Belenios with cast as intended

Véronique Cortier, Alexandre Debant, Pierrick Gaudry, Stéphane Glondu

Université de Lorraine, CNRS, INRIA, LORIA

Abstract. We propose the BeleniosCaI protocol, a variant of Belenios which brings the cast-as-intended property, in addition to other existing security properties. Our approach is based on a 2-part checksum that the voting device commits to, before being challenged to reveal one of them chosen at random by the voter. It requires only one device on the voter's side and does not rely on previously sent data like with return codes. Compared to the classical Benaloh audit-or-cast approach, we still have cast-as-intended with only some probability, but the voter's journey is more linear, and the audited ballot is really the one that is cast. We formally prove the security of BeleniosCaI w.r.t. end-to-end verifiability and privacy in a symbolic model, using the ProVerif tool.

1 Introduction

Electronic voting systems aim at guaranteeing simultaneously vote privacy (no one should know what/whom I voted for) and verifiability (my vote should be properly counted). These properties come with trust and distrust assumptions on the involved parties such as the voting server, the voting devices, the decryption authorities, or external auditors. A long-standing issue is the so-called *cast-as-intended* property: a voter should be able to control that their vote has been properly encoded and cast with their proper intention, even if their voting device tries to send a different vote.

A simple and appealing approach has been proposed by Benaloh [4] about 15 years ago. When a voter selects a vote v, their device produces a ballot b and the voter is given the choice to either cast the ballot or audit it. In the latter case, the device must produce the randomness used to form the ballot. The randomness as well as the ballot b are then sent to a second device or a third trusted party to control that b indeed encrypts v. This procedure is repeated an unpredictable number of times until the voter is convinced that their voting device behaves as expected. This approach is simple and versatile. However, user studies have shown that it is hard to use and understand in practice [26]. As a consequence, a few voters audit in real elections. Moreover, this mechanism may even threaten privacy in case voters are not properly instructed to audit ballots that contain votes that are independent of their real intention. For the Benaloh's mechanism to be truly secure, voters need to first roll a dice to decide whether they will vote or audit, and in the latter case, roll a dice to decide which candidate to use.

Our contribution. We propose an audit mechanism that is part of the voter's journey (no need to choose) and in which the audited ballot is the one actually cast. Our approach works as follows. When a voter selects a vote v, an integer a is chosen at random (between 1 and μ, a positive integer larger than the number of possible values for v) and the voter is given v, a, and b such that $b = v + a \bmod \mu$. The corresponding ballot bal is formed of the respective encryption of v, a, and b, together with a zero-knowledge proof zkp that guarantees that the three ciphertexts encrypt values x, y, z such that $z = x + y \bmod \mu$.

$$\mathsf{bal} = \mathsf{enc}(v), \mathsf{enc}(a), \mathsf{enc}(b), \mathsf{zkp}$$

The voting device commits to bal on the bulletin board. Then the voter asks their device to open either the second or the third encryption (chosen at random), by revealing the randomness used to produce the ciphertext. In order to modify v, the voting device needs to modify either a or b. Hence the voter will detect a malicious device with probability $1/2$. This audit mechanism does not leak any information on v since a and b (considered separately) are perfectly random.

We integrate this mechanism into the Belenios protocol [13], yielding BeleniosCaI. Belenios is an evolution of Helios [2] that additionally provides eligibility verifiability. Belenios is used each year in about 2000 elections, that include a German political party and some EU institutions [14]. We believe that our approach could be used to add cast-as-intended to other protocols as well. Interestingly, the computation overhead remains affordable. We show that we can encode the zero-knowledge proofs of modular equality into a standard proof of set membership. Overall, the computational cost and the size of ballots are about 2-3 times bigger than their counterparts in Belenios.

We formally prove that BeleniosCaI guarantees verifiability against a compromised voting device. We also show that BeleniosCaI preserves vote secrecy (assuming an honest voting device). Specifically, we provide a formal model of BeleniosCaI using the ProVerif tool [5], a well established tool for analyzing the security of protocols. This required to reflect the theory of modular arithmetic in ProVerif, which is typically out of range of this tool. Fortunately, we could re-use the model developed in [7]. Verifiability is shown by excluding traces that do not satisfy some of the properties of modular arithmetic, using restrictions, a feature recently introduced in ProVerif [6]. Vote privacy is more involved since it is expressed as an equivalence property where the attacker tries to distinguish between different voting choices. Considering sufficient conditions is no longer appropriate. Instead, we show (in ProVerif) lemmas on traces that then allow us to conclude thanks to a theorem of [7].

Related work. The idea of using a two-checksum a and b with $b = v + a \bmod \mu$ has been firstly developed by Neff in [28] for the specific subcase $\mu = 2$. Then, the general case has been sketched in [11] and more thoroughly introduced in [7] in the context of on-site voting. The proposed system involves printed papers with scratch-off parts, smartcards, and local observers. We adapt this idea to Internet voting, using the Belenios protocol, with many simplifications (e.g. we no longer request a and b to be of different parity).

Many Internet voting protocols have been proposed for cast-as-intended. We refer the reader to [27] for a nice and comprehensive survey. This survey splits existing protocols into five categories: audit-or-cast, tracking data, verification devices, code sheets, delegation. Interestingly, our approach introduced a novel category that could be called audit-and-cast. We only review here the main families. In the code sheet approach, voters obtain printed code sheets during the setup, with one code assigned to each candidate. When voting for a candidate v, the voter receives a code and checks on their sheet that it indeed corresponds to v. This is the approach followed in Switzerland with CHVote [20], as well as the protocol developed by SwissPost [1] or previously by Scytl [18]. Such systems require a heavy infrastructure (e.g. a distributed voting server) and assume a honest printer that is in charge of most of the setup. Other code-sheet based systems include Demos [24] and Pretty Good Democracy [31].

Some systems assume a (second) verification device. This is for example the case in Estonia [22] where voters can use a second device and the randomness of their ballot to check it is well-formed. In Australia [8], voters simply call a server that opens their ballot and gives the vote in clear (with associated threats on vote privacy). Intuitively, BeleniosCaI allows to simplify so much the work of the second device (checking an addition) that it can be discharged on the voter.

Tracking systems include in particular sElect [25], Selene [30] and its successor Hyperion [29]. They let the voter check that their vote appears on the public bulletin board, thanks to a voting tracker. The voter may detect a misbehavior only once the election is over. In Selene and Hyperion, the validity of the tracker needs to be checked with a second device (which, in return offers some protection against vote-buying).

2 Protocol description

We describe a voting protocol, BeleniosCaI, where voters have to select between k_1 and k_2 candidates among a list of n candidates. A vote can be represented as a vector $(v_i)_{1 \le i \le n}$, where $v_i = 1$ if candidate i has been selected and $v_i = 0$ otherwise; the condition $\sum v_i \in [k_1, k_2]$ must be satisfied. An overview of the protocol is presented in Figure 1.

2.1 Participants and setup

Voters are identified with their email addresses, that will be used for authentication by the **Server**. We consider two distinct roles for the Voter and their **Voting device** since our protocol is designed to protect against a malicious Voting device (w.r.t. verifiability).

During the setup phase, the **Registrar** sends a private credential cred (a signing key) to each voter. It also sends the list of corresponding public verification keys to the Server. The k **Decryption authorities** set up the public key of the election pk_E such that a threshold t of them can decrypt any message encrypted with pk_E.

We assume a **Public board** that can be accessed at any time by all the participants. How to realize a public board in practice is out of the scope of this paper and is discussed for example in [23]. We also assume that at least one honest **Auditor** checks the validity of the ballots and all the cryptographic material that appears on the public board.

2.2 Voting phase

Ballot. Given a vote $v = (v_i)_{1 \leq i \leq n}$ and a credential cred, a ballot bal $= (M, \text{zkp}, \sigma)$ is formed of an encrypted matrix M, a zero-knowledge proof zkp and a signature σ defined as follows. First, a vector of audit codes $(a_i)_{1 \leq i \leq n}$ is chosen uniformly at random, where $0 \leq a_i < \mu$, that is, each a_i is a small integer (smaller than a "modulus" μ than can be thought as 2 or 10).

- The encrypted matrix $M = \begin{pmatrix} V_1 \ A_1 \ B_1 \\ \vdots \\ V_n \ A_n \ B_n \end{pmatrix}$ contains n lines. Each line is
  formed of V_i, A_i, B_i where $V_i = \text{enc}(v_i, \text{pk}_E)$ encrypts the choice v_i, $A_i = \text{enc}(a_i, \text{pk}_E)$ encrypts the audit code a_i, while B_i ties V_i and A_i together. Namely, $B_i = \text{enc}(b_i, \text{pk}_E)$ where $b_i = v_i + a_i \bmod \mu$.
- The zero-knowledge proof zkp is formed of several zero-knowledge proofs that guarantee that:
  - each V_i encrypts either 0 or 1;
  - the V_i's encrypt values v_i such that $k_1 \leq \Sigma_{i=1}^n v_i \leq k_2$.
  - for each $1 \leq i \leq n$, V_i, A_i, B_i encrypt some values v_i, a_i, b_i such that $b_i = v_i + a_i \bmod \mu$.
- Finally, σ is the signature of M and zkp with the credential cred.

Voter experience. The voter selects their vote v on their voting device, that computes a ballot bal as described above. The voting device displays the vote v and $h = \text{hash}(\text{bal})$ to the voter, where h will be used as a tracking number. This step is illustrated in Figure 2a. The voter confirms their vote, and then the device sends the ballot bal to the voting server. The voter waits for an email from the server, that contains a confirmation challenge chal (used for authentication) and their tracking number h. If the tracking number is correct, they enter chal to their voting device and starts the audit phase, as illustrated in Figure 2b:

- the voting device displays the matrix in clear

$$m = \begin{pmatrix} v_1 + a_1 = b_1 \\ \vdots \\ v_n + a_n = b_n \end{pmatrix}$$

- the voter reviews the matrix m and checks that the sums are correct (taking $\mu = 10$ can make this step easier);
- for each line, they select either a_i or b_i, yielding a bit δ_i, where $\delta_i = 0$ if a_i is selected and 1 otherwise.
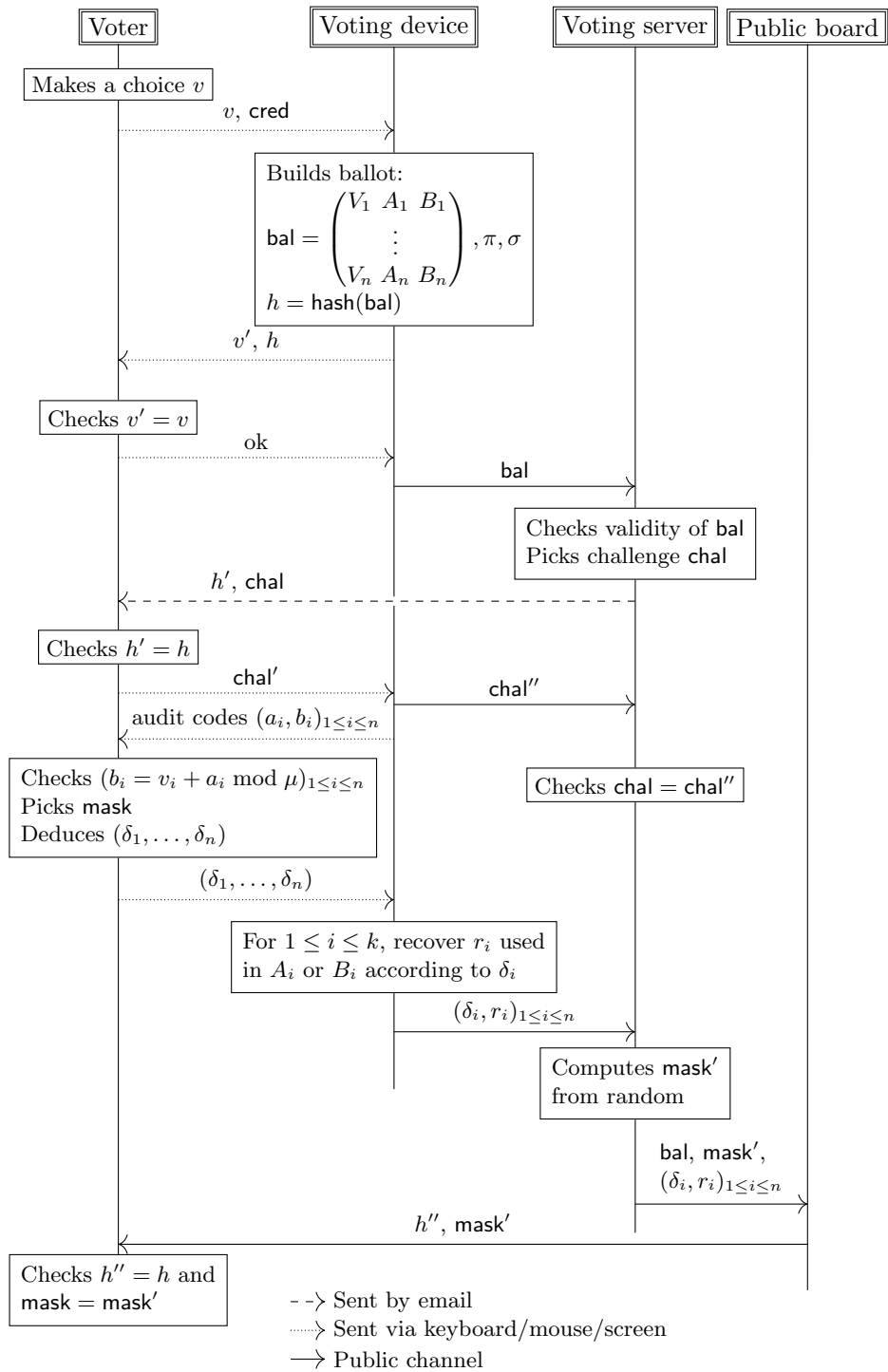
Voter | Voting device | Voting server | Public board

Makes a choice v

v, cred →

Builds ballot:
$$\mathsf{bal} = \begin{pmatrix} V_1 & A_1 & B_1 \\ & \vdots & \\ V_n & A_n & B_n \end{pmatrix}, \pi, \sigma$$
$h = \mathsf{hash}(\mathsf{bal})$

v', h

Checks $v' = v$

ok →

bal →

Checks validity of bal
Picks challenge chal

h', chal

Checks $h' = h$

chal' → chal'' →

audit codes $(a_i, b_i)_{1 \le i \le n}$

Checks $(b_i = v_i + a_i \bmod \mu)_{1 \le i \le n}$
Picks mask
Deduces $(\delta_1, \ldots, \delta_n)$

Checks $\mathsf{chal} = \mathsf{chal}''$

$(\delta_1, \ldots, \delta_n)$ →

For $1 \le i \le k$, recover r_i used
in A_i or B_i according to δ_i

$(\delta_i, r_i)_{1 \le i \le n}$ →

Computes mask'
from random

$\mathsf{bal}, \mathsf{mask}'$,
$(\delta_i, r_i)_{1 \le i \le n}$

h'', mask'

Checks $h'' = h$ and
$\mathsf{mask} = \mathsf{mask}'$

- -→ Sent by email
⋯→ Sent via keyboard/mouse/screen
⟶ Public channel

Fig. 1: Description of the voting phase of the BeleniosCaI protocol

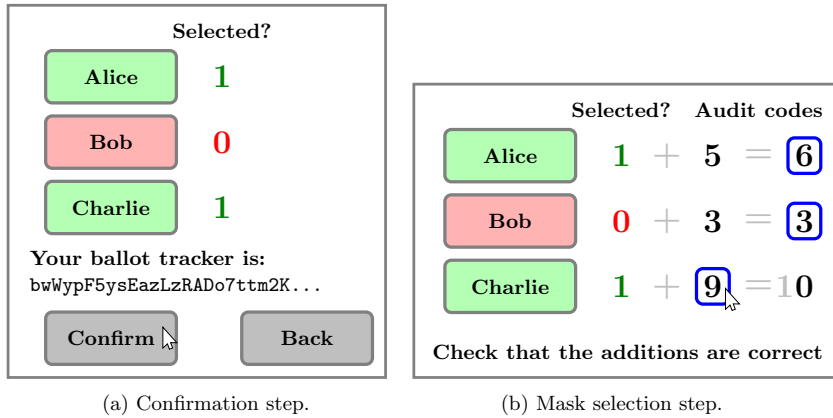(a) Confirmation step. (b) Mask selection step.

Fig. 2: Two steps of the voter's journey.

For each i, the voting device must reveal the randomness r_i used to encrypt either a_i or b_i according to the selection of the voter. The voting device sends to the server the selection $(\delta_i)_{1 \leq i \leq n}$ as well as the corresponding randomness $(r_i)_{1 \leq i \leq n}$. For each line i, the server decrypts A_i or B_i, according to δ_i, thanks to the randomness r_i and reveals on the public board the corresponding a_i or b_i. Finally, the voter checks on the public board that their tracking number h appears, as well as their selection $(\delta_i)_{1 \leq i \leq n}$ and the corresponding chosen integers. External auditors check that the decryptions of audit codes are valid, as well as the zero-knowledge proofs and signatures.

Intuitively, even if the voting device is malicious, it must commit to values v_i, a_i, b_i such that $b_i = v_i + a_i \mod \mu$. Hence if the voting device wishes to change the vote v_i of the voter, it must also modify a_i or b_i. This will be detected with probability $1/2$ by the voter since they chose to see either a_i or b_i at random. Therefore, while a few votes may be modified without being detected, the attacker has a probability of $1/2^k$ to change k votes without being detected. This does not diminish vote privacy since only a random mask (either a_i or b_i) is revealed. We provide a formal security analysis in the next section.

Server. As in Belenios, the Server only accepts well-formed ballots, that is, ballots such that the zero-knowledge proofs are valid and the signature corresponds to a valid public verification key. Moreover, the Server ensures that a voter always uses the same credential (in case of a revote) and that no two voters use the same credential.

2.3 Tally phase

As in Helios and Belenios, the encrypted votes $(V_i)_{1 \leq i \leq n}$ can be combined homomorphically in order to obtain an encrypted vector that corresponds to the total number of votes per candidates. The vector is decrypted by the Decryption authorities, who also provide a zero-knowledge proof of correct decryption.

2.4 Usability considerations

In theory, the modulus μ can be as small as 2. This is enough to perfectly mask a bit. However, we do not expect typical voters to be familiar with binary arithmetic. We suggest to use $\mu = 10$, so that voters can do a classical addition, and are instructed to consider the units digit only, e.g. if the sum is 10, then we forget the 1 and get 0. Printing the 1 in gray can help as depicted in Figure 2b.

Choosing a random mask could be difficult since we expect the voters to be bad random generators in case there are many candidates. Assuming that voters can better pick numbers at random, one option is to ask voters to enter an at least 4-digit number (or maybe longer, if there are more than 13 candidates) instead of directly selecting the mask, and convert it to a mask using a predefined deterministic function. The server must then print both the mask and the number, so that the voter can check them.

3 Security Analysis

We conduct a security analysis of the BeleniosCaI protocol using the tool ProVerif.

3.1 ProVerif

ProVerif [5,6] is a state-of-the-art, automatic verification tool to prove the security of cryptographic protocols, including industrial-scale protocols such as TLS or Signal. ProVerif proves the security of protocols thanks to a (sound) transformation into first order logic. It (often) reports an attack trace when the proof fails. We recall here its main specificities and we present an overview of our model of BeleniosCaI. The full models are available at [10]. A detailed description of the syntax and semantics of ProVerif can be found in [6].

Messages ProVerif is based on the notion of a Dolev-Yao attacker model [17] in which messages are abstracted by terms which are either an atomic data (e.g., a key, an unguessable random number, etc) or a function symbol applied to other terms. The semantics of a message is then provided by an equational theory and/or a rewriting system. For instance, a randomized asymmetric encryption scheme is modeled by a function symbol `aenc` of arity 3, a symbol `adec` of arity 2, and a symbol `pk` of arity 1. The possibility to decrypt an encrypted message using the correct secret key is then modeled by the rewriting rule: $\mathtt{adec}(y, \mathtt{aenc}(\mathtt{pk}(y), x, r)) \to x$ where x is the plaintext message, y the secret key corresponding to the public key $\mathtt{pk}(y)$, and r the randomness used for encryption. Most cryptographic primitives can be modeled similarly (e.g., signature, hash function, zero knowledge proof).

Roles The different roles of the protocol are modeled through a process algebra inspired by the applied pi-calculus which includes the commands $P|Q$ to model that the processes P and Q execute in parallel, and $!P$ to model that an

arbitrary number of process P can be executed in parallel. Moreover, communications on a (public or private) channel c are represented by actions $\mathtt{in}(c, x)$ and $\mathtt{out}(c, m)$ which respectively model that an agent is waiting for a message x on channel c, and an agent is sending a message m on c. In addition, the process may contain *event* actions that are used to identify specific steps in the process and to express some security properties. Finally, there are standard actions to model fresh unguessable name generations, conditionals, declarations, etc. For sake of simplicity, ProVerif supports the syntax $\mathtt{in}(c, =m); P$ as a shortcut for $\mathtt{in}(c, x); \mathtt{if}\ x = m\ \mathtt{then}\ P\ \mathtt{else}\ 0$. This notation will be used in Figure 3.

All the processes are executed using an operational semantics presented in [6]. For instance, it formally defines that a message u can be received through an input $\mathtt{in}(c, x)$ as soon as there exists an action $\mathtt{out}(c, u)$ to execute, or if the channel c is public and the attacker is able to deduce the message u from its knowledge. Roughly speaking, the semantics formalizes the intuitive execution of the processes that the reader might think of.

Example 1. Figure 3 presents as a concrete example the process used to model the voter role. We assume that the voter securely receives their secret signing key and knows their id and the channel they will use to communicate with their device (e.g. monitor, keyboard, mouse, etc). Finally, we assume a public channel c that will be used to communicate with the environment (i.e. the attacker).

The process then corresponds to the voter experience described in Section 2.2. First (l. 2-4), the voter sends their choice to their device. Second (l. 6-7), the voter reads on their device the ballot tracker and reviews their choice before confirming their vote. Third (l. 9-11), the voter receives an email from the server that contains a ballot tracker and a challenge. The ballot tracker must correspond to the one displayed on the reader. If this is the case, the voter enters their challenge in the device. Fourth (l. 13-18) the audit phase starts: the device displays the ballot and the corresponding audit codes, the voter checks the well-formedness of the ballot (i.e. $b = x + a$). The event $\mathtt{isSum}(x_B, v, x_A)$ is executed to model the check of the sum. The voter ends this phase by randomly choosing which code will be audited. Finally, (l. 20-29), the voter reads on the public bulletin board and look for an entry that matches their ballot tracker, and the expected audit codes. If everything is correct, then the voter is "happy" and can be sure that their vote will be counted, and that the ballot contains their intended choice.

The other roles of the protocol (i.e. voting device, server, tally) are modeled in a similar way.

Security properties ProVerif supports two main classes of security properties: trace properties and equivalence properties.

The trace properties express that specific *bad* states cannot be reached during the execution of the protocol. These are expressed using correspondence queries of the form

$$\bigwedge_{i \in \{1, \dots, n\}} E_i \Rightarrow \bigvee_{i \in \{1, \dots, m\}} \bigwedge_{j \in \{1, \dots, p\}} F_{i,j}$$

which models that whenever the events $E_1, \dots, E_n$ are executed during an execution then there must exist $i_0 \in \{1, \dots, m\}$ such that all the facts (i.e. events,

```
1    Voter(id, c_device, ssk_id) =
2      in(c, v);      (* voter's choice chosen by attacker, to consider all cases *)
3      event HasInitiatedVote(id,v);
4      out(c_device, (v,ssk_id));
5
6      in(c_device, (=v, x_ballot_tracker));
7      out(c_device, OK);
8
9      in(c_mail(id), (=x_ballot_tracker, x_chal));
10     event Voted(id,x_ballot_tracker,x_v);
11     out(c_device, x_chal);
12
13      in(c_device, x_ballot_plaintext);
14      let (=v, x_A, x_B) = x_ballot_plaintext in
15       if x_A <> blind_code && x_B <> blind_code then
16         event isSum(x_B,v,x_A);
17         in(c, audit_choice);     (* audit choice chosen attacker, to consider all cases *)
18         out(c_device, audit_choice);
19
20         in(cell_BB, (x_vk, x_ballot, x_ballot_tracker_bb, x_rands, x_codes));
21         if audit_choice = 0 then (
22           if x_ballot_tracker_bb = x_ballot_tracker && x_codes = (x_A,blind_code) then (
23             event HappyVoter(id_voter,x_ballot_tracker,v); 0
24           ) else out(c_error, ERROR)
25         ) else if audit_choice = 1 then (
26           if x_ballot_tracker_bb = x_ballot_tracker && x_codes = (blind_code,x_B) then (
27             event HappyVoter(id_voter,x_ballot_tracker,v); 0
28           ) else out(c_error, ERROR)
29         ) else out(c, ERROR)
```

Fig. 3: ProVerif process modeling the voter actions.

equalities, disequalities, etc) $F_{i_0,1}$, ..., $F_{i_0,p}$ hold too. This type of queries are used to express for instance authentication, confidentiality, or integrity. In the context of our security analysis it will be used to express verifiability properties.

The equivalence properties model that two processes P and Q cannot be distinguished by an attacker interacting with them, denoted $P \approx Q$. For sake of simplicity, we intentionally decide to not recall here the formal definition of the notion of equivalence that ProVerif verifies (see [6] for details). In our context, this notion of equivalence properties will be useful to model vote secrecy.

3.2   How to overcome ProVerif 's limitations?

The underlying symbolic model of the ProVerif tool has two main limitations to model the BeleniosCaI protocol. First, it does not allow to model associative and commutative operators which prevents an accurate modeling of the arithmetic operations in $\mathbb{Z}/\mu\mathbb{Z}$. Second, it does not model probabilistic actions that would be necessary to faithfully describe the audit mechanism of BeleniosCaI. To overcome these limitations, we leverage techniques developed in [7]. We recall here an overview of them but a detailed description and a proof of their correctness is available in the original paper.

Arithmetic operations   To model the sum in $\mathbb{Z}/\mu\mathbb{Z}$, we assume that an event $\mathrm{isSum}(x, a, b)$ is executed each time an agent verifies that $x = a+b \mod \mu$. These events records all the equalities that must hold. Based on these events, two

approaches are developed in [7] depending on the security property under study: first, for trace properties, it models the subset of properties which are relevant to make the audit mechanism secure. Specifically, it restricts the analysis to execution traces that satisfy these properties using *restrictions*. For example, we model that *"for all $a, b \in \mathbb{Z}/\mu\mathbb{Z}$ there exists a unique $x \in \mathbb{Z}/\mu\mathbb{Z}$ such that $x = a + b \mod \mu$"* using the restriction

$$\texttt{isSum}(x, a, b) \wedge \texttt{isSum}(x, a, b') \Rightarrow b = b'.$$

Other properties are modeled in the same way. These properties are trivially satisfied by the modular arithmetic, and actually sufficient for our verifiability properties.

Second, for equivalence properties, [7] proposes an approach based on *relation preservation* between the traces of the two processes P and Q we want to prove equivalent. Note that the previous approach, based on an over-approximation of the sum relation in $\mathbb{Z}/\mu\mathbb{Z}$ would be unsound for equivalence properties. Hence, [7] proposes to prove equivalence of processes P and Q as follows: (1) show that for any trace tr_P of P there exists an indistinguishable trace tr_Q of Q, and (2) if an event $\texttt{isSum}(x, a, b)$ is executed in P then the same event is executed in tr_Q. The first item corresponds to the standard trace equivalence property that can be proved as usual in ProVerif, and the second item is the relation preservation property. It can be proved in ProVerif too by defining a specific lemma. An immediate consequence of item (2) is that, if the relation induced by the events $\texttt{isSum}(x, a, b)$ models the sum in $\mathbb{Z}/\mu\mathbb{Z}$ in tr_P then it remains true in tr_Q. A detailed description of this approach and its soundness is provided in [7].

Probabilistic actions ProVerif does not handle probabilistic actions, and thus cannot faithfully model the random choice done by the voter to perform the audit. Like [7], to be able to conduct the security analysis, we decided to make the following reasoning on top of the modeling: because an attacker cannot know in advance which code the voter is about to audit, the attacker must be able to provide data that make the audits valid for both codes. Therefore, we model that the voter audits both codes for verifiability when it is necessary to avoid false attacks. Concretely, this means that, regarding Figure 3, a third case is added for audit:

```
1    if audit_choice = 2 then (
2      if x_ballot_tracker_bb = x_ballot_tracker && x_codes = (x_A,x_B) then (
3        event HappyVoter(id_voter,x_ballot_tracker,v); 0
4      ) else out(c, ERROR)
```

3.3 Security analysis and result

We proved the security of the BeleniosCaI protocol regarding vote secrecy and verifiability. Following the approach developed in [12,3], we consider 3 sub-properties to model E2E verifiability: *cast-as-intended* to model that the voter can verify their ballot contains their intended vote, *no clash attack* to model that two voters should not agree on the same ballot, and *recorded-as-cast* to model that an attacker cannot create nor modify a ballot in the name of an honest

voter. In this security analysis, we deliberately omit the eligibility property because it remains exactly the same as the current version of the Belenios protocol (eligibility is performed by both the registrar and the server).

To formalize these security properties, we define the following events:

- $\mathsf{onBoard}(vk, h, b, r, X)$ is executed each time a ballot b is added to the public bulletin board. vk is the public key associated to the signature occurring in b, h is the ballot tracker associated to b (i.e. $h = \mathsf{hash}(b)$) and (r, X) is the data published to conduct the audit.
- $\mathsf{Honest}(id, vk)$ is executed each time an honest voter id is registered with the public signing key vk.
- $\mathsf{HasInitiatedVote}(id, vk)$ is executed when the voter id with the public signing key vk has initiated a vote.
- $\mathsf{Voted}(id, vk, h)$ is executed when the voter id with the public signing key vk has confirmed their vote using the ballot tracker h.
- $\mathsf{HappyV}(X, id, h, v)$ is executed when voter id has completed the audit using the ballot tracker h and intended to vote for v. $X \in \{\mathsf{L}, \mathsf{R}, \mathsf{LR}\}$ records whether the voter audited the Left, the Right, or both (for modeling) codes.

Vote secrecy. As usual in the symbolic analyses, we consider the vote secrecy definition proposed by Kremer *et al.* [15]: an e-voting protocol ensure vote secrecy if an attacker is not able to distinguish whether Alice voted for 0 and Bob for 1, or conversely. Formally, we note $Alice(x)$ (resp. $Bob(x)$) the process modeling the role of Alice (resp. Bob) when voting x, and P the process modeling all the other roles involved in the protocol then we want to prove that:

$$P|Alice(0)|Bob(1) \approx P|Alice(1)|Bob(0).$$

Cast-as-intended. A protocol ensures cast-as-intended if the voter is able to verify that their ballot will be counted and contains their intended vote. We consider the following correspondence property to model cast-as-intended:

$$\mathsf{HappyV}(\mathsf{LR}, id, h, v) \wedge \mathsf{Honest}(id, vk) \Rightarrow$$
$$\mathsf{onBoard}(vk', b, h, r, X) \wedge \ (b \text{ encrypts candidate } v).$$

No clash attack. A protocol protects against clash attacks if two voters cannot agree on the same ballot. When honest devices are used, they generate ballots with different randomness, which ensure the no-clash property. This is no longer true if the voting devices are malicious and collude: they may use the same randomness and make Alice and Bob believe they own the same ballot. Interestingly, the no-clash property still holds in BeleniosCaI thanks to the fact that voters randomly choose whether they audit the left or the right code. We therefore model the no-clash property for voters that audit differently:

$$\mathsf{HappyV}(\mathsf{L}, id, h, v) \wedge \mathsf{HappyV}(\mathsf{R}, id', h, v') \Rightarrow \mathtt{false}.$$

Note that this property is weaker than the original no-clash property, that does not need to make assumptions on voter's (audit) choices.

Recorded-as-intended. A protocol ensures recorded-as-intended if an attacker is not able to forge a ballot in the name of an honest voter. This corresponds to the following correspondence property[1]:

$$\texttt{onBoard}(vk, h, b, r, X) \land \texttt{Honest}(id, vk) \Rightarrow \texttt{Voted}(id', vk, h). \qquad (1)$$

Unfortunately, this property is not satisfied by BeleniosCaI when the voting device and the server are compromised. Indeed, as soon as the voter initiates a vote, they reveal their credential which lets the attacker completely impersonate them. We thus define a weaker property, that says that an attacker is not able to forge a ballot in the name of an honest voter, unless the voter has started a voting session:

$$\texttt{onBoard}(vk, h, b, r, X) \land \texttt{Honest}(id, vk) \Rightarrow \texttt{HasInitiatedVote}(id', vk). \qquad (2)$$

Remark 1. Regarding the literature, it seems that protocols known to be verifiable assuming a compromised voting device guarantee only Property 2. This is the case, for instance, of Helios [2] or Selene [30]. Still, other protocols such as the Swiss Post [1] or BeleniosVS [12] ensure Property 1. Hence, we considered interesting to analyze the security of BeleniosCaI w.r.t. these two properties, as presented in Table 1 and 2. Recorded-as-intended corresponds to Property 2 and recorded-as-intended (strong) to Property 1.

Results Table 1 presents the main results of the security analysis: BeleniosCaI is as secure as Belenios if we assume that the voting device is honest, i.e. it requires that either the registrar or the server is honest to ensure verifiability and that the decryption authorities are honest for vote secrecy. Moreover, it still provides verifiability when considering a malicious voting device: if the server is honest then it meets the same verifiability property as Belenios (strong E2E verifiability), while if the registrar is honest, then it ensures (only) the E2E verifiability. From the vote secrecy point of view, BeleniosCaI and Belenios are both secure as soon as enough decryption authorities are honest.

In summary, these results demonstrate that BeleniosCaI provides strictly better security guarantees than Belenios.

For interested readers, Table 2 presents the detailed results of the security analysis conducted in ProVerif. The ProVerif files are available in [10].This table details the weakest trust assumptions in which each security property is ensured by BeleniosCaI: the less trustworthy agents there are, the more secure the protocol is. For instance, line 4 shows that the strong notion of recorded-as-cast is ensured as soon as the server is honest (i.e. even if the voting device or the registrar are compromised). Indeed, if the server is honest then the attacker will not be able to learn the challenge code sent by mail to the voter, and thus will not be able to confirm a maliciously created ballot in the name of the voter.

---

[1] This correspondence property identifies voters by their public signing key. This assumption is valid as long as the registrar is honest. Otherwise, when the server is honest, they can be identified by their *id* and a similar property is defined. This distinction corresponds to the approach developed in [12].

Trusted components	VD ∧ (R ∨ S) [T]	[VD ∧ T]	S [VD ∧ T]	R [VD ∧ T]
E2E verifiability (strong) (including cast-as-intended)	✔	✔	✗	✗
E2E verifiability (including cast-as-intended)	✔	✔	✔	✗
Vote secrecy	✔	✔	✔	✔

✔ = proved secure   VD=voting device, R=registrar, S=server, T=dec. auth.,
✗ = attack   [.]= extra trust assumption for vote secrecy only

Table 1: Security analysis of BeleniosCaI: minimal trust assumption.

	Voter	Voting device	Server	Registrar	Dec. auth.
Cast-as-intended	🙂	🙁	🙁	🙁	🙁
No clash	🙂	🙁	🙁	🙁	🙁
Recorded-as-cast (strong)	🙂	🙂	🙁	🙂	🙁
	🙂	🙁	🙂	🙁	🙁
Recorded-as-cast	🙂	🙁	🙁	🙂	🙁
	🙂	🙁	🙂	🙁	🙁
Vote secrecy	🙂	🙂	🙁	🙁	🙂

🙂 = trustworthy   🙁 = compromised

Table 2: Minimal trust assumptions for each security property.

4 Efficiency considerations

A ballot in Belenios or Helios essentially costs one ciphertext and zero-knowledge (individual) proof per candidate, plus one zero-knowledge (overall) proof that controls the number of selected candidates. BeleniosCaI requires two extra ciphertexts and a proof of modular equality for each candidate. This essentially adds a factor between 2 and 3 to compute the ballot, as we detail now.

The zero-knowledge proofs required for forming the ballots can all be expressed as statements of the form "The cleartext of this ciphertext belongs to this finite set of possible values".

- **Individual proofs.** Each v_i belongs to $\{0, 1\}$ is already in this form. This is therefore a list of 0/1 proofs. The basic approach requires 5 exponentiations. See [21,16] for ways to optimize them.
- **Overall proof.** The sum of all the v_i's is between k_1 and k_2. Due to the homomorphic property of ElGamal encryption, anyone can compute the ciphertext of this quantity, from the list of the V_i's, so that we are indeed in the claimed setting. Proving the property is then classically done as in Helios

or Belenios, by rewriting it as belonging to a set of integers, with a cost that is linear in $k_2 - k_1$. Techniques exist to do this at a cost in $O(\log(k_2 - k_1))$ [9].

- **Arithmetic proofs.** The fact that V_i, A_i, B_i encrypt some values v_i, a_i, b_i such that $b_i = v_i + a_i \bmod \mu$ can be rewritten as follows. First, by the homomorphic property, anyone can build the ciphertext that encrypts $v_i + a_i - b_i$ (without the modulo μ, that can not be computed homomorphically). By construction, if the ballot is correctly formed, this value must be equal to 0 or to μ. Therefore, this arithmetic proof can be rewritten as a proof of membership in a set of two elements, which is almost the same as the classical 0/1 proof. It therefore requires 5 exponentiations.

In Belenios, there is the possibility to have a special candidate representing a blank vote, and this comes with a zero-knowledge proof that either the blank vote is chosen and all the v_i's are zero, or the blank vote is not chosen, and then the sum of the v_i's is in $[k_1, k_2]$ (see [19]). Our method supports this setting: it is enough to add an arithmetic proof for this additional encrypted bit.

We also mention that in the security analysis, there is no need to have zero-knowledge proofs of the facts that the a_i's and the b_i's are in $[0, \mu-1]$. If, for some practical reasons, the server needs to detect early invalid ballots, these proofs can be added. They are of the same nature as the overall proof. However, they can be costly for the voting device of the voters, if there are many candidates.

To sum-up, compared to Belenios, for the voting device, the additional cost of forming a ballot is, for each candidate, to compute two ElGamal encryptions A_i and B_i, and to compute the arithmetic proof for them. This amounts to 9 additional exponentiations per candidate. Techniques like in [16] can be used to reduce this cost. As a rule of thumb, we can say that this will multiply the running time of forming a complete ballot by a factor between 2 and 3, the exact number depending on the number of candidates and the value of $k_2 - k_1$. For elections for which there are no more than a few dozens of candidates, this remains affordable with a Javascript/WebAssembly implementation running in a standard browser, and there is no need to have a native implementation (which would raise many issues, since this usually requires to install specific software).

References

1. Swiss post voting specification - version 1.1.1. https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/tree/master/System, 2022.
2. B. Adida. Helios: Web-based Open-Audit Voting. In *USENIX*, 2008.
3. S. Baloglu, S. Bursuc, S. Mauw, and J. Pang. Election verifiability revisited: Automated security proofs and attacks on Helios and Belenios. In *CSF'21*, 2021.
4. J. Benaloh. Simple verifiable elections. In *EVT'06*. Usenix, 2006.
5. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW'01*, pages 82–96. IEEE, 2001.
6. B. Blanchet, V. Cheval, and V. Cortier. ProVerif with lemmas, induction, fast subsumption, and much more. In *S&P'22*. IEEE, 2022.
7. M. Bougon, H. Chabanne, V. Cortier, A. Debant, E. Dottax, J. Dreier, P. Gaudry, and M. Turuani. Themis: an on-site voting system with systematic cast-as-intended verification and partial accountability. In *CCS'22*, 2022.

8. I. Brightwell, J. Cucurull, D. Galindo, and S. Guasch. An overview of the iVote 2015 voting system. New South Wales Electoral Commission, Australia, 2015.

9. S. Canard, I. Coisel, A. Jambert, and J. Traoré. New results for the practical use of range proofs. In *EuroPKI'2013*, pages 47–64. Springer, 2014.

10. V. Cortier, A. Debant, P. Gaudry, and S. Glondu. Belenios with cast as intended. In *8th Workshop on Advances in Secure Electronic Voting*, Bol, Brač, Croatia, May 2023. `https://hal.inria.fr/hal-04020110`.

11. V. Cortier, J. Dreier, P. Gaudry, and M. Turuani. A simple alternative to Benaloh challenge for the cast-as-intended property in Helios/Belenios. Preprint available at `https://hal.inria.fr/hal-02346420`, 2019.

12. V. Cortier, A. Filipiak, and J. Lallemand. BeleniosVS: Secrecy and Verifiability against a Corrupted Voting Device. In *CSF'19*, pages 367–36714. IEEE, 2019.

13. V. Cortier, D. Galindo, S. Glondu, and M. Izabachène. Election verifiability for Helios under weaker trust assumptions. In *ESORICS'14*. Springer, 2014.

14. V. Cortier, P. Gaudry, and S. Glondu. Features and usage of Belenios in 2022. In *E-Vote-ID'22*, 2022.

15. S. Delaune, S. Kremer, and M. D. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17:435–487, 2009.

16. H. Devillez, O. Pereira, and T. Peters. How to verifiably encrypt many bits for an election? In *ESORICS'2022*, pages 653–671. Springer, 2022.

17. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29:198–208, 1983.

18. D. Galindo, S. Guasch, and J. Puiggali. 2015 Neuchâtel's cast-as-intended verification mechanism. In *VoteID'15*, 2015.

19. P. Gaudry. Some ZK security proofs for Belenios. `https://hal.inria.fr/hal-01576379`, 2017. Informal note.

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

21. R. Haenni, P. Locher, and N. Gailly. Improving the performance of cryptographic voting protocols. In *VOTING'19*, pages 272–288. Springer, 2019.

22. S. Heiberg and J. Willemson. Verifiable internet voting in Estonia. In *EVOTE'14*. IEEE, 2014.

23. L. Hirschi, L. Schmid, and D. A. Basin. Fixing the Achilles heel of e-voting: The bulletin board. In *CSF'21*, 2021.

24. A. Kiayias, T. Zacharias, and B. Zhang. End-to-end verifiable elections in the standard model. In *EUROCRYPT'15*, pages 468–498. Springer, 2015.

25. R. Küsters, J. Müller, E. Scapin, and T. Truderung. sElect: A lightweight verifiable remote voting system. In *CSF'16*. IEEE, 2016.

26. K. Marky, O. Kulyk, K. Renaud, and M. Volkamer. What did I really vote for? On the usability of verifiable e-voting schemes. In *CHI '18*, 2018.

27. K. Marky, M. Zollinger, P. B. Roenne, P. Y. A. Ryan, T. Grube, and K. Kunze. Investigating usability and user experience of individually verifiable internet voting schemes. *ACM Trans. Comput. Hum. Interact.*, 2021.

28. C. A. Neff. Practical high certainty intent verification for encrypted votes, 2004.

29. P. Ryan, P. Roenne, and S. Rastikian. Hyperion: An enhanced version of the Selene end-to-end verifiable voting scheme. In *E-Vote-ID'21*, 2021.

30. P. Ryan, P. Rønne, and V. Iovino. Selene: Voting with transparent verifiability and coercion-mitigation. In *VOTING'16*, pages 176–192, 2016.

31. P. Y. A. Ryan and V. Teague. Pretty good democracy. In *Security Protocols'09*. Springer, 2009.