How to challenge *and* cast your e-vote

Sandra Guasch[1], Paz Morillo[2]

Scytl Secure Electronic Voting[1], Universitat Politecnica de Catalunya[2]
`sandra.guasch@scytl.com`, `paz@ma4.upc.com`

Abstract. An electronic voting protocol provides cast-as-intended verifiability if the voter can verify that her encrypted vote contains the voting options that she selected. There are some proposals of protocols with cast-as-intended verifiability in the literature, but all of them have drawbacks either in terms of usability or in terms of security. In this paper, we propose a new voting scheme with cast-as-intended verifiability which allows to audit the vote to be cast, while providing measures for avoiding coercion by allowing the voter to create fake proofs of the content of her vote. We provide an efficient implementation and formally analize its security properties.

1 Introduction

In remote e-voting schemes the vote is encrypted at the same voter device used to choose the selections and cast the vote. This way voter choices remain secret during their transmission and storage in the remote voting server. Encrypted voting options cast by the voters are anonymized prior to decryption at the counting phase, in order to maintain voter privacy.

This introduces new concerns on e-voting systems: how can voters be sure that (i) the vote that their device encrypted contains their selections, (ii) the vote which was stored in the remote voting server is the same that their device encrypted and (iii) the anonymization and decryption processes were done correctly? To provide assurance to the voters that the vote casting, vote storage and vote counting processes were done following the specified protocol, the notions of cast-as-intended verifiability, recorded-as-cast verifiability and counted-as-recorded verifiability have been introduced in the literature.

There are some satisfying solutions for both recorded-as-cast verifiability and counted-as-recorded verifiability. Recorded-as-cast verifiability can be achieved by publishing all encrypted votes received at the remote voting server in a Bulletin Board [17], where voters can check for their cast votes, and making sure that only the votes which are in the Bulletin Board are tallied. For examples of protocols providing counted-as-recorded verifiability, see [27] or [12], where verifiable mix-nets and verifiably homomorphic tally schemes are introduced, respectively.

However, proposed schemes achieving cast-as-intended verifiability still have some drawbacks, either in usability or security terms. For example, some of these systems, known as *challenge-or-cast*, do not allow to audit the same vote to be cast. In case such systems allowed to verify the same ballot to be cast, they would fail on fulfilling other security requirements for electronic voting systems such as protection against voter coercion and vote selling, given that their verification involves providing the randomness of the encrypted vote.

In this paper, we propose a new voting scheme which provides cast-as-intended verifiability. In our scheme, the voter can audit the same encrypted vote that she will later cast, what we think is an improvement from the point of view of soundness of the verification and of the usability of the system: it represents a more straightforward process for average voters to audit the vote that is going to be cast. Still, measures are applied in order to ensure that this verification does not provide the voter with a receipt that can be used to sell her vote. We call this variant **challenge-*and*-cast**.

Related Work There have been several proposals of cast-as-intended verification schemes during the last two decades. In Helios [2], the voter's device encrypts a vote and the voter is allowed to challenge the encryption and obtain the randomness used for encrypting the voting options, to check that the encrypted vote was constructed correctly. However, in order to prevent vote selling, the vote has to be encrypted again with new randomness, after auditing. In particular, this means that the voter's device has a small probability of cheating, which decreases with the number of challenged encryptions.

Other methods to provide cast-as-intended verification are those based in return codes, such as [18]. In these schemes, a secondary channel is used to deliver reference codes assigned to voting options to each voter before the voting phase. During voting, the remote voting server, computes return codes from received votes and sends them back to the voters, who verify that they match the expected reference codes. These solutions are more usable than previous proposals, but they require a secondary channel, which may not always be available.

Some code voting schemes (Surevote [8], Pretty Good Democracy [26]) or verifiable DRE-based schemes (MarkPledge [24], Moran-Naor's receipt-free voting system [23]) also provide cast-as-intended verifiability. However, they present their own limitations: while the first category relies on the voter having to enter one randomized code for each voting option she selects, with the corresponding drawbacks on usability, the second one requires specific hardware (such as printers with protected output trays) which cannot be assumed to be available in remote voting scenarios.

Finally, there are systems such as [10] which use trapdoor commitments, as our solution, in order to provide receipt-freeness in blind signature voting schemes. However they struggle on the way of providing the voter with the trapdoor key. Although we also use trapdoor commitments in our scheme, we have naturally associated the trapdoor key to a voting credential needed to cast a ballot, in order to improve the usability and understandability of the scheme. Moreover, the protocol we present here does not aim to solve the problem of receipt-freeness. Getting a receipt of your vote is a possibility inherent to most the electronic voting systems where the vote is encrypted at the voting device, such as Helios [2]. Instead, our motivation is to provide a method for cast-as-intended verification which does not involve providing a receipt to the voter (or that at least allows to fake it for a possible coercer). A future work will consist on analyzing how this cast-as-intended verification method can be combined with other systems providing receipt-freeness or even coercion-resistance, such as JCJ [21].

Overview The solution is the following: the voting device encrypts the vote and shows the resulting ballot to the voter, together with a zero-knowledge proof of knowledge (ZKPK) of the encryption randomness instead of revealing the plain value, as in the challenge-or-cast mechanisms. After the voter agrees on the proof, the ballot is cast and published on the bulletin board, so that the voter can check that her ballot has been correctly received at the voting platform. The voter agreement of the proof is represented with an authentication of the ballot, and only authenticated ballots are accepted in the system (posted on the bulletin board).

The cast-as-intended verification is still sound compared to prior systems, thanks to the properties of the proofs of knowledge: the verification of the proof will succeed only in case the voter's device is honest (i.e., the device is encrypting the voting options selected by the voter). In case of a dishonest device, the probability of the proof being successfully verified (and thus, the voter being cheated without notice) is negligible. At the same time, the scheme provides protection in front of vote selling/voter coercion scenarios thanks to the fact that it generates a ZKPK instead of providing the value itself. With the proof itself, the voter can be easily coerced or she can sell her vote. However, we take advantage of the fact that ZKPKs can be simulated to give a chance to the voter to cheat the coercers/vote buyers: In our scheme, the voter is allowed to generate *fake* proofs that will look like good proofs to anyone else.

This paper is organized as follows: Section 2 gives an introduction to the techniques used for the simulation of the proofs of the ballot content; Section 3 presents the syntax and description of the protocol, as well as the trust assumptions; Section 4 provides an efficient instantiation; a discussion about the voter experience using this protocol is provided in Section 5. Finally, an extension for multiple voting is provided in Section 6. The annexes contain some security definitions of the scheme and the result of the security analysis.

2 Proof simulation

The scheme uses Designated Verifier Proofs [20], which allow a designated proof verifier to get convinced of a statement, while she is able to simulate proofs for other statements (which are not true) to other verifiers. In our scheme, the prover is the voting device, who proves knowledge of the encryption randomness, and the designated verifier is the voter. Other verifiers such as possible coercers or vote buyers cannot be convinced by the proof. Designated Verifier Proofs use trapdoor commitments, also known as chameleon commitments [7]. The trapdoor information is only available to the designated verifier of the proof, who can use it to generate simulated proofs for other verifiers. In non-interactive settings, such as in non-interactive zero-knowledge proofs of knowledge (NIZKPKs), Chameleon hashes [22] are used rather than chameleon commitments.

Chameleon Hashes A chameleon hash function is a trapdoor collision-resistant hash function. Without knowledge of the trapdoor, the chameleon hash behaves as an ordinary collision-resistant hash function. However, using the trapdoor, collisions can be found efficiently.

A chameleon hash function is composed by three p.p.t. algorithms: $\mathsf{Gen}_{\mathsf{ch}}$ takes as input a security parameter 1^k, outputs an evaluation key $\mathrm{ek}_{\mathrm{ch}}$ and a trapdoor key $\mathrm{tk}_{\mathrm{ch}}$, and defines a message space $\mathcal{M}_{ch}$, a randomness space $\mathcal{R}_{ch}$ and a hash space $\mathcal{Y}_{ch}$; $\mathcal{H}_{ch}$ takes as input an evaluation key $\mathrm{ek}_{\mathrm{ch}}$, a message $m \in \mathcal{M}_{ch}$ and a random value $\mathrm{r}_{\mathrm{ch}} \in \mathcal{R}_{ch}$ and outputs a hash value $\mathrm{c}_{\mathrm{ch}} \in \mathcal{Y}_{ch}$; $\mathcal{H}_{ch}^{-1}$ takes as input the trapdoor $\mathrm{tk}_{\mathrm{ch}}$, two messages $m, m' \in \mathcal{M}_{ch}$ and a random $\mathrm{r}_{\mathrm{ch}} \in \mathcal{R}_{ch}$, and returns a value $\mathrm{r}_{\mathrm{ch}}' \in \mathcal{R}_{ch}$ such that $\mathcal{H}_{ch}(\mathrm{ek}_{\mathrm{ch}}, m, \mathrm{r}_{\mathrm{ch}}) = \mathcal{H}_{ch}(\mathrm{ek}_{\mathrm{ch}}, m', \mathrm{r}_{\mathrm{ch}}')$.
Chameleon hashes have the following properties:

COLLISION RESISTANCE. Provides that, given only the evaluation key $\mathrm{ek_{ch}}$, the probability of finding $(m, \mathrm{r_{ch}}) \neq (m', \mathrm{r_{ch}}')$ such that $\mathcal{H}_{ch}(\mathrm{ek_{ch}}, m, \mathrm{r_{ch}}) = \mathcal{H}_{ch}(\mathrm{ek_{ch}}, m', \mathrm{r_{ch}}')$ is negligible in polynomial time.

TRAPDOOR COLLISION. Provides that there is an efficient algorithm $\mathcal{H}_{ch}^{-1}$ which finds two pairs $(m, \mathrm{r_{ch}}) \neq (m', \mathrm{r_{ch}}')$ for which $\mathcal{H}_{ch}(\mathrm{ek_{ch}}, m, \mathrm{r_{ch}}) = \mathcal{H}_{ch}(\mathrm{ek_{ch}}, m', \mathrm{r_{ch}}')$, using the trapdoor key $\mathrm{tk_{ch}}$.

UNIFORMITY. For any message $m \in \mathcal{M}_{ch}$, and any $\mathrm{r_{ch}}$ uniformly distributed in $\mathcal{R}_{ch}$, the hash value $\mathrm{c_{ch}}$ is uniformly distributed in $\mathcal{Y}_{ch}$. Therefore the probability of an adversary of distinguishing between the hash value of m and m', both in $\mathcal{M}_{ch}$ is negligible in polynomial time.

2.1 A simulatable NIZK proof using chameleon hashes

Although examples of simulatable NIZKPK proofs are given by the authors in [20], here we provide a formal description of the algorithms that will be used in further sections, in order to prove their properties and those of the scheme where they are used.

In a Σ-protocol, in order to prove that a statement x belongs to $\mathcal{L}_{\mathcal{R}}$, a prover P and a verifier V engage in an interactive protocol where first, P sends a commitment message a to V; then V replies with a random challenge e; finally, P sends an answer z to V. Interactive zero-knowledge protocols such as Σ proofs can be turned into non-interactive using the Fiat-Shamir [16] transformation, where a hash function is used to compute the random challenge e.

The transformation into a (trapdoor) simulatable NIZKPK works by substituting the challenge e with the result of a chameleon hash: P chooses a random value $\mathrm{r_{ch}}$ and evaluates the chameleon hash function $\mathcal{H}_{ch}$ on the message $m = H(x, a)$ using the randomness $\mathrm{r_{ch}}$, where H is a regular collision-resistant hash function. The challenge of the Σ-protocol is then defined as $e = \mathcal{H}_{ch}(H(x, a); \mathrm{r_{ch}})$. In addition, P also sends the randomness $\mathrm{r_{ch}}$ which he used in the computation of the chameleon hash.

This non-interactive protocol allows to simulate valid proofs by means of the trapdoor key of the chameleon hash scheme: indeed, given a trapdoor $\mathrm{tk_{ch}}$ for the chameleon hash, the simulator can compute the triplet (a^*, e^*, z^*) as the simulator of the Σ-protocol would do. Then, by using the trapdoor of the chameleon hash, the simulator will be able to find a random value $\mathrm{r_{ch}}^*$ such that $e^* = \mathcal{H}_{ch}(H(x^*, a^*); \mathrm{r_{ch}}^*)$. The uniformity property of the chameleon hash scheme guarantees that simulated proofs have the same distribution than honest proofs.

Concretely, the trapdoor-simulatable NIZKPK scheme to be used in our protocol uses a Σ-protocol, a chameleon hash scheme ($\mathsf{Gen_{ch}}$, $\mathcal{H}_{ch}$, $\mathcal{H}_{ch}^{-1}$) and two hash functions $H_1 : \{0,1\}^* \to \mathcal{M}_{ch}$ and $H_2 : \{0,1\}^* \to \mathcal{CH}$ (the challenge space). Then, the NIZK proof is given by the following algorithms:

- GenCRS: on input a security parameter, it runs $\mathsf{Gen_{ch}}$ and outputs $\mathsf{crs} = \mathrm{ek_{ch}}$ and $\mathsf{tk} = \mathrm{tk_{ch}}$.
- $\mathsf{NIZKProve}$: on input the common reference string crs, a statement x and a witness w, it follows the next steps:
  1. Run the first phase of the prover P of the Σ-protocol, which outputs a commitment a.
  2. Sample a random $\mathrm{r_{ch}} \in \mathcal{R}_{ch}$ and compute $e = H_2(\mathcal{H}_{ch}(H_1(x, a), \mathrm{r_{ch}}))$.
  3. Run the second phase of the prover P of the Σ-protocol, obtaining an answer z.
  4. Define the proof $\pi = (a, e, \mathrm{r_{ch}}, z)$.
- $\mathsf{NIZKVerify}$: on input a proof π and a statement x, return 1 if $e = H_2(\mathcal{H}_{ch}(H_1(x, a), \mathrm{r_{ch}})$ and the verification checks of the Σ-protocol pass on (a, e, z), 0 otherwise.
- $\mathsf{NIZKSimulate}$: on input a statement x and a trapdoor tk, the simulator runs the following steps:
  1. Run the simulator $\mathcal{S}$ of the Σ-protocol to obtain a triplet (a^*, e^*, z^*).
  2. Use the trapdoor $\mathrm{tk_{ch}}$ to obtain a value $\mathrm{r_{ch}}^*$ s.t. $e^* = H_2(\mathcal{H}_{ch}(H_1(x, a^*), \mathrm{r_{ch}}^*))$
  3. Output a simulated proof $\pi^* = (a^*, e^*, \mathrm{r_{ch}}^*, z^*)$

A NIZKPK satisfies the properties of completeness, knowledge soundness and zero-knowledge [13], [28].

3 Protocol syntax

In this section we define a syntax for the proposed voting protocol. We use as a basis the syntax defined in [31] and [11] for analyzing the properties of the Helios voting protocol [2], and add an auditing phase for the cast as intended verification functionality.

The following are the participants of the voting protocol: the *Election Authorities* configure the election and tally the votes to produce the election result; the *Registrars* registers the voters and provide them with information for participating in the election; *Voters* participate in the election by providing their choices; the *Voting Device* generates and casts a vote given the voting options selected by the voter; an *Audit Device* is used by the voter to verify cryptographic evidences; the *Bulletin Board Manager* receives and publishes the votes cast by the voters in the bulletin board BB; finally the *Auditors* are responsible of verifying the integrity of the procedures run in the counting phase.

Consider that the list of voting options $V = \{v_1, \ldots, v_n\}$ in the election is defined in advance. The counting function $\rho : (V \cup \{\perp\})^* \to R$, where $\perp$ denotes an invalid vote and R is the set of results, is a multiset function which provides the set of cleartext votes cast by the voters in a random order [5].

The voting protocol uses an encryption scheme with algorithms ($\mathsf{Gen_e}$, Enc, Dec, $\mathsf{EncVerify}$), a signature scheme ($\mathsf{Gen_s}$, Sign, $\mathsf{SignVerify}$) and a mix-net with algorithms Mix and $\mathsf{MixVerify}$. It additionally uses a trapdoor-simulatable NIZKPK scheme denoted by the algorithms (GenCRS, $\mathsf{NIZKProve}$, $\mathsf{NIZKVerify}$, $\mathsf{NIZKSimulate}$).

- $\mathsf{Setup}(1^\lambda)$ chooses p and q for the ElGamal encryption scheme and runs $\mathsf{Gen_e}$. Then it sets the election public key $pk = (pk_\mathsf{e}, \mathbb{G})$ and the election private key $sk = (sk_\mathsf{e}, pk_\mathsf{e})$. Finally it generates the empty list of credentials ID.
- $\mathsf{Register}(1^\lambda, \mathtt{id})$ takes the public parameters defined by pk, runs GenCRS from the NIZKPK scheme and $\mathsf{Gen_s}$ from the signature scheme, and sets $pk_\mathtt{id} = (\mathsf{crs}, pk_\mathsf{s})$ and $sk_\mathtt{id} = (\mathsf{tk}, sk_\mathsf{s})$.
- $\mathsf{CreateVote}(v_i, pk_\mathtt{id})$ runs Enc from the encryption scheme with inputs pk and v_i and obtains the ciphertext c_s. Then it parses c_s as (c_1, c_2, h, z) and $pk_\mathtt{id}$ as $(\mathsf{crs}, pk_\mathsf{s})$, and runs $\mathsf{NIZKProve}$ from the NIZKPK scheme, using as input crs, the statement $(c_1, c_2/v_i)$ and the witness r, where r is the random element in $\mathbb{Z}_q$ used during encryption. The result is set to be σ, while the ballot b takes the value of c_s.
- $\mathsf{AuditVote}(v_i, b, \sigma, pk_\mathtt{id})$ parses b as (c_1, c_2, h, z) and $pk_\mathtt{id}$ as $(\mathsf{crs}, pk_\mathsf{s})$, then runs $\mathsf{NIZKVerify}$ from the NIZKPK scheme with inputs the proof σ, the common reference string crs and the statement $(c_1, c_2/v_i)$. It outputs the result of the proof verification.
- $\mathsf{CastVote}(b, sk_\mathtt{id}, \mathtt{id})$ runs Sign with inputs the voter's signing private key sk_s and the ballot b to be signed together with the voter identity $\mathtt{id}$. The output is the authenticated ballot $b_a = (\mathtt{id}, b, \psi)$.
- $\mathsf{FakeProof}(b, sk_\mathtt{id}, pk_\mathtt{id}, v_j)$ parses b as (c_1, c_2, h, z), $pk_\mathtt{id}$ as $(\mathsf{crs}, pk_\mathsf{s})$ and $sk_\mathtt{id}$ as $(\mathsf{tk}, sk_\mathsf{s})$. Then it runs $\mathsf{NIZKSimulate}$ from the NIZKPK scheme for the statement $(c_1, c_2/v_j)$. Then the simulated encryption proof data σ' is the simulated proof π^*.
- $\mathsf{ProcessBallot}(\mathsf{BB}, b_a)$ parses b_a as $(\mathtt{id}, b, \psi)$ and b as (c_1, c_2, h, z). Then it proceeds to perform some validations: It checks that there is not

already an entry in the bulletin board for the same id and that this id is present in the list ID, or with the same ciphertext (c_1, c_2). It also runs EncVerify to verify the proof (h, z) and the voter's signature running SignVerify (for which it picks the corresponding public key $pk_{\texttt{id}}$ from the list ID). If any of these validations fail, the process stops and outputs 0. Otherwise it outputs 1.

- VerifyVote(BB, b, id) checks that there is an entry in the bulletin board for the identity id. In the affirmative case, it parses the authenticated ballot b'_a as $(\texttt{id}, b', \psi')$ and checks that all the fields in b' are equal to all the fields in b.
- Tally(BB, sk) runs ProcessBallot over the individual entries of the ballot box. For those who passed the verifications, it parses each one as $(\texttt{id}, (c_1, c_2, h, z), \psi)$, it takes the pairs (c_1, c_2) and runs $\mathsf{Mix}((c_1^1, c_2^1), \ldots, (c_1^n, c_2^n))$, which denotes a verifiable mixnet such as [3] or [32]. Then the ciphertexts are decrypted running the Dec algorithm and a proof of correct decryption is produced. The outputs are the list of decrypted votes r and the proofs of correct mixing and decryption, Π.
- VerifyTally(BB, r, Π) in the first place performs the same validations than the ProcessBallot algorithm over the ballots in BB: for each one, it checks that there is only one entry in the ballot box per id and per (c_1, c_2). In case it founds any coincidence, it halts and outputs $\perp$. Otherwise, it continues with the validations and discards all the ballot box entries for which EncVerify or SignVerify output 0. Finally it verifies the proofs Π of correct mixing and decryption, using the ciphertexts of the entries which have passed the validation and the result r.

The voting protocol algorithms are organised in the following phases:

Configuration phase: in this phase, the election authorities set up the public parameters of the election such as the list of voting options $\{v_i\} \in V$ and the result function ρ. They also run the Setup algorithm and publish the resulting election public key pk and the empty credential list ID in the bulletin board. The private key sk is kept in secret by the electoral authorities.

Registration phase: in this phase the registrars register the voters to vote in the election. For each voter with identity id, the registrars run Register and update the credential list ID in the bulletin board with the pair $(\texttt{id}, pk_{\texttt{id}})$. The key pair $(pk_{\texttt{id}}, sk_{\texttt{id}})$ is provided to the voter.

Voting phase: in this phase the voter chooses a voting option $v_i \in V$ and interacts in the following way with the voting device, in order to cast a vote:

1. The voter provides her identity $\mathtt{id}$ and the voting option v_i to the voting device, which gathers the corresponding public key $pk_{\mathtt{id}}$ from the bulletin board and runs the CreateVote algorithm. The outputs b and σ are provided to the voter.
2. The voter uses an audit device to run AuditVote using b and σ provided by the voting device. The voter may enter $pk_{\mathtt{id}}$ herself, or her identity $\mathtt{id}$ so that the audit device picks the corresponding public key $pk_{\mathtt{id}}$ from the list ID. A positive result means that b is encrypting the voter's selection v_i and the voter can continue the process. Otherwise, the voter is instructed to abort the process and choose another voting device to cast her vote, since the one she is using is corrupted and did not encrypt what the she selected.
3. As a sign of approval of the generated ballot, the voter provides her private key $sk_{\mathtt{id}}$ to the voting device, which proceeds to run CastVote. The resulting authenticated ballot is sent to the bulletin board manager.
4. Then the voting device runs FakeProof using a voting option $v_j \in V$ as input (supposedly the one requested by the coercer/vote buyer, otherwise it may be a random value from the set V), and provides the simulated encryption data σ' to the voter.

The bulletin board manager runs the ProcessBallot algorithm. If the result is positive, the authenticated ballot b_a is posted in the bulletin board. Otherwise, the bulletin board is left unchanged, and the voter receives a negative response. From that point, the voter can run VerifyVote to check that her vote has been posted in the bulletin board.

From this point, the voter can provide the ballot b and the simulated encryption data σ' to a coercer, who might want to check that a ballot for the requested voting option v_j is present in the bulletin board by running the AuditVote and VerifyVote algorithms.

Counting phase: in this phase, the election authorities provide the election private key sk and run the Tally algorithm on the contents of the bulletin board. The obtained result r and the proof Π are posted in the bulletin board. The auditors then run the VerifyTally algorithm. In case the verification is satisfactory, the election result is considered to be correct. Otherwise, an investigation is opened in order to detect any manipulation that could lead to a corrupted result.

3.1 Trust model

Security definitions and analysis results are provided in the Annex, while complete demonstrations are included in the full version [19]. However, here we informally introduce the trust assumptions we make on the scheme regarding privacy and integrity:

We assume that voters follow the protocol in the correct way. We also assume the voter to follow the audit processes indicated and complain in case of any irregularity.

In order to simplify the analysis, we consider that the election authorities, and the registrars as well, behave properly in the sense that they generate correct and valid key pairs, and that they do not divulge the secret keys to unintended recipients. Multiparty computation techniques such as [25] or [14] can be used in order to distributely generate secrets among a set of trustees, ensuring their privacy and a correct generation in case a subset of them is honest.

From the point of view of privacy, the voting device is trusted not to leak the randomness used for the encryption of the voter's choices. While this assumption may seem too strong, it is in fact needed in any voting scheme where the voting options are encrypted at the voting device (no pre-encrypted ballots are used) and the vote is not cast in an anonymous way. However, for the point of view of integrity, we consider that a malicious voting device may ignore the selections made by the voter and put another content in the ballot to be cast.

As in similar schemes such as Helios [2] or Wombat [1], the audit device is trusted both from the point of view of privacy (it is assumed not to divulge the voter's selections) and from the point of view of integrity (it is assumed to honestly transmit the result of the proofs verification to the voter).

The bulletin board manager is trusted to accept and post on the bulletin board all the correct votes. No assumptions are done in the topic of privacy. Finally, auditors are assumed to honestly transmit the result of their verification. However, we assume them to be curious and try to find out the content of voter's votes from the information they get.

4 Concrete instantiation

In this Section, we provide a concrete instantiation based on ElGamal over a finite field.

Encryption scheme The Signed ElGamal encryption scheme [30] is used in our instantiation of the protocol. It is a combination of the ElGamal encryption scheme [15] and a proof of knowledge of the encryption randomness, which is based in the Schnorr signature scheme [29] (sometimes we will refer to this proof of knowledge as the Schnorr proof). In our notation, (c_1, c_2) represent the single ElGamal ciphertext, while (h, z) represent the Schnorr signature.

According to the work in [6], this variant of ElGamal is NM-CPA secure.

Chameleon hash The following instantiation of a chameleon hash $(\mathsf{Gen}_{\mathsf{ch}}, \mathcal{H}_{ch}, \mathcal{H}_{ch}^{-1})$ based on the discrete logarithm problem [22] is used: $\mathsf{Gen}_{\mathsf{ch}}$ receives a group $\mathbb{G}$ of prime order q of elements in $\mathbb{Z}_p^*$ with generator g. An element x is sampled uniformly from $\mathbb{Z}_q$ and $h = g^x$ is computed. Then, the evaluation key $\mathrm{ek}_{\mathrm{ch}}$ is defined as $\mathrm{ek}_{\mathrm{ch}} = (\mathbb{G}, g, h)$ and the trapdoor key $\mathrm{tk}_{\mathrm{ch}}$ is defined as $\mathrm{tk}_{\mathrm{ch}} = (\mathrm{ek}_{\mathrm{ch}}, x)$. The message space and the randomness space are $\mathbb{Z}_q$ and the hash space is $\mathbb{G}$. The algorithm $\mathcal{H}_{ch}$ is defined for $(m, \mathrm{r}_{\mathrm{ch}}) \in \mathbb{Z}_q \times \mathbb{Z}_q$ to output $\mathrm{c}_{\mathrm{ch}} = g^m \cdot h^{\mathrm{r}_{\mathrm{ch}}}$. Finally, $\mathcal{H}_{ch}^{-1}(m, \mathrm{r}_{\mathrm{ch}}, m')$ outputs ${\mathrm{r}_{\mathrm{ch}}}' = (m - m') \cdot x^{-1} + \mathrm{r}_{\mathrm{ch}}$.

***Σ*-proof** We use a simulatable NIZKPK based on a Σ-proof which proves that a specific plaintext corresponds to a given ciphertext. The Σ-proof computed over an ElGamal ciphertext of the form $(c_1, c_2) = (g^r, pk_{\mathsf{e}}^r \cdot m)$ is as follows:

1. Prover computes $(a_1, a_2) = (g^s, pk_{\mathsf{e}}^s)$, where s is a random element $\in \mathbb{Z}_q$, and provides them to the verifier.
2. Verifier provides a challenge e.
3. Prover provides to the verifier $z = s + re$. The verifier checks that $g^z = a_1 \cdot c_1^e$ and that $h^z = a_2 \cdot (c_2/m)^e$.

This Σ-proof can be simulated in the following way: the simulator samples a random $z^* \in \mathbb{G}$, a random $e^* \in \mathbb{Z}_q$ and computes $a_1^* = g^{z^*} \cdot c_1^{-e^*}$ and $a_2^* = h^{z^*} \cdot (c_2/m)^{-e^*}$. The resulting (a^*, e^*, z^*) values have the same distribution than the original ones.

Simulatable NIZKPK The algorithms of the NIZKPK scheme are then defined by using the discrete log-based chameleon hash scheme and the Σ-proof defined above, as well as two hash functions H_1, H_2 mapping inputs to $\mathbb{Z}_q$, as follows:

- GenCRS runs $\mathsf{Gen}_{\mathsf{ch}}$ and outputs $\mathsf{crs} = (\mathbb{G}, g, h)$ and $\mathsf{tk} = (\mathsf{crs}, x)$;

- $\mathsf{NIZKProve}$ receives crs, the statement $x = (c_1, c_2/m)$ and the witness r, and computes: (1) the commitment $(a_1, a_2) = (g^s, pk_{\mathsf{e}}^s)$, (2) the non-interactive challenge $e = H_2(g^{(H_1(x,a))} \cdot h^{\mathrm{r_{ch}}})$, where $\mathrm{r_{ch}}$ is picked at random from $\mathbb{Z}_q$, (3) the answer $z = s + re$, and (4) provides the proof $\pi = (a, e, \mathrm{r_{ch}}, z)$;
- $\mathsf{NIZKVerify}$ checks that $g^z = a_1 \cdot c_1^e$, $h^z = a_2 \cdot (c_2/m)^e$, and that $e = H_2(g^{(H_1(x,a))} \cdot h^{\mathrm{r_{ch}}})$;
- $\mathsf{NIZKSimulate}$ receives as input a statement $x^* = (c_1, c_2/m^*)$ and the trapdoor tk, and does the following: takes at random $z^* \in \mathbb{G}$ and random pair $(\alpha, \beta) \in \mathbb{Z}_q$, and sets $e^* = H_2(g^\alpha \cdot h^\beta)$. Then it computes $a_1^* = g^{z^*} \cdot c_1^{-e^*}$ and $a_2^* = h^{z^*} \cdot (c_2/m^*)^{-e^*}$, and finally it obtains $\mathrm{r_{ch}}^* = (\alpha - H_1(x^*, a^*)) \cdot x^{-1} + \beta$. The simulated proof is then $\pi^* = (a^*, e^*, \mathrm{r_{ch}}^*, z^*)$.

The full version of this paper [19] provides a proof that the described simulatable NIZK proof fulfills the properties of completeness, knowledge soundness and zero-knowledge of NIZKPKs.

Additionally, we use the *RSA Full Domain Hash (RSA-FDH)* [4] algorithm for the signature scheme $(\mathsf{Gen_s}, \mathsf{Sign}, \mathsf{SignVerify})$, and a proof of correct decryption based on the Chaum-Pedersen protocol [9], as described in [12].

4.1 Performance

This instantiation is simple and efficient. For a *k-out-of-n* voting scheme, where k options can be encrypted into one ElGamal ciphertext, the encryption of the voter selections using the Signed ElGamal encryption scheme requires 3 exponentiations. The computation of the NIZKPK requires 6 additional exponentiations (2 of them for the computation of the chameleon hash), and 6 more for verification. Each proof simulation costs 6 exponentiations.

An important detail is that, for efficiency purposes, the prime group and the generator of such group used in all these primitives must be the same.

5 Voting Experience

It is important to recall the criticity of the voter's trapdoor key. A voter who has not access to it will not be able to simulate a proof. Thus, the cast-as-intended verification mechanism will no longer protect the privacy of the voter. On the other hand, the voter device has to learn the trapdoor key only after it has already generated a honest proof for the voter. Otherwise, the device could simulate a proof the voter expects to be honest, and the scheme would no longer be cast-as-intended verifiable.

In order to present an easy and intuitive voting process for the voter, we have related the private information she uses to authenticate her vote (for example, her private signing key) with the trapdoor key which is used to generate false proofs. We think that it is meaningful that the voter provides both secrets at the same time, as a confirmation that she agrees to cast that vote (which she is expected to do only after verifying the honest proof). Before the voter provides these secrets, the voting device can neither cast a valid vote, nor cheat the voter by generating a fake proof.

Therefore, at the voter registration stage each voter may be issued both key pairs (the signing key pair to authenticate their vote, and the evaluation/trapdoor key pair for the NIZKPK scheme), where the private keys are password-protected. Later on, during the voting stage, the voter's selections are encrypted by the voting device, and the resulting ciphertext and the proof of content are shown to the voter, who then can use an audit device to check that the ciphertext contents match her selections (for example, her smartphone). After a positive audit, the voter enters her password into the voting device, which recovers both private keys, using the private signing key part to digitally sign the vote to be cast, and the NIZKPK trapdoor key part to generate one or several fake proofs for alternative voting options which may be defined by the voter. The fake proofs have to be presented in the same way than the honest one, so that they cannot be distinguished by a potential coercer. Finally, the vote is sent to the bulletin board manager which publishes it the bulletin board, so that the voter can check that her audited vote has been correctly received.

6 Protocol extension for multiple voting

The possibility of multiple voting may be interesting in case something goes wrong or for anti-coercion measures. However, it has to be taken into account that in this case the voting device learns the trapdoor key after the first vote, and could cheat the voter in further ballot generations.

An approach for allowing multiple voting consists on delegating the generation of simulated proofs to the bulletin board manager, who keeps the voters' trapdoor keys and provides simulated proofs to the voting devices *only* when receiving confirmed ballots (which means that voters already verified their contents and agreed with them). Although this does not endanger the voter's privacy, a collusion of the bulletin board manager and the voting device may defeat the property of cast-as-intended verifiability by simulating proofs in advance or refuse to generate them. A distributed setting, where multiple bulletin board managers hold shares of the voters' trapdoor keys generated with a threshold scheme can be used to enforce this property, even if a subset of the bulletin board managers are malicious.

References

1. Wombat voting system. Available at http://www.wombat-voting.com/ (2015)
2. Adida, B.: Helios: Web-based open-audit voting. In: van Oorschot, P.C. (ed.) USENIX Security Symposium. pp. 335–348. USENIX Association (2008)
3. Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Advances in Cryptology–EUROCRYPT 2012, pp. 263–280. Springer (2012)
4. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: CCS'93, Proceedings. pp. 62–73. NY, USA (1993)
5. Bernhard, D., Cortier, V., Galindo, D., Pereira, O., Warinschi, B.: A comprehensive analysis of game-based ballot privacy definitions. IACR Cryptology ePrint Archive 2015, 255 (2015)
6. Bernhard, D., Pereira, O., Warinschi, B.: On necessary and sufficient conditions for private ballot submission. IACR Cryptology ePrint Archive 2012, 236 (2012)
7. Brassard, G., Chaum, D., Crépeau, C.: Minimum disclosure proofs of knowledge. J. Comput. Syst. Sci. 37(2), 156–189 (Oct 1988)
8. Chaum, D.: Surevote: technical report (2001), http://www.iavoss.org/mirror/wote01/pdfs/surevote.pdf
9. Chaum, D., Pedersen, T.P.: Wallet databases with observers. In: Advances in Cryptology - CRYPTO '92, Proceedings. LNCS, vol. 740, pp. 89–105. Springer (1992)
10. Chen, X., Wu, Q., Zhang, F., Tian, H., Wei, B., Lee, B., Lee, H., Kim, K.: New receipt-free voting scheme using double-trapdoor commitment. Information Sciences 181(8), 1493–1502 (2011)
11. Cortier, V., Galindo, D., Glondu, S., Izabachène, M.: Election verifiability for helios under weaker trust assumptions. In: Computer Security - ESORICS 2014, Proceedings Part II. LNCS, vol. 8713, pp. 327–344. Springer (2014)
12. Cramer, R., Gennaro, R., Schoenmakers, B.: A secure and optimally efficient multi-authority election scheme. In: EUROCRYPT. LNCS, vol. 1233, pp. 103–118. Springer (1997)
13. Damgård, I.: Commitment schemes and zero-knowledge protocols. In: Lectures on Data Security, LNCS, vol. 1561, pp. 63–86. Springer (1999)
14. Damgård, I., Mikkelsen, G.L.: Efficient, robust and constant-round distributed RSA key generation. In: TCC 2010, Proceedings. LNCS, vol. 5978, pp. 183–200. Springer (2010)
15. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: CRYPTO '84, Proceedings. LNCS, vol. 196, pp. 10–18. Springer (1984)

16. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: CRYPTO '86, Proceedings. LNCS, vol. 263, pp. 186–194. Springer (1986)
17. Gharadaghy, R., Volkamer, M.: Verifiability in electronic voting - explanations for non security experts. In: EVOTE 2010, Proceedings. LNI, vol. 167, pp. 151–162. GI (2010)
18. Gjøsteen, K.: Analysis of an internet voting protocol. IACR Cryptology ePrint Archive 2010, 380 (2010)
19. Guasch, S., Morillo, P.: How to challenge *and* cast your e-vote. IACR Cryptology ePrint Archive, to be published (2016)
20. Jakobsson, M., Sako, K., Impagliazzo, R.: Designated verifier proofs and their applications. In: EUROCRYPT'96, Proceedings. pp. 143–154 (1996)
21. Juels, A., Catalano, D., Jakobsson, M.: Coercion-resistant electronic elections. In: WPES'05, Proceedings. pp. 61–70. ACM (2005)
22. Krawczyk, H., Rabin, T.: Chameleon hashing and signatures. IACR Cryptology ePrint Archive 1998, 010 (1998)
23. Moran, T., Naor, M.: Receipt-free universally-verifiable voting with everlasting privacy. In: CRYPTO 2006, Proceedings. LNCS, vol. 4117, pp. 373–392. Springer (2006)
24. Neff, C.A.: Practical high certainty intent verification for encrypted votes (2004)
25. Pedersen, T.: A threshold cryptosystem without a trusted party. In: Advances in Cryptology EUROCRYPT 91, LNCS, vol. 547, pp. 522–526. Springer (1991)
26. Ryan, P.Y.A., Teague, V.: Pretty good democracy. In: Security Protocols XVII, 17th International Workshop, Cambridge, UK, 2009. Revised Selected Papers. LNCS, vol. 7028, pp. 111–130. Springer (2009)
27. Sako, K., Kilian, J.: Receipt-free mix-type voting scheme - a practical solution to the implementation of a voting booth. In: EUROCRYPT '95, Proceedings. LNCS, vol. 921, pp. 393–403. Springer (1995)
28. Santis, A.D., Persiano, G.: Zero-knowledge proofs of knowledge without interaction (extended abstract). In: FOCS. pp. 427–436. IEEE Computer Society (1992)
29. Schnorr, C.: Efficient identification and signatures for smart cards. In: CRYPTO '89, Proceedings. LNCS, vol. 435, pp. 239–252. Springer (1989)
30. Schnorr, C.P., Jakobsson, M.: Security of signed elgamal encryption. In: ASIACRYPT 2000, Proceedings. LNCS, vol. 1976, pp. 73–89. Springer (2000)
31. Smyth, B., Frink, S., Clarkson, M.R.: Computational election verifiability: Definitions and an analysis of helios and JCJ. IACR Cryptology ePrint Archive 2015, 233 (2015)
32. Wikström, D.: A commitment-consistent proof of a shuffle. IACR Cryptology ePrint Archive 2011, 168 (2011)

A Security Definitions and Analysis results

A.1 Definitions

In this section we define the notions of ballot privacy, and cast as intended verifiability for an electronic voting scheme such as that described in Section 3. We take as basis the definitions from [5] and then adapt them to the particularities of our scheme. Other defintions such as strong consistency or strong correctness are available in the full version [19].

Ballot privacy It is defined by means of an experiment where an adversary is presented with two experiments and has to be able to distinguish between them. In each experiment the adversary has indirect access to a ballot box which receives the ballots created by honest voters, as well as ballots cast by the adversary itself on behalf of corrupted voters. In the case of honest voters, we let the adversary choose two possible votes which they will use to create their ballots. Which vote is used to cast a voter's ballot that goes to a specific ballot box depends on the experiment that is taking place.

At the end of the experiment, the adversary is presented with the result of tallying the ballot box, which is the same regardless of the experiment. As noted in [5], revealing the true tally in each experiment would easily allow the adversary to distinguish between both ballot boxes. Additionally, for the votes cast by honest voters, we provide the resulting encryption proof data to the adversary in order to model a coercer which uses it to learn something about the vote. We will use a simulation functionality to generate fake proofs when required.

$\mathsf{Exp}^{\mathsf{priv},\beta}_{\mathcal{A},V}$:

1. **Setup phase:** The challenger $\mathcal{C}$ sets up two empty bulletin boards BB_0 and BB_1 and runs the $\mathsf{Setup}(1^\lambda)$ algorithm to obtain the election key pair (pk, sk) and the empty list of credentials ID. $\mathcal{A}$ is given access to BB_0 when $\beta = 0$ and to BB_1 when $\beta = 1$.
2. **Registration phase:** The adversary may make the following query:
   – $\mathcal{O}$**Register**($\mathtt{id}$): $\mathcal{A}$ provides an identity $\mathtt{id} \notin \mathsf{ID}$. $\mathcal{C}$ runs $\mathsf{Register}(1^\lambda,$ $\mathtt{id})$, provides the voter key pair $(pk_{\mathtt{id}}, sk_{\mathtt{id}})$ to $\mathcal{A}$, and adds $(\mathtt{id}, pk_{\mathtt{id}})$ to ID.
3. **Voting phase:** The adversary may make the following types of queries:
   – $\mathcal{O}$**VoteLR**($\mathtt{id}, v_0, v_1$): this query models the votes cast by honest voters. $\mathcal{A}$ provides an identity $\mathtt{id} \in \mathsf{ID}$ and two possible votes v_0, $v_1 \in V$. The challenger $\mathcal{C}$ does the following:
     - It picks the corresponding $pk_{\mathtt{id}}$ from ID and executes $\mathsf{CreateVote}$ $(v_0, pk_{\mathtt{id}})$ and $\mathsf{CreateVote}(v_1, pk_{\mathtt{id}})$ which produce the ballots b^0 and b^1 respectively and their encryption proof data σ_0 and σ_1.
     - Then it executes $\mathsf{CastVote}(b^0, sk_{\mathtt{id}}, \mathtt{id})$, $\mathsf{CastVote}(b^1, sk_{\mathtt{id}}, \mathtt{id})$ to obtain the authenticated ballots b^0_a and b^1_a, and $\mathsf{ProcessBallot}$ (BB_0, b^0_a) and $\mathsf{ProcessBallot}(BB_1, b^1_a)$. If both processes return 1, the ballot boxes BB_0 and BB_1 are updated with b^0_a and b^1_a respectively. Otherwise, $\mathcal{C}$ stops and returns $\perp$.

  - Finally, $\mathcal{C}$ executes $\mathsf{FakeProof}(b^{\beta}, sk_{\mathtt{id}}, pk_{\mathtt{id}}, v_{\overline{\beta}})$ and provides σ_{β} and the simulated encryption proof data σ^{*}_{β} to $\mathcal{A}$.
- $\mathcal{O}$**Cast**(b_a): this query models the votes cast by corrupted voters. $\mathcal{A}$ provides an authenticated ballot b_a, and then $\mathcal{C}$ executes $\mathsf{ProcessBallot}$ with b_a and each ballot box. If both algorithms return 1, both ballot boxes are updated with b_a. Otherwise, $\mathcal{C}$ halts and none of the ballot boxes are updated.

4. **Counting phase:** The challenger runs $\mathsf{Tally}(\mathsf{BB}_0, sk)$ and obtains the result r and the tally proof Π, which are provided to $\mathcal{A}$ in case $\beta = 0$. In case $\beta = 1$, $\mathcal{C}$ runs $\mathsf{SimProof}(\mathsf{BB}_1, r)$ to obtain Π^{*}, and provides (r, Π^{*}) to $\mathcal{A}$.
5. **Output:** The output of the experiment is the guess of the adversary for the bit β.

We say that a voting protocol as defined in Section 3 has ballot privacy if there exists an algorithm $\mathsf{SimProof}$ such that for any probabilistic polynomial time (p.p.t.) adversary $\mathcal{A}$, the following advantage is negligible in the security parameter λ:

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathcal{A}} = |\ \Pr[\mathsf{Exp}^{\mathsf{priv},0}_{\mathcal{A},V} = 1] - \Pr[\mathsf{Exp}^{\mathsf{priv},1}_{\mathcal{A},V} = 1]\ |$$

Cast-as-Intended verifiability A voting system is defined to be cast-as-intended verifiable if a corrupt voting device is unable to cast a vote on behalf of a voter, with a voting option different than the one chosen by the voter, without being detected. In our definition, we consider an adversary who posts ballots in the bulletin board on behalf of honest and corrupt voters. In case of honest voters, they follow the protocol and perform some validations before approving the ballot to be cast. Corrupt voters provide their approval without doing any prior verification.

$\mathsf{Exp}^{\mathsf{CaI}}_{\mathcal{A},V}$**:**

1. **Setup phase:** The challenger $\mathcal{C}$ runs the $\mathsf{Setup}(1^{\lambda})$ algorithm and provides the election key pair (pk, sk) to $\mathcal{A}$. Then it publishes the empty lists of voter credentials ID_h and ID_c such that $\mathsf{ID} = (\mathsf{ID}_h \cup \mathsf{ID}_c)$. Finally $\mathcal{A}$ is given read access to BB.
2. **Registration phase:** The adversary may make the following queries:
   - $\mathcal{O}$**RegisterHonest**$(\mathtt{id})$: $\mathcal{A}$ provides an identity $\mathtt{id} \notin \mathsf{ID}$. The challenger $\mathcal{C}$ runs $\mathsf{Register}(1^{\lambda}, \mathtt{id})$, and adds $(\mathtt{id}, pk_{\mathtt{id}})$ to ID_h.
   - $\mathcal{O}$**RegisterCorrupt**$(\mathtt{id})$: $\mathcal{A}$ provides an identity $\mathtt{id} \notin \mathsf{ID}$. The challenger $\mathcal{C}$ runs $\mathsf{Register}(1^{\lambda}, \mathtt{id})$, and adds $(\mathtt{id}, pk_{\mathtt{id}})$ to ID_c.

3. **Voting phase:** The adversary may make the following types of queries:
   – $\mathcal{O}$**VoteHonest**$(\texttt{id}, v_i, b, \sigma)$: this query models the votes cast by honest voters. $\mathcal{A}$ provides an identity $\texttt{id} \in \mathsf{ID}_h$, a ballot b, an encryption proof data σ and the voting option v_i. The challenger $\mathcal{C}$ runs $\mathsf{AuditVote}(v_i, b, \sigma, pk_{\texttt{id}})$, and only if the result is 1 it provides $sk_{\texttt{id}}$ to $\mathcal{A}$.
   – $\mathcal{O}$**VoteCorrupt**$(b, \texttt{id})$: this query models the votes cast by corrupted voters. $\mathcal{A}$ provides a ballot b and an identity $\texttt{id} \in \mathsf{ID}_c$. $\mathcal{C}$ answers with $sk_{\texttt{id}}$.
4. **Output:** The adversary submits an authenticated ballot $b'_a = (\texttt{id}', b', \psi')$. The output of the experiment is 1 if the following conditions hold:
   – $\texttt{id}' \in \mathsf{ID}_h$
   – $\mathsf{ProcessBallot}(\mathsf{BB}, b'_a) = 1$
   – $\mathsf{VerifyVote}(\mathsf{BB}, b', \texttt{id}') = 1$
   – $\mathsf{Extract}(b'_a; sk) \neq v'_i$, where v'_i is the voting option submitted by the adversary in the $\mathcal{O}$VoteHonest query.

We say that a voting protocol as defined in Section 3 has cast-as-intended verifiability if, given an $\mathsf{Extract}$ algorithm for which the protocol is consistent with respect to ρ, the following advantage is negligible in the security parameter λ for any probabilistic polynomial time (p.p.t.) adversary $\mathcal{A}$:

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{CaI}} = |\ \Pr[\mathsf{Exp}_{\mathcal{A},V}^{\mathsf{CaI},0} = 1]\ |$$

A.2 Security analysis results

In this section we provide the results of our security analysis, which is available in the full version of this paper [19].

Theorem 1. *Let* $(\mathsf{Gen_e}, \mathsf{Enc}, \mathsf{Dec})$ *be an NM-CPA secure encryption scheme and* $(\mathsf{GenCRS}, \mathsf{NIZKProve}, \mathsf{NIZKVerify}, \mathsf{NIZKSimulate})$ *a NIZKPK which provides zero-knowledge. Then the protocol presented in Section 3 satisfies the ballot privacy property.*

Theorem 2. *Let* $(\mathsf{Gen_e}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{EncVerify})$ *be a probabilistic encryption scheme,* $(\mathsf{GenCRS}, \mathsf{NIZKProve}, \mathsf{NIZKVerify}, \mathsf{NIZKSimulate})$ *a NIZKPK which is sound and* $(\mathsf{Gen_s}, \mathsf{Sign}, \mathsf{SignVerify})$ *an unforgeable signature scheme. Then the protocol presented in Section 3 satisfies the cast-as-intended verifiability property.*