# Belenios: a simple private and verifiable electronic voting system

Véronique Cortier, Pierrick Gaudry, Stephane Glondu

# Belenios: a simple private and verifiable electronic voting system

Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu

CNRS, Inria, Univ. Lorraine, France

**Abstract.** We present the electronic voting protocol Belenios together with its associated voting platform. Belenios guarantees vote privacy and full verifiability, even against a compromised voting server. While the core of the voting protocol was already described and formally proved secure, we detail here the complete voting system from the setup to the tally and the recovery procedures.
We comment on the use of Belenios in practice. In particular, we discuss the security choices made by election administrators w.r.t. the decryption key and the delegation of some setup tasks to the voting platform.

## 1 Introduction

Electronic voting facilitates counting and enables elections with multiple questions or sophisticated tally functions like approval voting or single transferable vote. Ballots may be quite complex since voters may have to give a score to each candidate or rank them according to their preference. Tallying such complex ballots is a tedious job if done by hand while it is an easy task for a computer. Electronic voting also provides flexibility: an election may last from a few minutes to several weeks and voters may vote from any place. Consequently, e-voting is now often used at least as a replacement for postal voting.

Besides, electronic voting is subject to heavy controversy. The main reason is that existing systems are not sufficiently secure, as exemplified by several severe attacks. For example, the Washington, D.C., Internet voting system has been attacked [40], during a trial just before the election. The research team successfully replaced existing ballots, modified the code, and retrieved the login and passwords of all (real) voters. Similarly, important security concerns have been raised in the voting systems used respectively in Estonia [38] and Australia [28].

Modern electronic voting systems aim at two main properties: vote privacy (no one should know my vote) and verifiability (it is possible to check that the votes are correctly counted). Verifiability is often divided into three subproperties:

- individual verifiability: a voter can check that her vote has been properly counted;
- universal verifiability: everyone can check that the result corresponds to the ballots on the public bulletin board;
- eligibility verifiability: ballots come from legitimate voters.

Privacy and verifiability are difficult to achieve simultaneously.

Many e-voting systems have been proposed in the literature. Each protocol solves some security issues. They are either designed to vote in polling stations or remotely. We focus here on Internet voting, although the models and techniques developed in this context also apply to on-site systems whose security does not assume trusted machines (e.g. STAR-vote [7], Prêt-à-voter [35]).

Systems like CHVote [27] and the Neuchâtel protocol [24] protect voters against a corrupted device. Even if a voter uses a corrupted computer or smartphone, she should be able to check that her *intended vote* has been correctly recorded on the voting platform thanks to return codes: after casting a vote, a voter receives a code (a short sequence of characters) and checks using a previously received sheet of paper that the code corresponds to her vote intent. Such protocols rely on a rather heavy infrastructure and for the Neuchâtel protocol, voters cannot check that the election result corresponds to the received ballots. In a contrast, Helios [3] is a simple protocol that aims at privacy and end-to-end verifiability in low coercion environment. A voter may audit her voting device by generating mock ballots and sending them to a third (trusted) party. Another simple system is sElect [31], where voters can easily check that their vote has been counted as intended thanks to a tracking number displayed next to their vote, once the election result is published. A drawback is that vote buying is then straightforward. Selene [36] also uses tracking numbers to ease verifiability, together with a cryptographic mechanism that provides receipt-freeness: voters cannot prove for whom they voted. Demos [30] also aims at both verifiability and receipt-freeness. Actually, all the aforementioned systems admit a way for a voter to sell her vote to a buyer (or a coercer). Civitas [15] is the only system that provably achieves both verifiability and coercion-resistance. The idea is that voters may produce fake voting credentials such that the corresponding ballots will eventually be deleted without the coercer noticing. Other systems aim at everlasting privacy [22] (will my vote remain secret if the underlying cryptography is broken?) or accountability [32].

We present here the Belenios system. It offers a good compromise between simplicity and security. Belenios has been deployed on an online platform [1] that has already been used in more than 200 elections, in academia, education, and in sport associations. Belenios is built upon Helios. Like in Helios, the voters can check that their ballots appear on the bulletin board, and that the result corresponds to the ballots on the board, while vote secrecy is guaranteed. In addition, Belenios provides eligibility verifiability: anyone can check that ballots come from legitimate voters, whereas in Helios, a dishonest bulletin board could add ballots without anyone noticing. Helios is thus vulnerable to ballot stuffing. Eligibility verifiability can be added to voting systems through a signature mechanism and additional credentials [18]. Belenios is an instance of this generic construction, applied to Helios. Note that, like in Helios, Belenios is not coercion-resistant: voters may prove for whom they voted by providing the randomness used to produce their ballot or they may simply sell their voting material. Therefore Belenios should not be used in high stake elections. More generally, we believe

that electronic voting systems still do not achieve an appropriate security level for high stake elections, such as politically binding national elections. At least, we believe that e-voting does not yet achieve the same level of security as paper-based elections, organized in physical polling stations, where people may watch the ballot box and manually count the ballots. A more detailed security comparison needs to be carried out for paper-based elections where the ballot box cannot be properly monitored or when ballots are counted through electronic devices.

As it is often the case for protocols, the specification of Belenios can be retrieved only by expanding a series of papers [18,17,3,11] that still omit many implementation choices. Alternatively, one may dive directly in the code specification [26]. To fill the gap between these two highly technical (in a different way) descriptions, in the first part of this article, we provide a detailed presentation of Belenios, from the setup to the tally and the recovery procedures. We discuss practical implementation choices. For example, Belenios involves several entities: voters, of course, but also a registrar, and decryption trustees. None of these roles require special cryptographic skills. We therefore describe here which adaptions had to be made for our system to be usable. Moreover, our voting platform offers several levels of security: the registration may be done directly by the voting server, the decryption key may or may not be split into several shares. This yields different tradeoffs between security and simplicity. We discuss these choices and we report on Belenios usage in various elections.

The security of Belenios has been formally proved in [16] w.r.t. vote privacy and verifiability. We do not reproduce here the formal security models but we provide a detailed overview of the properties that have been proved and the associated security assumptions. In particular, these high security guarantees are provided when the decryption key as well as the setup phase are distributed among several entities. Yet, the voting platform still offers some guarantees when the server is entrusted with more tasks.

## 2 Description of Belenios

The full description of Belenios can be found in the specification document [26], and this article refers to the version 1.6. We provide here a high level description where some cryptographic details are omitted.

### 2.1 Preliminaries: cryptographic tools

Belenios relies on a couple of rather standard cryptographic primitives, namely hash functions, encryption, signature, and zero-knowledge proof. For the public-key part, we work in a cyclic group $G$ of order $q$ for which a generator $g$ is given, and we assume that the decisional Diffie-Hellman problem is hard in $G$. In the current implementation, the only choice for $G$ is a subgroup of a multiplicative group of a prime finite field. Everything is in place to implement other instances of $G$, for instance elliptic curves, if needed for efficiency or security reasons.

**Encryption** In Belenios, votes are encrypted using El Gamal encryption. To generate a private key, one simply picks $x$ uniformly (as always in this paper) at random in $\mathbb{Z}_q$. The associated public key is $y = g^x$.

Given a vote $v$ encoded as an integer in $\{0, \ldots, q-1\}$ and a public key $y$, the encryption of $v$ is defined as follows: pick $r$ at random in $\mathbb{Z}_q$ and compute

$$\mathsf{enc}(v, y, r) = (g^r, y^r g^v).$$

Note that compared to the textbook El Gamal encryption where the message is a group element, the vote $v$ is encrypted as $g^v$. So, to decrypt a ciphertext $c = (a, b)$ using the private key $x$, we should first compute $b/a^x = g^v$ and then retrieve $v$ by a discrete logarithm computation. This is possible only if $v$ is taken from a small subset and not the entire interval $\{0, \ldots, q-1\}$.

This encryption enjoys an homomorphic property, which is particularly useful in the context of voting, namely:

$$\prod_{i=1}^{n} \mathsf{enc}(v_i, y, r_i) = \mathsf{enc}\Big(\sum_{i=1}^{n} v_i,\ y,\ \sum_{i=1}^{n} r_i\Big),$$

where the product of encrypted messages is defined coordinate-wise as $(a_1, b_1) \cdot (a_2, b_2) = (a_1 a_2, b_1 b_2)$. This property is used in Belenios to compute the encrypted sum of the votes directly from the encrypted ballots.

**Hash function** A hash function is used in several places, including as an internal operation for signatures and zero-knowledge proofs. We denote by $h(m)$ the hash of $m$. To avoid any collision when the hash function is used in different contexts, a message $m$ is actually prefixed by a tag indicating the context and no tag is a prefix of another tag. For example, if $m$ is hashed inside the signature function then $h(\texttt{sigmsg} \mid m)$ is computed instead of $h(m)$. These tags will be omitted in the rest of the paper for the sake of readability but they are important for the security analysis to be valid.

**Signature** Each voter signs her encrypted ballot with a Schnorr signature to avoid any ballot stuffing. A private signing key $\mathsf{sk}$ can be generated as a random element of $\mathbb{Z}_q$ and the associated verification key is $\mathsf{vk} = g^{\mathsf{sk}}$. The signature of a message $m$ with signing key $\mathsf{sk}$ is denoted $\mathsf{sign}(m, \mathsf{sk})$ and is computed as follows:

- pick a random $w \in \mathbb{Z}_q$ ;
- compute $c = h(m \mid g^w)$ and $r = w - \mathsf{sk} \cdot c \mod q$ ;
- return $(r, c)$ .

Given a message $m$, a signature $(r, c)$, and a verification key $\mathsf{vk}$, the verification algorithm $\mathsf{verifsign}$ computes $A = g^r\, \mathsf{vk}^c$ and checks that $c = h(m \mid A)$.

**Zero-knowledge proofs** Zero-knowledge proofs are used in several places in Belenios. First, voters must show that they encrypt a valid vote (e.g. they prove that they selected at most 4 candidates, as allowed by the election). Second, the decryption trustees must prove that they correctly decrypted the result of the election. All the zero-knowledge proofs are made non-interactive using the Fiat-Shamir technique.

A basic zero-knowledge proof is a proof of knowledge of a discrete logarithm. For example, a voter may need to prove that she knows the randomness $r$ used to encrypt her vote $v$ as $(g^r, y^r g^v)$. Given $g^r$ and $r$, she proceeds, as a prover, as follows:

- pick a random $w \in \mathbb{Z}_q$ ;
- compute $c = h(g^r \mid g^w)$ and $s = w - r\,c \mod q$ ;
- return $(s, c)$ .

The verifier, given the proof $(s, c)$ and the message $z = g^r$, checks that $c = h(z \mid A)$ where $A = g^s\, z^c$.

Given a finite set $\mathcal{V}$ of valid votes, a voter may similarly prove that her encrypted vote $v$ belongs to $\mathcal{V}$, providing a proof $\mathsf{proofv}(v, r, \mathsf{enc}(v, \mathsf{pk}, r), \mathsf{pk}, \mathsf{vk})$. In particular, the associated verification algorithm $\mathsf{verifproofv}$ is such that $\mathsf{verifproofv}(\mathsf{proofv}(v, r, \mathsf{enc}(v, \mathsf{pk}, r), \mathsf{pk}, \mathsf{vk}), \mathsf{enc}(v, \mathsf{pk}, r), \mathsf{pk}, \mathsf{vk})$ returns true if $v \in \mathcal{V}$ and false in any other situation. Note that in Belenios, we chose to make the zero-knowledge proof depend also on the verification key $\mathsf{vk}$ of the voter. We will explain why in Section 3.2.

Zero-knowledge proofs are also used by the decryption trustees. First, during the setup, they prove knowledge of their secret key. Second, during the tally, they produce a proof of correct decryption.

The reader is referred to [26,25] for the precise description of the corresponding algorithms and e.g. to [8] for more scientific background on zero-knowledge proofs.

### 2.2 Participants

Belenios includes four main participants: the server, the voters and their voting device, the registrar, and the decryption trustees. We describe them informally. The role of each participant is explained in more details in the next section.

**Registrar** The registrar, also called credential authority on the voting platform, generates and sends privately a signing key to each voter. This key is used by voters to sign their ballot. The registrar also sends the corresponding verification keys to the voting server.

**Voters** The voters select their vote. Their voting device encrypts and signs their vote. The resulting ballot is sent on an authenticated channel to the voting server (thanks to a login and password mechanism). Voters may check at any time that

their ballot is present on the bulletin board. They may also revote, in which case only the last ballot is retained. In Belenios, voting devices are assumed to be honest, hence we will not distinguish between a voter and her voting device in the rest of the paper.

**Voting server** The voting server is in charge of maintaining the bulletin board, that is, the list of accepted ballots. Upon receiving a ballot from a voter, the voting server checks that the ballot is valid (e.g. the signature is valid) and adds it to the bulleting board.

**Decryption trustees** No single authority detains the private key of the election. Instead, a set of $m$ decryption trustees are selected, out of which $t+1$ are needed to decrypt the result of the election. For example, if 5 out of 7 trustees are needed to decrypt the election then 2 trustees may lose their key without having to cancel the whole election.

### 2.3 Protocol

The voting protocol Belenios is divided in three main phases. During the setup, the election material is sent to the voters and the election public key is computed. During the voting phase, voters may cast their vote. Once the voting phase is over, the result is tallied thanks to the decryption trustees. For simplicity, we present Belenios in a simplified version, where voters simply express their vote as a number. For example, in the context of a single-question referendum, 1 means "yes" while 0 means "no". We explain later how to use Belenios for more complex elections.

**Election material generation** For each voter $\mathsf{id}$, the registrar generates a signing key $\mathsf{sk}_{\mathsf{id}} \in \mathbb{Z}_q$ and sends it privately (in practice, by email) to the voter. The registrar transmits the corresponding list of verification keys $\mathsf{vk}_{\mathsf{id}_1}, \ldots, \mathsf{vk}_{\mathsf{id}_n}$ to the voting server, in some random order, where $\mathsf{vk}_{\mathsf{id}} = g^{\mathsf{sk}_{\mathsf{id}}}$.

The voting server publishes the list of verification keys, that is, this list is part of the public election data. Moreover, the voting server generates a password for each voter and sends it privately (in practice, again by email) to the voter.

This phase is depicted in Figure 1.

**Key generation** The decryption key of the election is never computed in any form. Instead, $m$ decryption trustees are selected, out of them a threshold of $t+1$ suffices to decrypt the election. In case $t+1=m$, that is, all trustees need to contribute to the decryption, the key generation phase can be simplified. We present here the general case, following the scheme proposed by Pedersen [34] and proved to yield a secure encryption scheme when combined with ElGamal encryption in [17].

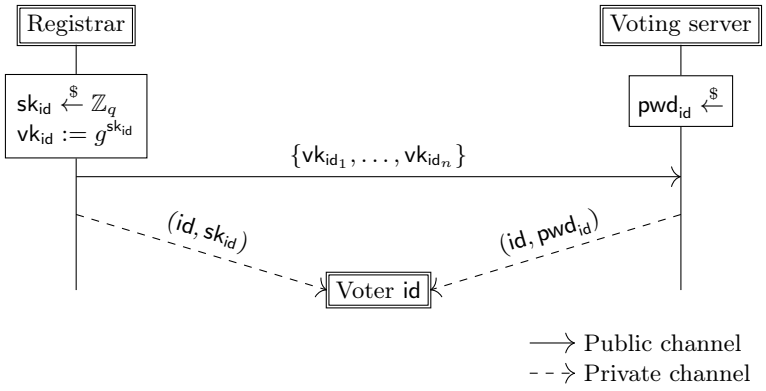


**Fig. 1.** Election material generation.

Each decryption trustee $i$ chooses at random a polynomial $f_i(x) = a_{i0} + a_{i1}x + \cdots + a_{it}x^t$ of degree $t$ and sends privately the value of the polynomial $s_{ij} = f_i(j)$ to the trustee $j$. Intuitively, the secret of the trustee $i$ is $a_{i0}$ and any $t+1$ evaluations of the polynomial will allow to reconstruct the polynomial by Lagrange interpolation and hence $a_{i0}$ (even if it is not done this way). The public key of the election is set to $\mathsf{pk} = \prod_{i=1}^{m} g^{a_{i0}}$. To avoid potentially malicious trustees to corrupt the election key, the key generation includes further checks: each trustee $i$ commits to her polynomial by publishing $A_{i0} = g^{a_{i0}}, \ldots, A_{it} = g^{a_{it}}$. This way, the trustee $i$ can verify the consistency of each received private contribution $s_{ji}$ by checking that $g^{s_{ji}} = \prod_{k=0}^{t} (A_{jk})^{i^k}$. Finally, each trustee $i$ computes her public key $\mathsf{pk}_i = g^{\mathsf{dk}_i}$ with associated decryption key $\mathsf{dk}_i = \sum_{j=1}^{m} s_{ji}$ and sends $\mathsf{pk}_i$ to the server, together with a proof of knowledge $\mathsf{pok}$ of $\mathsf{dk}_i$.

This protocol is depicted in Figure 2. The last consistency checks made by the server are omitted and can be found in [17].

**Voting phase** The list $\mathsf{BB}$ of accepted ballots, the *public board*, is public and can be accessed at any time. Of course, $\mathsf{BB}$ is initially empty. The voting server also displays the election data, namely:

- the set of verification keys $\{\mathsf{vk}_{\mathsf{id}_1}, \ldots, \mathsf{vk}_{\mathsf{id}_n}\}$,
- the public key of the election $\mathsf{pk}$.

The voting server initially does not know the link between a verification key and the corresponding voter. It will memorize this link in a private database $\mathsf{log}$.

To vote, a voter simply encrypts her vote yielding a ciphertext $c = \mathsf{enc}(v, \mathsf{pk}, r)$, produces a proof $\pi = \mathsf{proofv}(v, r, \mathsf{enc}(v, \mathsf{pk}, r), \mathsf{pk}, \mathsf{vk})$ that the vote belongs to the set of valid votes, and signs $c$, yielding a signature $s = \mathsf{sign}(c, \mathsf{sk})$. The ballot $(c, \pi, s), \mathsf{vk}$ is sent to the voting server over an authenticated channel thanks to a login and password mechanism.

Upon receiving a ballot $b, \mathsf{vk}$ from voter $\mathsf{id}$, the server checks whether $\mathsf{id}$ already voted, by looking for an entry of the form $(\mathsf{id}, \mathsf{vk}') \in \mathsf{log}$. If $\mathsf{vk}' \neq \mathsf{vk}$, the

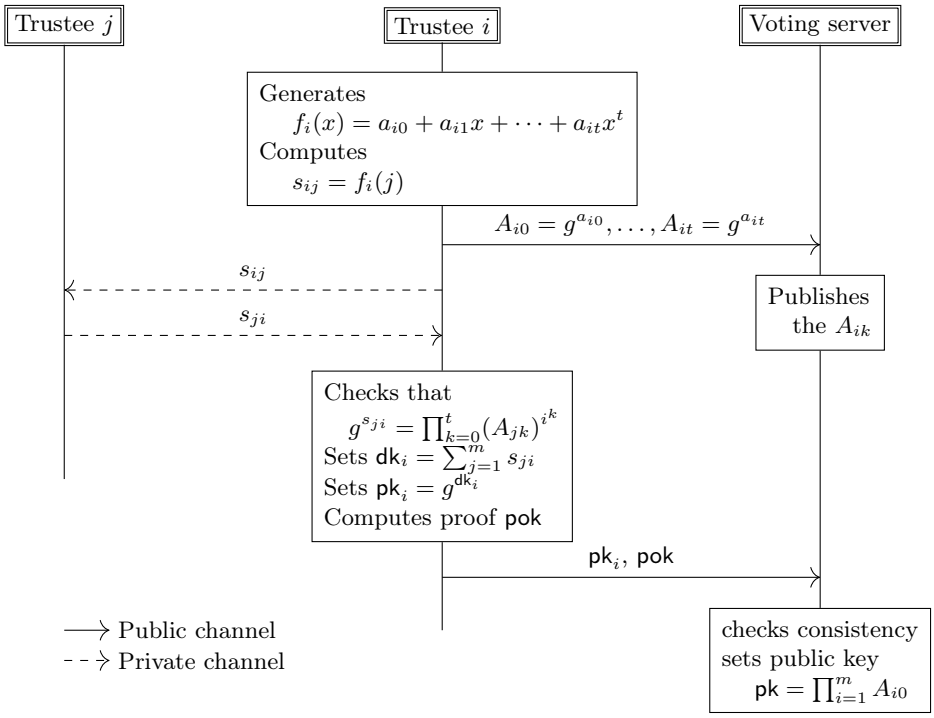


**Fig. 2.** Election key generation.

ballot is rejected: a voter cannot use different verification keys. The server also checks that no other voter used $\mathsf{vk}$ as signing key, otherwise the ballot is also rejected. Then the server checks the consistency of the signature and the proof and rejects the ballot if one of the checks fails. If no such entry exists, the server adds $(\mathsf{id}, \mathsf{vk})$ to $\mathsf{log}$. Then, if there is already a ballot of the form $(b', \mathsf{vk})$ in $\mathsf{BB}$ then this ballot is removed: $\mathsf{BB} := \mathsf{BB} \backslash \{(b', \mathsf{vk})\}$ (only the last ballot is kept for each voter). Finally, the new ballot is added: $\mathsf{BB} := \mathsf{BB} || (b, \mathsf{vk})$.

At any time, voters may check that their last submitted ballot appears in the public board $\mathsf{BB}$.

The voting phase is depicted in Figure 3.

**Tally phase** Once the voting phase is over, the list $\mathsf{BB}$ of accepted ballots is of the form

$$((c_1, \pi, s_1), \mathsf{vk}_1), \ldots, ((c_p, \pi_p, s_p), \mathsf{vk}_p),$$

where the $\mathsf{vk}_j$ are all distinct, the proofs and the signature are valid. Anyone can compute the encrypted result $\mathsf{res}_e = \prod_1^p c_i$. Since each $c_i$ is the encryption of a vote $c_i = \mathsf{enc}(v_i, \mathsf{pk}, r_i)$, we have that $\mathsf{res}_e$ corresponds to the encryption of the

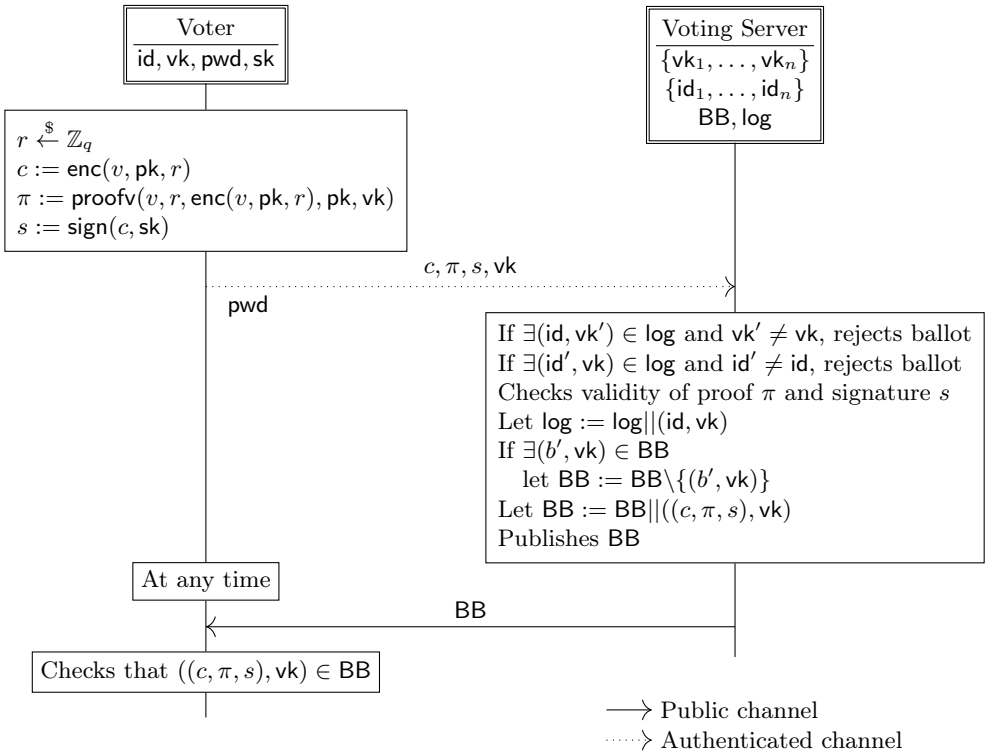


**Fig. 3.** Voting phase.

result:

$$\mathsf{res}_e = \mathsf{enc}(\sum_{i=1}^{p} v_i, \mathsf{pk}, \sum_{i=1}^{p} r_i).$$

Then each trustee $i$ (or at least $t+1$ trustees) contributes to the decryption by providing $\mathsf{res}_e^{\mathsf{dk}_i}$ together with a proof $\mathsf{pok}$ of correct decryption. As explained in [17,26], from these contributions, it is possible to compute the decryption of $\mathsf{res}_e$, that is $\sum_{i=1}^{p} v_i$. The tally phase is depicted in Figure 4.

### 2.4 Elections with several candidates

For simplicity, we have presented Belenios when voters express their vote as a (small) integer. Actually, in Belenios, voters have to select between $k_1$ and $k_2$ candidates out of $l$. A vote is represented by a vector in $\{0,1\}^l$. For example, if there are 5 candidates, the vote $(0,1,1,0,0)$ means that the voter has selected the second and third candidates. Then the encryption of a vote $v = (v_1, \ldots, v_l)$ with the public key $\mathsf{pk}$ is simply

$$\mathsf{enc}(v_1, \mathsf{pk}, r_1), \ldots, \mathsf{enc}(v_l, \mathsf{pk}, r_l),$$

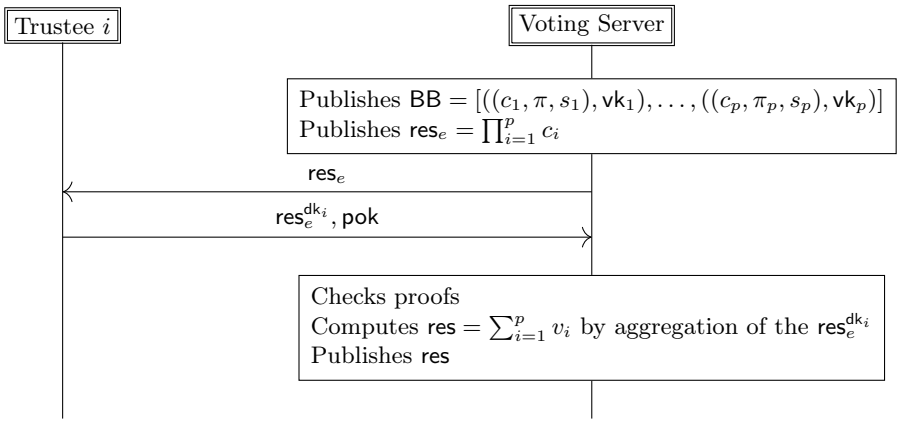


**Fig. 4.** Tally phase.

and the associated zero-knowledge proof guarantees that:

- each $v_i$ is either 0 or 1;
- the voter has selected at least $k_1$ and at most $k_2$ candidates, that is, $k_1 \leq \sum_{i=1}^{l} v_i \leq k_2$.

Then, during the tally, the final ballot box $\mathsf{BB}$ contains ballots $b_i$ of the form

$$b_i = ((c_i^1, \ldots, c_i^l), \pi, s_i, \mathsf{vk}_i).$$

An encrypted result $\mathsf{res}_e^j = \prod_{i=1}^{p} c_i^j$ is computed for each candidate $j$. Each $\mathsf{res}_e^j$ is then decrypted by the decryption trustees, yielding the sum of the votes received by each candidate.

## 3 Design choices and variants

### 3.1 The log file

One of the security goals of Belenios is to prevent ballot stuffing. Intuitively, the authentication of a voter is split into two parts: the login and password authentication on the one hand, and the signature of the ballot on the other hand. Belenios guarantees that no ballot can be added unless both the registrar and the voting server are corrupted.

In the case where revoting is allowed (which is the case in Belenios), then the voting server needs to store the correspondence between a voter and her verification key. This correspondence is used to enforce that a voter does not vote with two distinct verification keys and that no two voters use the same key. This is absolutely necessary to avoid the following attack. Assume the registrar is dishonest, as well as one voter $C$. Assume also that the voters who received the verification keys $\mathsf{vk}_{i_1}, \ldots, \mathsf{vk}_{i_m}$ from the registrar will not vote (in many elections,

the turnout is low). Then, using the login and password of the corrupted voter $C$, the registrar may cast $m$ ballots using successively $\mathsf{vk}_{i_1}, \ldots, \mathsf{vk}_{i_m}$, pretending $C$ is re-voting. This way, the registrar would insert $m$ ballots instead of one.

We could as well provide the voting server with the correspondence between voters and their verification keys at the setup phase. This would not change any security property in the case where revoting is allowed. However, in case revoting is disallowed, the voting sever does not need to log the correspondence between voters and their keys anymore. This provides better *everlasting privacy* [33] guarantees. Indeed, even if cryptography is broken later on and if all data stored on the voting server are lost, it is no longer possible to retrieve who voted what. In this scenario (no revote, no $\mathsf{log}$ file), only the registrar may break everlasting privacy if he does not destroy his initial file that contains the correspondence between voters and signing keys. Note that none of these security claims are currently supported by proofs so they should be used with care.

### 3.2 No weeding

An expert reader may know that Helios (on which Belenios elaborates) requires weeding: the voting server must check that no ciphertext is submitted twice. This is to avoid copy attacks [21]. Imagine that Alice is voting 1 and casts ballot $b_1$, Bob is voting 0, and casts ballot $b_2$. Then, if a dishonest voter Charlie (re)casts $b_1$ pretending it is his own ballot, in the end, the result of the election would be 2 and Charlie deduces that Alice voted 1.

This ballot privacy attack is no longer possible in Belenios thanks to the zero-knowledge proofs. Remember that a ballot is of the form $b = (c, \pi, s), \mathsf{vk}$ where $c = \mathsf{enc}(v, \mathsf{pk}, r)$, $\pi = \mathsf{proofv}(v, r, \mathsf{enc}(v, \mathsf{pk}, r), \mathsf{pk}, \mathsf{vk})$, and $s = \mathsf{sign}(c, \mathsf{sk})$. Assume a dishonest voter $\mathsf{id}'$ wishes to copy the ciphertext $c$ already submitted by a voter $\mathsf{id}$. Then he also needs to use the same proof $\pi$ (since he does not know the randomness $r$). However, since $\pi$ embeds the verification key $\mathsf{vk}$ of voter $\mathsf{id}$, he also needs to include the signature $s$. Now, if voter $\mathsf{id}'$ submits $b$, the ballot would be rejected since $(\mathsf{id}, \mathsf{vk}) \in \mathsf{log}$ with $\mathsf{id} \neq \mathsf{id}'$.

In other words, the zero-knowledge proof guarantees that the ballot has been produced by the voter that received the signing key $\mathsf{sk}$.

### 3.3 The BeleniosRF variant

In Belenios (as in Helios), a voter may prove for whom she voted. Indeed, if Alice publishes the randomness $r$ used to form the encryption of her vote, anyone can re-encrypt using this randomness and check the value of the vote. Note that, of course, our implementation does not provide the voter with a direct tool to obtain her randomness. However, it would be easy for a malicious voter to write her own voting client.

To avoid this issue, a variant of Belenios has been proposed, named BeleniosRF [14], that offers both receipt-freeness and verifiability. It is receipt-free in the sense that even dishonest voters cannot prove how they voted. It relies on re-randomizable signed encryption [13]. Namely, given an (ElGamal) encryption

and its signature, anyone can produce a re-randomized encryption together with a valid signature (without knowing the signing key). Then the key ingredient of BeleniosRF is that the voting server re-randomizes the ballots before publishing them on the ballot box. This way, no voter can provide the corresponding randomness since part of it has been generated by the voting server. Hence the randomness is not known to the voter nor to her voting client. Therefore BeleniosRF prevents behaviours where voters may e.g. tweet for whom they voted. Note however that this is not sufficient to prevent vote buying. Indeed, a voter may still sell her credentials (password and signing key) to an attacker.

### 3.4 The BeleniosVS variant

Another limitation of Belenios is that the voting device of the voter (typically her computer) needs to be trusted. Indeed, a malicious voting device may learn the vote of a voter or even modify it. Building upon BeleniosRF, another variant, BeleniosVS [23], has been proposed, in which the voter receives a voting sheet as part of her voting material. This sheet is generated by the registrar and contains the list of candidates together with a corresponding, signed, encrypted vote next to each candidate. A voter then simply provides her encrypted ballot to her voting device by scanning exactly this ballot and nothing else. This way, the voting device cannot learn the value of the vote (since it is encrypted) nor modify it (since it does not have the signing key). Both the voting device and the voting server re-randomize the ballot in order to break the correspondence between the initial ballot and the vote. Moreover, a voter may audit the voting sheet using another device (e.g. her smartphone) or delegate this audit to a third party, to check that each encrypted ballot does actually correspond to the vote written next to it. For the sake of auditing, the randomness used for encryption is also provided on the voting sheet.

BeleniosVS guarantees both privacy and verifiability even against a dishonest voting device.

## 4 Security proofs

The design of security protocols in general is known to be error-prone and voting protocols make no exception to this rule. For example, the well-known Helios protocol, from which Belenios builds upon, was first proposed in [9] in 2006 and implemented as Helios in 2008 [3]. In 2011, it was found [21] to be subject to a replay attack, which compromises privacy. Namely, dishonest voters can collude and can all vote as Alice (without knowing Alice's vote). Then dishonest voters may infer information on Alice's vote from the result of the election.

Therefore, the state-of-the-art practice consists in *proving* the security of protocols. A security proof identifies in particular what are the security guarantees and the trust assumptions. For example, the Swiss Chancellerie requires [2] that "there exists a cryptographic proof and a symbolic proof [of the voting protocol]". What does this mean? Two distinct approaches have been developed for

analysing and proving security protocols, developed by two distinct communities (resp. logic and cryptography): symbolic and computational models. Symbolic models analyse the logical flow of protocols, with an abstract representation of the cryptographic primitives, based on rewriting or logic. Mature push-button tools such as ProVerif [12] or Tamarin [37] can automatically find flaws or formally prove security in symbolic models, possibly with some user guidance for Tamarin. Computational models are based on complexity theory. Namely, the security of a protocol is reduced to some algorithmically hard problem such as discrete logarithm or factorisation. The execution model is specified down to the bitstring level, yielding higher guarantees but also more complex proofs. Computational proofs of protocols are typically done by hand (e.g. [29,3] for voting protocols) with a recent attempt of a machine-checked framework [6] using the EasyCrypt tool [6]. EasyCrypt is an interactive theorem prover specialized in proofs of probabilistic equivalence of programs and well adapted to cryptographic security proofs. Reading hand-written proofs in this domain requires a lot of expertise and spotting mistakes is difficult. Therefore using a tool like EasyCrypt provides a higher level of confidence in the proofs.

Both symbolic and computational proofs have been conducted for Belenios. However, the existing symbolic proofs of Belenios have been developed as an illustration of a proof technique and remain quite abstract. For example, [5] shows that Belenios preserves vote privacy in a simplified model where the registrar is not represented explicitly. There is no proof of verifiability. Since BeleniosVS has been proved verifiable, the corresponding symbolic proof could probably be adapted to Belenios but this has not been done yet.

Therefore, in the rest of this section, we will focus on the proof of both privacy and verifiability, conducted in a cryptographic model [16]. These proofs have been established with the aforementioned EasyCrypt tool. In the remainder of this section, we first sketch the formal definitions of privacy and verifiability and we then detail the security guarantees and the corresponding trust assumptions for Belenios.

### 4.1 Overview of the privacy and verifiability definitions

In cryptographic models, messages are bitstrings and the adversary is any probabilistic polynomial time Turing machine. This represents the fact that an adversary may use any algorithm, provided it runs in a reasonable amount of time. Of course, the adversary controls all public communications and may send any message it can compute. Security proofs work by reduction: breaking the security of a (voting) protocol should be as hard as breaking some well known algorithmic problems. For example, Belenios uses ElGamal encryption and its security relies on the difficulty of solving the decisional Diffie-Hellman problem.

**Vote privacy** There is no well established consensus on how to define vote privacy. Several definitions have been proposed, often through games: the attacker should not observe any difference when Alice is voting 0 or 1. We chose here to consider the privacy definition BPRIV [10].

Intuitively, BPRIV defines an experiment where the adversary tries to distinguish between two worlds: a "real world" and a "simulated world". For this, we give to the adversary the power to ask the (honest) voters to vote differently in both worlds, and all the votes in both worlds are known to the adversary (they can actually be chosen by him). At any time in the experiment, the adversary can look at the public board of the world he is in. In a secure scheme, the public board contains only the encrypted ballots, so there is no direct way for the adversary to deduce in which world he is, even if the votes are different.

At any time in the experiment, the adversary can also emulate a dishonest voter and cast any ballot, not necessarily coming from the legitimate voting algorithm. For instance, he can attempt to forge a ballot from what he has previously seen in the public board of his world. As long as it is recognized as a valid ballot by the validation algorithm of the protocol, it will be cast in both worlds.

In the end of the experiment, the adversary gets the result of the election. To avoid a trivial attack, the adversary is always given the tallying function applied to the public board of the real world, even if he is in the simulated world (otherwise, this would immediately reveal the answer to the adversary, since he knows the votes and they can be different in both worlds). The additional data, for instance, the proof of correct decryption, is computed by the legitimate algorithm in the real world, or is computed by a simulator (an algorithm to be defined in the security proof) in the simulated world. If it can be proven that no polynomial-time adversary can guess with a non-negligible advantage in which world he is, then the scheme respects privacy.

This definition is meant to capture the fact that, besides the result of the election, no other data should leak information about the votes.

**Verifiability** Again, several notions of verifiability have been proposed in the literature, surveyed for example in [19]. Intuitively, verifiability ensures that votes are correctly reflected in the result of the election. We distinguish between three types of voters:

- Honest voters that follow the voting protocol exactly as expected. In particular, they perform the required checks. In Belenios, honest voters are supposed to check that their ballot is included in the (public) ballot box.
- Honest voters that do not check. Unfortunately this corresponds to the majority of voters: voters follow the protocol but not entirely, they stop once they have cast their ballot.
- Dishonest voters are fully controlled by the attacker and may submit anything as their own ballot (if they wish to).

In what follows, we will say that a protocol is *verifiable* if the result of the election corresponds to:

- all the votes from voters who checked;
- a subset of the votes from voters who did not check;

– an arbitrary set of valid votes, of size smaller than the number of corrupted voters. This last part guarantees that there is *no ballot stuffing*: the attacker cannot control more votes than the number of dishonest voters.

We refer the reader to [18] for a formal definition. Since the attacker controls all the public communications, Belenios cannot guarantee that votes of voters that did not check will be counted. Indeed, the corresponding ballots may have been dropped by an attacker. However, Belenios guarantees that these votes cannot be modified by the attacker.

### 4.2 Security guarantees of Belenios

As mentioned earlier, the security definitions as well as the corresponding proofs have been fully developed through the EasyCrypt tool, forming the first machine-checked proof of both verifiability and privacy of a deployed voting protocol. We now spell out our trust assumptions, summarised in Figure 5.

| | Number of dishonest authorities | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Decryption trustees | $\leq t$ | $\leq t$ | $\leq t$ | $\leq t$ | $> t$ | $> t$ | $> t$ | $> t$ |
| Registrar | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| Voting Server | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| Verifiability | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ |
| Privacy | ✓ | . | . | . | ✗ | ✗ | ✗ | ✗ |

✓ indicates that the property is satisfied. ✗ indicates that the property is not satisfied. . indicates that there is no formal proof, yet no attack is known. As in Section 2.3, $t$ is the threshold decryption parameter, *i.e.* at least $t + 1$ contributions are required to be able to decrypt.

**Fig. 5.** Trust assumptions for Belenios.

**Verifiability** Belenios is verifiable provided that the registrar or the voting server are honest and the voting device of the voter is honest. The decryption trustees may all be corrupted.

**Privacy** Belenios guarantees vote privacy provided that both the registrar and the voting server are honest, that the voting device of the voter is honest, and that at most $t$ decryption trustees are corrupted (where $t$ is the threshold used to generate the key, as explained in Section 2.3).

**Discussion** Why do we need to assume that both the registrar and the voting server are honest for privacy? Intuitively, there is no reason for that. On the contrary, Belenios is designed to preserve vote privacy even if both the registrar

and the voting server are corrupted since these two authorities are rather in charge of ensuring that only legitimate voters can vote. The first reason is that existing definitions of privacy in a computational setting all implicitly assume an honest voting server. Thus we cannot prove privacy in a setting that has not been defined yet. The second, deeper, reason is that there are subtle relations between verifiability and privacy. In a scenario where an attacker may selectively drop votes, he can thereby learn information from the result. In particular, it has been recently shown that the current definitions of privacy imply individual verifiability, that is, they imply that all honest votes are counted (including votes from voters that do not check) [20]. These limitations apply to the other voting schemes as well.

Note that as stated in the description of Belenios, we also assume, for both properties, that the voting device is honest. Indeed, since the voter selects her voting choice thanks to her voting device, the voting device automatically learns the vote. Moreover, a corrupted voting device may easily change the vote by encrypting 1 when a voter selects 0 for example. Helios includes a cast-or-audit mechanism. Indeed, a voter can interact with her voting device to check that it behaves as expected. When she is satisfied, she can then use her device for the actual vote. This mechanism could easily be added to Belenios. We chose not to include it as it is not really used in practice and it is easy to target attacks to voters that are more likely to avoid checks.

## 5 Implementation and deployment of the public platform

### 5.1 Source code and system aspects

We have written a full implementation of the Belenios protocol following the specification. The current version is 1.8 and corresponds to the version 1.6 of the specification (the versioning numbers are independent). The source code is written in OCaml and is regularly checked and updated if necessary to work with the latest version of OCaml. The few non-OCaml dependencies (`wget`, `zip`, `openssl`) are standard tools easily available, for instance in a Debian Linux distribution. The implementation provides a command-line tool called `belenios-tool` that allows to perform all the algorithmic steps required in the protocol. It can be compiled separately from the other part which contains a web server allowing the deployment of elections. For this web part, the same back-end is called for the algorithmic operations of the protocol, the `http` server is the OCaml `ocsigen` server, and the web application is programmed within the `eliom` OCaml package. This use of a consistent framework for the whole code of the platform allows to share several parts of the code between the server (for which the OCaml code is compiled to the native machine language of the host) and the web browser on the client side for which the OCaml code is compiled to Javascript. These Javascript clients depend on the classical libraries `jsbn` and `sjcl` for low-level big integer arithmetic and cryptographic functions.

The command-line program `belenios-tool` is available as a Debian package, but the web part is not, due to difficulties with some OCaml dependencies within the Debian distribution.

The web interface allows an election administrator to setup a new election, in interaction with the registrar and the decryption trustees. She is responsible in providing the list of e-mail addresses of voters, to which a login/password is sent by the server. The signing keys are sent to the voters by the registrar; usually this is also done by e-mail, but it could be sent via any other channel. This use of e-mail as an implementation of the private channels in Figure 1 is certainly a weakness of our on-line platform. For a high-stake election, the way to send the election material must be adapted, taking into account what is realistically feasible (postal mail, use of existing e-IDs, ...). In a context where all the voters belong to a same entity that provides a single-sign-on solution, this can be a replacement to the login/password authentication. There is actually support for the CAS protocol in our implementation.

On the voting side, the easiest way to vote is to use the web-interface, following the URL of the election. Then, the ballot is prepared entirely on the client side with a Javascript code sent by the server. Since it is not easy to provide guarantees that the Javascript code is really the one that it is supposed to be, there is, in principle, a safer way to prepare the ballot. Indeed, the interface proposes to upload directly an encrypted ballot prepared externally from the browser, for instance with `belenios-tool`. This tool can be installed from a Debian distribution, with the standard package signature mechanism, which gives guarantees on the authenticity of the code. Yet another possibility is to download the sources of Belenios, compile them, and use the generated Javascript code directly instead of the one provided by the server.

The online platform [1] which is available freely for anyone who wants to run an election, with a limitation on the number of voters, is running on a machine that is hosted in our research laboratory. The software deployed is exactly the same as the Belenios package that is freely distributed. The additional features that had to be added are the configuration for the network (a reverse proxy) and the e-mail, some backup mechanism, and monitoring and statistics tools. Although the platform is reasonably monitored for suspicious behaviours, we do not provide a strong hardening, for instance against denial of service, and we do not guarantee 24/7 uptime. Therefore, our online platform does not claim the robustness that one could expect for a high stake election.

### 5.2 Implementation issues related to voters

While working on Belenios, both as a protocol specification and as an online platform, we took into account the usability for the voters as a strong criterion. Some of the features that we list here were actually implemented after feedback from our users.

**Size of the voting material** A first important issue for the voters is the size of the voting material. In the current setting of our platform where it is sent

by e-mail, everything can be copy-pasted. So limiting the size of the material is not such a strong requirement. But we keep in mind that, in some context, the voter might have to type their voting material. Therefore, the registrar does not send the signing key itself to the voter but a 15-character string that includes a checksum. The corresponding 88 bits of entropy are used to derive the signing key with the PBKDF2 primitive. In the same spirit, the encrypted ballot is not presented as such to the voter, but only its hashed version is shown during the preparation, sent by e-mail, and printed in the ballot-box (of course, the raw ballots are also easily available for verifiability).

**Multilingual support** Since our first users were from the academic world which is highly international, we quickly felt the need to have multilingual support for the part of the interface that is exposed to the voters. Indeed, although most of them are comfortable with the basic English used in scientific articles, the vocabulary of an election (voting booth, credential, ballot, tally, ...), is not well known to non-native speakers. We currently support English, French, German, Italian, Romanian, and adding a new language is not difficult. For the moment, the web interfaces for the election administrator, the registrar, and the decryption trustees are in English only.

**Re-sending the voting material** Another predictable request from users is to have a way to receive their voting material again, in case they have lost it. It is even more important in the setting of our platform, where the material is sent by e-mail and the messages are considered as spam by some automatic filters due to the presence of URLs and the key-words login/password. There is no difficulty on the voting server side which can easily generate a new password and send it to the voter. For the registrar, there is no need to keep the list of signing keys once they have been sent to the voters, except if a voter loses her key. Since we did not want to impose to the registrar the need to keep secrets for a long time, the specification contains a protocol of credential recovery (see Section 3.3 of [26]) in which the registrar generates a new signing key and sends the corresponding updates to the server, to ensure that the old key had not been used in the past and will not be used in the future. We remark however that, in practice, the registrars usually prefer keeping the list of signing keys to be able to send them again instead of running this credential recovery protocol with the server.

### 5.3 Other features of Belenios

**Counting the blank votes** In an early version of the Belenios specification, the only elections that could be setup were the one described in Section 2.4, namely choosing $k$ candidates among $l$, where $k$ lies in a prescribed set of values. Allowing $k$ to be 0 was a way for voters to express a blank vote. The problem with this easy solution is that counting the number of blank votes is not always possible. This was a major missing feature for some potential users, who complained.

Therefore we added a bit for encoding a blank vote in the ballots, together with the corresponding zero-knowledge proof that, if this bit is activated, no other one is (see Section 4.10 of [26] or [25]). Other frequent requests include various advanced counting functions, but most of them are not compatible with the homomorphic decryption, and we postpone their support to a future inclusion of verifiable mixnets in Belenios.

**Secure channels between different parties** In the formal protocol description, some messages are sent via a secure channel from one authority to another, in particular for the election key setup, in the threshold mode (Figure 2). In practice, everything is organized around the web server that plays the role of a hub through which all the messages are transmitted. Unfortunately, it is not realistic in our setting to assume that all the decryption trustees possess signing and encryption keys that can be used for ensuring secure channels to and from them. Therefore, in a first step of the key generation protocol, each decryption trustee starts by generating a random secret seed from which she derives cryptographic keys that are the basis of a custom PKI (see Section 4.5.1 of [26]). Then, the messages from one trustee to another go through the server, and thanks to the PKI, the server is just part of the (untrusted) network in the abstract model.

Although the support for threshold decryption has been implemented for more than a year, it has been only recently added to the web interface of the online platform. We have tried to keep the tasks of the decryption trustees as easy as possible, and the PKI is part of this effort, but we do not have feedback yet from users. To our knowledge, this state-of-the-art threshold decryption protocol is the first one to be implemented on a public voting platform.

**Degraded mode for testing purpose** For very low stake elections (which pizza for tonight?) or for testing purposes, the web server can emulate the roles of the registrar and of the decryption trustees. This mode is tempting for election administrators since it makes their life simpler. However, in this mode, the system administrators of the machine that runs the web server become powerful attackers. If we do not assume that they can be trusted, almost no security property is preserved, apart from some verifiability. Namely, as in Helios, voters who check the presence of their ballot are guaranteed that their vote is counted but extra ballots may have been added, taking advantage of abstention. To mitigate the danger of the presence of such a degraded mode, the server forgets the signing keys once they are sent to the voters. As a consequence, ballot-stuffing by the system administrators must be planned in advance to be successful. Also, the voters can not ask the server to send this signing key again if they lose it. This last part is actually a strong incentive not to use the degraded mode for real elections. Unfortunately, we still observe many elections being run in degraded mode while not of so low stake as we would expect. When asked, the users simply answer that they are happy to trust the system administrators.

**Auditing and monitoring** A strong assumption in the security analysis of Belenios is that all the process is monitored and audited by sufficiently many independent participants. For this, tools are needed and ideally, they should be written by developers different from the one of Belenios, directly from the specification document. For the moment, the only available program is `belenios-tool` which provides two auditing commands: `verify` and `verify-diff`. The first one checks the consistency of all the public data at a given time during an election, including the decrypted tally if the election if finished. All the zero-knowledge proofs and all the signatures are checked; no two ballots can be signed by the same key. The `verify-diff` command takes as input two snapshots of the public data and checks that the second one is a valid future state of the first one. This includes checking that the size of the ballot box is increasing, and more precisely that if a ballot has disappeared, another one has been cast with the same signature key (i.e. someone can re-vote, but the voting server did not drop ballots). This command is also aware of the credential recovery protocol and will check that if the list of verification keys is modified, everything is consistent with the protocol.

### 5.4 Usage statistics

After a long test period of more than one year in 2016-2017 where various elections were organized with a big help from us, we have now reached a point where the organizers are autonomous and most elections are done without interaction with us. In the rest of this section, all the data and comments are related to the year 2018 and refer to this period where our on-line platform was no longer considered in test. General statistics for this 2018 year are:

| | |
|---|---|
| Number of elections | 142 |
| Total number of voters | 17 650 |
| Total number of tallied ballots | 5 579 |

**Type of elections** Given the academic nature of the Belenios project, it is not a surprise that most of the elections organized on the platform are related to (mostly French) academia. This includes elections for representatives in councils of research laboratories, elections for the head or for representatives of scientific working groups or learned societies, or elections during committees for promotions. A few not-so-small elections are for representatives in associations that are unrelated to academia. For numerous small elections we are not aware of the context at all.

**Sizes** We have a limit to 1000 voters for a single election. Occasionally, upon request, we increase this limit for a larger election to be held on our platform. The picture in Figure 6 shows how the sizes of the 142 elections held in 2018 are distributed and the turnout for them. Statistics for five intervals of the logarithm of the number of voters are given in the following table:

| Number of voters | Number of elections | Average turnout | Elections with external trust party |
|---|---|---|---|
| $\cdot < 12$ | 15 | 84.5% | 46.7% |
| $12 \leq \cdot < 50$ | 62 | 62.9% | 29.0% |
| $50 \leq \cdot < 200$ | 45 | 41.6% | 51.1% |
| $200 \leq \cdot < 800$ | 14 | 25.6% | 21.4% |
| $800 \leq \cdot$ | 6 | 22.5% | 67.7% |
| Total | 142 | 31.6% | 38.7% |

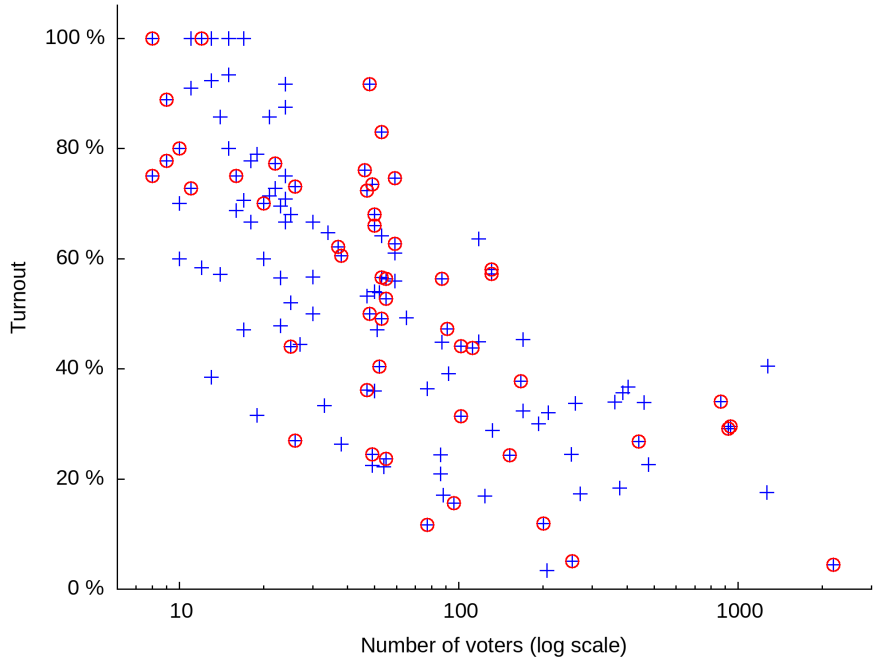


**Fig. 6.** Elections run on the Belenios platform in 2018. Each + symbol corresponds to an election. When a red circle ○ surrounds it, it means that at least one of the roles (registrar or decryption trustee) was not emulated by the server but was held by a third party.

**Use of the degraded mode** In Figure 6, we see that the majority of the elections are configured in such a way that the server plays all the roles, so that the security is severely degraded: the trust on the administrators of the server is almost total.

We summarize the number of elections that were run for various security configurations.

| Security configuration | # elections | Comment |
|---|---|---|
| Server plays all the roles (no circle on Fig. 6) | 87 | Organizer made no effort regarding security |
| Other configurations with a unique decrypt. auth. | 27 | Privacy issue |
| At least 2 decrypt. auth. and server is cred. auth. | 4 | Ballot stuffing issue |
| At least 2 decrypt. auth. and external cred. auth. | 24 | Scenario corresponding to security proofs |

We noticed that for elections where the administrators "have to" care about security due to the regulations, they usually do it. This is especially the case for the administrators who have been using the platform during the testing phase where we had a lot of interaction with them: now that they use it by themselves, they continue to follow our advices. This corresponds to the cluster of circles for elections of about 50 voters on Figure 6. Some of them have a high turnout, which might indicate that they are indeed considered important and deserve a high level of security.

## 6 Further possible developments

**Public board in practice** Belenios, like several e-voting protocols, relies on the notion of public board were the ballots are recorded. In the current platform, it is implemented in the most naive way, as a public web page, and it is assumed that enough parties will monitor it, so that it is consistent. A more advanced decentralized public board, together with a state-of-the-art consensus protocol would certainly be preferable. Since this is not specific to Belenios, it makes no sense to develop a specific tool, though. In the meantime, providing better tools for monitoring and auditing the public board would be useful to push more users to contribute to the security of their election.

**More trust in the software** All the parties, the voters, the registrar, the decryption trustees, use software during their participation to the protocol. The easiest way for the voters and the decryption trustees is to use the Javascript code that is provided by the web platform and do everything in the browser (for the registrar, this is not possible, since she must send e-mails to the voters). A cast-or-audit (or more generally compute-or-audit) mechanism could be added. But we still need to find the appropriate mechanism to ensure that the users will really perform the checks. It might then be better to allow ourselves to modify the protocol, and, for instance, design a practical variant of BeleniosVS that would be robust in the case of a corrupting device.

An easier, but less elegant approach, would be to provide standalone applications that work outside the browser and rely on the security mechanisms of the application store of the operating system for ensuring the traceability of the software that has been installed and run.

Another possible direction to enhance the amount of trust in the software is to formally prove part of it with tools like F*[39]. The difficulty might be to get a proof that goes all along to the GUI (and ensures that "yes" and "no" are not swapped on the screen of the voter).

**More types of elections via mixnets** The use of homomorphic encryption does not allow complex counting functions. Switching to verifiable mixnets is not a problem from a theoretical point of view, and in fact, the security proofs mentioned in Section 4 have also been done in this context. The implementation is work in progress, and the list of counting functions that we plan to support in the platform is not yet decided.

We had also demands from users for votes with weights. Depending on the situation, it might be necessary to include some weight randomization to avoid privacy issues due to underlying Knapsack problems [4]. Adding this kind of feature on an online platform requires great care, because it can lead to confusion of the users.

## References


1. Belenios – Verifiable online voting system. `http://www.belenios.org/`.
2. Exigences techniques et administratives applicables au vote électronique. Chancellerie fédérale ChF, 2014. Swiss recommendation on e-voting.
3. Ben Adida. Helios: Web-based open-audit voting. In *17th USENIX Security Symposium (Usenix'08)*, pages 335–348, 2008.
4. Ben Adida, Olivier de Marneffe, Olivier Pereira, and Jean-Jacques Quisquater. Electing a University President Using Open-Audit Voting: Analysis of Real-World Use of Helios. In *Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*. Usenix, 8 2009.
5. Myrto Arapinis, Véronique Cortier, and Steve Kremer. When are three voters enough for privacy properties? In *21st European Symposium on Research in Computer Security (Esorics'16)*, volume 9879 of *LNCS*, pages 241–260. Springer, 2016.
6. Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design (FOSAD'13)*, pages 146–166, 2013.
7. Susan Bell, Josh Benaloh, Michael D. Byrne, Dana Debeauvoir, Bryce Eakin, Philip Kortum, Neal McBurnett, Olivier Pereira, Philip B. Stark, Dan S. Wallach, Gail Fisher, Julian Montoya, Michelle Parker, and Michael Winn. STAR-Vote: A secure, transparent, auditable, and reliable voting system. In *Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE'13)*, 2013.
8. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM-CCS'93*, 1993.
9. Josh Benaloh. Simple verifiable elections. In *USENIX Security Symposium (EVT'06)*, 2006.
10. David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. A comprehensive analysis of game-based ballot privacy definitions. In *36th IEEE Symposium on Security and Privacy (S&P'15)*, pages 499–516. IEEE Computer Society Press, May 2015.

11. David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In *Advances in Cryptology - ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 626–643. Springer, 2012.
12. Bruno Blanchet. Automatic verification of security protocols in the symbolic model: The verifier ProVerif. In *Foundations of Security Analysis and Design (FOSAD'13)*, volume 8604 of *LNCS*, pages 54–87. Springer, 2013.
13. Olivier Blazy, Georg Fuchsbauer, David Pointcheval, and Damien Vergnaud. Signatures on randomizable ciphertexts. In *Public Key Cryptography - PKC 2011*, pages 403–422, Taormina, Italy, 2011.
14. Pyrros Chaidos, Véronique Cortier, Georg Fuchsbauer, and David Galindo. BeleniosRF: A non-interactive receipt-free electronic voting scheme. In *23rd ACM Conference on Computer and Communications Security (CCS'16)*, pages 1614–1625, Vienna, Austria, 2016.
15. M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy (S&P'08)*, pages 354–368. IEEE Computer Society, 2008.
16. Véronique Cortier, Constantin Catalin Dragan, Pierre-Yves Strub, Francois Dupressoir, and Bogdan Warinschi. Machine-checked proofs for electronic voting: privacy and verifiability for Belenios. In *31st IEEE Computer Security Foundations Symposium (CSF'18)*, pages 298–312, 2018.
17. Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachene. Distributed ElGamal à la Pedersen - application to Helios. In *Workshop on Privacy in the Electronic Society (WPES 2013)*, Berlin, Germany, 2013.
18. Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachene. Election verifiability for Helios under weaker trust assumptions. In *19th European Symposium on Research in Computer Security (ESORICS'14)*, volume 8713 of *LNCS*, pages 327–344. Springer, 2014.
19. Véronique Cortier, David Galindo, Ralf Küsters, Johannes Müller, and Tomasz Truderung. SoK: Verifiability notions for e-voting protocols. In *36th IEEE Symposium on Security and Privacy (S&P'16)*, pages 779–798, San Jose, USA, May 2016.
20. Véronique Cortier and Joseph Lallemand. Voting: You can't have privacy without individual verifiability. In *25th ACM Conference on Computer and Communications Security (CCS'18)*, pages 53–66. ACM, 2018.
21. Véronique Cortier and Ben Smyth. Attacking and fixing Helios: An analysis of ballot secrecy. *Journal of Computer Security*, 21(1):89–148, 2013.
22. Edouard Cuvelier, Olivier Pereira, and Thomas Peters. Election verifiability or ballot privacy: Do we need to choose? In *18th European Symposium on Research in Computer Security (ESORICS'13)*, pages 481–498, 2013.
23. Alicia Filipiak. *Design and formal analysis of security protocols, an application to electronic voting and mobile payment.* PhD thesis, Université de Lorraine, March 2018.
24. David Galindo, Sandra Guasch, and Jordi Puiggali. 2015 Neuchâtel's cast-as-intended verification mechanism. In *5th International Conference on E-Voting and Identity, (VoteID'15)*, pages 3–18, 2015.
25. Pierrick Gaudry. Some ZK security proofs for Belenios. `https://hal.inria.fr/hal-01576379`, 2017.
26. Stéphane Glondu. Belenios specification - version 1.6. `http://www.belenios.org/specification.pdf`, 2018.

27. Rolf Haenni, Reto E. Koenig, Philipp Locher, and Eric Dubuis. CHVote system specification. Cryptology ePrint Archive, Report 2017/325, 2017.
28. J. Alex Halderman and Vanessa Teague. The New South Wales iVote system: Security failures and verification flaws in a live online election. In *5th International Conference on E-voting and Identity (VoteID '15)*, 2015.
29. A. Juels, D. Catalano, and M. Jakobsson. Coercion-resistant electronic elections. In *Workshop on Privacy in the Electronic Society (WPES'05)*, pages 61–70. ACM, 2005.
30. Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. DEMOS-2: Scalable E2E verifiable elections without random oracles. In *ACM Conference on Computer and Communications Security (CCS'15)*, 2015.
31. Ralf Küsters, Johannes Müller, Enrico Scapin, and Tomasz Truderung. sElect: A lightweight verifiable remote voting system. In *29th IEEE Computer Security Foundations Symposium (CSF'16)*, pages 341–354, 2016.
32. Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Accountabiliy: Definition and relationship to verifiability. In *17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 526–535, 2010.
33. T. Moran and M. Naor. Receipt-free universally-verifiable voting with everlasting privacy. In *CRYPTO 2006*, volume 4117 of *LNCS*, pages 373–392. Springer, 2006.
34. Torben Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO 1991*, pages 129—140, 1991.
35. Peter Ryan. Prêt à Voter with Paillier encryption. *Mathematical and Computer Modelling*, 48(9–10):1646–1662, 2008.
36. Peter Y. A. Ryan, Peter B. Roenne, and Vincenzo Iovino. Selene: Voting with transparent verifiability and coercion-mitigation. In *1st Workshop on Secure Voting Systems (VOTING'16)*, pages 176–192, 2016.
37. Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *25th IEEE Computer Security Foundations Symposium (CSF'12)*, pages 78–94, 2012.
38. Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J. Alex Halderman. Security analysis of the Estonian Internet voting system. In *11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 703–715, 2004.
39. Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*, pages 256–270. ACM, 2016.
40. Scott Wolchok, Eric Wustrow, Dawn Isabel, and J. Alex Halderman. Attacking the Washington, D.C. Internet voting system. In *Financial Cryptography and Data Security (FC'12)*, 2012.