Coercion Mitigation for Voting Systems with Trackers: A Selene Case Study

Kristian Gjøsteen[1], Thomas Haines[2], and Morten Rotvold Solberg[1(✉)]

[1] Norwegian University of Science and Technology, Trondheim, Norway
{kristian.gjosteen,mosolb}@ntnu.no
[2] Australian National University, Canberra, Australia
thomas.haines@anu.edu.au

Abstract. An interesting approach to achieving verifiability in voting systems is to make use of tracking numbers. This gives voters a simple way of verifying that their ballot was counted: they can simply look up their ballot/tracker pair on a public bulletin board. It is crucial to understand how trackers affect other security properties, in particular privacy. However, existing privacy definitions are not designed to accommodate tracker-based voting systems. Furthermore, the addition of trackers increases the threat of coercion. There does however exist techniques to mitigate the coercion threat. While the term *coercion mitigation* has been used in the literature when describing voting systems such as Selene, no formal definition of coercion mitigation seems to exist. In this paper we formally define what coercion mitigation means for tracker-based voting systems. We model Selene in our framework and we prove that Selene provides coercion mitigation, in addition to privacy and verifiability.

Keywords: E-voting · Coercion mitigation · Selene

1 Introduction

Electronic voting has seen widespread use over the past decades, ranging from smaller elections within clubs and associations, to large scale national elections as in Estonia. It is therefore necessary to understand the level of security that electronic voting systems provide. In this paper, we define precisely what verifiability, privacy and coercion mitigation means for voting systems using so-called *trackers*, and we prove that Selene provides these properties.

Verifiability is an interesting voting system property, allowing a voter to verify that their particular ballot was counted and that the election result correctly reflects the verified ballots. One example of a system with verifiability is Helios [2], which is used in the elections of the International Association for Cryptologic Research [1], among others. However, the Benaloh challenges used to achieve verifiability in Helios are hard to use for voters [26].

Schneier [33] proposed using human-readable *tracking numbers* for verifiability. Each voter gets a personal tracking number that is attached to their ballot.

M. Volkamer et al. (Eds.): E-Vote-ID 2023, LNCS 14230, pp. 69–86, 2023.
https://doi.org/10.1007/978-3-031-43756-4_5

At the end of the election, all ballots with attached trackers are made publicly available. A voter can now trivially verify that their ballot appears next to their tracking number, which gives us verifiability as long as the trackers are unique. Multiple voting systems making use of tracking numbers have been proposed and deployed. Two notable examples are sElect [27] and Selene [31]. Tracking numbers intuitively give the voters a simple way of verifying that their ballot was recorded and counted. However, other security properties must also be considered. In particular, it is necessary to have a good understanding of how the addition of tracking numbers affects the voters' *privacy*.

Verifiable voting may exacerbate threats such as *coercion*, in particular for remote electronic voting systems (e.g. internet voting) where a coercer might be present to "help" a coerced voter submit their ballot. *Coercion resistant* voting systems [9,25] have been developed. Coercion resistance typically involves voters re-voting when the coercer is not present, but this often complicates voting procedures or increases the cost of the tallying phase. Furthermore, re-voting might not always be possible and may even be prohibited by law.

Like verifiability in general, tracking numbers may make coercion simpler: if a coercer gets access to a voter's tracker, the coercer may also be able to verify that the desired ballot was cast. While tracking numbers complicate coercion resistance, it may be possible to *mitigate* the threat of coercion. For instance, if the voter only learns their tracking number after the result (ballots with trackers) has been published, as in Selene, they may lie to a coercer by observing a suitable ballot-tracker pair. *Coercion mitigation* is weaker than coercion resistance, but may be appropriate for low-stakes elections or where achieving stronger properties is considered to be impractical.

1.1 Related Work

Privacy. Bernhard *et al.* [6] analysed then-existing privacy definitions. They concluded that previous definitions were either too weak (there are real attacks not captured by the definitions), too strong (no voting system with any form of verifiability can be proven secure under the definition), or too narrow (the definitions do not capture a wide enough range of voting systems).

The main technical difficulty compared to standard cryptographic privacy notions is that the result of the election must be revealed to the adversary. Not only could the result reveal information about individual ballots, but it also prevents straight-forward cryptographic real-or-random definitions from working. Roughly speaking, there are two approaches to defining privacy for voting systems, based on the two different questions: "Does anything leak out of the casting and tallying prosesses?" *vs.* "Which voter cast this particular ballot?" The first question tends to lead to simulation-based security notions, while the second question can lead to more traditional left-or-right cryptographic definitions.

Bernhard *et al.* [6] proposed the BPRIV definition, where the adversary plays a game against a challenger and interacts with two worlds (real and fake). The adversary first specifies ballots to be cast separately for each world. In the real world, ballots are cast and then counted as usual. In the fake world, the specified

ballots are cast, but the ballots from the left world are counted and any tally proofs are simulated. The adversary then gets to see one of the worlds and must decide which world it sees. The idea is that for any secure system, the result in the fake world should be identical to what the result would have been in the real world, proving that – up to the actual result – the casting and tallying processes do not leak anything about the ballots cast, capturing privacy in this sense.

Bernhard *et al.* [6] proposed MiniVoting, an abstract scheme that models many voting systems (e.g. Helios), and proved that it satisfies the BPRIV definition. Cortier *et al.* [10] proved that Labelled-MiniVoting, an extension of MiniVoting, also satisfies BPRIV. Belenios [13] also satisfies BPRIV [11].

The original BPRIV definition does not attempt to model corruption in any part of the tally process. Cortier *et al.* [15] proposed mb-BPRIV which models adversarial control over which encrypted ballots should go through the tally process. Drăgan *et al.* [18] proposed the du-mb-BPRIV model which also covers systems where verification happens after tallying.

The other approach to privacy is a traditional left-or-right game, such as Benaloh [4], where the adversary interacts with the various honest components of a voting system (voters, their computers, shuffle and decryption servers, etc.), all simulated by an experiment. Privacy is captured by a left-or-right query, and the adversary must determine if the left or the right ballots were cast. The game becomes trivial if the left and the right ballots would give different tallies, so we require that the challenge queries taken together yield the same tally for left and right. In the simplest instantiation, the left and right ballots contain distinct permutations of the same ballots, so showing that they cannot be distinguished shows that the election processes do not leak who cast which ballots. Smyth [36] and Gjøsteen [20] provide examples of this definitional style. As far as we know, no definition in this style captures tracker-based voting systems.

The advantage of the traditional cryptographic left-or-right game relative to the BPRIV approach is that it is easier to model adversarial interactions with all parts of the protocol, including the different parts of the tally process, though authors before Gjøsteen [20] do not seem to do so. In principle, the BPRIV requirement that the tally process be simulatable is troublesome, since such simulators cannot exist in the plain model, which means that the definition itself technically exists in some unspecified idealised model (typically the random oracle model). In practice, this is not troublesome. Requiring balanced left and right ballots is troublesome for some systems with particular counting functions, but not if the system reveals plaintext ballots.

Verifiability. Verifiability intuitively captures the notion that if a collection of voters verify the election, the result must be consistent with their cast ballots. For voters that do not verify or whose verification failed, we make no guarantees.

Several definitions of verifiability have appeared in the literature, see e.g. [12] or [37] for an overview. Furthermore, the verifiability properties of Selene have been thoroughly analysed both from a technical point of view (e.g. [3,31]) and with respect to the user experience (e.g. [17,38]).

Coercion. Coercion resistance models a coercer that controls the voter for a period of time. We refer to Smyth [35] for an overview of definitions. A weaker notion is receipt-freeness, where the coercer does not control the voter, but asks for evidence that the voter cast the desired ballot. This was introduced by Benaloh and Tuinstra [5], while Chaidos *et al.* [7] gave a BPRIV-style security definition. Selene, as generally instantiated, is not receipt-free. *Coercion mitigation* is a different notion, where we assume that the coercer is not present during vote casting and is somehow not able to ask the voter to perform particular operations (such as revealing the randomness used to encrypt). This could allow the voter to fake information consistent with following the coercer's demands. While the term coercion mitigation has been used to describe the security properties provided by Selene (e.g. in [23,31,38]), there seems to be no formal definition of coercion mitigation in the literature.

Selene. Selene as a voting system has been studied previously, in particular with respect to privacy [18]. But a study of the complete protocol, including the tally phase, is missing. The coercion mitigation properties of Selene have also been extensively discussed [23,31], but have not received a cryptographic analysis.

1.2 Our Contribution

We define security for cryptographic voting systems with trackers, capturing privacy, verifiability and coercion mitigation. An experiment models the adversary's interaction with the honest players through various queries.

To break privacy, the adversary must decide who cast which ballot. Our definition is based on a similar definition by Gjøsteen [20, p. 492], adapted to properly accommodate voting systems using trackers. To break verifiability, the adversary must cause verifying voters to accept a result that is inconsistent with the ballots they have cast (similar to Cortier *et al.* [12]). To break coercion mitigation, the adversary is allowed to reveal the verification information of coerced voters and must decide if the coerced voter lied or not. Selene is vulnerable to collisions among such lies; e.g. multiple coerced voters claim the same ballot. We want to factor this attack out of the cryptographic analysis, so we require that the coercer organises the voting such that collisions do not happen. For schemes that are not vulnerable, we would remove the requirement.

Our definitions are easy to work with, which we demonstrate by presenting a complete model of Selene (expressed in our framework) and prove that Selene satisfies both privacy, verifiability and coercion mitigation. Selene has seen some use [32], so we believe these results are of independent interest.

We developed our definitions with Selene in mind, but they also accommodate other tracker based voting systems such as Hyperion [30] and (with some modifications to accomodate secret key material used in the shuffles) sElect [27]. Furthermore, our models also capture voting systems that do not use trackers.

2 Background

2.1 Notation

We denote tuples/lists in bold, e.g. $\mathbf{v} = (v_1, \ldots, v_n)$. If we have multiple tuples, we denote the jth tuple by $\mathbf{v}_j$ and the ith element of the jth tuple by $v_{j,i}$.

2.2 Cryptographic Building Blocks

We briefly introduce some cryptographic primitives we need for our work. Due to space constraints we omit much of the details.

To protect voters' privacy, ballots are usually encrypted. Selene makes use of the ElGamal public key encryption system [19], which is used to encrypt both ballots and trackers. Throughout this paper, we will denote an ElGamal ciphertext by $(x, w) := (g^r, m \cdot \mathsf{pk}^r)$, where g is the generator of the cyclic group $\mathbb{G}$ (of prime order q) we are working in, m is the encrypted message, $\mathsf{pk} = g^{\mathsf{sk}}$ is the public encryption key (with corresponding decryption key sk) and r is a random element in $\mathbb{Z}_q$ (the field of integers modulo q).

Cryptographic voting systems typically make use of zero-knowledge proofs to ensure that certain computations are performed correctly. We refer to [16] for general background on zero-knowledge proofs. In particular, we use *equality of discrete logarithm* proofs and correctness proofs for *shuffles* of encrypted ballots. The former ensures correctness of computations. The latter preserves privacy by breaking the link between voters and their ballots. It is necessary that the shuffles are *verifiable* to ensure that no ballots are tampered with in any way. We refer to [22] for an overview of verifiable shuffles. In Selene it is necessary to shuffle two lists of ciphertexts (ballots and trackers) in parallel. Possible protocols are given in [29] and we detail a convenient protocol in the full version [21].

Furthermore, in Selene, the election authorities make use of Pedersen-style commitments [28] to commit to tracking numbers.

3 Voting Systems with Trackers

We model a voting protocol as a simple protocol built on top of a cryptographic voting scheme in such a way that the protocol's security properties can be easily inferred from the cryptographic voting scheme's properties. This allows us to separate key management (who has which keys) and plumbing (who sends which message when to whom) from the cryptographic issues, which simplifies analysis.

Due to space limitations, we model a situation with honest setup and tracker generation, as well as a single party decrypting. The former would be handled using a bespoke, verifiable multi-party computation protocol (see [31] for a suitable protocol for Selene), while the latter is handled using distributed decryption.

3.1 The Syntax of Voting Systems with Trackers

A verifiable voting system $\mathcal{S}$ consists of the following algorithms (extending Gjøsteen [20]):

- Setup: takes as input a security parameter and returns a pair $(\mathsf{pk}, \mathsf{sk})$ of election public and secret keys.
- $\mathsf{UserKeyGen}$: takes as input an election public key pk and returns a pair $(\mathsf{vpk}, \mathsf{vsk})$ of voter public and secret keys.
- $\mathsf{TrackerGen}$: takes as input an election public key pk and a list $(\mathsf{vpk}_1, \ldots, \mathsf{vpk}_n)$ of voter public keys and returns a list $\mathbf{t}$ of trackers, a list $\mathbf{et}$ of ciphertexts, a list $\mathbf{ct}$ of commitments, a list $\mathbf{op}$ of openings and a permutation π on the set $\{1, \ldots, n\}$.
- $\mathsf{ExtractTracker}$: takes as input a voter secret key vsk, a tracker commitment ct and an opening op and returns a tracker t.
- $\mathsf{ClaimTracker}$: takes as input a voter secret key vsk, a tracker commitment ct and a tracker t and returns an opening op.
- Vote: takes as input an election public key pk and a ballot v and returns a ciphertext ev, a ballot proof Π^v and a receipt ρ.
- $\mathsf{Shuffle}$: takes as input a public key pk and a list $\mathbf{evt}$ of encrypted ballots and trackers, and returns a list $\mathbf{evt}'$ and a proof Π^s of correct shuffle.
- $\mathsf{DecryptResult}$: takes as input a secret key sk and a list $\mathbf{evt}$ of encrypted ballots and trackers and returns a result res and a result proof Π^r.
- $\mathsf{VoterVerify}$: takes as input a receipt ρ, a tracker t, a list $\mathbf{evt}$ of encrypted ballot/tracker pairs, a result res and a result proof Π^r and returns 0 or 1.
- $\mathsf{VerifyShuffle}$: takes as input a public key pk, two lists $\mathbf{evt}, \mathbf{evt}'$ of encrypted ballots and trackers and a shuffle proof Π^s and returns 0 or 1.
- $\mathsf{VerifyBallot}$: takes as input a public key pk, a ciphertext ev and a ballot proof Π^v and returns 0 or 1.
- $\mathsf{VerifyResult}$: takes as input a public key pk, a list $\mathbf{evt}$ of encrypted ballots and trackers, a result res and a result proof Π^r and returns 0 or 1.

We say that a verifiable, tracker-based voting system is (n_v, n_s)-*correct* if for any $(\mathsf{pk}, \mathsf{sk})$ output by Setup, any $(\mathsf{vpk}_1, \mathsf{vsk}_1), \ldots, (\mathsf{vpk}_{n_v}, \mathsf{vsk}_{n_v})$ output by $\mathsf{UserKeyGen}$, any lists $\mathbf{t}, \mathbf{et}, \mathbf{ct}, \mathbf{op}$ and permutations $\pi\colon \{1, \ldots, n_v\} \to \{1, \ldots, n_v\}$ output by $\mathsf{TrackerGen}(\mathsf{pk}, \mathsf{sk}, \mathsf{vpk}_1, \ldots, \mathsf{vpk}_{n_v})$, any ballots $v_1, \ldots, v_{n_v}$, any (ev_i, Π_i^v, ρ_i) output by $\mathsf{Vote}(\mathsf{pk}, v_i), i = 1, \ldots, n_v$, any sequence of n_s sequences of encrypted ballots and trackers $\mathbf{evt}_i$ and proofs Π_i^s output by $\mathsf{Shuffle}(\mathsf{pk}, \mathbf{evt}_{i-1})$, and any (res, Π^r) possibly output by $\mathsf{DecryptResult}(\mathsf{sk}, \mathbf{evt}_{n_s})$, the following hold:

- $\mathsf{DecryptResult}(\mathsf{sk}, \mathbf{evt}_{n_s})$ did not output $\perp$,
- $\mathsf{VoterVerify}(\rho_i, t_i, \mathbf{evt}_{n_s} \mathsf{res}, \Pi^r) = 1$ for all $i = 1, \ldots, n_v$,
- $\mathsf{VerifyShuffle}\left(\mathsf{pk}, \mathbf{evt}_{j-1}, \mathbf{evt}_j, \Pi_j^s\right) = 1$ for all $j = 1, \ldots, n_s$,
- $\mathsf{VerifyResult}\left(\mathsf{pk}, \mathbf{evt}_{n_s}, \mathsf{res}, \Pi^r\right) = 1$,
- $\mathsf{VerifyBallot}(\mathsf{pk}, ev_i, \Pi_i^v) = 1$ for all $i = 1, \ldots, n_v$, and

for any voter key pair $(\mathsf{vpk}, \mathsf{vsk})$, ct in $\mathbf{ct}$ and tracker t in $\mathbf{t}$, we have that

$$\mathsf{ExtractTracker}(\mathsf{vsk}, ct, \mathsf{ClaimTracker}(\mathsf{vsk}, ct, t)) = t.$$

We will describe later how Selene fits into our framework, but we note that this framework also captures voting systems that do not use trackers for verification. Such protocols are simply augmented with suitable dummy algorithms for TrackerGen, ExtractTracker and ClaimTracker.

3.2 Defining Security

We use a single experiment, found in Fig. 1, to define privacy, integrity and coercion mitigation. Verifiability is defined in terms of integrity. The experiment models the cryptographic actions of honest parties.

The test query is used to model integrity. The challenge query is used to define privacy. The coerce and coercion verification queries are used to model coercion, again modified by freshness. The coerce query specifies two voters (actually, two indices into the list of voter public keys) and two ballots. The first voter is the coerced voter. The first ballot is the coerced voter's intended ballot, while the second ballot is the coercer's desired ballot. The second voter casts the opposite ballot of the coerced voter. In the *coercion verification query*, the coerced voter either reveals an opening to their true tracker, or an opening to the tracker corresponding to the coercer's desired ballot, cast by the second voter, thereby ensuring that the coerced voter can lie about its opening without risking a collision (as discussed in Sect. 1.2). We note that this does not capture full coercion resistance, as that would require that the adversary is able to see exactly which ciphertext the coerced voter submitted (as, for example in [14]). In our definition, however, the adversary gets to see two ciphertexts, where one is submitted by the coerced voter, but he receives no information about which of the two ciphertexts the coerced voter actually submitted.

We make some restrictions on the order and number of queries (detailed in the caption of Fig. 1), but the experiment allows the adversary to make combinations of queries that do not correspond to any behaviour of the voting protocol. Partially, we do so because we can, but also in order to simplify definitions of certain cryptographic properties (such as uniqueness of results).

The adversary decides which ballots should be counted. We need to recognise when the adversary has organised counting such that it results in a trivial win. We say that a sequence $\mathbf{evt}$ of encrypted ballots and trackers is *valid* if

- L_s contains a sequence of tuples $(\mathbf{evt}_{j-1}, \mathbf{evt}_j, \Pi_j^s)_{j=1}^{n_s}$, not necessarily appearing in the same order in L_s, with $\mathbf{evt}_{n_s} = \mathbf{evt}$;
- L_v contains tuples

$$(i_1, j_1, v_{0,1}, v_{1,1}, ev_1, \Pi_1^v, \rho_1), \ldots, (i_{n_c}, j_{n_c}, v_{0,n_c}, v_{1,n_c}, ev_{n_c}, \Pi_{n_c}^v, \rho_{n_c})$$

  such that $\mathbf{evt}_0 = (ev_1, \ldots, ev_{n_c})$; and

The experiment proceeds as follows:

- Sample $b, b'' \xleftarrow{r} \{0,1\}$. Let L_r, L_v, L_s, L_d be empty lists.
  We denote by $(\mathsf{vpk}_i, \mathsf{vsk}_i)$ the ith entry in L_r.
- Compute $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}$ and send pk to the adversary.
- On a *register query*, compute $(\mathsf{vpk}, \mathsf{vsk}) \leftarrow \mathsf{UserKeyGen}(\mathsf{pk})$, append $(\mathsf{vpk}, \mathsf{vsk})$ to L_r and send vpk to the adversary.
- On a *chosen voter key query* vpk, append $(\mathsf{vpk}, \bot)$ to L_r.
- On a *tracker generation query*, compute $(\mathbf{t}, \mathbf{et}, \mathbf{ct}, \mathbf{op}, \pi) \leftarrow \mathsf{TrackerGen}(\mathsf{pk}, \mathsf{vpk}_1, \ldots, \mathsf{vpk}_{n_r})$ where $\mathsf{vpk}_1, \ldots, \mathsf{vpk}_{n_r}$ are the public keys from L_r, and send $(\mathbf{t}, \mathbf{et}, \mathbf{ct})$ to the adversary.
  We denote by t_i, et_i, ct_i and op_i the ith entries in the corresponding lists.
- On a *chosen ciphertext query* (i, ev, Π^v), if $\mathsf{VerifyBallot}(ev, \Pi^v) = 1$, append $(i, \bot, \bot, \bot, ev, \Pi^v, \bot)$ to L_v.
- On a *challenge query* (i, v_0, v_1), compute $(ev, \Pi^v, \rho) \leftarrow \mathsf{Vote}(\mathsf{pk}, v_b)$, append $(i, \bot, v_0, v_1, ev, \Pi^v, \rho)$ to L_v and send (ev, Π^v) to $\mathcal{A}$.
- On a *coerce query* (i, j, v_0, v_1), compute $(ev_i, \Pi_i^v, \rho_i) \leftarrow \mathsf{Vote}(\mathsf{pk}, v_b)$ and $(ev_j, \Pi_j^v, \rho_j) \leftarrow \mathsf{Vote}(\mathsf{pk}, v_{1-b})$, append $(i, j, v_0, v_1, ev_i, \Pi_i^v, \rho_i)$ and $(j, i, v_1, v_0, ev_j, \Pi_j^v, \rho_j)$ to L_v, and send (ev_i, Π_i^v) and (ev_j, Π_j^v) to $\mathcal{A}$.
- On a *shuffle query* $\mathbf{evt}$, compute $(\mathbf{evt}', \Pi^s) \leftarrow \mathsf{Shuffle}(\mathsf{pk}, \mathbf{evt})$, append $(\mathbf{evt}, \mathbf{evt}', \Pi^s)$ to L_s and send $(\mathbf{evt}', \Pi^s)$ to $\mathcal{A}$.
- On a *chosen shuffle query* $(\mathbf{evt}, \mathbf{evt}', \Pi^s)$, if $\mathsf{VerifyShuffle}(\mathbf{evt}, \mathbf{evt}', \Pi^s) = 1$, append the query to L_s.
- On a *result query* $\mathbf{evt}$, compute $(\mathsf{res}, \Pi^r) \leftarrow \mathsf{DecryptResult}(\mathsf{sk}, \mathbf{evt})$, send (res, Π^r) to $\mathcal{A}$ and append $(\mathbf{evt}, \mathsf{res}, \Pi^r)$ to L_d.
- On a *voter verification query* $(k, \mathbf{evt}, \mathsf{res}, \Pi^d)$ with $(i, \bot, v_0, v_1, ev, \Pi^v, \rho)$ being the kth entry in L_v, compute $t \leftarrow \mathsf{ExtractTracker}(\mathsf{vsk}_i, op_i, ct_i)$ and $d \leftarrow \mathsf{VoterVerify}(\rho_i, t, \mathbf{evt}, \mathsf{res}, \Pi^r)$ and send d to $\mathcal{A}$.
- On a *coercion verification query* k, with $(i, j, \ldots)$ being the kth entry in the L_v list, then if $b = 0$ send op_i to $\mathcal{A}$, otherwise compute $op \leftarrow \mathsf{ClaimTracker}(\mathsf{vsk}_i, ct_i, t_{\pi(j)})$ and send op to $\mathcal{A}$.
- On a *test query* $(\mathbf{evt}, \mathsf{res}, \Pi^r)$, compute $d \leftarrow \mathsf{VerifyResult}(\mathbf{evt}, \mathsf{res}, \Pi^r)$ and send d to $\mathcal{A}$.
- On a *voter key reveal query* i, send $(\mathsf{vpk}_i, \mathsf{vsk}_i)$ to $\mathcal{A}$.
- On a *tracker reveal query* i, compute $t \leftarrow \mathsf{ExtractTracker}(\mathsf{vsk}_i, ct_i, op_i)$ and send t to $\mathcal{A}$.
- On a *receipt reveal query* k, where the kth entry of L_v is $(\cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \rho_k)$, send ρ_k to $\mathcal{A}$.
- On an *election key reveal query*, send sk to $\mathcal{A}$.

Eventually, the adversary outputs a bit b'.

Fig. 1. Security experiment for privacy, integrity and coercion mitigation. The bit b'' is not used in the experiment, but simplifies the definition of advantage. The adversary makes register and chosen voter key queries, followed by a single tracker generation query, followed by other queries. Queries in framed boxes are only used for privacy and coercion mitigation. Queries in dashed boxes are only used for coercion mitigation. Queries in doubly framed boxes are only used for privacy and integrity (with b fixed to 0). Queries in shaded boxes are only used for integrity.

– for any $k, k' \in \{1, \ldots, n_c\}$ with $k \neq k'$, we have $i_k \neq i_{k'}$ (only one ballot per voter public key).

In this case, we also say that $\mathbf{evt}$ *originated* from $\mathbf{evt}_0$, alternatively from

$$(i_1, j_1, v_{0,1}, v_{1,1}, ev_1, \Pi_1^v, \rho_1), \ldots, (i_{n_c}, j_{n_c}, v_{0,n_c}, v_{1,n_c}, ev_{n_c}, \Pi_{n_c}^v, \rho_{n_c}).$$

Furthermore, we say that a valid sequence $\mathbf{evt}$ is *honest* if at least one of the tuples $(\mathbf{evt}_{j-1}, \mathbf{evt}_j, \Pi_j^s)$ comes from a shuffle query. A valid sequence is *balanced* if the ballot sequences $(v_{0,1}, \ldots, v_{0,n_c})$ and $(v_{1,1}, \ldots, v_{1,n_c})$ are equal up to order.

An execution is *fresh* if the following all hold:

– If a voter secret key, a receipt or a tracker is revealed, then any challenge query for that voter contains the same ballot on the left and the right side.
– For any result query $\mathbf{evt}$ that does not return $\perp$, $\mathbf{evt}$ is balanced and honest.
– For any voter verification query $(j, \mathbf{evt}, \mathsf{res}, \Pi^r)$, $\mathbf{evt}$ contains an encryption of $v_{b,j}$ and $\mathsf{VerifyResult}(\mathsf{pk}, \mathbf{evt}, \mathsf{res}, \Pi^r)$ evaluates to 1.
– For any encrypted ballot returned by a coerce query, if it is in an origin of any result query, the other encrypted ballot returned by the coerce query is also in the same origin of the same result query.
– There is no election key reveal query.

We define the joint privacy and coercion mitigation event E_p to be the event that after the experiment and an adversary has interacted, the execution is fresh and $b' = b$, or the execution is not fresh and $b' = b''$. In other words, if the adversary makes a query that results in a non-fresh execution of the experiment, we simply compare the adversary's guess to a random bit, giving the adversary no advantage over making a random guess.

In the integrity game, the adversary's goal is to achieve inconsistencies:

– The *count failure* event F_c is that a result query for a valid sequence of encrypted ballots and trackers results in $\perp$.
– The *inconsistent result* event F_r is that a test query $(\mathbf{evt}, \mathsf{res}, \Pi^r)$ evaluates to 1, $\mathbf{evt}$ originated from

$$(i_1, \cdot, v_{0,1}, v_{1,1}, ev_1, \Pi_1^v, \rho_1), \ldots, (i_{n_c}, \cdot, v_{0,n_c}, v_{1,n_c}, ev_{n_c}, \Pi_{n_c}^v, \rho_{n_c})$$

  and there is no permutation π on $\{1, \ldots, n_c\}$ such that for $i = 1, \ldots, n_c$, either $v_{b,i} = \perp$ or $\mathsf{Dec}(\mathsf{sk}, ev_{\pi(i)}) = v_{b,i}$.
– The *no unique result* event F_u is that two test queries $(\mathbf{evt}, \mathsf{res}_1, \Pi_1^r)$ and $(\mathbf{evt}', \mathsf{res}_2, \Pi_2^r)$ both evaluate to 1, $\mathbf{evt}$ and $\mathbf{evt}'$ have a common origin, and res_1 and res_2 are not equal up to order.
– The *inconsistent verification* event F_v is that a sequence of voter verification queries $\{(k_j, \mathbf{evt}, \mathsf{res}, \Pi^r)\}_{j=1}^n$ all return 1, $\mathbf{evt}$ is valid, and with $L_v = \big((i_1, \perp, v_{0,1}, v_{1,1}, ev_1, \Pi_1^v, \rho_1), \ldots, (i_{n_c}, \perp, v_{0,n_c}, v_{1,n_c}, ev_{n_c}, \Pi_{n_c}^v, \rho_{n_c})\big)$ there is no permutation π on $\{1, \ldots, n_c\}$ such that $\mathsf{Dec}\left(\mathsf{sk}, ev_{\pi(k_j)}\right) = v_{b,k_j}$ for all $j = 1, \ldots, n$, i.e. that all the specified voters think their ballots are included in the tally, but at least one of the ballots is not.

We define the advantage of an adversary $\mathcal{A}$ against a voting system $\mathcal{S}$ to be

$$\mathsf{Adv}_{\mathcal{S}}^{\mathsf{vote}-x}(\mathcal{A}) = \begin{cases} 2 \cdot |\Pr[E_p] - 1/2| & x = \mathsf{priv} \text{ or } x = \mathsf{c{-}mit}, \text{ or} \\ \Pr[F_c \vee F_r \vee F_u \vee F_v] & x = \mathsf{int}. \end{cases}$$

3.3 The Voting Protocol

The different parties in the voting protocol are the n_v voters and their devices, a trusted *election authority* (EA) who runs setup, registration, tracker generation and who tallies the cast ballots, a collection of n_s shuffle servers, one or more auditors, and a public append-only bulletin board BB. There are many simple variations of the voting protocol.

In the *setup phase*, the EA runs Setup to generate election public and secret keys pk and sk. The public key pk is posted to BB.

In the *registration phase*, the EA runs UserKeyGen(pk) to generate per-voter keys (vpk, vsk) for each voter. The public key vpk is posted to BB and the secret key vsk is sent to the voter's device.

In the *tracker generation phase*, the EA runs TrackerGen(pk, sk, $\mathsf{vpk}_1, \ldots, \mathsf{vpk}_{n_v}$) to generate trackers, encrypted trackers, tracker commitments and openings to the commitments. To break the link between voters and their trackers, the trackers are encrypted and put through a re-encryption mixnet before they are committed to. Each encrypted tracker and commitment is assigned to a voter public key and posted to BB next to this key. Plaintext trackers are also posted to BB.

In the *voting phase*, a voter instructs her device on which ballot v to cast. The voter's device runs the Vote algorithm to produce an encrypted ballot ev and a proof of knowledge Π^v of the underlying plaintext. The encrypted ballot and the proof are added to the web bulletin board next to the voter's public key, encrypted tracker and tracker commitment.

In the *tallying phase*, the auditors first verify the ballot proofs Π_i^v, subsequently ignoring any ballot whose ballot proof does not verify. The pairs (ev_i, et_i) of encrypted ballots and trackers are extracted from the bulletin board and sent to the first shuffle server. The first shuffle server uses the shuffle algorithm Shuffle on the input encrypted ballots and trackers, before passing the shuffled ballots on the next shuffle server, which shuffles the ballots again and sends the shuffled list to the next shuffle server, and so on. All the shuffle servers post their output ciphertexts and shuffle proofs on the bulletin board, and the auditors verify the proofs. If all the shuffles are correct, the EA runs DecryptResult on the output from the final shuffle server, to obtain a result res and a proof Π^r. The auditors verify this too and add their signatures to the bulletin board.

In the *verification phase*, the EA tells each voter which tracker belongs to them (the exact details of how this happens depends on the underlying voting system). The voters then run VoterVerify to verify that their vote was correctly cast and counted. For voting systems without trackers (such as Helios [2] and Belenios [13]), voters simply run VoterVerify without interacting with the EA.

Security Properties. It is easy to see that we can simulate a run of the voting protocol using the experiment. It is also straight-forward for anyone to verify, from the bulletin board alone, if the list of encrypted ballots and trackers that is finally decrypted in a run of the protocol is valid.

For simplicity, we have assumed trusted setup (including tracker generation) and no distributed decryption. We may also assume that any reasonable adversary against the voting scheme has negligible advantage.

It follows, under the assumption of trusted tracker generation, that as long as the contents of the bulletin board verifies, we have verifiability in the sense that the final result is consistent with the ballots of voters that successfully verify. (Though we have not discussed this, one can also verify eligibility by verifying the bulletin board against the electoral roll. When Selene is used without voter signatures, it does not protect against voting on behalf of abstaining honest voters, though such voters could detect this.)

If at least one of the shuffle servers is honest and the election secret key has not been revealed, and the adversary does not manage to organise the voting to get a trivial win, we also have *privacy* and *coercion mitigation*.

4 The Selene Voting System

We provide a model of Selene and analyse it under our security definition. Relative to the original Selene paper, there are three interesting differences/choices: (1) We do not model distributed setup and tracker generation, nor distributed decryption. (2) The voter proves knowledge of the ballot using an equality of discrete logarithm proof. (3) We assume a particular shuffle described in the full version [21] is used. The latter two simplify the security proof by avoiding rewinding. The first is due to lack of space (though see [31] for distributed setup protocols, and [20] for how to model distributed decryption).

4.1 The Voting System

Let $\mathbb{G}$ be a group of prime order q, with generator g. Let $\mathsf{E} = (\mathsf{Kgen}, \mathsf{Enc}, \mathsf{Dec})$ be the ElGamal public key encryption system. Let $\Sigma_{dl} = (\mathcal{P}_{dl}, \mathcal{V}_{dl})$ be a proof system for proving equality of discrete logarithms in $\mathbb{G}$ (e.g. the Chaum-Pedersen protocol [8]). We abuse notation and let $\Sigma_s = (\mathcal{P}_s, \mathcal{V}_s)$ denote both a proof system for shuffling ElGamal ciphertexts and a proof system for shuffling pairs of ElGamal ciphertexts. Our instantiation of Selene works as follows:

- Setup: sample $h_v \xleftarrow{r} \mathbb{G}$ and compute $(\mathsf{pk_v}, \mathsf{sk_v}) \leftarrow \mathsf{Kgen}(1^\lambda)$ and $(\mathsf{pk_t}, \mathsf{sk_t}) \leftarrow \mathsf{Kgen}(1^\lambda)$. The election public key is $\mathsf{pk} = (\mathsf{pk_v}, \mathsf{pk_t}, h_v)$ and the election secret key is $\mathsf{sk} = (\mathsf{sk_v}, \mathsf{sk_t})$.
- $\mathsf{UserKeyGen}(\mathsf{pk})$: compute $(\mathsf{vpk}, \mathsf{vsk}) \leftarrow \mathsf{Kgen}(1^\lambda)$.
- $\mathsf{TrackerGen}(\mathsf{pk}, \mathsf{vpk}_1, \ldots, \mathsf{vpk}_n)$: set $\mathbf{t} \leftarrow (1, \ldots, n)$. Choose a random permutation π on the set $\{1, \ldots, n\}$. For each i, choose random elements $r_i, s_i \xleftarrow{r} \{0, \ldots, q-1\}$, compute ElGamal encryptions $et_i \leftarrow (g^{r_{\pi(i)}}, \mathsf{pk_t}^{r_{\pi(i)}} g^{t_{\pi(i)}})$ and

commitments $ct_i \leftarrow \mathsf{vpk}_i^{s_i} \cdot g^{t_{\pi(i)}}$. Set $op_i = g^{s_i}$. The public output is the list of trackers $\mathbf{t}$, the list of encrypted trackers $\mathbf{et}$ and the list of tracker commitments $\mathbf{ct}$. The private output is the list of openings $\mathbf{op}$ to the commitments and the permutation π.

- $\mathsf{ExtractTracker}(\mathsf{vsk}, ct, op)$: compute $g^t \leftarrow ct \cdot op^{-\mathsf{vsk}}$.
- $\mathsf{ClaimTracker}(\mathsf{vsk}, ct, g^t)$: compute $op \leftarrow (ct/g^t)^{1/\mathsf{vsk}}$.
- $\mathsf{Vote}(\mathsf{pk}, v)$: sample $r \xleftarrow{\mathrm{r}} \{0, \ldots, q-1\}$ and compute $x \leftarrow g^r$, $\hat{x} \leftarrow h_v^r$ and $w \leftarrow \mathsf{pk}_v^r v$. Compute a proof $\Pi^{dl} \leftarrow \mathcal{P}_{dl}((g, h_v, x, \hat{x}), r)$ showing that $\log_g x = \log_{h_v} \hat{x} = r$. Output $c = (x, w)$, $\Pi^v = (\hat{x}, \Pi^{dl})$ and $\rho = v$.
- $\mathsf{Shuffle}(\mathsf{pk}, \mathbf{evt})$: sample two lists $\mathbf{r}_v, \mathbf{r}_t \xleftarrow{\mathrm{r}} \{0, \ldots, q-1\}^n$ and a random permutation on the set $\{1, \ldots, n\}$. For each $((x_{v,i}, w_{v,i}), (x_{t,i}, w_{t,i})) \in \mathbf{evt}$, compute $x'_{v,i} \leftarrow g^{r_{v,\pi(i)}} x_{v,\pi(i)}, w'_{v,i} \leftarrow \mathsf{pk}_v^{r_{v,\pi(i)}} w_{v,\pi(i)}, x'_{t,i} \leftarrow g^{r_{t,\pi(i)}} x_{t,\pi(i)}$ and $w'_{t,i} \leftarrow \mathsf{pk}_t^{r_{t,\pi(i)}} w_{t,\pi(i)}$. Compute a proof $\Pi^s \leftarrow \mathcal{P}_s((\mathbf{evt}, \mathbf{evt}'), (\mathbf{r}_v, \mathbf{r}_t, \pi))$ of correct shuffle and output $(\mathbf{evt}', \Pi^s)$.
- $\mathsf{DecryptResult}(\mathsf{sk}, \mathbf{evt})$: for each $((x_{v,i}, w_{v,i}), (x_{t,i}, w_{t,i})) \in \mathbf{evt}$, compute $v_i \leftarrow \mathsf{Dec}(\mathsf{sk}_v, (x_{v,i}, w_{v,i}))$, $t_i \leftarrow \mathsf{Dec}(\mathsf{sk}_t, (x_{t,i}, w_{t,i}))$ and proofs $\Pi^{dl}_{v,i} \leftarrow \mathcal{P}_{dl}((g, x_{v,i}, \mathsf{pk}_\mathsf{v}, w_{v,i}/v_i), \mathsf{sk}_\mathsf{v})$ and $\Pi^{dl}_{t,i} \leftarrow \mathcal{P}_{dl}((g, x_{t,i}, \mathsf{pk}_\mathsf{t}, w_{t,i}/t_i), \mathsf{sk}_\mathsf{t})$, proving that $\log_g \mathsf{pk}_\mathsf{v} = \log_{x_{v,i}}(w_{v,i}/v_i) = \mathsf{sk}_\mathsf{v}$ and $\log_g \mathsf{pk}_\mathsf{t} = \log_{x_{t,i}}(w_{t,i}/t_i) = \mathsf{sk}_\mathsf{t}$. Set $\mathsf{res} \leftarrow \mathbf{v}$ and $\Pi^r \leftarrow (\{\Pi^{dl}_{v,i}\}, \{\Pi^{dl}_{t,i}\}, \mathbf{t})$ and output (res, Π^r).
- $\mathsf{VoterVerify}(\rho, t, \mathbf{evt}, \mathbf{v}, \Pi^r)$: parse Π^r as $(\{\Pi^{dl}_{v,i}\}, \{\Pi^{dl}_{t,i}\}, \mathbf{t})$ and check if $\rho \in \mathbf{v}$, and $t \in \mathbf{t}$, and that if $t = t_i$ then $\rho = v_i$, i.e. the ballot appears next to the correct tracker.
- $\mathsf{VerifyShuffle}(\mathsf{pk}, \mathbf{evt}, \mathbf{evt}', \Pi^s)$: compute $d \leftarrow \mathcal{V}_s(\mathsf{pk}, \mathbf{evt}, \mathbf{evt}', \Pi^s)$.
- $\mathsf{VerifyBallot}(\mathsf{pk}, ev, \Pi^v)$: parse Π^v as $(\hat{x}, \Pi^{dl})$ and compute $d \leftarrow \mathcal{V}_{dl}((g, h, x, \hat{x}), \Pi^{dl})$.
- $\mathsf{VerifyResult}(\mathsf{pk}, \mathbf{evt}, \mathsf{res}, \Pi^r)$: parse Π^r as $(\{\Pi^{dl}_{v,i}\}, \{\Pi^{dl}_{t,i}\}, \mathbf{t})$ and compute $d_{v,i} \leftarrow \mathcal{V}_{dl}((g, x_{v,i}, \mathsf{pk}_\mathsf{v}, w_{v,i}/v_i), \Pi^{dl}_{v,i})$ and $d_{t,i} \leftarrow \mathcal{V}_{dl}((g, x_{t,i}, \mathsf{pk}_\mathsf{t}, w_{t,i}/t_i), \Pi^{dl}_{t,i})$ for all $i = 1, \ldots, n$, where $((x_{v,i}, w_{v,i}), (x_{t,i}, w_{t,i})) \in \mathbf{evt}, v_i \in \mathsf{res}, t_i \in \mathbf{t}$.

The correctness of Selene follows from the correctness of ElGamal, the completeness of the verifiable shuffles and the straight-forward computation

$$\mathsf{ExtractTracker}(\mathsf{vsk}, ct, \mathsf{ClaimTracker}(\mathsf{vsk}, ct, g^t)) = ct \cdot \left(\left(ct/g^t \right)^{1/\mathsf{vsk}} \right)^{-\mathsf{vsk}} = g^t.$$

Note that in the original description of Selene [31], the exact manner of which the voters prove knowledge of their plaintext in the voting phase is left abstract. However, several different approaches are possible. One may, for example, produce a Schnorr proof of knowledge [34] of the randomness used by the encryption algorithm. We choose a different approach, and include a check value $\hat{x}$ and give a Chaum-Pedersen proof that $\log_{h_v} \hat{x} = \log_g x$. Both are valid approaches, however our approach simplifies the security proof by avoiding rewinding.

4.2 Security Result

We say that an adversary against a voting scheme is *non-adaptive* if every voter key reveal query is made before the tracker generation query.

Theorem 1. *Let $\mathcal{A}$ be a non-adaptive $(\tau, n_v, n_c, n_d, n_s)$-adversary against Selene, making at most n_v registration and chosen voter key queries, n_c challenge and coerce queries, n_d chosen ciphertext queries, and n_s shuffle/chosen shuffle queries, and where the runtime of the adversary is at most τ. Then there exist a τ'_1-distinguisher $\mathcal{B}_1$, a $\tau'_{2,1}$-distinguisher $\mathcal{B}_{2,1}$, a $\tau'_{2,2}$-distinguisher $\mathcal{B}_{2,2}$ and a τ'_3-distinguisher $\mathcal{B}_3$, all for DDH, $\tau'_1, \tau'_{2,1}, \tau'_{2,2}, \tau'_3$ all essentially equal to τ, such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{vote-x}}_{\mathsf{Selene}}(\mathcal{A}) \leq\ & \mathsf{Adv}^{\mathsf{ddh}}_{\mathbb{G},g}(\mathcal{B}_1) + 2n_s(\mathsf{Adv}^{\mathsf{ddh}}_{\mathbb{G},g}(\mathcal{B}_{2,1}) + \mathsf{Adv}^{\mathsf{ddh}}_{\mathbb{G},g}(\mathcal{B}_{2,2})) \\ & + \mathsf{Adv}^{\mathsf{ddh}}_{\mathbb{G},g}(\mathcal{B}_3) + \textit{negligible terms},\end{aligned}$$

where $\mathsf{x} \in \{\mathsf{priv}, \mathsf{c{-}mit}, \mathsf{int}\}$.

(Better bounds in the theorem are obtainable, but these are sufficient.)

4.3 Proof Sketch

We begin by analysing the integrity events. *Count failures* cannot happen. If we get an inconsistent result, then either the equality of discrete logarithm proofs used by the decryption algorithm or the shuffle proofs are wrong. The soundness errors of the particular proofs we use are negligible (and unconditional), so an *inconsistent result* happens with negligible probability. The same analysis applies to *non-unique results* as well as *inconsistent verification*.

We now move on to analysing the privacy event. The proof is structured as a sequence of games. We begin by simulating the honestly generated non-interactive proofs during ballot casting. This allows us to randomize the check values $\hat{x}_v$ in honestly generated ballot proofs, so that we afterwards can embed a trapdoor in h_v. The trapdoors allow us to extract ballots from adversarially generated ciphertexts. The shuffle we use also allows us to extract permutations from adversarially generated shuffles by tampering with a random oracle. This allows us to use the ballots from chosen ciphertext queries to simulate the decryption, so we no longer use the decryption key. The next step is to also simulate the honest shuffles, before randomising the honestly generated ciphertexts (including encrypted trackers) and the re-randomisations of these ciphertexts. Finally, we sample tracker commitments at random and compute the openings from tracker generation using the ClaimTracker algorithm. This change is not observable, and makes the computation of tracker commitments and openings independent of the challenge bit. This makes the entire game independent of the challenge bit, proving that the adversary has no advantage.

The complete security proof can be found in the full version [21].

5 Other Variants of Selene

There are [30,31] some challenges tied to the use of trackers in Selene. First, if the coercer is also a voter, there is a possibility that a coerced voter points to

the coercer's own tracker when employing the coercion evasion strategy. Second, publishing the trackers in the clear next to the ballots might affect the voters' *perceived* privacy, and some might find this troublesome.

To address the first challenge, the authors of Selene have proposed a variant they call Selene II. Informally, the idea is to provide each voter with a set of alternative (or dummy) trackers, one for each possible candidate, in a way that the set of alternative trackers is unique to each voter. This way, it is not possible for a coerced voter to accidentally point to the coercer's tracker. However, trackers are still published in the clear.

Both challenges are also addressed by Ryan *et al.* [30], who have proposed a voting system they call Hyperion. The idea is to only publish commitments next to the plaintext ballots, rather than plaintext trackers. Furthermore, to avoid the issue that voters might accidentally point to the coercer's own tracker, each voter is given their unique view of the bulletin board.

For both Selene II [31] and Hyperion [30], we refer to the original papers for the full details of the constructions, but we briefly describe here how these systems fit into our framework. We first remark that in Selene II, it is necessary that the encryption system used to encrypt the ballots supports *plaintext equivalence tests* (PETs). As in the original description of Selene, we use ElGamal encryption to encrypt the ballots, so PETs are indeed supported (see e.g. [24]).

For Selene II, we need to change the TrackerGen algorithm so that it outputs $c+1$ trackers for each voter, where c is the number of candidates, and c "dummy" ciphertexts, one ciphertext for each candidate. We let the last tracker be the one that is sent to the voter to be used for verification. By construction, for all voters there will be an extra encrypted ballot for each candidate. Thus, the DecryptResult algorithm works similarly as for Selene, except that it needs to subtract n_v votes for each candidate, where n_v is the number of voters. The *voting protocol* must also be changed. Before notifying the voters of their tracking numbers, the EA must now perform a PET between each voter's submitted ciphertext, and each of the "dummy" ciphertexts belonging to the voter, before removing the ciphertext (and the corresponding tracker) containing the same candidate as the voter voted for. This way, all voters receive a set of trackers, each pointing to a different candidate, which is unique to them. The opening to their real trackers is transmitted as usual, and thus the ExtractTracker algorithm works as in Selene. The ClaimTracker algorithm also works exactly as in Selene, except that voters now can choose a tracker from their personal set of dummy trackers, thus avoiding the risk of accidentally choosing the coercer's tracker.

For Hyperion, the modification of the TrackerGen algorithm is straight forward: we simply let it compute tracker commitments as described in [30], namely by (for each voter) sampling a random number r_i and computing the commitment as $\mathsf{vpk}_i^{r_i}$. At the same time, an opening is computed as $op_i \leftarrow g^{r_i}$. The Shuffle algorithm still shuffles the list of encrypted ballots and tracker commitments in parallel, in the sense that they are subjected to the same permutation. However, the encrypted ballots are put through the same re-encryption shuffle as before, but the tracker commitments are put through an *exponentiation mix*,

raising all commitments to a common secret power s. The DecryptResult algorithm now performs additional exponentiation mixes to the commitments, one mix for each voter (by raising the commitment to a secret power s_i, unique to each voter), giving the voters their own unique view of the result. For each voter, it also computes the final opening to their commitments, as $op_i \leftarrow g^{r_i \cdot s \cdot s_i}$. Again, we need to change the voting protocol, this time so that each voter actually receives their own view of the bulletin board. The ExtractTracker algorithm raises the opening op_i to the voter's secret key and loops through the bulletin board to find a matching commitment. The ClaimTracker algorithm uses the voter's secret key to compute an opening to a commitment pointing to the coercer's desired ballot.

Acknowledgments. We thank the anonymous reviewers at E-Vote-ID for their helpful comments. This work was supported by the Luxembourg National Research Fund (FNR) and the Research Council of Norway (NFR) for the joint project SURCVS (FNT project ID 11747298, NFR project ID 275516). Thomas Haines is the recipient of an Australian Research Council Australian Discovery Early Career Award (project number DE220100595).

References

1. Final report of IACR electronic voting committee. https://www.iacr.org/elections/eVoting/finalReportHelios_2010-09-27.html. Accessed 05 May 2023
2. Adida, B.: Helios: web-based open-audit voting. In: van Oorschot, P.C. (ed.) USENIX Security 2008, pp. 335–348. USENIX Association (2008)
3. Baloglu, S., Bursuc, S., Mauw, S., Pang, J.: Election verifiability in receipt-free voting protocols. In: 2023 2023 IEEE 36th Computer Security Foundations Symposium (CSF) (CSF), pp. 63–78. IEEE Computer Society, Los Alamitos (2023). https://doi.org/10.1109/CSF57540.2023.00005
4. Benaloh, J.: Verifiable Secret-Ballot Elections. Ph.D. thesis (September 1987). https://www.microsoft.com/en-us/research/publication/verifiable-secret-ballot-elections/
5. Benaloh, J.C., Tuinstra, D.: Receipt-free secret-ballot elections (extended abstract). In: 26th ACM STOC, pp. 544–553. ACM Press (1994). https://doi.org/10.1145/195058.195407
6. Bernhard, D., Cortier, V., Galindo, D., Pereira, O., Warinschi, B.: A comprehensive analysis of game-based ballot privacy definitions. Cryptology ePrint Archive, Report 2015/255 (2015). https://eprint.iacr.org/2015/255
7. Chaidos, P., Cortier, V., Fuchsbauer, G., Galindo, D.: BeleniosRF: a non-interactive receipt-free electronic voting scheme. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016, pp. 1614–1625. ACM Press (2016). https://doi.org/10.1145/2976749.2978337
8. Chaum, D., Pedersen, T.P.: Wallet databases with observers. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 89–105. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-48071-4_7
9. Clarkson, M.R., Chong, S., Myers, A.C.: Civitas: toward a secure voting system. In: 2008 IEEE Symposium on Security and Privacy, pp. 354–368. IEEE Computer Society Press (2008). https://doi.org/10.1109/SP.2008.32

10. Cortier, V., Dragan, C.C., Dupressoir, F., Schmidt, B., Strub, P.Y., Warinschi, B.: Machine-checked proofs of privacy for electronic voting protocols. In: 2017 IEEE Symposium on Security and Privacy, pp. 993–1008. IEEE Computer Society Press (2017). https://doi.org/10.1109/SP.2017.28
11. Cortier, V., Dragan, C.C., Dupressoir, F., Warinschi, B.: Machine-checked proofs for electronic voting: privacy and verifiability for Belenios. In: Chong, S., Delaune, S. (eds.) CSF 2018 Computer Security Foundations Symposium, pp. 298–312. IEEE Computer Society Press (2018). https://doi.org/10.1109/CSF.2018.00029
12. Cortier, V., Galindo, D., Küsters, R., Mueller, J., Truderung, T.: SoK: verifiability notions for E-voting protocols. In: 2016 IEEE Symposium on Security and Privacy, pp. 779–798. IEEE Computer Society Press (2016). https://doi.org/10.1109/SP.2016.52
13. Cortier, V., Gaudry, P., Glondu, S.: Belenios: a simple private and verifiable electronic voting system. In: Guttman, J.D., Landwehr, C.E., Meseguer, J., Pavlovic, D. (eds.) Foundations of Security, Protocols, and Equational Reasoning. LNCS, vol. 11565, pp. 214–238. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-19052-1_14
14. Cortier, V., Gaudry, P., Yang, Q.: Is the JCJ voting system really coercion-resistant? Cryptology ePrint Archive, Report 2022/430 (2022). https://eprint.iacr.org/2022/430
15. Cortier, V., Lallemand, J., Warinschi, B.: Fifty shades of ballot privacy: privacy against a malicious board. In: Jia, L., Küsters, R. (eds.) CSF 2020 Computer Security Foundations Symposium, pp. 17–32. IEEE Computer Society Press (2020). https://doi.org/10.1109/CSF49147.2020.00010
16. Damgård, I.: Commitment schemes and zero-knowledge protocols. In: Damgård, I.B. (ed.) EEF School 1998. LNCS, vol. 1561, pp. 63–86. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48969-X_3
17. Distler, V., Zollinger, M.L., Lallemand, C., Roenne, P.B., Ryan, P.Y.A., Koenig, V.: Security - visible, yet unseen? CHI '19, pp. 1–13. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3290605.3300835
18. Dragan, C.C., et al.: Machine-checked proofs of privacy against malicious boards for Selene & co. In: CSF 2022 Computer Security Foundations Symposium, pp. 335–347. IEEE Computer Society Press (2022). https://doi.org/10.1109/CSF54842.2022.9919663
19. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Blakley, G.R., Chaum, D. (eds.) CRYPTO'84. LNCS, vol. 196, pp. 10–18. Springer, Heidelberg (Aug (1984). https://doi.org/10.1007/3-540-39568-7_2
20. Gjøsteen, K.: Practical Mathematical Cryptography. Chapman and Hall/CRC, Boca Raton (2023)
21. Gjøsteen, K., Haines, T., Solberg, M.R.: Coercion mitigation for voting systems with trackers: a Selene case study. Cryptology ePrint Archive, report 2023/1102 (2023). https://eprint.iacr.org/2023/1102
22. Haines, T., Müller, J.: SoK: techniques for verifiable mix nets. In: Jia, L., Küsters, R. (eds.) CSF 2020 Computer Security Foundations Symposium, pp. 49–64. IEEE Computer Society Press (2020). https://doi.org/10.1109/CSF49147.2020.00012
23. Iovino, V., Rial, A., Rønne, P.B., Ryan, P.Y.A.: Using Selene to verify your vote in JCJ. In: Brenner, M., et al. (eds.) FC 2017 Workshops. LNCS, vol. 10323, pp. 385–403. Springer, Heidelberg (Apr (2017). https://doi.org/10.1007/978-3-319-70278-0_24

24. Jakobsson, M., Juels, A.: Mix and match: secure function evaluation via ciphertexts. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 162–177. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44448-3_13
25. Juels, A., Catalano, D., Jakobsson, M.: Coercion-resistant electronic elections. Cryptology ePrint Archive, Report 2002/165 (2002). https://eprint.iacr.org/2002/165
26. Karayumak, F., Olembo, M.M., Kauer, M., Volkamer, M.: Usability analysis of Helios-an open source verifiable remote electronic voting system. In: EVT/WOTE, vol. 11, no. 5 (2011)
27. Küsters, R., Müller, J., Scapin, E., Truderung, T.: sElect: a lightweight verifiable remote voting system. In: Hicks, M., Köpf, B. (eds.) CSF 2016 Computer Security Foundations Symposium, pp. 341–354. IEEE Computer Society Press (2016). https://doi.org/10.1109/CSF.2016.31
28. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 129–140. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_9
29. Ramchen, K.: Parallel shuffling and its application to Prêt à voter. In: EVT/WOTE (2010)
30. Ryan, P.Y.A., Rastikian, S., Rønne, P.B.: Hyperion: an enhanced version of the Selene end-to-end verifiable voting scheme. E-Vote-ID 2021, 285 (2021)
31. Ryan, P.Y.A., Rønne, P.B., Iovino, V.: Selene: voting with transparent verifiability and coercion-mitigation. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 176–192. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_12
32. Sallal, M., et al.: VMV: augmenting an internet voting system with Selene verifiability (2019)
33. Schneier, B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd edn. John Wiley & Sons Inc, Hoboken (1995)
34. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 239–252. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_22
35. Smyth, B.: Surveying definitions of coercion resistance. Cryptology ePrint Archive, Report 2019/822 (2019). https://eprint.iacr.org/2019/822
36. Smyth, B.: Ballot secrecy: security definition, sufficient conditions, and analysis of Helios. J. Comput. Secur. **29**(6), 551–611 (2021). https://doi.org/10.3233/JCS-191415
37. Smyth, B., Clarkson, M.R.: Surveying definitions of election verifiability. Cryptology ePrint Archive, Report 2022/305 (2022). https://eprint.iacr.org/2022/305
38. Zollinger, M.L., Distler, V., Rønne, P., Ryan, P., Lallemand, C., Koenig, V.: User experience design for E-voting: how mental models align with security mechanisms (2019). https://doi.org/10.13140/RG.2.2.27007.15527