and non-termination. We therefore make *sound transformations* of our framework, in order to prove privacy, without changing the semantics of our model. This was already the case for proving verifiability. Indeed, the verifiability framework comes with a library of 40+ lemmas and axioms to aid termination. A large part of these lemmas are inspired from the GSVerif tool [41] that generates (correct) axioms that avoid some of the overapproximations done by ProVerif. For the privacy proof, we identify which lemmas and axioms are still useful and we develop a few new ones. Finally, we need to write a protocol-specific lemma that helps ProVerif to track the honest ballots from their emission to their tally, in order to prove privacy.

### A. Transformations of the verifiability framework

Verifiability requires to prove injective properties, hence needs to be very precise. Intuitively, it needs to track that all votes in the tally have not been modified, and that the verified votes indeed all reached the tally. As a basic building block to reach this precision, the verifiability framework [17] extensively uses channels to count (e.g. see `cell_tally`) and to store state (e.g. see `cell_BB`). To increase their precision, they are often annotated with GSVerif-like events or the [`precise`] option, additionally to some custom overall lemmas.

For example, the `Tally` process is annotated as follows.

```
let Tally(elec: elec_id) =
  !
  in(cell_tally(elec), i: nat);
  let () = PH_counter_change(count_tally(elec), i) in

  if i = 0 then (
    ! get log_i_c(elec,i,c) [precise] in
      ( out(ch_pub,c) | Verify_Decrypt(elec, i, c) )
  ) else (
    in(cell_BB(elec, i), b: ballot);

    Decrypt(elec, i, b) |
    in(res_pre_decrypt(elec, i), c: bitstring) [precise];
    insert log_i_c(elec,i,c);
    out(cell_tally(elec), i-1)
  )
```

Functions such as `PH_counter_change` raise events that allow to write GSVerif axioms of the following form:

```
(* axiom for reachability (e.g. verifiability) *)
axiom
  event(CounterChange(cell, st, i)) &&
  event(CounterChange(cell, st', i))
    ==> st = st';
  event(CounterChange(cell, st, i)) &&
  event(CounterChange(cell, st, i'))
    ==> i = i';
```

where `st` is a fresh name generated when executing the function. These axioms express, for instance, that in a cell with decreasing values, a value can be changed at most once, and each function call identifies a unique value.

Importantly, these axioms are correct hence adding these events and axioms does not change the semantics of the reference model [17]. A large part of the work of [17] consists in identifying useful (and sound) axioms, as well as developing a library of lemmas, that are then used as proof helpers.

### B. Architecture of the voting framework

We start from a reference framework, that models a generic voting system as sketched in Sections III and IV. The user-defined processes and functions are left unspecified since they need to be populated for each considered voting protocol.

For privacy, we transform the reference model as follows:

- swap ballots in the `Tally` process (see Section V-C) to avoid false attacks due to the fact that ProVerif proves a too strong notion of equivalence.
- we remove the `Count` process as it only contains non observable actions (see Proposition 3 in Appendix D).
- use some GSVerif-like axioms (see Section V-D), whose soundness has been proved in [17],
- design some lemmas, in particular the CellBB lemma (see Section V-E), that states an important invariant for proving privacy,

We summarize our approach in Figure 4.

### C. Ballot swap

ProVerif does not prove observational equivalence, but the stronger notion diff-equivalence [42], which checks whether the processes behave the same *step-by-step*. This defeats the fact that we model mixnet-based tally through parallel output of decrypted votes. In particular, ProVerif cannot prove:

```
out(ch_pub, choice[voteA, voteB]) |
out(ch_pub, choice[voteB, voteA])
```

because it will try to match the first output on the left with the first output on the right.

To circumvent this well known issue, a standard technique consists in swapping terms in between the processes, for parts that are executed in parallel. This has been proven sound by Blanchet and Smyth [15]. Hence, we replace the lines

```
    ! get log_i_c(elec,i,c) in
      ( out(ch_pub,c) | Verify_Decrypt(elec, i, c) )
```

of process `Tally` by

```
get honest_voters_index(=elec,i1,i2) in
(
  ! get log_i_c(=elec,=i1,c1) in
    get log_i_c(=elec,=i2,c2) in
      out(ch_pub,diff[c1,c2]) |
      Verify_Decrypt(elec, diff[i1,i2], diff[c1,c2]) |
      out(ch_pub,diff[c2,c1]) |
      Verify_Decrypt(elec, diff[i2,i1], diff[c2,c1]) |
) | (
```

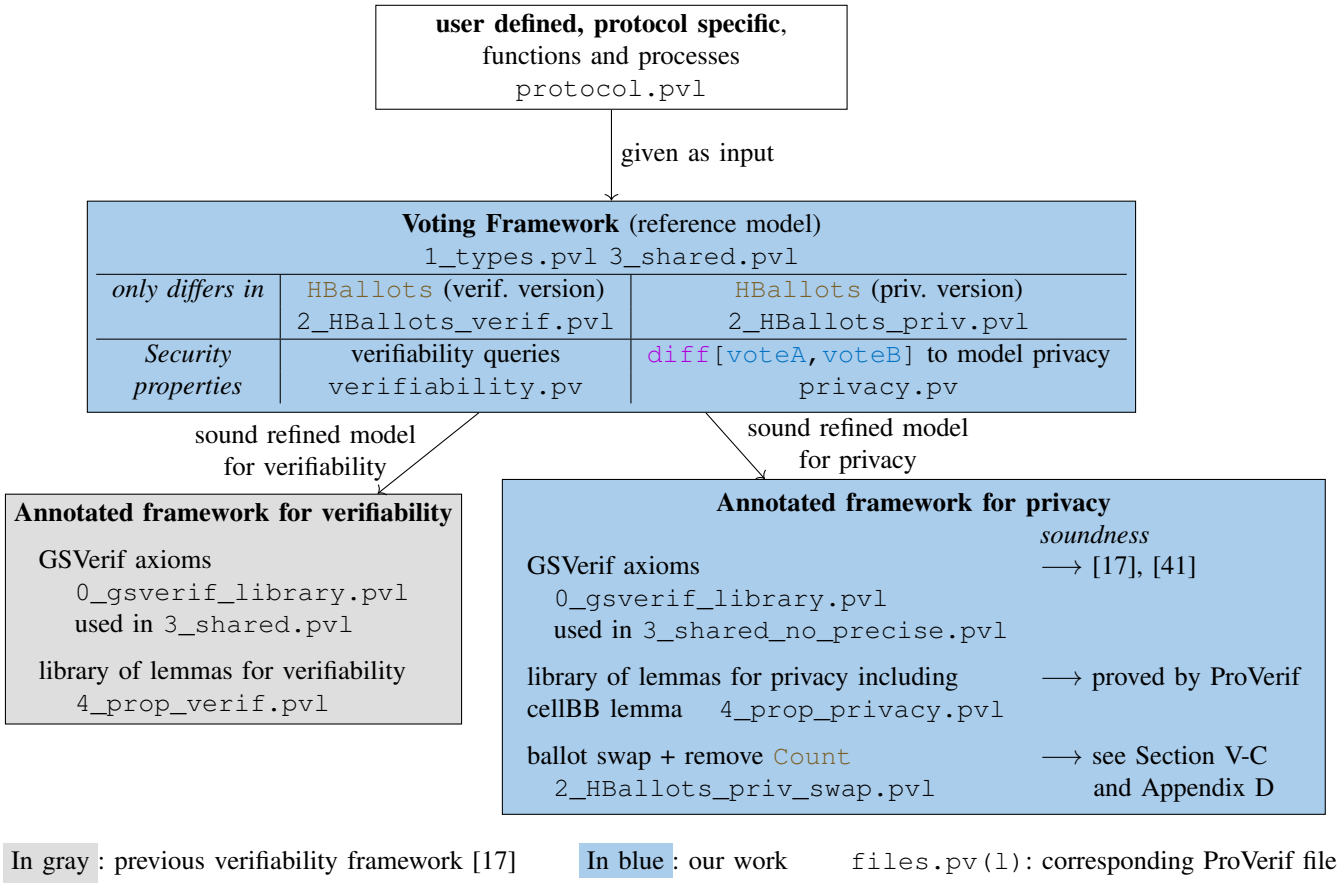



Fig. 4. Architecture of the voting framework.

```
    ! get log_i_c(elec,i,c) suchthat i <> i1 && i <> i2 in
      ( out(ch_pub,c) | Verify_Decrypt(elec, i, c) )
)
```

If the index does not correspond to indices `i1` or `i2` of the ballots of the two honest voters, the behavior is unchanged. Otherwise, we retrieve the two ballots of the honest voters, and swap them.

Intuitively, this swap is sound because the processes `Verify_Decrypt` and the outputs are all launched in parallel, hence

```
out(ch_pub,c1) |
Verify_Decrypt(elec, i1, c1) |
out(ch_pub,c2) |
Verify_Decrypt(elec, i2, c2)
```

is equivalent to

```
out(ch_pub,c2) |
Verify_Decrypt(elec,i2,c2) |
out(ch_pub,c1) |
Verify_Decrypt(elec,i1,c1)
```

A formal proof of soundness of this swap is provided in Appendix D.

### D. Balancing precision and non-termination

*Axioms for privacy.* For privacy, the precision developed for proving verifiability is not only largely unnecessary for the proof, it even leads to a non-terminating proof. A contributing reason is that the diff-equivalence attacker, stronger than the real adversary, needs to explore all executions, and check that they perform the same actions in both worlds. If branches of the proof are not cut off early (e.g., because precision measures prevent it), the proof therefore quickly explodes in size.

We therefore ignore most of the GSVerif axioms and events for the privacy proof. We however need to keep a GSVerif-like event at a crucial place: over the `cell_BB` channel. This prevents that two different honest ballots are contained in the same cell, which is impossible by construction of our model. Otherwise, the adversary finds a false attack: While the swap is applied to one of the two ballots, the other ballot remains untouched and is published. The diff-adversary can use this to again trivially distinguish the worlds. While this is exactly the case for which GSVerif was built, reintroducing the GSVerif-like events for all cells on the bulletin board is however too coarse-grained, and again leads to non-termination. The solution is to restrict the GSVerif-like events to only the relevant ballots, i.e., the ballots of honest voter 1 and 2.

```
in(cell_BB(elec, i), b: ballot);
if i <= 2 then (
  let () = PH_synch_value_bi(count_BB(elec, i), b) in
)
```

We adapt this GSVerif-like axiom to diff-equivalence.

```
(* GSVerif-like axiom for equivalence (e.g. privacy) *)
axiom
```

```
    event(SyncValueBi(cell, diff[bL, bR])) &&
    event(SyncValueBi(cell, diff[bL', bR'])))
    ==> bL = bL' && bR = bR'
```

The soundness of this axiom, simply stated for bi-processes, follows directly from its soundness for standard processes [17].

*Lemmas to guide natural numbers and equalities.* Another source of non-termination are overapproximations around natural numbers and equalities. They lead to generation of traces where the cell itself is different in the left and right side of bitraces. Further, some traces include cells that are not indexed by natural numbers, but arbitrary terms. This leads to an explosion of possible traces, where notably the GSVerif-like axioms are no longer applied (as the cells are not equal in both worlds). We guide ProVerif to avoid these overapproximations with corresponding lemmas over the cells.

```
lemma
  mess(cell_BB(diff[eidL, eidR],  i), diff[bL, bR])
    ==> eidR = eidL;
  mess(cell_BB(elec, diff[iL, iR]), diff[bL, bR])
    ==> is_nat(iL) && is_nat(iR) && iL = iR
```

*Guide resolution algorithm.* After all these changes, ProVerif will still loop on counters. We therefore need to prevent the resolution algorithm to pick the counter terms for resolution. This can be done by adding appropriate noselects statements, that intuitively change in which order ProVerif applies its resolution algorithm, without breaking its soundness.

```
noselect mess(cell_BB(elec, i), diff[bL, bR]).
noselect mess(cell_tally(elec), diff[nL, nR]).
```

However, by preventing the resolution on counters, the model becomes very imprecise. Concretely, it can no longer relate the ballots cast by the voters to the ballots in the tally. This is manifested as another source of non-termination, where traces are explored that contain ballots which could not exist in a real execution.

We therefore need a way to connect the ballots cast by the voters to the ballots found in the tally phase. More precisely, we only need to enable ProVerif to track the ballots cast by the honest voters, as those are the only ballots that might allow the diff-equivalence adversary to separate the two worlds. If the corresponding traces of these ballots can be explored exhaustively by the adversary, our proof will terminate and potentially succeed. We approach this by writing a custom lemma, that we describe in detail in the next section.

### E. The cellBB lemma

To help ProVerif establish a link between the ballots cast by the honest voters 1 and 2, and the swapped ballots during the tally, we write a lemma over `cell_BB`. The lemma unpacks the plain vote from the ballot. This must include some cryptographic structure that binds the ballot to the concrete bulletin board cell (e.g. a zero-knowledge proof that binds the ballot to the voter identifier `i`, which is also used to index the `cell_BB(elec, i)` cell). The plain vote is then asserted to be equal in the left and the right world for all voters except the honest voters 1 and 2. For the honest voter's ballots, it defines the plain votes to be exactly as expected. For example, honest voter 1 has cast their ballot given the instruction `diff[voteA, voteB]`, hence the expected vote in the left world is `voteA`, and in the right `voteB`.[2]

For our running example, the lemma looks like this:

```
mess(cell_BB(elec, i), diff[bL, bR]) ==>
  (bL = empty && bR = empty) (* either empty... *) ||
  (
    id = get_vid(i) &&

    (* ...or correctly formed ballot in that cell... *)
    (i, (cL, pL)) = bL &&
    (i, (cR, pR)) = bR &&

    (* ...encrypting some vote... *)
    cL = encrypt(pk(elec), vL, rL) &&
    cR = encrypt(pk(elec), vR, rR) &&

    (* ...cryptographically bound to the voter... *)
    pL = zkp_encrypt(pk(elec), (cL, id), (vL, rL)) &&
    pR = zkp_encrypt(pk(elec), (cR, id), (vR, rR)) &&

    (
      (* ...has equal vote left/right... *)
      (vL = vR && (vL = voteA || vL = voteB)) ||

      (* ...except for honest voters 1 and 2 *)
      ((id = 1) && (vL = voteA && vR = voteB)) ||
      ((id = 2) && (vL = voteB && vR = voteA))
    )
  )
```

This lemma has however some clear drawbacks. First of all, it is complex, even for the comparatively simple ballots used by our mock example. Notably, the ballot structure has to be defined twice, once for the left and once for the right world. As this ballot structure is further protocol dependent, the lemma must be written by the user of the framework. Therefore, the user needs to know the precise internal workings of the framework, which is something we have been able to avoid until now. We improve on this by adding a new feature to the lemmas of ProVerif, and also slightly changing the framework, as described in VI.

## VI. Extending ProVerif to ease writing lemmas

### A. A generic cellBB lemma

The cellBB lemma presented in the last section is complex. However, the actual protocol-specific part only concerns the extraction of the vote. This could be formulated much simpler as a destructor `extract_vote` which rewrites a given ballot `b` into the vote `v` it represents:

```
fun extract_vote(ballot): vote
  reduc
    let c = encrypt(pk(elec), v, _) in
    let p = zkp_encrypt(pk(elec), (c, id), _(v, _)) in
    extract_vote((c, p)) = v
  [private]
```

Further, the function `get_vid`, which attributes the cell index to the voter identifier, is mostly an artifact of the framework. In the verifiability framework, this attribution can be customized with the custom process `Gen_Voter_Ident`. However, most models do not use this capability (opting for a trivial implementation), and, if there is a need, this mapping logic can also be moved to the custom function

[2]Note how this lemma assumes that the bulletin board stores the ballots of the honest voters 1 and 2 in a predictable cell. While this assumption is true for all our case studies, it might not be true for all secure protocols.

get_ident_from_ballot. We therefore decide to get rid Gen_Voter_Ident entirely, and thereby also alleviate the need for get_vid.

So, without the need for get_vid, and given an implementation of extract_ballot, the cellBB lemma is much simpler. Further, it can now be provided directly by the framework, as no user customization is required anymore.

```
mess(cell_BB(elec, i), diff[bL, bR])@i ==>
  (bL = empty && bR = empty) ||
  (
    vL = extract_vote(bL) && vR = extract_vote(bR) &&
    (
      (attacker(diff[vL, vR])@j && i > j) ||
      (i = 1) || (i = 2)
    )
  )
```

However, destructors such as extract_vote in lemmas are not supported in ProVerif. Further, neither are letfun constructs, which would be especially useful for more complex ballots, as they allow conditionals and variable assignments.

We therefore add support for both destructors and letfun to ProVerif queries, which then allows their use also in lemmas, axioms, and restrictions. This is of general interest, and contributes to writing concise and comprehensible lemmas. In the following section, we provide an informal description of our transformation. We formally prove the soundness of it in Section C.

### B. Add support for Destructors and Letfun to ProVerif queries

In ProVerif, only constructor function symbols can appear in queries, such as encrypt. One of the reasons is that the semantics of queries and the algorithm for verifying them rely exclusively on unification and equality between messages, modulo the equational theory. However, destructor function symbols, that is functions whose semantics are defined with a rewrite rule using the construct reduc, can be seen as functions that modify messages and may fail.

For example, consider the definition of decrypt below. When the arguments given do not match the left-hand side of the rule, e.g., decrypt(sk(e), e), the term is considered by ProVerif to fail, meaning it does not produce any message.

```
reduc forall e, m, r;
 decrypt(sk(e),encrypt(pk(e),m,r)) = m
```

*Adding destructors.* When a destructor occurs in a fact of the query, such as in events, we extend the semantics of queries by only considering the possible instantiations of the destructor's arguments that produce a message, that is, cases in which the destructor does not fail. For instance, observe how the following query $\psi$ is rewritten.

```
query x,y; event(A(x,y)) ==> event(B(decrypt(x,y)))
(* is equivalent and rewritten to *)
query x,y,e; event(A(x,y)) ==>
  event(B(m)) && x = sk(e) && y = encrypt(pk(e),m,r)
```

Now, a destructor's semantics can be defined via multiple reduction rules, intuitively representing the different ways in which the destructor may be rewritten. For example, instead of failing, the decrypt destructor could be encoded to return the constant error when decryption is not possible:

```
fun decrypt(bitstring, bitstring): bitstring
  reduc forall e, m, r;
    decrypt(sk(e), encrypt(pk(e), m, r)) = m
  otherwise forall t, u; decrypt(t, u) = error
```

Hence, a term decrypt(x, y) may have multiple ways of instantiating its arguments to produce a message. When such a destructor occurs in the query, this is translated as a disjunction over all these possible instantiations. Internally, these possibilities are generated by computing the variants [43] of the terms containing the destructors, modulo the equational theory. Our previous query $\psi$ would then be rewritten to

```
query x,y,e; event(A(x,y)) ==>
  (event(B(m)) && x = sk(e) && y = encrypt(pk(e),m,r))
  || (event(B(error)) && (forall e',m',r';
       (x,y) <> (sk(e'),encrypt(pk(e'),m',r'))))
```

*Adding letfun.* The construct letfun can be seen as an extension to term generation that incorporates some components of process calculi, such as conditional tests, variable assignment, name generation, and events. In general, the evaluation of a letfun construct depends on where it is called in the executed process. For queries, however, we restrict the use of letfun to functions that are protocol independent, meaning those that are built solely from terms, conditionals, and variable assignments. Internally, this class of letfun can be expressed as a single term with natively defined destructors, whose semantics are as described above.

## VII. Case Studies

We apply our unified framework for verifiability and privacy to several voting protocols. We consider all protocols previously proven using the verifiability framework, namely the ones listed in [17] as well as two new ones: Vote&Check [19] and Short Voting Codes [20]. We were able to prove both verifiability and privacy for all of them with two exceptions: Vote&Check and CHVote [24] For CHVote, the verifiability model omits many important details for privacy (such as encrypting the votes). There is no existing privacy model, and there are actually unresolved privacy issues [25] hence we did not attempt to model privacy. Vote&Check highlights a limitation of the verifiability framework that assumes that verification is performed before or during the tally. We further comment on this case in Section VII-F.

An overview of the considered models and their respectively proven properties is given in Table I. Interestingly, for some models, proving equivalence is faster than proving verifiability. This can be explained by the fact that the cellBB lemma allows ProVerif to easily reason where the ballots of Alice and Bob are, but also because it remains demanding for ProVerif to track the counters necessary for verifiability. Compared to the verifiability framework, our novel framework has mostly the same performance, but may be slower in some scenarios (e.g., when proving that the tally counter is decreasing), likely because of the additional counter introduced for Count.

The models are provided as supplementary material. The test cases can be rerun using the util driver.py in the tests directory. The measurements have been done using ProVerif commit de4036ba, the newest commit on the

TABLE I
PROVEN PROPERTIES PER SYSTEM.

| | Verifiability | | Privacy | |
|---|---|---|---|---|
| **Running example** Helios-like [28] | | | | |
| with honest decryption 1 | ✔ | 6s | ✔ | 11s |
| with honest decryption 2 | ✔ | 6s | ✔ | 7s |
| with honest decryption 3 | ✔ | 143s | ✔ | 5s |
| **Belenios** [17], [27] | | | | |
| with honest server | ✔ | 13s | ✔ | 20s |
| with honest registrar | ✔ | 13s | ✔ | 13s |
| with honest server & registrar | ✔ | 13s | ✔ | 16s |
| **Swiss Post Protocol** [17], [29] | | | | |
| with honest CCR1 & CCM1 | ✔ | 1092s | ✔ | 18s |
| with honest CCR1 & CCM2 | ✔ | 1089s | ✔ | 11s |
| with honest CCR2 & CCM1 | ✔ | 1093s | ✔ | 18s |
| with honest CCR2 & CCM2 | ✔ | 1091s | ✔ | 11s |
| **Short Voting Codes** [20] | ✔ | 851s | ✔ | 1228s |
| CHVote [17], [24] | ✔ | | open issues in [25] | |
| Vote & Check [19] | ✔ | | not supported | |

branch necessary for the verifiability framework, on a Laptop with a 11th Gen Intel i7 at 4.70 GHz. The reported time includes everything necessary to fully reason about the respective property, hence for verifiability notably includes the time to prove the lemmas, which are sometimes proven separately to the main verifiability queries.

### A. Method

Thanks to our backwards-compatibility with the verifiability framework, existing models need only superficial extensions:

- As required by the privacy definition, we need to add the channel `res_vote_ballot` for honest voters.
- For models that implement a verifiable shuffle, we need to implement the `Pre_HBallots` and the `Verify_Decrypt` process.
- To let the cellBB lemma help with termination, we further need to implement `extract_vote(ballot)`.

This is best demonstrated on the Short Voting Codes model. This model already fully included privacy-related aspects, and therefore adding these small extensions was sufficient to directly reach a proof in our new framework. The models that were more difficult to extend delivered either comparatively weak guarantees (e.g., Belenios allows to misattribute ballots, which made detection of the ballot of Alice and Bob by the framework difficult), or did not fully model privacy-related aspects in their verifiability model (e.g., the Swiss Post System model did not encrypt the votes in their verifiability model).

### B. Running example (Helios-like)

Our running example is based on [28], but adds a zero-knowledge proof to prevent ballot copying [1]. Further, to showcase the framework's capabilities, we implement a tally phase that supports a decryption shuffle. We note that the same structure can also be reused to support a reencryption shuffle. Due to the complexity of the tally in this example, compared to the other models, we had to further guide ProVerif by adding a couple of standard lemmas that say that messages in the left and right parts of the equivalence remain equal.

### C. Belenios

Belenios [27] is very close to our running example, but it adds the registrar as an additional authority. The registrar is in charge of delivering private credentials to the voters and publishing the corresponding public parts (and whom they belong to). To cast their ballot, the voter signs it using the registrar's private credential, and then sends the ballot over the authenticated channel to the server. The protocol is secure if either the server or the registrar is honest, by notably preventing ballot stuffing by either authority. However, it allows a malicious registration authority to cast a ballot in another voter's name. Concretely, the attacker can intercept an honest voter's ballot, and recast it over the authenticated channel of a malicious voter, while claiming that the public part of the credential of the honest voter actually belongs to the malicious voter. Thereby, the ballot is validated successfully by the honest server, who attributes it wrongly to the malicious voter. While this does not break privacy (as e.g., copying ballots is prevented), it is hard to prove in our framework, as we rely on detecting the honest voter's ballots to perform the swap.

We build upon the model of [17]. To ensure that the ballots of honest voters 1 and 2 are attributed correctly, we let our framework read the private credential used by honest voters 1 and 2 to attribute the ballots, and for all other ballots, allow any other attribution.

```
letfun get_ident_from_ballot(elec: elec_id, b: ballot) =
  let c = get_credential(b) in
  get credentials(=elec, id, =c)
    suchthat (id = 1 || id = 2) in ( id )
  else (
    get public_credentials(=elec, id, _) in ( id )
  )
```

We highlight that this function never blocks, which is important for soundness: `public_credentials` will always contain at least one entry, as the server uses this same table to check whether to add a given ballot to the ballot box. Concretely, the function delivers a superset of all real traces, by allowing to attribute a given ballot to any `id` instead of only the `id` where the corresponding public key matches. We choose this approach to avoid placing any assumption on the consistency of the attacker-controlled `credentials` and `public_credentials` tables. As ProVerif overapproximates the else branch (it does not negate the if condition), we need to increase its precision. We use the same technique as presented in [19], for which we provide a proof in our setting in Section B. For verifiability, we keep the trivial implementation that uses the public credential attached to the ballot to query `public_credentials` for the corresponding `id`.

### D. Swiss Post Protocol

The Swiss Post Protocol [29] is set in the demanding setting enforced by the Swiss chancellery [6], and bases its security

on return codes and a verifiable decryption shuffle in the tally phase. In a nutshell, an honest setup component prepares the election key material with four online components, of which one is honest. The setup component sends for each voting option a return code to the voter over the postal channel. The voter selects the candidates on their voting device, which sends the encryption thereof to the online components. These jointly derive the corresponding return code, and send it back to the voting device. Finally, if the return codes shown on the device match with the return codes received over the postal channel, the voter confirms their vote. After the voting phase, the confirmed ballots are verifiably shuffled and decrypted.

In this setting, there is an additional trust assumption for privacy: The voting device is assumed honest, as the voter directly selects their candidates on its screen. As it is not assumed honest for verifiability, it is not modelled in the existing verifiability model of [17]. Therefore, we need to first implement the voting device for the privacy proof. Further, the verifiability model omits the lexicographic ordering of the list with the return codes, hence deriving the return code would trivially leak the voter's choice, by their corresponding position in the list. ProVerif indeed immediately finds an attack. We therefore model the lexicographic ordering by making the ordering indistinguishable relative to `voteA` and `voteB` of the honest voters.

```
letfun sort(id: voter_id, return_codes: list3_t) =
  let (voteA, voteB, voteC) = return_codes in
  if (id = 1) then (
    (diff[voteA, voteB], diff[voteB, voteA], voteC)
  ) else if (id = 2) then (
    (diff[voteB, voteA], diff[voteA, voteB], voteC)
  ) else (voteA, voteB, voteC)
```

Further, we need to adapt the modelling of the tally slightly to be able to prove privacy. In the verifiability model, the adversary is given all decryption keys, and only the correctness of the decryption is checked. For privacy, we need to fully model the honest decrypting component, and cannot simply give its decryption key to the adversary, as then the adversary could trivially decrypt all ballots. It now however matters for the model which decryption authority is honest. When the first decryption authority is honest, we perform the decryption as before in `Decrypt`. However, when the second decryption authority is honest, we need to verify the proofs of the preceding (dishonest) decryption authority *before* the swap, in `Pre_Dec`. This looks as follows.

```
let Pre_HBal(elec: elec_id, id: voter_id, b: ballot) =
  let c = get_ciphertext(b) in
  in(ch_pub, (p, c'));
  if verify_zkp(pk(elec), p, (c, c')) then (
    out(res_pre_hballots(elec, id), wrap_m_ballot(c'))
  )

let Decrypt(elec: elec_id, id: voter_id, b: m_ballot) =
  let wrap_m_ballot(c') = b in
  let v = decrypt(sk(elec), c') in
  out(res_decrypt(elec, id), v)
```

If we would perform the zero-knowledge proof verification after the swap, they would fail for the honest ballots, as these have been swapped. For the same reason we highlight that the honest decryption authority does not create or publish a corresponding zero-knowledge proof in the model. Again, this zero-knowledge proof would make the swap (trivially) visible to the adversary, and therefore break the privacy proof. For our purposes, this is sound, as the zero-knowledge proof of the honest control component will always be correct and not reveal any other information.

### E. Short Voting Codes

The Short Voting Codes [20] protocol is also set in the Swiss setting. However, the voter enters voting codes, which the adversary cannot relate to the corresponding plain candidates. Consequentially, the voting device remains untrusted for privacy. When we take over the verifiability model [20], this simplifies our work, as we do not need to implement the voting device. Further, as the existing privacy model of [20] is close to the verifiability model, it is straightforward to integrate its existing capabilities (which prevent trivial attacks on the return codes table, as also necessary for the Swiss Post model). We are also able to directly take over the implementation of `Decrypt`, as the tally phase is not modelled (instead, the ballots on the bulletin board are simply decrypted). Overall, the model allowed to test our backwards-compatibility: besides not needing any non-superficial adaption, it also demonstrates that the execution time of our unified framework has not changed compared to the verifiability framework.

### F. Vote&Check

Vote&Check [19] is a verifiable postal voting protocol. Its individual verification relies on trackers: At the end of the election, each vote is published in plain together with a unique tracking code. Each voter may then check that their personal tracker appears next to their intended choice.

However, strictly speaking, the (verifiability) framework does not support voter verification *after* the tally. This verification may notably be done over *decrypted* votes. Concretely, the framework launches the user-defined verification process directly after the voter cast their vote. Further, the framework publishes the tallied votes simply to the public channel, and does not store them on a public bulletin board or similar.

For Vote&Check, the authors slightly abused the verifiability framework by performing the verification *before* the tally. This hack is possible thanks to the particular design of the protocol: it does not encrypt the votes, but assumes instead an anonymous postal channel. Therefore, the ballots do not need to be decrypted as part of the tally, and performing the verification before or after the tally is equivalent. This equivalence however no longer holds true for privacy, where the ballots of the honest voters have been swapped. This yields an interesting issue: swapping the honest clear votes after tally is necessary for privacy, however a verification mechanism that needs to read a particular line (cell) of the bulletin board immediately breaks this privacy. Hence, it is necessary to model that voters retrieve the full bulletin board, not a particular line, which first requires to extend the verifiability framework.

## VIII. Conclusion

We have provided a generic framework to prove both verifiability and privacy for voting protocols. We have successfully applied it to several protocols of the literature, notably Swiss Post, used in Switzerland for politically binding elections.

Our framework could not be applied however to the Vote&Check protocol, which highlights the fact that the verifiability framework, on which we build upon, does not cover protocols where voters verify *after* the tally. This is a case of a rather large class of protocols such as Select [44], Selene [45], or Hyperion [46], where voters check their vote in clear after the tally, thanks to a tracker. As future work, it would be interesting to study how to extend the framework to tracker-based protocols, both for verifiability and privacy.

To avoid trivial false attacks, our vote privacy definition assumes that honest ballots are always included in the tally. Alternatively, we could analyze privacy only for voters who check that their ballots are included in the tally. In case individual verifiability checks fail, privacy would be broken as well (as highlighted in [47]). So vote privacy would be defined under the restriction that honest voters successfully verified that their ballots are counted. This would probably require to characterize under which conditions equivalence properties with restriction [18] indeed model the desired property.

In order to more faithfully model verifiable mixers, we would need an equational theory that reflects the corresponding zero-knowledge proof. These proofs assert the equality of the plaintexts of two lists of encrypted ciphertexts. In our unbounded setting, these lists would however need to be of arbitrary size, which seems difficult to encode without using AC (associative and commutative) symbols, that are currently out of scope of ProVerif.

## References


[1] V. Cortier and B. Smyth, "Attacking and fixing helios: An analysis of ballot secrecy," in *2011 IEEE 24th Computer Security Foundations Symposium (CSF'11)*, 2011, pp. 297–311.

[2] 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'21)*, 2021, pp. 1–15.

[3] A. Sutopo, T. Haines, and P. Roenne, "On the auditability of the Estonian IVXV system and an attack on individual verifiability," in *Workshop on Advances in Secure Electronic Voting 2023*, 2023.

[4] J. Mueller, "Breaking and fixing vote privacy of the Estonian e-voting protocol IVXV," in *Workshop on Advances in Secure Electronic Voting*, 2022.

[5] V. Cortier, A. Debant, and P. Gaudry, "A privacy attack on the Swiss Post e-voting system," in *Real World Crypto Symposium (RWC'22)*. Amsterdam, Netherlands: IACR, 2022.

[6] Federal Chancellery, "Federal Chancellery Ordinance on Electronic Voting," https://www.fedlex.admin.ch/eli/cc/2022/336/en, December 2013.

[7] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, pp. 1–135, 2016.

[8] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *25th IEEE Computer Security Foundations Symposium (CSF'12)*, 2012, pp. 78–94.

[9] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *S&P'17*, 2017, pp. 483–502.

[10] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated analysis and verification of tls 1.3: 0-rtt, resumption and delayed authentication," in *IEEE Symposium on Security and Privacy (S&P'16)*, 2016.

[11] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of tls 1.3," in *24th ACM Conference on Computer and Communications Security (CCS'17)*, 2017.

[12] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5g authentication," in *25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[13] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *EuroS&P'17*. IEEE, 2017, pp. 435–450.

[14] S. Delaune, S. Kremer, and M. D. Ryan, "Verifying privacy-type properties of electronic voting protocols," *Journal of Computer Security*, vol. 17, no. 4, pp. 435–487, 2009.

[15] B. Blanchet and B. Smyth, "Automated reasoning for equivalences in the applied pi calculus with barriers," *Journal of Computer Security*, vol. 26, no. 3, pp. 367–422, 2018.

[16] D. Basin, J. Dreier, and R. Sasse, "Automated Symbolic Proofs of Observational Equivalence," in *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM, Oct. 2015, pp. 1144–1155.

[17] V. Cortier, A. Debant, and V. Cheval, "Election Verifiability with ProVerif," in *Computer Security Foundations Symposium (CSF'23)*, 2023.

[18] B. Blanchet, V. Cheval, and V. Cortier, "ProVerif with Lemmas, Induction, Fast Subsumption, and Much More," in *Symposium on Security and Privacy (S&P'22)*, 2022.

[19] V. Cortier, A. Debant, P. Gaudry, and L. Louistisserand, "Vote & check: Secure postal voting with reduced trust assumptions," in *Privacy Enhancing Technologies Symposium (PETS'25)*, 2025.

[20] V. Cortier, A. Debant, and F. Moser, "Code voting: when simplicity meets security," in *European Symposium on Research in Computer Security (Esorics'24)*. Springer, 2024, pp. 410–429.

[21] Swiss Post, "Swiss Post Voting System: Symbolic Analysis of the Swiss Post Voting System. Version 1.4.0," Swiss Post, Tech. Rep., 2025. [Online]. Available: https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/tree/master/Symbolic-models?ref_type=heads

[22] 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.

[23] L. Hirschi and C. Cremers, "Improving automated symbolic analysis for e-voting protocols: A method based on sufficient conditions for ballot secrecy," in *EuroS&P'19*, 2019.

[24] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis, "CHVote Protocol Specification." *Cryptology ePrint Archive*, 2017. [Online]. Available: https://eprint.iacr.org/2017/325.pdf

[25] V. Cortier, A. Debant, and P. Gaudry, "Chvote: breaking privacy," in *European Symposium on Research in Computer Security (Esorics'25*. Springer, 2025.

[26] A. Juels, D. Catalano, and M. Jakobsson, "Coercion-resistant electronic elections," in *WPES'05*. ACM, 2005, pp. 61–70.

[27] V. Cortier, P. Gaudry, and S. Glondu, "Belenios: a simple private and verifiable electronic voting system," *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows*, 2019.

[28] B. Adida, "Helios: Web-based Open-Audit Voting," in *USENIX*, 2008.

[29] Swiss Post, "Swiss Post Voting System: System Specification. Version 1.1.1," Swiss Post, Tech. Rep., October 2022. [Online]. Available: https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/blob/afbd7555b3f87092596a5203f022c00dcc0fd390/System/System_Specification.pdf

[30] J. Benaloh, "Verifiable secret-ballot elections," Ph.D. dissertation, Yale University, 1987.

[31] S. Kremer and M. Ryan, "Analysis of an electronic voting protocol in the applied pi calculus," in *European Symposium on Programming (ESOP'05)*. Springer, 2005, pp. 186–200.

[32] 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, pp. 327–344.

[33] 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)*, June 2019.
[34] R. Kuesters, T. Truderung, and A. Vogt, "Clash attacks on the verifiability of e-voting systems," in *S&P'12*. IEEE, 2012.
[35] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Proc. 28th ACM Symp. on Principles of Programming Languages (POPL'01)*. ACM Press, 2001, pp. 104–115.
[36] D. Wikström, "A universally composable mix-net," in *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004*, ser. Lecture Notes in Computer Science, M. Naor, Ed., vol. 2951. Springer, 2004, pp. 317–335.
[37] J. Groth, "A verifiable secret shuffle of homomorphic encryptions," *Journal of Cryptology*, 2010.
[38] S. Bayer and J. Groth, "Efficient zero-knowledge argument for correctness of a shuffle," in *EUROCRYPT 2012*, ser. Lecture Notes in Computer Science, 2012.
[39] R. Küsters, J. Liedtke, J. Müller, D. Rausch, and A. Vogt, "Ordinos: A verifiable tally-hiding e-voting system," in *IEEE European Symposium on Security and Privacy, (EuroS&P 2020)*, 2020, pp. 216–235.
[40] V. Cortier, P. Gaudry, and Q. Yang, "A toolbox for verifiable tally-hiding e-voting systems," in *ESORICS 2022 - 27th European Symposium on Research in Computer Security*, Copenhague, Denmark, 2022.
[41] V. Cheval, V. Cortier, and M. Turuani, "A little more conversation, a little less action, a lot more satisfaction: Global states in proverif," in *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF'18)*, 2018, pp. 344–358.
[42] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, "Proverif 2.00: automatic cryptographic protocol verifier, user manual and tutorial," *Version from*, vol. 16, pp. 05–16, 2018.
[43] H. Comon-Lundh and S. Delaune, "The finite variant property: How to get rid of some algebraic properties," in *Term Rewriting and Applications, 16th International Conference, (RTA'05)*, 2005.
[44] R. Küsters, J. Müller, E. Scapin, and T. Truderung, "sElect: A lightweight verifiable remote voting system," in *CSF*. IEEE, 2016.
[45] P. Ryan, P. Rønne, and V. Iovino, "Selene: Voting with transparent verifiability and coercion-mitigation," in *VOTING (FC workshop)*. Springer, 2016.
[46] P. Ryan, P. Roenne, and S. Rastikian, "Hyperion: An enhanced version of the Selene end-to-end verifiable voting scheme," in *EvoteID*, 2021.
[47] V. Cortier, J. Lallemand, and B. Warinschi, "Fifty shades of ballot privacy: Privacy against a malicious board," in *33rd IEEE Computer Security Foundations Symposium (CSF'20)*, Boston / virtual, USA, 2020.
[48] V. Cheval and B. Blanchet, "Proving more observational equivalences with proverif," in *Principles of Security and Trust - Second International Conference, POST 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, ser. Lecture Notes in Computer Science, D. A. Basin and J. C. Mitchell, Eds., vol. 7796. Springer, 2013, pp. 226–246. [Online]. Available: https://doi.org/10.1007/978-3-642-36830-1_12

## APPENDIX

### A. Running example: The rest of the framework interface

In the setup, we use the default implementations, which semantically do nothing.

```
(* nothing known by voter beforehand *)
letfun gen_initial_vd(elec: elec_id, id: voter_id) = ()

(* take over (empty) value from gen_initial_vd *)
let Gen_VD(elec: elec_id, id: voter_id, vd: voter_data) =
  insert voting_data(elec, id, vd)
```

The voter verification checks whether the exact ballot as cast is contained on the bulletin board.

```
let Verify(elec, id: voter_id, v: vote, b: ballot) =
  in( cell_BB (elec, id), =b);
  event Verified(elec, id, v);
  out( cell_BB (elec, id), b)
```

Finally, we implement the bulletin board functions as expected, mostly using trivial implementations.

```
(* extract v_id from the tuple in ballot_box. *)
letfun get_id(elec: elec_id, b: ballot) =
  let (id,_) = b in id

(* ballots on the bulletin board have a valid ZKP *)
letfun is_valid(elec: elec_id, b: ballot) =
  let (id, (c, p)) = b in verify_zkp(p, (c, id))

(* store the full ballot on the bulletin board. *)
letfun update_BB(elec: elec_id, b: ballot) = b

(* leave the public bulletin board empty. *)
letfun update_gBB(elec: elec_id, b: ballot) = empty
```

### B. Proving soundness of the axiom

In order to prevent over-approximations when verifying protocols with `else` branches, we need to refine our ProVerif model. In the spirit of [19], we follow a hybrid approach based on a pen-and-paper proof and additional lemmas proved by ProVerif. In the following we sketch this the approach, while we refer interested readers to [19] for the precise details.

*a) Main idea:* To prevent ProVerif from over-approximating `else` branches, we define a generic axiom enforcing the `else` branch semantics in `get` commands. For sake of simplicity, we state it for `get` commands of the form: `get tab(x1,...,xn) suchthat xi = y in P else Q` where `y` is a bound variable.

We introduce two events: `Inserted(x)`, which is placed each time an entry `x` is added in the table, and `NotFound(y)`, which is placed in the `else` branch with `y` being the bound variable occurring in the get condition. Finally, we add the following axiom[3]:

```
axiom
  event(Inserted(x)) && event(NotFound(y)) && x = y
    ==> false
```

*b) Proof Sketch:* Because ProVerif does not check soundness of axioms, we do a pen-and-paper proof to prove it. We manually prove that if some simple properties hold, then our main axiom holds as well. These simple properties are proven as lemmas with ProVerif.

To do this proof, we need to define extra events and weaken the axiom we declare in ProVerif. Concretely, we define two extra events: $\texttt{CreatedFresh}(x, r)$ is triggered as soon as the fresh value $r$ is created with `new` $r$ in a context $x$, and $\texttt{Received}(b)$ when the message $b$ is received at a specific step of the processes. In the spirit of GSVerif [17], [41], we will then be able to guarantee that if $r$ is a subterm of $b$, denoted $\mathsf{subterm}(r, b)$, then $\texttt{CreatedFresh}(x, r)$ must be executed before $\texttt{Received}(b)$ in any execution trace of the protocol. Moreover, we extend the `NotFound` event to record an extra value $b$ used when executing the `get` command leading to the `else` branch.

Finally, we define the following lemmas to prove our main axiom which we recall as well:

- lemma 1: $\texttt{Received}(b) \wedge \texttt{CreatedFresh}(x, r) \wedge \mathsf{subterm}(r, b) \implies \texttt{Inserted}(x)$
- lemma 2: $\texttt{NotFound}(x, b) \implies \texttt{Received}(b)$

[3]The axioms defined in the ProVerif models are actually weaker according to the changes explained just after.

- axiom: Inserted($x$) $\wedge$ CreatedFresh($x$, $r$) $\wedge$ NotFound($y$, $b$) $\wedge$ subterm($r$, $b$) $\wedge$ $x = y$ $\implies$ false

The proof is done by contradiction. Assuming that the axiom does not hold, but the two lemmas do, we prove that we can derive a contradiction. Concretely, given a execution trace which falsifies the axiom, we can derive that it must contain the following events:

| | |
|---|---|
| Inserted(x)@t1 $\wedge$ | (by $\neg$ axiom) |
| CreatedFresh(x, r)@t2 $\wedge$ | (by $\neg$ axiom) |
| NotFound(x, b)@t3 $\wedge$ | (by $\neg$ axiom) |
| subterm(r,b) $\wedge$ | (by $\neg$ axiom) |
| Received(b)@t4 $\wedge$ t4 $<$ t3 | (by lemma 2) |
| Inserted(x)@t5 $\wedge$ t5 $<$ max(t2, t4) | (by lemma 1) |

We now show that such an execution trace cannot exist:

- We first observe that max(t2, t4) = t4. As $b$ contains the fresh term $r$ by subterm$(r, b)$, and there exists CreatedFresh($x$, $r$)@t2, based of GSVerif-like reasoning [17], [41], we have that CreatedFresh($x$, $r$)@t2 occurs before Received($b$)@t4. Hence, t2 $<$ t4.
- Combining this with lemma 2, which asserts that t4 $<$ t3, we can conclude that t5 $<$ t4 $<$ t3. However, semantically, NotFound($x$, $b$)@t3 is not accessible if Inserted($x$)@t5 has been raised before. This leads to the contradiction.

Hence, either one of the two lemma does not hold, or the axiom holds. Since the lemmas are proven by ProVerif, we can conclude that the axiom holds. □

*c) Proving lemmas:* Following the reasoning presented in [19], the two lemmas previously defined need to be proven in a model without the axiom. Unfortunately, when proving equivalence queries, this leads to non-termination issues.

In order to overcome this limitation, we decided to slightly restrict our model: we only consider traces in which the setup and the voting phases are disjoint, i.e., all the events Inserted($x$) are executed before all the events CreatedFresh($x$, $r$). In our model, event Inserted($x$) corresponds to the registration of the voter public signing key, while the event CreatedFresh($x$, $r$) is the generation of the randomness used to form the ballot. When considering an implementation of the Belenios protocol in practice, this restriction seems very realistic.

```
1  restriction
2    event(Inserted(x))@t1 && event(CreatedFresh(x, r))@t2
3      ==> t1 < t2.
```

Assuming such a model restriction and that lemma 2 holds (it is syntactically true) then we can prove that all traces which might falsify lemma 1 must satisfy the axiom, with the same reasoning as above.

Hence, by contraposition, it means that any trace that does not satisfy the axiom must satisfy lemma 1. So if we prove now, with ProVerif, that any trace that satisfies the axiom (stated as a restriction) satisfy lemma 1, we can conclude that all traces indeed satisfy lemma 1, whether or not they satisfy the axiom. This allows to conclude that lemma 1 holds, hence the axiom.

### C. Proving soundness of query transformation

In this section we formally define the semantics of our extended queries that support Destructors and Letfun. In addition we show that the automatic transformation from our extended queries to the queries of Vanilla ProVerif is sound and complete. This shows that we do not need to modify ProVerif internal algorithms for proving a query and that our extended queries can be seen as a front-end to Vanilla ProVerif.

*1) Semantics of Vanilla ProVerif:* We recall here some of the notions from [18] that are relevant to describe the syntax and semantics of correspondence queries in Vanilla ProVerif. We may omit some details when they are independent of our extension.

$$\begin{aligned} F ::= & \mathsf{event}(ev) \\ & \mathsf{inj}_k\text{-}\mathsf{event}(ev) \\ & \mathsf{attacker}(M) \\ & \mathsf{mess}(M, N) \\ & \mathsf{table}(tbl) \end{aligned}$$

where $M, N$ are constructor terms, that are terms that only contains constructor function symbols, names, and variables. Similarly, $ev$ and $tbl$ are respectively events and tables, which are constructor terms with a special function symbol as root representing either the triggered event or the entry of a table.

In addition to facts, ProVerif also allows for some generic formulas, consisting in particular of

$$\phi ::= \forall \tilde{x}.(M \neq N) \mid M = N$$

where $\tilde{x}$ is a set of universally quantifier variables. Notice that all these generic formulas once again only contain constructor terms.

The semantics of facts and generic formulas is defined by a relation $\vDash$ over traces $T$ and ground facts and generic formulas, which naturally follows the intuition behind all facts. For example, $T \vDash \mathsf{event}(ev)$ holds when the execution trace $T$ has triggered the event $ev$, and $T \vDash \mathsf{attacker}(M)$ holds when $M$ can be deduced from the knowledge of the attacker at the end of the trace $T$. For generic formulas, the relation is in fact independent of $T$ as it only focuses on terms. For instance, $T \vDash M = N$ if and only if $M$ is equal to $N$ modulo the equational theory $E$. The exact definition of the relation $\vDash$ can be found in [18]. The relation $\vDash$ is then naturally extended to boolean conjunction/disjunction over facts and generic formulas.

A (non-injective) correspondence query is thus a query of the form $F_1 \wedge \ldots \wedge F_n \Rightarrow \psi$ where $\psi$ is a boolean disjunction and conjunction over facts and generic formulas. Its semantics are given as follows: The query $F_1 \wedge \ldots \wedge F_n \Rightarrow \psi$ holds when for all traces $T$, for all substitutions $\sigma$, if $T \vDash F_1\sigma \wedge \ldots \wedge F_n\sigma$ then there exists $\sigma'$ such that:

- for all $x \in \mathrm{vars}(F_1, \ldots, F_n)$, $x\sigma =_E x\sigma'$
- $T \vDash \psi\sigma'$

*2) Destructors:* Destructors as described in this paper have a semantics in ProVerif based on reduction rules. In the previous section, we provided as example the destructor:

```
82    reduc forall elec, m, r;
83       decrypt(sk(elec),encrypt(pk(elec),m,r)) = m.
```

defined based on one single rewrite rule. The evaluation of a destructor is then given by matching its arguments to the arguments of the rewrite rule. For example, $\mathrm{decrypt}(M_1, M_2)$ evaluates to $m\sigma$ when $M_1 =_E \mathrm{sk}(\mathrm{elec})\sigma$ and $M_2 =_E \mathrm{encrypt}(\mathrm{pk}(elec\,\sigma), m\sigma, r\sigma)$ where $elec$, $m$ and $r$ are taken as fresh variables. When these conditions are not met, the (evaluation of the) term $\mathrm{decrypt}(M_1, M_2)$ fails.

In all generality, the semantics of destructors in ProVerif are given by a list of rewrite rules that are evaluated in order before failing. This was introduced in [48, Figure 2] under a semantically equivalent form with the notion of a set of rewrite rules with constraints. Importantly, [48] provides two evaluation relations on terms with destructors: *close and open evaluation*. For a term $D$ containing possibly destructors, the *close evaluation* of $D$ either yields a constructor term $M$, denoted $D \Downarrow M$, or fails, denoted $D \not\Downarrow$ (or $D \Downarrow$ **fail**) ( [48, Figure 2]). Close evaluation is intuitively used to evaluation ground terms, that are terms without variables.

Open evaluation of a term $D$ [48, Section 4.4] intuitively evaluates $D$ into all possible ways an instantiation $D$ could be rewritten. Formally, the open evaluation is a relation $\Downarrow'$ between a term $D$ and a tuple $(M, \sigma, \phi)$, denoted by $D \Downarrow (M, \sigma, \phi)$ where $M$ is a constructor term, $\sigma$ is a substitution and $\phi$ are conjunctions of generic inequality formulas. It typically indicates if $D$ is instantiated by $\sigma$ then it will be evaluated in $M$ provided that $\phi$ holds.

Crucially, the link between open and close evaluation is given by the following two results [48]:

**Proposition 1.** *For all expressions $D$, for all ground substitutions closing $D$, if $D\sigma \Downarrow M$ then there exists $D \Downarrow' (M', \sigma_1, \phi)$ and $\sigma'_1$ such that $M =_E M'\sigma'_1$, $\sigma =_E \sigma_1\sigma'_1$ and $\phi\sigma'_1$ is true.*

**Proposition 2.** *For all expressions $D$, if $D \Downarrow' (M, \sigma, \phi)$ then for all $\sigma'$ such that $\phi\sigma'$ is true then $D\sigma\sigma' \Downarrow M\sigma'$.*

*3) Extended syntax / semantics of our queries:* .

A mentioned in Section VI-B, a query $F_1 \wedge \ldots \wedge F_n \Rightarrow \Psi$ can now contain destructors in its conclusion $\Psi$. We impose a restriction over destructors within inequalities $\forall \tilde{x}.(D \neq D')$: no universally quantified variables in $\tilde{x}$ can occur in arguments of destructors in $D$ and $D'$.

The semantics of facts is a natural extension of the semantics over terms. For example,

- $T \models \mathsf{attacker}(D)$ when there exists $M$ such that $D \Downarrow M$ and $T \models \mathsf{attacker}(M)$
- $T \models \mathsf{mess}(D, D')$ when there exist $M, N$ such that $D \Downarrow M$, $D' \Downarrow N$ and $T \models \mathsf{mess}(M, N)$
- $T \models \mathsf{table}(D)$ when there exists $tbl$ such that $D \Downarrow tbl$ and $T \models \mathsf{table}(tbl)$

The semantics of generic formulas follows the same principle:

- $T \models D = D'$ when there exists $M, N$ such that $D \Downarrow M$, $D' \Downarrow N$ and $T \models M = N$
- $T \models \forall \tilde{x}.(D \neq D')$ when there exist $M, N$ such that $D \Downarrow M$, $D' \Downarrow N$ and $T \models \forall \tilde{x}.(M \neq N)$

From the semantics and the previous two propositions, we can always transform an extended query into an equivalent query for Vanilla ProVerif by replacing every fact or generic formula with a disjunction over their open evaluation. Formally every occurrence of the fact $\mathsf{attacker}(D)$ in $\Psi$ is replaced by

$$\bigvee_{D\Downarrow'(M,\sigma,\phi)} \mathsf{attacker}(M) \wedge \phi \wedge \bigwedge_{x\in dom(\sigma)} x = x\sigma$$

Every occurrence of $\mathsf{mess}(D, D')$ in $\Psi$ is replaced by

$$\bigvee_{(D,D')\Downarrow'((M,N),\sigma,\phi)} \mathsf{mess}(M, N) \wedge \phi \wedge \bigwedge_{x\in dom(\sigma)} x = x\sigma$$

This transformation applies to all other facts and generic formulas. For example every occurrence of $D = D'$ is replaced by:

$$\bigvee_{(D,D')\Downarrow'((M,N),\sigma,\phi)} M = N \wedge \phi \wedge \bigwedge_{x\in dom(\sigma)} x = x\sigma$$

The equivalence between these formulas in our semantics are directly given by Propositions 1 and 2.

### D. Soundness of the swap technique

In this section we prove the soundness of the transformation described in Section V-C. To do so, we consider the following processes[4]:

- `Tally()` which describes the tally process and is recalled in Figure 5. It builds upon two protocol-dependent processes, namely `Decrypt(i,b)` and `Verify_Decrypt(i,c)`, described in Section V-C. For the sake of generality, we make as little assumptions as possible on these two processes.
- `Count()` described in Section V-C and whose purpose is to process all the votes and raise the event `Counted(v)`, necessary to state the verifiability queries.
- `S` an arbitrary process which models the rest of the system.

In order to prove the soundness of our transformation, we make a few assumptions about the different processes:

1) channels `cell_tally` and `cell_result` are private channels which do not appear elsewhere in `S`, except once for initialization. Moreover, they are initialized at the same time and with the same value.
2) `res_decrypt(i)` is a private channel which does not appear elsewhere in `S`. Moreover, only `Verify_Decrypt(i,c)` contains an output on this channel, as its final action.

[4]Since all the channels and processes take `elec`, the election identifier, as argument, w.l.o.g, we omit it and consider each symbol different for each election.

```
let Tally() =
  ! (
      in(cell_tally, i:nat);
      if i = 0 then (
        ! get log_i_c(i, c) in
          ( out(ch_pub, c) | Verify_Decrypt(i, c) )
      ) else
        Q;
        insert log_i_c(i, c);
        out(cell_tally, i-1)
  ).

let Count() =
  ! (
      in(cell_result, i:nat);
      if i = 0 then event Finish()
      else
        in(res_decrypt(i), v:vote);
        event Counted(v);
        out(cell_result, i-1)
  ).
```

Fig. 5. Our ideal processes to model the tally phase where `Q` is a linear process (i.e., without `else` branch).

3) process `Verify_Decrypt(i,c)` does not contain an input on private channels, nor a `get` command, nor an `insert` command, nor an output except the one on the channel `res_decrypt(i)` (see above).
4) events occurring in `Tally()`, `Count()`, `Decrypt(i,b)`, and `Verify_Decrypt(i,c)` do not interfere with the semantics (i.e., do not appear in restrictions), except the event `Finish()`.
5) the indices recorded by the table `honest_voters_index(i1,i2)` (introduced in Section IV-B) are not greater than the initialization value on the channel `cell_tally`.

Even if these assumptions may seem restrictive, all our case studies satisfy them. Moreover, they are easy to check: items 3 and 4 can be entirely syntactically checked, while items 1 and 2 can be partly syntactically checked and partly verified by ProVerif itself by defining secrecy lemmas (called *secrecy assumption* in the ProVerif manual). Finally, item 5 is ensured by design in our framework (it could also be proved by ProVerif too).

Given a bi-process $P$, we note $\mathtt{fst}(P)$ (resp. $\mathtt{snd}(P)$) the process obtained by replacing in $P$, all the bi-terms $\mathtt{diff}[u_1, u_2]$ by $u_1$ (resp. $u_2$).

**Proposition 3.** *Given bi-processes `S`, `Tally()`, `Count()` (as introduced above). We have:*

$$\mathtt{fst}(S \mid Tally'()) \approx_o \mathtt{snd}(S \mid Tally'()) \Rightarrow \mathtt{fst}(S \mid Tally() \mid Count()) \approx_o \mathtt{snd}(S \mid Tally() \mid Count())$$

*where `Tally'()` is defined in Figure 6.*

*Proof.* The main differences between the two bi-processes are: (1) the process `Count()` is removed, and (2) the event `Finish()` is placed at the end of the decreasing loop on `cell_tally` (while it was at the end of the loop on `cell_result` previously).

First, one can note that the process `Count()` does not contain possibly visible actions except the two events `Finish()` and `Counted(v)`. Indeed, it only contains communications on private channels and conditionals which are invisible actions when considering the standard observational equivalence definition, as defined in [7] for instance.

Moreover, among the two events, `Counted(v)` is an invisible action too. Indeed, by assumption (4) the event does not appear in restrictions, and thus does not impact the semantics. We thus deduce that it cannot interfere with the observational equivalence of the two processes.

Finally, only `Finish()` may be a visible action. However, this action is not removed from the global bi-process. Instead, it is placed in the `Tally'()` process. Let us focus on the execution of this action to prove the proposition by induction on the length of the trace: let $tr = tr_1.....tr_n$ be an execution trace of `S|Tally()|Count()` such that $tr_n$ is the execution of the event `Finish()`. Let $tr' = tr'_1.....tr'_m$ be an execution trace of `S|Tally'()` with the same observables as $tr_1.....tr_{n-1}$. Let us show that there exist $tr'_{m+1}, \ldots, tr'_{m+p}$ such that $tr'_1.....tr'_{m+p}$ is an execution trace of `S|Tally'()` and such that $tr'_{m+p}$ execute the `Finish()` event.

Because $tr_n$ executes the `Finish` event, and because it is a private channel (assumption (1)), we know that the loop on `cell_result` has been fully unfolded, i.e., an input `in(cell_result,0)` has been executed.

$\Rightarrow$ an output `out(cell_result,i)` has been executed as well for all $i \geq 0$.

$\Rightarrow$ the input on `res_decrypt(i)` has been executed for all $i > 0$.

$\Rightarrow$ because only the process `Verify_Decrypt(i,c)` executes an output on `res_decrypt(i)` (assumption (2)), and because `cell_tally` is a private channel (assumption (1)), we know that the process `Tally()` has unfolded the decreasing loop to reach the case `i = 0`. Therefore, following the exact same trace, w.l.o.g. we can assume that the process `Tally'()` has unfolded the loop too.

$\Rightarrow$ the event `Finish()` can be executed too as it is placed in parallel with all the `Verify_Decrypt(i,c)` processes.

Finally, there exist $tr'_{m+1}, \ldots tr'_{m+p}$ such that $tr'_{m+p}$ executes the event `Finish` and $tr'_{m+1}, \ldots tr'_{m+p-1}$ are invisible actions. The two bi-processes have the exact same set of execution traces, up to invisible actions.

This ends the proof of the proposition. □

Even if the first transformation eases the verification by removing an extra process, it is not sufficient to let ProVerif prove equivalence of the two processes. Indeed, ProVerif proves *diff-equivalence*, a stronger notion of equivalence which, intuitively, requires indistinguishability of the two processes assuming the same scheduler. In order to avoid false attacks we apply a second transformation which applies a *swap*.

```
let Tally'() =
  ! (
    in(cell_tally,i:nat);
    if i = 0 then (
      event Finish() |
      ! get log_i_c(i,c) in
        ( out(ch_pub, c) | Verify_Decrypt(i,c) )
    ) else
      Q;
      insert log_i_c(i,c);
      out(cell_tally, i-1)
  ).
```

Fig. 6. `Tally'()`: first transformation to make the analysis possible

This transformation relies on the fact that $P \mid Q \approx_o Q \mid P$.

**Proposition 4.** *Given bi-processes* `S`, `Tally'()` *(as introduced above). We have:*

$$\mathtt{fst}(S \mid \mathit{Tally''}()) \approx_o \mathtt{snd}(S \mid \mathit{Tally''}()) \Rightarrow \mathtt{fst}(S \mid \mathit{Tally'}()) \approx_o \mathtt{snd}(S \mid \mathit{Tally'}())$$

*where* `Tally''()` *is defined in Figure 7.*

```
let Tally''() =
  get honest_voters_index(i1, i2) in
  !(
   in(cell_tally, i:nat);
   if i = 0 then (
     event Finish() |
     (
       get log_i_c(=i1, c1) in
       get log_i_c(=i2, c2) in
         out(diff[c1,c2]) |
         out(diff[c2,c1]) |
         Verify_Decrypt(diff[i1,i2],diff[c1,c2]) |
         Verify_Decrypt(diff[i2,i1],diff[c2,c1])
     ) |
     ! get log_i_c(i,c) suchthat i <> i1 && i <> i2 in
       ( out(ch_pub,c) | Verify_Decrypt(i,c) )
   )
   else
     Q;
     insert log_i_c(i, c);
     out(cell_tally, i-1)
  )
```

Fig. 7. `Tally''()`: second transformation to make the analysis possible where `aux_swap` is a fresh private channel for each election and `HonestVotersIndices(.,.)` is a table populated exactly once (before any output on `cell_tally`) by two distinct natural numbers not greater than the max value sent over `cell_tally`.

*Proof.* We first notice that the process

```
! get log_i_c(i,c) in
( out(ch_pub,c) | Verify_Decrypt(i,c) )
```

is structurally equivalent to the process

```
(
  ! get log_i_c(=i1, c1) in
    ( out(ch_pub, c1) | Verify_Decrypt(i1, c1) )
) | (
  ! get log_i_c(=i2, c2) in
    ( out(ch_pub, c2) | Verify_Decrypt(i2, c2) )
) | (
  ! get log_i_c(i, c) suchthat i <> i1 && i <> i2 in
    ( out(ch_pub, c) | Verify_Decrypt(i, c) )
)
```

Indeed, this last process solely particularizes the replications corresponding to the two specific indices `i1` and `i2`.

Second, we notice that `i1` and `i2` are not greater than the initial value on `cell_tally`. Hence, by the design of the `Tally'()` process, we know that two entries `log_i_c(i1, c1)` and `log_i_c(i2, c1)` have necessarily been inserted. Therefore, because the execution of a `get` command is an invisible action, the process

```
(
  ! get log_i_c(=i1, c1) in
    ( out(ch_pub, c1) | Verify_Decrypt(i1, c1) )
) | (
  ! get log_i_c(=i2, c2) in
    ( out(ch_pub, c2) | Verify_Decrypt(i2, c2) )
)
```

is equivalent (i.e., it has the same set of traces, up to invisible actions) to

```
! get log_i_c(=i1, c1) in
  get log_i_c(=i2, c2) in
    ( out(ch_pub, c1) | Verify_Decrypt(i1, c1) ) |
    ( out(ch_pub, c2) | Verify_Decrypt(i2, c2) )
```

Finally, we conclude the proof by referring back to the initial remark "$P \mid Q \approx_o Q \mid P$", which justifies the use of the `diff` symbols (one can convince themselves that, by projecting on the first (resp. second) component, we obtain the same processes put in parallel.). □

The following corollary justifies the soundness of our transformed bi-process.

**Corollary 1.** *Given bi-processes* `S`, `Tally()`, `Count()` *(as introduced above). We have:*

$$\mathtt{fst}(S \mid \mathit{Tally''}()) \approx_o \mathtt{snd}(S \mid \mathit{Tally''}()) \Rightarrow \mathtt{fst}(S \mid \mathit{Tally}() \mid \mathit{Count}()) \approx_o \mathtt{snd}(S \mid \mathit{Tally}() \mid \mathit{Count}())$$