Hyperion: Transparent End-to-End Verifiable Voting with Coercion Mitigation

Aditya Damodaran[1][0000−0003−4030−6859], Simon Rastikian[2], Peter B. Rønne[1][0000−0002−2785−8301], and Peter Y. A. Ryan[1][0000−0002−1677−9034]

[1] SnT, University of Luxembourg, Luxembourg
[2] Near One Limited, United Kingdom

Abstract. We present *Hyperion*, an end-to-end verifiable e-voting scheme that allows the voters to identify their votes in cleartext in the final tally. In contrast to schemes like *Selene* or *sElect*, identification is not via (private) tracker numbers but via cryptographic commitment terms. After publishing the tally, the *Election Authority* provides each voter with an individual dual key. Voters identify their votes by raising their dual key to their secret trapdoor key and finding the matching commitment term in the tally. The dual keys are self-certifying in that, without the voter's trapdoor key, it is intractable to forge a dual key that, when raised to the trapdoor key, will match an alternative commitment. On the other hand, a voter can use their own trapdoor key to forge a dual key to fool any would-be coercer.

We provide new improved definitions of privacy and verifiability for e-voting schemes and prove the scheme secure against these, as well as proving security with respect to earlier definitions in the literature.

We provide a prototype implementation and provide measurements which demonstrate that our scheme is practical for large scale elections.

1 Introduction

Many democracies are moving towards voting over the internet, and some, e.g. Estonia, has fully adopted it. While internet voting has many attractions it introduces new, poorly understood threats. The internet is inherently insecure and remote voting introduces coercion threats not present in in-person voting. To counter these threats, cryptographic mechanisms and protocols have been proposed. However, designing and analysing such protocols is very challenging, and we have not reached consensus on rigorous definitions of security properties such as *vote secrecy*, *verifiability*, *receipt-freeness*, *coercion-resistance* and *dispute resolution*.

A good voting system should not only deliver the correct result w.r.t. the legitimately cast votes, but also provide sufficient evidence to convince all observers of the announced result. Ensuring both vote secrecy and verifiability is complex, and indeed, many technologies sacrifice the latter, forcing the stakeholders to place total, blind trust in the correct behaviour of the code, for example direct-recording electronic (DRE) machines. Such observations motivated the development of end-to-end verifiable (E2E V) schemes [21] and the notion of software independence [28].

E2E V schemes usually involve the creation of an encryption or encoding of the vote at the time of casting, a copy of which is retained by the voter. Later the voter can check that her "receipt" appears correctly on an append-only public ledger called the *Bulletin Board* (BB). After this, a universally verifiable, anonymising tally is performed on the posted, encrypted ballots to reveal the result. Voters can also perform some form of ballot auditing before casting to gain assurance that their vote is correctly represented in their ballot. Putting these steps together ensures that the corruption of any vote during recording and tallying is detectable. Along with mechanisms to prevent ballot stuffing and clash attacks (ballot collisions) etc. we can detect any inaccuracy in the announced outcome.

Such schemes, while technically appealing, have at least two drawbacks. First, the fact that errors can be detected does not guarantee that they will be: it is essential that sufficient numbers of voters and observers actually perform the checks diligently and report anomalies. Second, a voting scheme must be easily understandable and usable by voters and voting officials. The assurance argument outlined above is rather subtle, and not easy for many voters or stakeholders to digest. Many find the idea of voters having to perform checks on encrypted ballots unreasonable.

These observations prompted the exploration of more direct and transparent forms of verification, in particular based on the idea of private tracker numbers to identify votes in cleartext in the tally. Examples of such schemes include the CNRS scheme [3], *Selene* [30] and sElect [25]. Of these, *Selene* is of particular interest as it provides mitigation of the coercion threats that tracker based schemes otherwise exhibit: the coercer demands the voter to reveal her tracker. Notably, the Selene construction has been trialled in elections in The Royal College of Nursing and The College of Podiatrists in the UK [32] and for elections in the ESORICS steering committee.

1.1 Contribution

We present a novel, E2E verifiable scheme, inspired by the *Selene* scheme [30], that not only provides a highly transparent verification, but also affords voters a greater sense of privacy than with *Selene*. *Hyperion* is significantly more efficient as it greatly simplifies the setup (section 2.2) of *Selene*, eliminating all the encryption, mixing, decryption and ZK proofs computations implied by the use of tracker numbers.

Hyperion, in contrast to *Selene*, does not publicly reveal trackers, indeed, we do away entirely with trackers. Instead, the voter identifies her vote in the tally by identifying the row in the tally containing the commitment that opens, with her trapdoor secret key and dual key, to a constant, e.g. 1. This is rather like identifying your house by finding the door that opens to your key. This is still deniable, but the mechanism is now different: a coerced voter identifies a commitment paired with the coercer's required vote and, if necessary computes, using her trapdoor key, the fake dual key that opens the chosen commitment to 1. Doing away with the trackers also improves the situation for coerced voters, since they do not need to equivocate and lie about trackers that they have seen and which could have easy-to-remember or characteristic features. As an example, a voter might accidentally reveal having a tracker with consecutive numbers.

The *Hyperion* construction has further advantages over *Selene*. In particular, variants have been created that exploit the fact that the cryptographic commitments are perfectly randomly distributed to an observer only seeing the bulletin board. This can be used for quantum-safety against future attackers or even satisfying everlasting privacy (see the long version of this paper [14] where further variants are explored), and has even been used to construct the first scheme with everlasting receipt-freeness [26].

Importantly, we also contribute novel definitions, providing a ballot privacy definition allowing maliciously generated public keys by corrupted voters, and considering stronger adversaries with access to information about whether voters verify successfully or not. We also give a definition of verifiability against a malicious voting board and consider malware on the user side, and we prove verifiability when either the vote-casting or the vote-verification device is uncorrupted. We also provide proofs of security against established definitions in the literature, especially we prove privacy against a malicious board as in [12, 16].

Finally we provide a prototype implementation along with performance data.

Structure of the Paper We first discuss related work and introduce the notation used in the paper. We then describe the voter experience in Sec. 2: casting and verifying a vote and, where necessary, evading the coercer. The precise instantiation used for security proofs is presented in Sec. 3. The remainder of the paper presents the security definitions for ballot privacy (Sec. 4) and integrity (Sec. 5) followed by game-based proofs. These definitions are also novel and represent a contribution to the state of the art in the field. For the ballot privacy definition, we consider adversaries who get information on whether the verification of voters failed or not. Such information can lead to privacy attacks, as demonstrated in other protocols, and are important to counter in *Hyperion*. For verifiability, we craft a definition considering that each voter has two devices – one for vote casting and one for verification, and we demonstrate that both need to be corrupted for successful verifiability attacks against *Hyperion*.

Finally, we include some performance statistics for a prototype implementation that demonstrate that the scheme is practical for large scale elections, e.g of the order of a million voters, see App. A.

Related Work Most of the end-to-end verifiable schemes proposed to date involve a rather indirect verification by the voter: checking that an encryption of their vote appears on the BB in the input to a (universally verifiable) tally process. Some recent schemes seek a more direct and arguably more compelling voter verification process: identifying the vote in plaintext in the final tally. Here we focus on the latter class.

Schneier [33], proposes the idea of voters attaching a password to their vote which is then posted alongside the vote on the BB. Later, [3] elaborate on this in a boardroom context. sElect [25], is also tracker-based but with the additional feature of having an accountable tally process. All of these systems are vulnerable to the obvious threat of the coercer demanding the voter reveal her tracker. *Selene* [30] introduced the idea of delayed notification of the trackers to mitigate the coercion threat, along with constructions to guarantee uniqueness and deniability of the trackers.

Some adaptations of *Selene* have been presented in an in-person variant [29, 38] and in a JCJ-like variant [22] which offers greater coercion-resistance. *Selene* has been analysed symbolically in [7] and implemented (using a distributed ledger) in [32].

Hyperion, like *Selene*, provides a direct and intuitive way for voters to verify their votes, however, it does away with the need for trackers. This modification greatly simplifies the setup and, more importantly, voters should feel much more comfortable about the privacy of their vote. Studies, [15, 2, 37] suggest that some voters are troubled by having their vote appear publicly beside their tracker. We hypothesise that voters will be more comfortable with the *Hyperion* verification, but this needs to be investigated by a complementary user study.

Selene has the problem that the coercer might claim ownership of a faked tracker offered by a coerced voter, or that it coincides with one offered by another victim. Several enhancements to *Selene* to counter this have been suggested including adding extra dummy trackers [30] and shrouding parts of the trackers or votes [23]. The *Hyperion* construction presented here, combined with a further innovation: individual bulletin boards, elaborated in the full version of this paper, provides a more elegant solution.

Regarding the definitions, our verifiability definition builds on [10], which following [11] is the best choice for our case. Our definition benefits from a detailed model that allows corruption of vote casting and usage of verification devices.

For the ballot privacy definition, the state of the art was summarised in the SoK paper [5] which also presented a game-based definition BPRIV that implies an ideal functionality under certain conditions and which covers all types of tally functions. This definition was further extended to considering more general attacks during vote casting and malicious boards in [12]. Unfortunately, that definition does not capture schemes where the verification happens after the tally as in *Hyperion*. Recently, a new definition was presented in [16] allowing late verification and which also included a machine-checked proof of ballot privacy for Selene. Whereas the last definition would be applicable to *Hyperion*, it does not capture attacks where the attacker has access to whether the voter's verification is successful or not. Since Hyperion allows a direct check of the tallied plaintext vote, this would immediately cause privacy problems if the adversary manages to cast a vote on behalf of the voter. However, the BPRIV type of definitions, especially [16], does not capture these types of attack, and are not well-suited to do this due to being based on a simulated view. Instead, we here go back to a very early definition by Benaloh [4], but update this with inspiration from [16], also taking into account that the adversary can register maliciously generated keys.

Notation This paper includes writing program code. Besides standard notation for assigning ‘ $\leftarrow$ ’ and random sampling ‘ $\twoheadleftarrow$ ’, we will also use $\mathrm{X} \overset{\cup}{\leftarrow} \mathrm{Y}$ as shorthand for $\mathrm{X} \leftarrow \mathrm{X} \cup \mathrm{Y}$. Similarly, we write $m \overset{\shortparallel}{\leftarrow} n$ shorthand for $m \leftarrow m \shortparallel n$. Security games invoke an efficient adversary $\mathcal{A}$ with access to some oracles. The games terminate when executing **Stop with** $\cdot$ command. Each game is associated to a certain winning probability. We write $\Pr[G(\mathcal{A})]$ for the probability that game G invoked with adversary $\mathcal{A}$ stops with $\top$. Game codes will be compacted by introducing the instructions **Require** $\cdot$ which stands for ‘ if not $\cdot$ then **Stop with** $\bot$ ’ and **Promise** $\cdot$ which stands for ‘ if not $\cdot$ then **Stop with** $\top$ ’.

2 Details of the Scheme

In this section, we describe the main variant of *Hyperion*. We note that the *Hyperion* verification mechanism is versatile and could be incorporated in an existing voting scheme. For concreteness, we present it as a self contained scheme. A protocol flow diagram can be found in the long version of this paper [14] (Fig. 7) along with additional details.

2.1 Parties Involved

Election Authority (EA). Performs the general election setup, i.e. defines the election parameters, the ballot styles etc. and sets up the initial Bulletin Board.
Bulletin Board (BB). We consider an append-only board with a consistent view for all participants.
Voters. Each voter i is identified uniquely with an id_i and holds two secret keys: a signing key used to authenticate the ballot and a verification trapdoor key used to verify the plaintext vote. These keys can be stored on two different devices/apps that assist the voters in casting and verifying their votes.
Registration Authority. Identifies the eligible voters and posts their public keys on BB.
Tally Tellers (TT). Are responsible for setting up a shared (threshold) public election key pk_{EA} which will be used for encryption. They also perform a verifiable decryption during the tally phase.
Mix-Tellers. Is a set of mix tellers that perform a verifiable parallel mix in order to anonymise the votes.

2.2 The Setup

The election authority publishes the relevant details of the election including a cryptographic setup of a secure prime order group on the BB. A set of tally tellers create a threshold public key pair for the election $(sk_{\mathsf{EA}}, pk_{\mathsf{EA}})$ and publishes pk_{EA}. We assume here that each eligible voter holds a valid private signing key with corresponding certification key pk_i published along with unique voter identifiers id_i on BB. The unique identifiers enable *universal eligibility verifiability*. We trust the registration authority to set this up correctly[3]. Note that the setup here is much simpler than that of *Selene* which requires additional verified generation, encryption and mixing of tracking numbers.

2.3 Voting

Each voter generates an ephemeral trapdoor key x_i using her device. The public component $h_i := g^{x_i}$ will be registered during vote casting, along with a Zero Knowledge Proof of Knowledge (ZKPoK) of x_i. For all proofs that follow, we assume that these proofs are non-malleable and include binding to a unique election identifier and the public election key pk_{EA}. The proofs here should also be bound to the identity id_i of the voters to prevent the public keys from being copied. In our case a simple Schnorr proof [34] is sufficient, made non-interactive via the (strong) Fiat-Shamir transformation [18, 6] and including all the necessary information in the hash for non-malleability.

Voting proceeds as follows: voter i sends her trapdoor key h_i along with a ZKPoK of x_i, an encryption $\{v_i\}_{pk_{\mathsf{EA}}}$ of her vote v_i (e.g. ElGamal [17]) and the well-formedness ZK proofs of encryption, i.e. a proof of the vote be in the correct space and a proof of plaintext-knowledge[4]. Recall that these proofs are non-malleable and bound to the voter id_i to prevent vote copy attacks[5] [13]. The encryption scheme should support verifiable mixing and together with the ZKPs be IND-1-CCA (see App. A in [14]). We denote the concatenation of the ZK proofs by Π_i. Registering the (ephemeral) trapdoor keys at the same time as casting the vote avoids the need for an extra registration phase. All of this is signed, sent to the EA and appended next to the appropriate pk_i on the BB (for brevity $\mathsf{sign}(m)$ means the signature with the message m included):

$$id_i,\ pk_i,\ \mathsf{sign}_i(\{v_i\}_{pk_{\mathsf{EA}}},\ h_i,\ \Pi_i)$$

2.4 Tallying

Once the voting phase has closed, ballots posted to the BB with valid signatures and proofs are identified. For these, the Tally Tellers now take each public trapdoor key h_i and privately raise this to a fresh, random, secret r_i, encrypt it and post the output on BB together with Π_i^{TT}, a ZKPoK of honest construction with knowledge of r_i and the encryption random coins.[6] For ElGamal this proof

[3] In Estonia, each voter has her keys integrated in her identity card.

[4] A simple choice is Chaum-Pedersen proofs of discrete log equality using OR Sigma protocols for the different vote choices.

[5] Vote copy attacks would undermine coercion resistance with plaintext verification.

[6] This can easily be distributed over the Tally Tellers for ElGamal. For instance, each TT_j posts $\{h_i^{r_{i,j}}\}_{pk_{\mathsf{EA}}}$ together with the appropriate ZKPoK, then these ciphertexts

can be efficiently implemented, see e.g. [8]. The Tellers keep the corresponding g^{r_i} (dual key) terms secret, for the verification phase. The BB now contains, for the rows with valid ballots, the following:

$$id_i,\ pk_i,\ \mathsf{sign}_i(\{v_i\}_{pk_{\mathsf{EA}}},\ h_i,\ \Pi_i),\ \{h_i^{r_i}\}_{pk_{\mathsf{EA}}},\ \Pi_i^{\mathsf{TT}}$$

The pairs $(\{v_i\}_{pk_{\mathsf{EA}}},\ \{h_i^{r_i}\}_{pk_{\mathsf{EA}}})$ are shuffled in parallel by a verifiable mix-net and verifiably decrypted to obtain the final Tally Board

$$v_i,\ h_i^{r_i}$$

together with the ZKP of correct parallel mixing and decryption, e.g. using Verificatum [36]. If an element $h_i^{r_i} = 1$ an error is output which only happens with negligible probability if at least one Tally Teller is honest.

2.5 Notification and Verification

After a suitable delay we move to the notification phase: g^{r_i} dual key is sent[7] to voter i over a private channel at a randomly chosen time during the notification period. The voter raises this to her secret trapdoor key x_i and finds the match among the $h_j^{r_j}$ terms, so identifying her vote in the tally column.

2.6 Coercion Mitigation

Suppose a coercer instructs voter i to submit the vote v^*.[8] Voter i identifies a row in the tally that contains the pair $(v_k,\ h_k^{r_k})$ s.t. $v_k = v^*$. Using her trapdoor key x_i, she computes the *fake* dual key that when raised to x_i will match this row $(h_k^{r_k})^{x_i^{-1}}$.

As with *Selene*, care has to be taken in designing the notification channel to avoid a coercer being able to observe the notification of the *real* dual key. In contexts in which we anticipate extreme coercion, where for example the coercer demands access to the channel, coerced voters could be provided with means to request a fake dual key be sent over the channel instead of the real one.

We note that the vote casting method presented here is not fully coercion-resistant, but is software-dependent receipt-free, i.e. like Helios [1] would rely on the vote-casting device or app not leaking the randomness used in the vote encryption. However, *Hyperion* can be combined with different forms of vote-casting to achieve better receipt-freeness e.g. using the BeleniosRF construction, [9]. Also, better coercion-resistance can be achieved providing protection against a coercer even trying to vote on behalf of the coerced voter, e.g. by holding the signing key, for example using JCJ style credentials [24], see [22] but at the cost of an interactive vote verification.

are multiplied together to obtain $\{h_i^{r_i}\}_{pk_{\mathsf{EA}}}$ where $\sum_j r_{i,j} = r_i$. Each Teller then keeps $g^{r_{i,j}}$.

[7] With multiple Tally Tellers, TT_j can send $g^{r_{i,j}}$ to the voter or they can be collected and sent to the voter under encryption of h_i.

[8] This presumes that some votes v^* are cast by other voters otherwise it will, in any case, be evident that voter i did not cast v^*. For techniques to deal with the situation of unpopular candidates, see [23, 31].

2.7 Dispute Resolution

It is possible when verifying that a voter either fails to find the matching term or finds it but the associated vote does not match the vote they cast. The voter should notify this to the appropriate authority for the matter to be investigated.

Possible causes:

1. The voter's ballot was not correctly posted to the BB.
2. The voter's device did not encrypt the correct vote.
3. The voter's ballot was not correctly processed during the mixing and tallying.
4. The g^{r_i} term was corrupted.

Regarding the first, we should remark that voters should be encouraged to check the presence of their ballot on the BB before tallying starts, as with other E2E V schemes. Early detection of such problems makes them easier to resolve, but *Hyperion* (and indeed *Selene*) is less reliant than conventional E2E verifiable schemes on such checks being performed diligently.

It is of course possible that a voter claims falsely to have found a problem in which case we hit dispute resolution problems: it is not clear whether the problem is with the system, the voter's device or the voter, either lying or misremembering. We will discuss mechanisms to resolve disputes in Section 7.4.

3 Hyperion Instantiation

We will here present the algorithms, $\mathsf{EASetup}$, Setup, $\mathsf{ValidCred}$, Vote, $\mathsf{ValidBallot}$, Tally, $\mathsf{GetSecret}$, $\mathsf{Publish}$, Verify, $\mathsf{VerifyVote}$ and $\mathsf{VerifyBallot}$, which we will use in security games, and how they are instantiated for Hyperion.

$\mathsf{EASetup}$ sets up the secure cyclic DH group (of prime order) (G, g) and creates the threshold public and secret keys $(pk_{\mathsf{EA}}, sk_{\mathsf{EA}})$. Setup uses the EA keys to generate for each id a "unique" signing key pair (sk, pk) along with the proof of well-formedness; the voter also picks a random exponent x_i and computes $h_i := g^{x_i}$ along with Π_{x_i} the proof of knowledge of x_i bound non-malleably to the voter id_i. The previous algorithms should be randomized when generating the keys. $\mathsf{ValidCred}$ outputs $\top$ if Π_{x_i} is valid, and $\perp$ otherwise. Vote extracts h_i from pk and Π_{x_i}, encrypts the vote v with ElGamal encryption scheme using pk_{EA}, generates the proof of well-formedness and plaintext knowledge Π_v which is bound to the voter id_i, and signs these elements using sk. It finally outputs the signed elements along with the signature as a ballot blt and an empty state st. $\mathsf{ValidBallot}$ verifies the correctness of the signature using pk and the validity of Π_v and Π_x: if both verifications pass, then the function outputs $\top$, otherwise, it outputs $\perp$.

The Tally function has two main jobs, first computing the mix-net inputs while updating the BB, and second inserting some computed values to the decryption mix-nets and outputting the result along with the vote count. In its first functionality, Tally extracts h_i from BB, picks a random exponent r_i for each row i, computes $h_i^{r_i}$, g^{r_i}, internally stores g^{r_i}, then, it computes the encryption

$\{h_i^{r_i}\}_{pk_{\mathsf{EA}}}$ with the proof of knowledge of r_i and correct encryption Π_i^{TT} and sends $(\{h_i^{r_i}\}_{pk_{\mathsf{EA}}}, \Pi_i^{\mathsf{TT}})$ to BB. In the second functionality, the pair $(\{h_i^{r_i}\}_{pk_{\mathsf{EA}}}, \{v_i\}_{pk_{\mathsf{EA}}})$ are put through the mix-net and decryption to output $(h_{\sigma_i}^{r_{\sigma_i}}, v_{\sigma_i}, \Pi_{\text{mix}}, \Pi_{\text{dec}})$ as the final tally.

Further, $\mathsf{GetSecret}$ outputs g^{r_i}. $\mathsf{VerifyVote}$ extracts $h_i^{r_i}$ from the bulletin board, raises the input g^{r_i} to the secret key input x_i and outputs the equality check. $\mathsf{Publish}$ outputs the verifiable mix and the decryption of $(v_i, h_i^{r_i})$ along with the BB.

Finally, Verify will verify all public evidence on BB, $\mathsf{VerifyBallot}$ will generally verify that a ballot appears correctly on BB for a given voter, however in Hyperion this can often be relaxed to check that some valid ballot has appeared for the given voter *id* which we denote $\mathsf{VerifyVoted}$. In the privacy game we ignore this and it will always output $\top$. By ρ we denote the election result function.

In our privacy games we choose ρ to compute the array of votes created by extracting, from each element in the input array, the last submitted vote in the concatenated sequence.

4 Ballot Privacy

In this section, we introduce the game-based definition of ballot privacy Ballot-Priv. In this definition, we take into account voters having secret credentials sk_i and capture privacy leaks from verification, especially plaintext verification (as in *Hyperion*, *Selene* and the Estonian e-voting system).

Even though ballot privacy is a fundamental property in secure voting, it is hard to come up with a generic definition which supports standard proof techniques and encompasses large classes of voting systems and tally functions. A good overview of game-based definitions can be found in [5], which also concludes with a privacy definition (BPRIV) for general tally functions. BPRIV is however not directly applicable to the current context of post-tally verification. Instead we take advantage of *Hyperion* having a simple tally function, namely revealing all plaintext votes. This means we can use a much earlier definition as starting point, namely Benaloh's definition [4], which was rewritten in modern game-based notation in [5].

Figure 1 is a rework of Benaloh's definition, with inspiration from [12] and especially [16], allowing voters to hold secret key material and adding a verification phase. This verification phase has the potential to introduce privacy attacks, if the adversary has access to whether the verification was successful or not. This is a realistic real-world scenario even without compromised parties since voters might share a failed verification with others, perhaps even on social media.

For transparency and to detect wide-spread attacks such behaviour should even be endorsed, and hence better not invalidate privacy.

We also want to model robust voting systems in which the voting process proceeds even if individual verification fails (as would probably happen in larger elections). In the case where a covert attacker model against privacy is preferred, the definition can easily be updated to only let an adversary win the game if verification attempts are successful, which interestingly can still lead to an information leak of the vote. In the Ballot-Priv definition, Figure 1, we assume a trusted BB and secure channels between the voters and BB, meaning that an honest ballot will arrive unchanged to BB. We also assume an initial setup giving each voter a unique identity id_i.

```
Game Ballot-Priv$^b(\mathcal{A})$
00 $\mathrm{SK}, \mathrm{PK}, \mathrm{ST}, \mathrm{V}^0, \mathrm{V}^1 \leftarrow [\,]_\perp$
01 $\mathrm{HV}, \mathrm{DV} \leftarrow \{\}$
02 $(pk_{\mathsf{EA}}, sk_{\mathsf{EA}}) \leftarrow \mathsf{EASetup}()$
03 $st_{\mathcal{A}} \leftarrow \mathcal{A}_1(pk_{\mathsf{EA}}, \mathrm{PK})$
04 $b' \leftarrow \mathcal{A}_2(st_{\mathcal{A}})$
05 Stop with $b = b'$

Oracle$_1$ HonestSetup$(id)$
06 $(sk, pk) \leftarrow \mathsf{Setup}(id, sk_{\mathsf{EA}}, pk_{\mathsf{EA}})$
07 Promise $\mathsf{ValidCred}(id, pk, pk_{\mathsf{EA}})$
08 $\mathrm{SK}[id] \leftarrow sk$; $\mathrm{PK}[id] \leftarrow pk$
09 $\mathrm{HV} \overset{\cup}{\leftarrow} \{id\}$; $\mathrm{DV} \overset{\setminus}{\leftarrow} \{id\}$

Oracle$_1$ DishonestSetup$(id, pk)$
10 Require $id \notin \mathrm{HV}$
11 Require $\mathsf{ValidCred}(id, pk, pk_{\mathsf{EA}})$
12 $\mathrm{PK}[id] \leftarrow pk$
13 $\mathrm{DV} \overset{\cup}{\leftarrow} \{id\}$

Oracle$_2$ LoR$(id, v^0, v^1)$
14 Require $id \in \mathrm{HV}$
15 $sk \leftarrow \mathrm{SK}[id]$; $pk \leftarrow \mathrm{PK}[id]$
16 $(blt, st) \leftarrow \mathsf{Vote}(pk_{\mathsf{EA}}, id, sk, pk, v^b)$
17 Promise $\mathsf{ValidBallot}(\mathrm{BB}, blt)$
18 $\mathrm{V}^0[id] \overset{\shortparallel}{\leftarrow} v^0$; $\mathrm{V}^1[id] \overset{\shortparallel}{\leftarrow} v^1$
19 $\mathrm{ST}[id] \overset{\shortparallel}{\leftarrow} st$; $\mathrm{BB}[id] \overset{\shortparallel}{\leftarrow} blt$

Oracle$_2$ Tally()
20 Require $\rho(\mathrm{V}^0) = \rho(\mathrm{V}^1)$
21 Return $\mathsf{Tally}(\mathrm{BB}, sk_{\mathsf{EA}}, pk_{\mathsf{EA}}, \mathrm{PK})$

Oracle$_2$ VerifyVote$(id)$
22 Require $id \in \mathrm{HV} \cup \mathrm{DV}$
23 $pk \leftarrow \mathrm{PK}[id]$
24 $s \leftarrow \mathsf{GetSecret}(id, pk, sk_{\mathsf{EA}}, pk_{\mathsf{EA}}, \mathrm{BB})$
25 if $id \in \mathrm{DV}$ then Return $s$
26 $(r, \pi) \leftarrow \mathrm{Tally}()$ // scheme dependent
27 $sk \leftarrow \mathrm{SK}[id]$; $st \leftarrow \mathrm{ST}[id]$
28 Return $\mathsf{VerifyVote}(id, sk, st, s, \mathrm{BB}, r, \pi)$

Oracle$_2$ Board()
29 $\mathrm{BB}' \leftarrow \mathsf{Publish}(\mathrm{BB})$
30 Return $\mathrm{BB}'$

Oracle$_2$ Cast$(id, blt)$
31 Require $id \in \mathrm{HV} \cup \mathrm{DV}$
32 Require $\mathsf{ValidBallot}(\mathrm{BB}, blt)$
33 $\mathrm{BB}[id] \overset{\shortparallel}{\leftarrow} blt$

Oracle$_2$ VerifyBlt$(id)$
34 Require $id \in \mathrm{HV}$
35 $sk \leftarrow \mathrm{SK}[id]$; $pk \leftarrow \mathrm{PK}[id]$
36 $st \leftarrow \mathrm{ST}[id]$
37 Return $\mathsf{VerifyBallot}(id, st, sk, pk, \mathrm{BB})$
```

Fig. 1. The game-based security definition of Ballot Privacy. The adversary wins if it distinguishes the left world from the right one, by guessing bit b. In line 16, the Left-or-Right (LoR) oracle either inputs the left vote v^0 or the right one v^1 based on the bit b. We divide our adversary into $\mathcal{A}_1$ and $\mathcal{A}_2$ in 03, 04 and assume that they **respectively** have access to the oracles sub-indexed by 1 and 2.

First, in line 02, the EA prepares the master keys that are used to generate the voters credentials (lines 06-09), to verify a voter's credentials (lines 07, 11), to allow the voting process (line 16), to tally the BB (line 21) and to allow the generation of the voters' verification secrets (line 24).

Lines 10-13 give the adversary the possibility to dynamically register dishonest credentials for some voters: lines 09 and 10 prevent the adversary from registering a set of credentials as both honest and dishonest at the the same time e.g. by calling HonestSetup on a specific id and then DishonestSetup on the same id, causing the voter to be honest and dishonest at the same time. Notice that, for a voter $id \in \mathrm{ID}$, checks of honesty occur in lines 14, 25, 34.

Line 17 ensures that the ballots created in the left or right voting oracle are well-formed. Notice that in lines 18-19, the elements are concatenated to the history: this provides more generality then just overwriting the previous value using the $\leftarrow$ operator to accommodate elections that take into consideration the whole history of vote submissions.

In line 20, the ρ function guarantees that both V^0 and V^1 have the same count: this prevents $\mathcal{A}$ from trivially winning by querying $\mathrm{LoR}(v^0, v^1)$, $\mathrm{LoR}(v^0, v^0)$ and then querying Tally(). Additionally, $\mathcal{A}$ is capable of querying the ballot casting oracle (31-33), the board publishing oracle (29-30), the ballot verification oracle (34-37) and the vote verification oracle (22-28). The ballot verification oracle and the vote verification oracle, both, can provide the adversary with extra information about the honest and dishonest voters (secret s and/or verification output).
We define the advantage of the adversary $\mathsf{Adv}_{\mathcal{A}}^{\text{Ballot-Priv}} := |\Pr[\text{Ballot-Priv}^0(\mathcal{A})] - \Pr[\text{Ballot-Priv}^1(\mathcal{A})]|$. The generality of this type of Benaloh definition is limited to certain types of result functions, see [11], which however is fulfilled for Hyperion where we output all votes after mixing. Especially, we notice that, a necessary condition on ρ is that it should fulfill the following relation $\rho(\mathsf{V}_0) = \rho(\mathsf{V}_1) \implies \rho(\mathsf{V}_0 \shortparallel \mathsf{V}') = \rho(\mathsf{V}_1 \shortparallel \mathsf{V}')$. Probably the definition could be extended to general cases, with an assumption of extraction properties of the ballots.

Theorem 1. *For all $\mathcal{A}$ playing* Ballot-Priv *(Fig. 1 instantiated with Hyperion, there exists adversaries $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$, $\mathcal{F}$ such that the following relation holds:*

$$\mathsf{Adv}_{\mathcal{A}}^{\text{Ballot-Priv}} \leq \mathsf{Adv}_{\mathcal{B}}^{\text{ZK}} + \mathsf{Adv}_{\mathcal{C}}^{\text{EUF-CMA}} + \mathsf{Adv}_{\mathcal{D}}^{\text{Mix}} + \mathsf{Adv}_{\mathcal{E}}^{\text{ZK}'} + \mathsf{Adv}_{\mathcal{F}}^{\textsf{poly-IND-1-CCA}}$$

Further, in the long version [14] we also prove that our scheme satisfies du-mb-BPRIV against a malicious board from [12, 16]:

Theorem 2. *For all $\mathcal{A}$ playing* du-mb-BPRIV *[16] instantiated with Hyperion, there exists adversaries $\mathcal{B}$, $\mathcal{D}$, $\mathcal{D}'$, $\mathcal{E}$, $\mathcal{F}$ such that the following relation holds:*

$$\mathsf{Adv}_{\mathcal{A}}^{\text{du-mb-BPRIV}} \leq \mathsf{Adv}_{\mathcal{B}}^{\text{ZK}} + \mathsf{Adv}_{\mathcal{D}}^{\text{Mix}} + \mathsf{Adv}_{\mathcal{D}'}^{\text{Mix}} + \mathsf{Adv}_{\mathcal{E}}^{\text{ZK}'} + \mathsf{Adv}_{\mathcal{F}}^{\textsf{poly-IND-1-CCA}}$$

The main difference for the bounding is that du-mb-BPRIV does not capture attacks for the verification success seen as a side-channel to the adversary, and hence the ballot signatures are not necessary.

4.1 Proof of Ballot Privacy

We now prove that our scheme meets Ballot-Priv property. In order to do so, we instantiate the algorithms as described in section 3. We use game hopping technique to bound the adversary advantage. Since oracles can be called multiple times by the adversary, we will suppress pre-factors in the advantage bounds. We note by G_0 the instantiated Ballot-Priv game: $\mathsf{Adv}_{\mathcal{A}}^{\text{Ballot-Priv}} = \mathsf{Adv}_{\mathcal{A}}^{G_0}$.

In the first game hop, we remove line 07 in G_1. In fact under the assumption stated in Section 2.2 we have that line 07 will always pass, and thus $\mathsf{Adv}_{\mathcal{A}}^{G_0} = \mathsf{Adv}_{\mathcal{A}}^{G_1}$.
In G_2, by the zero knowledge property, the proofs Π_x and Π_{r_i} are simulated for honest voters. This is possible since these proofs are created by the challenger. Now, the adversary cannot extract any information from the simulated proofs

and $\mathsf{Adv}_{\mathcal{A}}^{G_1} \leq \mathsf{Adv}_{\mathcal{A}}^{G_2} + \mathsf{Adv}_{\mathcal{B}}^{\mathrm{ZK}}$ ($\mathsf{Adv}_{\mathcal{B}}^{\mathrm{ZK}}$ is the advantage of $\mathcal{B}$ to distinguish simulation from real proofs).
In game G_3, we modify line 31 to require $id \in \mathrm{DV}$ only. In this case, because of the requirement in 32, when casting a ballot for an honest voter, the adversary has to be capable of forging a valid signature for the honest voter. If the signature scheme is existentially unforgeable, we have $\mathsf{Adv}_{\mathcal{A}}^{G_2} \leq \mathsf{Adv}_{\mathcal{A}}^{G_3} + \mathsf{Adv}_{\mathcal{C}}^{\mathrm{EUF-CMA}}$.
In the fourth game G_4, we replace the verification step on line 28 to always output success. Because our scheme is correct, and since the adversary is not capable of submitting ballots on behalf of honest voters, then we have $\mathsf{Adv}_{\mathcal{A}}^{G_3} = \mathsf{Adv}_{\mathcal{A}}^{G_4}$.
In game G_5, we modify line 21 in the tally oracle: rather than picking r_i at random for each voter and computing $h_i^{r_i}$ then encrypting the computed value, we sample a uniformly random group element g_i and then encrypt it. Since we are working in a cyclic group of prime order, then the distributions of g_i and $h_i^{r_i}$ are exactly the same for all registered voters i, thus $\mathsf{Adv}_{\mathcal{A}}^{G_4} = \mathsf{Adv}_{\mathcal{A}}^{G_5}$. [9]
In the next game G_6, we modify again line 21, by replace the secure mix-net by its ideal functionality. We thus have $\mathsf{Adv}_{\mathcal{A}}^{G_5} \leq \mathsf{Adv}_{\mathcal{A}}^{G_6} + \mathsf{Adv}_{\mathcal{D}}^{\mathrm{Mix}}$.[10]
In G_7, analogously to G_2, we simulate the decryption proofs Π_{dec} for all the ciphertexts output by the mix-net. Further, for the honest voters, the decryption values for the plaintext votes are taken from the calls to the LoR oracle. Due to the correctness of the encryption scheme, the adversary's advantage is $\mathsf{Adv}_{\mathcal{A}}^{G_6} \leq \mathsf{Adv}_{\mathcal{A}}^{G_7} + \mathsf{Adv}_{\mathcal{E}}^{\mathrm{ZK}'}$.
In the final game hop, we require that the mix-nets in G_8 output the honest votes (taken from the LoR oracle) concatenated with the decryption of the dishonest votes. The views in the left world and the right world should be the same, thus we require the decryption mix to output $\rho(\mathrm{V}^b)$ concatenated with the dishonest votes. We have that $\mathsf{Adv}_{\mathcal{A}}^{G_8} = \mathsf{Adv}_{\mathcal{A}}^{G_7}$.
Finally, we argue that the advantage of the final game is exactly $\mathsf{Adv}_{\mathcal{F}}^{\mathsf{poly\text{-}IND\text{-}1\text{-}CCA}}$ in poly-IND-1-CCA since we can remove all simulated proofs. The label is the id of the voters. We assume that the the encryption scheme with the non-malleable proofs of plaintext knowledge including the id satisfy poly-IND-1-CCA security. We conclude the following:

$$\mathsf{Adv}_{\mathcal{A}}^{\text{Ballot-Priv}} \leq \mathsf{Adv}_{\mathcal{B}}^{\mathrm{ZK}} + \mathsf{Adv}_{\mathcal{C}}^{\mathrm{EUF-CMA}} + \mathsf{Adv}_{\mathcal{D}}^{\mathrm{Mix}} + \mathsf{Adv}_{\mathcal{E}}^{\mathrm{ZK}'} + \mathsf{Adv}_{\mathcal{F}}^{\mathsf{poly\text{-}IND\text{-}1\text{-}CCA}}$$

5 Integrity

5.1 Correctness

We first note that the scheme satisfies correctness in the sense that if the voting protocol is run honestly, the tally will give the correct result on the intended votes

[9] We can allow the adversary to have the dual key g^{r_i} for *all voters* in line 28 and still prove security of the scheme under the DDH assumptions: $h_i := g^{x_i}$ adversary cannot distinguish $(g^{x_i}, g^{r_i}, g^{x_i \cdot r_i})$ and (g^{x_i}, g^{r_i}, g_i).

[10] Alternatively, we could model the mix-net as a re-encryption mix with a NIZKP, and use IND-CPA of the encryptions of g_i to ignore these ciphertexts.

and all voters will verify correctly. This assumes correctness of the underlying zero-knowledge proofs, signatures, mix-net and correctness of the encryption scheme. This could be relaxed to non-perfect correctness if needed.

5.2 Verifiability

For an overview of verifiability definitions see [11], especially we will use the definition in [10]. This is because the specific voting-casting construction that we instantiate *Hyperion* with here is close to Helios-C (i.e. Helios with signatures) presented and proven verifiable in [10].

The election schemes in [10] are defined via algorithms $\mathsf{Setup} = \mathsf{EASetup}$, $\mathsf{Credential} = \mathsf{Setup}$, Vote, $\mathsf{VerifyVote}_{\mathsf{CGGI}}$, $\mathsf{Validate} = \mathsf{ValidBallot}$, Tally, Verify, where we have indicated by which algorithms they correspond to in our scheme, see Sec. 3. Verify will simply check all proofs on BB. The main difference is in $\mathsf{VerifyVote}$: in Helios-like constructions this corresponds to checking that your actual ballot *blt* has appeared on BB. Here, it corresponds to performing the *Hyperion* verification and will involve getting the dual key from the EA. Since we will define security against a malicious BB, we further need that the voters check that a valid vote was registered under their *id*, but without having to check which specific cryptographic ballot is recorded (for improved usability). We denote this $\mathsf{VerifyVoted}$. As in [10] we consider schemes without vote updates for simplicity.

In [10] combined individual and universal verifiability is defined against a malicious BB. This means the board is completely malicious up until the Tally, where it will be output by the adversary, and there will be a unified view of BB. This also models that vote casting channels might not be secure. The definition is via a game $\mathrm{Exp}_{\mathcal{A}}^{verb}$ for which the adversary has negligible chance in creating a valid BB and tally where it is not true that 1) the vote count will contain the honest verifying voters' votes, 2) for the non-verifying honest voters their votes can maximally be deleted, and 3) there is maximally one vote per corrupted voter in the tally. To count this it is assumed that the result function ρ allows partial tally. The main assumption is that the Registration Authority is honest i.e. signing keys are setup honestly and not leaked and are existentially unforgeable, EUF-CMA. This will also hold for *Hyperion*.

Theorem 3. *Hyperion will satisfy Verifiability against a dishonest bulletin board [10] if the signing keys are not leaked, the signature scheme is EUF-CMA and the ballot verification is via* $\mathsf{VerifyVoted}$*, i.e. the voter only checks if a valid vote was cast.*

The proof follows as for Helios-C, however, we use mix-nets instead of homomorphic tally, which still ensures one-vote per voter due to the soundness of the mixes. Also, we can replace $\mathsf{VerifyVote}_{\mathsf{CGGI}}$ with $\mathsf{VerifyVoted}$ since the adversary cannot forge a signature.[11]

[11] Since we are in a single pass setting this is particularly simple. With vote updates more care needs to be taken. Either we need to assume an append only board or that a vote update number is included in the signature and remembered by the voter.

However, this did not take into account the actual *Hyperion* verification check which also allows a voter to verify if the vote intent was captured directly in the tally. We now extend the verifiability definition to fully incorporate this. The main point will be that an honest checking voter can rely on her vote being counted correctly if either her signing key or *Hyperion* secret key is not compromised. In particular, we get a resistance against malware if we have separate devices for vote casting (containing the signing key) and vote verification (containing the *Hyperion* key), and not both devices are corrupted.

We now introduce a verifiability definition against a malicious voting bulletin board in the presence of malware with separate vote casting and vote verification devices. We stress that in the definition it is only the vote casting part of the bulletin board which is determined by the adversary, the registered public keys cannot be altered for honest devices.[12]

The security is defined via the experiment Verif-MBM in Fig. 2, where the advantage of the adversary is $\mathsf{Adv}_{\mathcal{A}}^{\text{Verif-MBM}} = \Pr[\text{Verif-MBM}(\mathcal{A}) = 1]$ The malicious bulletin board and corrupted authorities are modeled by the adversary outputting the bulletin board as well as the tally result and proofs in line 05. $\mathcal{H}_c$ and $\mathcal{D}_c$ respectively denote the voter IDs with honest and corrupted vote casting devices which have registered public verification keys and hence constitute the eligible voters. Correspondingly, $\mathcal{H}_v$ and $\mathcal{D}_v$ are the voter IDs with honest and corrupted vote verification devices. We split the algorithm Setup into a part for the signing key and for the verification key denoted respectively Setup_c and Setup_v, and we do the same split for the $\mathsf{ValidCred}$ algorithm.

$\mathcal{V}_i$ denotes the set of voters intending to vote and V_i captures their intended vote, with $\mathbb{V}$ the allowed vote space. $\mathcal{V}_{\text{Chkd}}$ denotes the voters who make successful verification checks. Failing checks would lead to complaints and the adversary loses the game. As in [10], the set of voters who are going to check can be input to the game to capture that not all voters verify. For simplicity we assume only voters with a vote intention will try to verify. Those who check will try to do both $\mathsf{VerifyVote}$ and $\mathsf{VerifyBallot}$ which we here simplify to $\mathsf{VerifyVoted}$.[13]

We assume $\mathsf{VerifyVoted}$ is unaffected by malware since it just requires access to BB (and could be delegated). For $\mathsf{VerifyVote}$, if the voter's verification device is corrupted, or the voter is not registered for verification, the check will be assumed successful.

For uncorrupted devices, since EA is corrupted, the adversary can choose which dual key the voter receives. We do not need a corrupted category since if both devices are corrupted and the voter does not perform verifications then the voter is completely controlled by the adversary. A stronger version can let the adversary choose the election setup, but here it is honestly created.

[12] In practice this can be secured in a full malicious board setting by forwarding the public keys to proxies at registration time, who will check later that these appear correctly.

[13] The definition can use $\mathsf{VerifyBallot}$ by defining which part of the voter state the adversary can control.

```
**Game** Verif-MBM($\mathcal{A}$)
00 $\mathrm{SK}_c, \mathrm{PK}_c \leftarrow [\,]_\perp$
01 $\mathrm{SK}_v, \mathrm{PK}_v, \mathrm{ST}, \mathrm{V} \leftarrow [\,]_\perp$
02 $\mathcal{H}_c, \mathcal{D}_c, \mathcal{H}_v, \mathcal{D}_v, \mathcal{V}_i \leftarrow \{\}$
03 $(pk_{\mathsf{EA}}, sk_{\mathsf{EA}}) \leftarrow \mathsf{EASetup}()$
04 $st_{\mathcal{A}} \leftarrow \mathcal{A}_1(pk_{\mathsf{EA}}, \mathrm{PK})$
05 $(\mathrm{BB}, r, \pi, st_{\mathcal{A}}) \leftarrow \mathcal{A}_2(st_{\mathcal{A}})$
06 **Require** $\mathsf{Verify}(\mathrm{BB}, r, \pi)$
07 for $id \in \mathcal{V}_{\mathrm{Chkd}} \cap \mathcal{V}_i$
08   **Require** $\mathsf{VerifyVoted}(id, \mathrm{BB})$
09   if $id \in \mathcal{H}_v$ then:
10     $sk \leftarrow \mathrm{SK}_v[id], s \leftarrow \mathcal{A}_3(st_{\mathcal{A}})$
11     $v \leftarrow \mathrm{V}[id]$
12     **Require** $\mathsf{VerifyVote}(id, sk, v, s, \mathrm{BB}, r, \pi)$
13 **Require** $r = \top$
14 $\mathcal{M} \leftarrow \mathcal{V}_{\mathrm{Chkd}} \cap (\mathcal{H}_c \cup \mathcal{H}_v)$
15 $a \leftarrow |\mathcal{V}_{\mathrm{Chkd}} \cap \mathcal{D}_v|$
16 $b \leftarrow |\mathcal{D}_c \setminus (\mathcal{V}_{\mathrm{Chkd}} \cap \mathcal{H}_v)|$
17 **Require**
18   $\exists n \in \{a, \ldots, b\}$
19   $\exists v_1, \ldots, v_n \in \mathbb{V}$ // vote set
20   $\exists \mathcal{S} \subseteq (\mathcal{V}_i \setminus \mathcal{V}_{\mathrm{Chkd}}) \cap \mathcal{H}_c$
21   s.t. r= $\rho([\mathrm{V}[j]]_{j \in \mathcal{S}}) \star \rho([v_j]_{j=1}^{n}) \star \rho([\mathrm{V}[j]]_{j \in \mathcal{M}})$
22 **Stop with** $\top$

**Oracle₁** HonestCastSetup($id$)
23 **Require** $id \notin \mathcal{D}_c$
24 $(sk, pk) \leftarrow \mathsf{Setup}_c(id, pk_{\mathsf{EA}})$
25 **Promise** $\mathsf{ValidCred}_c(id, pk, pk_{\mathsf{EA}})$
26 $\mathrm{SK}_c[id] \leftarrow sk;\ \mathrm{PK}_c[id] \leftarrow pk$
27 $\mathcal{H}_c \overset{\cup}{\leftarrow} \{id\}$

**Oracle₁** DishonestCastSetup($id, pk$)
28 **Require** $id \notin \mathcal{H}_c$
29 **Require** $\mathsf{ValidCred}_c(id, pk, pk_{\mathsf{EA}})$
30 $\mathrm{PK}_c[id] \leftarrow pk$
31 $\mathcal{D}_c \overset{\cup}{\leftarrow} \{id\}$

**Oracle₁** HonestVerSetup($id$)
32 **Require** $id \notin \mathcal{D}_v$
33 $(sk, pk) \leftarrow \mathsf{Setup}_v(id, pk_{\mathsf{EA}})$
34 **Promise** $\mathsf{ValidCred}_v(id, pk, pk_{\mathsf{EA}})$
35 $\mathrm{SK}_v[id] \leftarrow sk;\ \mathrm{PK}_v[id] \leftarrow pk$
36 $\mathcal{H}_v \overset{\cup}{\leftarrow} \{id\}$

**Oracle₁** DishonestVerSetup($id, pk$)
37 **Require** $id \notin \mathcal{H}_v$
38 **Require** $\mathsf{ValidCred}_v(id, pk, pk_{\mathsf{EA}})$
39 $\mathrm{PK}_v[id] \leftarrow pk$
40 $\mathcal{D}_v \overset{\cup}{\leftarrow} \{id\}$

**Oracle₂** Vote($id, v$)
41 **Require** $id \in \mathcal{H}_c \cup \mathcal{D}_c$
42 $\mathcal{V}_i \overset{\cup}{\leftarrow} \{id\};\ \mathrm{V}[id] \leftarrow v$
43 if $id \in \mathcal{H}_c$ then
44   $sk \leftarrow \mathrm{SK}[id];\ pk \leftarrow \mathrm{PK}[id]$
45   $(blt, st) \leftarrow \mathsf{Vote}(pk_{\mathsf{EA}}, id, sk, pk, v)$
46   $\mathrm{ST}[id] \leftarrow st$
47   Return $blt$
```

Fig. 2. The game-based definition of verifiability against a malicious voting board and malware. The indices on the oracles denote which adversary can use them. We use sub-index v to denote verify, c for cast and i for intended.

Note that this definition does not capture the probability of detecting the presence of malware, but the guarantee given to a successfully verifying voter and what votes the adversary can choose for the rest. In particular, the adversary will win if he can output a valid BB and tally and manages to either 1) change the vote of a voter who has at least one honest device and who verified (line 14), or 2) for voters with honest vote casting devices, he manages to stuff votes or change a cast vote in another way than simply deleting it (line 20), or 3) for the remaining eligible voters can cast more than one vote per voter (line 18). The lower bound on votes in the last category comes from the voters with both devices being corrupted, and who are successfully verifying, will know that some vote arrived on their behalf, but not which plaintext vote it contains. Finally, in line 21 the $\star$ denotes the combination of partial tallies in the result function ρ.

The verifiability of *Hyperion* relies on the computational 1-Diffie-Hellman Inversion Problem (1-DHI) [27] for a cyclic prime order group of order q and generator (G, g).

Definition 1 (Computational 1-DHI). *Given* $g^x \in G$ *with* $x \twoheadleftarrow \mathbb{Z}_q$ *compute* $g^{1/x}$. *Under the* 1-DHI *assumption the advantage* $\mathsf{Adv}_{\mathcal{A}}^{1-DHI} = \Pr[x \twoheadleftarrow \mathbb{Z}_q : g^{1/x} = \mathcal{A}(g^x)]$ *is negligible for all PPT algorithms.*

If we use ballot verification $\mathsf{VerifyBallot}$ via $\mathsf{VerifyVoted}$, i.e. the voter only checks if a valid vote was cast, then we have the following theorem

Theorem 4. *With EUF-CMA signatures, sound mix-nets, encryption correctness, simulation-sound extractability [19, 6] for the proofs of knowledge and under the 1-DHI assumption, the advantage in verifiability against a malicious* BB *and malware,* $\mathsf{Adv}_{\mathcal{A}}^{\mathrm{Verif\text{-}MBM}}$ *is negligible.*

The proof can be found in App. B.

6 Conclusion

We present a new end-to-end verifiable scheme, inspired by the *Selene* tracker based scheme, that provides a similar, highly transparent, intuitive way for voters to verify their vote: by identifying their vote in cleartext in the tally. Our new construction however allows us to achieve this without the need for trackers and allows us to neatly avoid the tracker collision problem that undermined the *Selene* scheme. The collision threat however could re-emerge as collision of commitments rather than trackers. This prompts and enables a further innovation, described in the full version of this paper: the idea of individual voter views, that entirely avoids the collision threat of *Selene* and should afford voters a greater sense of privacy. Voters should feel more comfortable with *Hyperion* as it does not involve the public posting of all the tracker numbers paired with the votes.

While we do not advocate the use of *Hyperion* for high-stakes elections, we do believe that it is well suited to many less critical contexts. The transparency of the verification and the underlying simplicity of the constructions should be appealing to many stakeholders: the voters, the election officials, the candidates etc. The individual views version introduces some additional computation and complexity, but is efficient for small elections, and in any case could be done on demand when a voter seeks to verify their vote.

We have proven that the system satisfies ballot privacy and verifiability, the latter even under partial malware corruption of the voters' vote casting and verification devices. In the full version, [14], we sketch how the *Hyperion* scheme can be made everlasting private, see lso [26], or post-quantum secure. We also outline some possible variants of the core scheme, including the individual *BB* views, the re-introduction of trackers and the use of return or confirmation codes to address dispute resolution.

Future work will include full analysis of the current scheme and detailing the variants and formally proving them. We will also perform focus groups and user trials to gauge user response and preferences amongst the variants and w.r.t. to *Selene*.

Acknowledgments. This paper acknowledges the Luxembourg National Research Fund (FNR) CORE project EquiVox (C19/IS/13643617/EquiVox/Ryan) and the CORE project (C21/IS/16221219/ ImPAKT).

References

1. Adida, B., De Marneffe, O., Pereira, O., Quisquater, J.J., et al.: Electing a university president using open-audit voting: Analysis of real-world use of Helios. EVT/WOTE **9**(10) (2009)
2. Alsadi, M., Schneider, S.: Verify my vote: voter experience. E-Vote-ID 2020 p. 280 (2020)
3. Arnaud, M., Cortier, V., Wiedling, C.: Analysis of an electronic boardroom voting system. In: International Conference on E-Voting and Identity. pp. 109–126. Springer (2013)
4. Benaloh, J.D.C.: Verifiable secret-ballot elections. Ph.D. thesis, Yale University (1987)
5. Bernhard, D., Cortier, V., Galindo, D., Pereira, O., Warinschi, B.: Sok: A comprehensive analysis of game-based ballot privacy definitions. In: 2015 IEEE Symposium on Security and Privacy. pp. 499–516. IEEE (2015)
6. Bernhard, D., Pereira, O., Warinschi, B.: How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 626–643. Springer (2012)
7. Bruni, A., Drewsen, E., Schürmann, C.: Towards a mechanized proof of Selene receipt-freeness and vote-privacy. In: International Joint Conference on Electronic Voting. pp. 110–126. Springer (2017)
8. Camenisch, J.: Group signature schemes and payment systems based on the discrete logarithm problem. Ph.D. thesis, ETH Zurich (1998)
9. Chaidos, P., Cortier, V., Fuchsbauer, G., Galindo, D.: Beleniosrf: A non-interactive receipt-free electronic voting scheme. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1614–1625 (2016)
10. Cortier, V., Galindo, D., Glondu, S., Izabachene, M.: Election verifiability for Helios under weaker trust assumptions. In: European Symposium on Research in Computer Security. pp. 327–344. Springer (2014)
11. Cortier, V., Galindo, D., Küsters, R., Müller, J., Truderung, T.: Sok: Verifiability notions for e-voting protocols. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 779–798. IEEE (2016)
12. Cortier, V., Lallemand, J., Warinschi, B.: Fifty shades of ballot privacy: Privacy against a malicious board. In: 2020 IEEE 33rd Computer Security Foundations Symposium (CSF). pp. 17–32. IEEE (2020)
13. Cortier, V., Smyth, B.: Attacking and fixing Helios: An analysis of ballot secrecy. Journal of Computer Security **21**(1), 89–148 (2013)
14. Damodaran, A., Rastikian, S., Rønne, P.B., Ryan, P.Y.: Hyperion: transparent end-to-end verifiable voting with coercion mitigation. Cryptology ePrint Archive (2024)
15. Distler, V., Zollinger, M.L., Lallemand, C., Roenne, P.B., Ryan, P.Y., Koenig, V.: Security-visible, yet unseen? In: Proceedings of the 2019 CHI conference on human factors in computing systems. pp. 1–13 (2019)
16. Dragan, C.C., Dupressoir, F., Estaji, E., Gjøsteen, K., Haines, T., Ryan, P.Y.A., Rønne, P.B., Solberg, M.R.: Machine-checked proofs of privacy against malicious boards for Selene & co. In: 35th IEEE Computer Security Foundations Symposium, CSF 2022. pp. 335–347. IEEE (2022)
17. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on information theory **31**(4), 469–472 (1985)

18. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194. Springer (1986)
19. Groth, J.: Simulation-sound nizk proofs for a practical language and constant size group signatures. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 444–459. Springer (2006)
20. Haenni, R., Locher, P., Koenig, R., Dubuis, E.: Pseudo-code algorithms for verifiable re-encryption mix-nets. In: Financial Cryptography and Data Security. pp. 370–384. Springer (2017)
21. Hao, F., Ryan, P.Y.: Real-world electronic voting: Design, analysis and deployment. CRC Press (2016)
22. Iovino, V., Rial, A., Rønne, P.B., Ryan, P.Y.: Using Selene to verify your vote in JCJ. In: International Conference on Financial Cryptography and Data Security. pp. 385–403. Springer (2017)
23. Jamroga, W., Roenne, P.B., Ryan, P.Y., Stark, P.B.: Risk-limiting tallies. In: International Joint Conference on Electronic Voting. pp. 183–199. Springer (2019)
24. Juels, A., Catalano, D., Jakobsson, M.: Coercion-resistant electronic elections. In: Towards Trustworthy Elections, pp. 37–63. Springer (2010)
25. Küsters, R., Müller, J., Scapin, E., Truderung, T.: sElect: a lightweight verifiable remote voting system. In: Computer Security Foundations Symposium (CSF), 2016 IEEE 29th. pp. 341–354. IEEE (2016)
26. Mosaheb, R., Rønne, P.B., Ryan, P.Y., Sarfaraz, S.: Direct and transparent voter verification with everlasting receipt-freeness. In: International Joint Conference on Electronic Voting. pp. 124–140. Springer Nature Switzerland Cham (2024)
27. Pfitzmann, B.P., Sadeghi, A.R.: Anonymous fingerprinting with direct non-repudiation. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 401–414. Springer (2000)
28. Rivest, R.L.: On the notion of 'software independence' in voting systems. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **366**(1881), 3759–3767 (2008)
29. Rønne, P.B., Ryan, P.Y., Zollinger, M.L.: Electryo, in-person voting with transparent voter verifiability and eligibility verifiability. arXiv preprint arXiv:2105.14783 (2021)
30. Ryan, P.Y.A., Rønne, P.B., Iovino, V.: Selene: Voting with transparent verifiability and coercion-mitigation. In: International Conference on Financial Cryptography and Data Security. pp. 176–192. Springer (2016)
31. Ryan, P.Y., Roenne, P.B., Ostrev, D., Orche, F.E.E., Soroush, N., Stark, P.B.: Who was that masked voter? The tally won't tell! In: International Joint Conference on Electronic Voting. pp. 106–123. Springer (2021)
32. Sallal, M., Schneider, S., Casey, M., Dupressoir, F., Treharne, H., Dragan, C., Riley, L., Wright, P.: Augmenting an internet voting system with Selene verifiability using permissioned distributed ledger. In: 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS). pp. 1167–1168. IEEE (2020)
33. Schneier, B.: Applied cryptography: protocols, algorithms, and source code in C. john wiley & sons (2007)
34. Schnorr, C.P.: Efficient signature generation by smart cards. Journal of cryptology **4**(3), 161–174 (1991)
35. Terelius, B., Wikström, D.: Proofs of restricted shuffles. In: Progress in Cryptology – AFRICACRYPT 2010. pp. 100–113. Springer (2010)
36. Wikström, D.: User manual for the verificatum mix-net version 1.4. 0. Verificatum AB, Stockholm, Sweden (2013)

37. Zollinger, M.L., Distler, V., Roenne, P.B., Ryan, P.Y., Lallemand, C., Koenig, V.: User experience design for e-voting: How mental models align with security mechanisms. arXiv preprint arXiv:2105.14901 (2021)
38. Zollinger, M.L., Rønne, P.B., Ryan, P.Y.: Short paper: mechanized proofs of verifiability and privacy in a paper-based e-voting scheme. In: International Conference on Financial Cryptography and Data Security. pp. 310–318. Springer (2020)

A Implementation

We implement and instantiate *Hyperion* [14] in Python, using the GNU Multiple Precision Arithmetic library, and evaluate its performance on a server equipped with a 32 core AMD EPYC 7302P CPU clocked at 3 GHz and 256 gigabytes of RAM. The implementation is parameterized by the P-256 curve. An implementation of a Terelius-Wikström mixnet [35, 20] was employed for parallel shuffling in the tallying phase. Analogously, we also implement and instantiate *Selene*[15] in order to compare the performance of both schemes; these measurements are provided in App. E of [14]. Table 1 presents measurements collected during the

Phase	$N = 1000$	$N = 10000$	$N = 100000$
Setup	0.0004s	0.0005s	0.0005s
Voting	0.0085s	0.0090s	0.0160s
Tallying (Mix)	42.205s	1541.62s	6886.90s
Tallying (Decrypt)	5.4640s	33.889s	1092.32s
Coercion-Mitigation	0.0008s	0.0008s	0.0007s
Individual Views	14.091s	256.72s	3498.16s

Table 1. Execution times of each phase of the Hyperion scheme in seconds.

course of 3 trial runs of the *Hyperion* scheme for 1000 voters, 10000 voters, and 100000 voters. We comment that though this is a prototype implementation, the mixnet code has been parallelised to run faster on multi-core systems. The explicit ZKPs employed in our implementation can be found in [14].

B Proof of Verifiability, Theorem 4

We here give a short proof, the finite advantage bound can easily be inferred. The proof is done without reference to whether a CRS or RO setup is used. By line 06 and 13 in Fig. 2 we can assume that the adversary outputs a valid BB with a result and valid proof. Since the mix-net proofs validate, by the soundness of the mix-net proofs, decryption proofs and correctness of the encryption scheme,

[14] https://github.com/hyperion-voting/hyperion
[15] https://github.com/hyperion-voting/selene

we have that the resulting multiset of votes are equal to the plaintext inputs. For honestly cast ballots we have correctness and they will validate if added to BB. Also, honestly generated key will validate. All votes will be in the correct vote space, this can either be directly checked after decryption or derived from the soundness of the ballot proof of well-formedness.

For all voters in $\mathcal{H}_c$ we have valid signatures if they cast votes and by EUF-CMA the adversary cannot forge any signature. Hence for $\mathcal{H}_c$ no votes can be stuffed and cast votes can never be altered, only deleted. Thus for successfully checking voters with honest vote casting device, $\mathcal{V}_{\text{Chkd}} \cap \mathcal{H}_c$, all votes has to appear unaltered (remember $\mathcal{V}_{\text{Chkd}} \subseteq \mathcal{V}_i$ i.e. the checking voters are part of the voters intending to vote), this proves the $\mathcal{V}_{\text{Chkd}} \cap \mathcal{H}_c$ part of line 14. For the remaining cast votes from voters in $(\mathcal{V}_i \setminus \mathcal{V}_{\text{Chkd}}) \cap \mathcal{H}_c$ the adversary can choose which to delete, ensuring line 20.

We can now consider the voters with a malicious vote-casting device $\mathcal{D}_c$. If these voters are not checking, we have no guarantees. If they check and have a corrupted verification device, then there has to be a ballot for their *id* due to VerifyVoted, however there is no guarantee which vote it contains. This explains the lower bound on the number of maliciously created ballots in line 18.

Finally, we need to consider voters successfully verifying with an uncorrupted verification device. We want to show that they will be able to verify their plaintext vote, hence proving the $\mathcal{V}_{\text{Chkd}} \cap \mathcal{H}_v$ part of line 14 and the upper bound in line 18. We first simulate the ZKPoKs of x_i for the honestly registered Hyperion keys $h_i = g^{x_i}$.

We will give a proof by contradiction, i.e. we assume that a voter in $\mathcal{V}_{\text{Chkd}} \cap \mathcal{H}_v$ will get pointed to another vote than her intended vote by the Hyperion verification with some non-negligible advantage $\mathsf{Adv}_{\mathcal{A}}$. We will use this to create an adversary against computations 1-DHI. To this end, we take a 1-DHI challenge g^x and use this as the key for a random voter in $\mathcal{V}_{\text{Chkd}} \cap \mathcal{H}_v$ with a simulated proof. Since the key g^x is indistinguishable from random this voter will be targeted by the attack with probability at least $1/|\mathcal{V}_{\text{Chkd}} \cap \mathcal{H}_v|$.

For all the voters with corrupted casting devices, we now extract their secret keys x_i from the ZKPoKs using the simulation sound extractability (for the honest verification devices, we know their secret keys). Let α denote the dual key term sent by the adversary to the voter. By the soundness of the ZKPoK for the encryption of the elements $h_i^{r_i}$, the soundness of the mix-net and the correctness of the encryption, the output commitments are all of the form $h_i^{r_i}$.

We further extract all r_is from the ZKPoKs. If the voter gets pointed to another vote we have that $\alpha^x = h_i^{r_i} = g^{x_i r_i}$ for some i with $x_i \neq x$. We don't know which i this is, but we guess at random between the k choices. Hence we can compute $\alpha^{1/(x_i r_i)}$ which will be equal $g^{1/x}$ with a non-negligible probability $\mathsf{Adv}_{\mathcal{A}}/(|\mathcal{V}_{\text{Chkd}} \cap \mathcal{H}_v| \cdot k)$ breaking the computational 1-DHI assumption and concluding the proof.