Cryptographic Methods for Storing Ballots on a Voting Machine

John Bethencourt [*] Carnegie Mellon University
Dan Boneh [†] Stanford University
Brent Waters [‡] SRI International

Abstract

A direct recording electronic (DRE) voting machine must satisfy several requirements to ensure voter privacy and the integrity of the election. A recent proposal for a vote storage system due to Molnar et al. provides tamper-evidence properties while maintaining voter privacy by storing ballots on a programmable, read-only memory (PROM). We achieve the same properties and protect against additional threats of memory replacement through cryptographic techniques, without the use of special hardware. Our approach is based on a new cryptographic primitive called History-Hiding Append-Only Signatures.

1 Introduction

The deployment of electronic voting terminals introduces the problem of adequately storing votes in digital form. A vote storage system must store votes on the voting terminal during the election and possibly in archive form long after the end of the election. While there is some debate as to the precise requirements of a vote storage system, some deployed systems [12] have been shown to be clearly inadequate.

Molnar et al. [15] recently identified seven requirements that a vote storage system must satisfy. The four primary requirements are: (1) **durable** — the storage system must not lose stored votes if the terminal crashes, (2) **tamper-evident** — an audit will detect if stored votes were modified or deleted, (3) **history-hiding** — the layout of ballots on the storage medium should reveal no information about the order in which ballots were cast, and (4) **subliminal-free** — a malicious implementation or user should not be able to embed covert information in the voting record.

The **history-hiding** property is necessary for voter privacy. The concern is that during the election a coercer might keep track of the order in which voters visited a particular voting terminal. If the history-hiding property does not hold, the coercer could inspect the voting record post-election and learn how each voter voted. Thus, history-hiding is necessary to prevent coercion and vote buying. The **subliminal-free** property ensures that malicious software loaded on the terminal cannot leak covert information exposing how voters voted. Note that we must assume the user of the voting machine is actively trying to leak information about their vote, since they may be under coercion or trying to sell their vote.

Molnar et al. [15] propose a vote storage system based on PROM storage, a form of write-once memory. The data structures used to store the ballots ensure that any illicit writes to the memory will be detected, thus providing tamper-evidence. Properties (3) and (4) are also maintained by the data structures used. However, the PROM approach does not address the additional threat of one PROM being replaced with another. Since the PROM's must be transported from the voting machines to the canvassing facility at the end of the polling period, this threat must not be underestimated.

Our contributions. In this paper, we propose a new vote storage system which builds on the Molnar et al. proposal by providing additional tamper-evidence properties. Specifically, it protects against the threat of physical replacement of the memory storing the ballots while maintaining the tamper-evidence, voter privacy, and durability provided by the previous system. The new proposal incurs no additional cost in the hardware requirements of the voting machines and furthermore removes the need for disposable PROM memories (which must be purchased for each election) by employing cryptographic techniques implemented in software.

The proposed vote storage system is based on a new cryptographic primitive we call **History-Hiding Append-Only Signatures** (HHAOS), which is of independent interest. We provide two HHAOS schemes and prove them secure. The first is based on any existing digital signature system in a straightforward manner, but requires the voting terminal to store large state and does not satisfy the subliminal-free property. The second system is space efficient and can

[*] Supported by NSF CNS-0524252.
[†] Supported by NSF CNS-0524252.
[‡] Supported by NSF CNS-0524252 and the US Army Research Office Grant No. W911NF-06-1-0316.

be made subliminal-free at election close time. This second system makes use of aggregate signatures in groups with a bilinear map.

The HHAOS primitive builds on an earlier primitive called append-only signatures, introduced by Kiltz et al. [11]. The basic idea is as follows (we give precise definitions in Section 2).

- An HHAOS is initialized by generating a public key PK and an "empty" signature Φ_1. The voting terminal performs this initialization prior to election start. PK is printed on paper and stored in a trusted facility. It can be replicated for higher assurance. Φ_1 is stored in non-volatile memory in the voting machine.
- Let Φ_i be the value in non-volatile memory when voter number i walks up to the terminal. After the voter casts his vote v_i, the terminal runs an append algorithm $\mathsf{Append}(\Phi_i, v_i)$. This algorithm outputs a new Φ_{i+1} which contains the ballot v_i. This new Φ_{i+1} is written to non-volatile memory and replaces Φ_i.
- At election close the terminal runs algorithm $\mathsf{Finalize}$ to update the last Φ and prevent any further appends. This finalized Φ is published or stored in an archive. At any time post-election, anyone can validate authenticity of the set of ballots $X = \{v_1, \ldots, v_n\}$ using PK and the final Φ.

To satisfy the tamper-evidence and history-hiding requirements, an HHAOS must satisfy two security properties:

- Append-only: given a signature Φ on a set of messages X it is difficult to remove messages from X. That is, it is difficult to create a valid Φ' for a set X' satisfying $X \not\subseteq X'$.
- History-hiding: given a signature Φ on a set of messages X, an adversary cannot determine the order in which messages were added to X.

Note that when a new Φ is computed and stored within the voting machine, the previous value is deleted. While securely deleting data on a commodity system takes some care, it is certainly possible [7, 5].

Relation to append-only signatures (AOS). Kiltz et al. [11] recently introduced the concept of an append-only signature (AOS) for the purpose of securing routing protocols. They give several elegant constructions and prove that AOS is equivalent to hierarchical identity-based signatures. An AOS is closely related to HHAOS — it satisfies the same append-only property, but need not be history-hiding or subliminal-free. Not surprisingly, the constructions in [11] are not history-hiding.

Relation to forward secure signatures. Forward secure signatures [1] enable one to periodically update the signing key so that a key compromise at day n does not invalidate signatures issued prior to day n. One may be tempted to use forward-secure signatures for vote storage: after signing a vote v the terminal discards the signing key and moves on to the signing key for the next time period. Unfortunately, most forward secure signatures are inherently history-preserving. One exception is a forward-secure system due to Itkis and Reyzin [9] which could be made history-independent. The resulting HHAOS, however, is less efficient than our second construction.

2 History-Hiding, Append Only Signatures

We start by precisely defining a history hiding append only signature system. We then explain how this definition implies the properties discussed in the introduction. Formally, an HHAOS scheme consists of three algorithms:

$\mathsf{KeyGen}(1^\kappa) \to \mathrm{PK}, \Phi$
: Given a security parameter κ, produce a public key PK and an initial signature Φ, which corresponds to the empty set.

$\mathsf{Append}(\Phi, x) \to \Phi'$
: Given a signature Φ for some set X and a new string $x \in \{0,1\}^*$, produce a new signature Φ' for the set $X' = X \cup \{x\}$.

$\mathsf{Verify}(\mathrm{PK}, X, \Phi) \to \{\mathrm{True}, \mathrm{False}\}$
: Given a the public key PK and a set X, determine whether Φ is a correct signature for X.

The system must satisfy the following correctness property:

Definition 1 (Correctness). Let $X = \{x_1, \ldots x_n\} \subseteq \{0,1\}^*$. Compute $\mathrm{PK}, \Phi_0 \leftarrow \mathsf{KeyGen}(1^\kappa)$ and $\Phi_i \leftarrow \mathsf{Append}(\Phi_{i-1}, x_i)$ for $i = 1, \ldots, n$. We require that $\mathsf{Verify}(\mathrm{PK}, X, \Phi_n) = \mathrm{True}$. If this holds for all finite $X \subseteq \{0,1\}^*$, then the scheme $(\mathsf{KeyGen}, \mathsf{Append}, \mathsf{Verify})$ is *correct*.

We define security for an HHAOS system using two games. The first game captures the append-only property.

Game 1:

Setup The challenger computes $\mathrm{PK}, \Phi_0 \leftarrow \mathsf{KeyGen}(1^\kappa)$ and gives PK to the adversary.

Corrupt The adversary sends the challenger an ordered set of strings $X = \{x_1, x_2, \ldots x_n\}$. The challenger computes $\Phi_i \leftarrow \mathsf{Append}(\Phi_{i-1}, x_i)$ for each $i \in \{1, \ldots n\}$, then returns Φ_n to the adversary.

Forge The adversary returns a set Y and a signature Φ_Y.

If $\mathsf{Verify}(\mathrm{PK}, Y, \Phi_Y) = \mathrm{True}$ and $X \not\subseteq Y$, then the adversary has won Game 1.

Definition 2 (Append Only Unforgeability). An HHAOS scheme $(\mathsf{KeyGen}, \mathsf{Append}, \mathsf{Verify})$ is (t, ϵ)-append only unforgeable if every probabilistic algorithm with running time at most t wins Game 1 with probability at most ϵ. We say the scheme is append only unforgeable if the scheme is (t, ϵ)-append only unforgeable where t is a polynomial in the security parameter κ and ϵ is negligible in κ.

Note that in Game 1 we only give the adversary the power to issue a single query X. This captures the voting terminal settings where every append-only chain is used only once, namely for one machine in one election. One can extend Game 1 to allow the adversary to adaptively issue queries for multiple sets X, as in [11]. Since here we focus on the application to voting, the definition using Game 1 is sufficient. The second game captures the history-hiding property.

Game 2:

Setup The challenger computes $\mathrm{PK}, \Phi_0 \leftarrow \mathsf{KeyGen}(1^\kappa)$ and gives PK to the adversary.

Challenge The adversary returns an ordered set $X = \{x_1, x_2, \ldots x_n\}$ and two permutations λ_0 and λ_1 on X. The challenger flips a coin $b \in \{0, 1\}$ and then computes $\Phi_i \leftarrow \mathsf{Append}(\Phi_{i-1}, \lambda_b(x_i))$ for $i \in \{1, \ldots n\}$. The challenger returns Φ_n to the adversary.

Guess The adversary outputs a guess b'.

We define the advantage of the adversary in Game 2 to be $|\Pr[\, b' = b\,] - \frac{1}{2}|$.

Definition 3 (History-Hiding). An HHAOS scheme $(\mathsf{KeyGen}, \mathsf{Append}, \mathsf{Verify})$ is (t, ϵ)-history-hiding if every probabilistic algorithm with running time at most t has advantage at most ϵ in Game 2. We say that the scheme is history-hiding if it is (t, ϵ)-history-hiding where t is a polynomial in the security parameter κ and ϵ is negligible in κ.

Much related work refers to data structures which are history-*independent*, i.e., independent of the sequence of operations used to construct them in an information theoretic sense [15, 14, 16, 4, 8]. Our definition of history-hiding is based on a weaker notion of computational indistinguishability. While history-hiding is all that is needed in practice, both our constructions satisfy history-independence in the information theoretic sense.

2.1 Extensions

We describe two simple methods of adapting an HHAOS scheme. We first show how one can disallow further appends to a signature. Second, we describe a method for handling multisets, that is, producing signatures that verify that certain messages were added multiple times.

Set finalization. For the voting application we need to "finalize" the ballots and prevent any further appends. We implement this $\mathsf{Finalize}$ operation as follows. Let Φ be a signature for a set X. Define

$$\mathsf{Finalize}(\Phi) \stackrel{\mathrm{def}}{=} \mathsf{Append}(\ \Phi, \quad \text{"\_finalize\_"} \parallel |X|\ )$$

where $|X|$ is the number of elements in the set X.[1] We modify the $\mathsf{Verify}(\mathrm{PK}, X, \Phi)$ algorithm to return False if a string "_finalize_" $\parallel \ell$ is included in X and $\ell \neq |X| - 1$. Note that without embedding the size of X in the finalize message there is nothing to prevent additional messages from being appended.

Now if Φ is a signature for some set X and $\Phi' = \mathsf{Finalize}(\Phi)$, then Φ' may be given to Verify but may not be used with Append to produce further signatures. This finalization operation may optionally be performed after every append, thus producing two signatures — one signature Φ_A which may be used for further appends and one signature Φ_V which may only be used to verify the current set.

Multiset semantics. Multisets may be supported by simply appending a nonce to each string added, thus maintaining the uniqueness of each element in the set. Alternatively, a serial number ℓ may be appended to each element, where ℓ is the number of instances of that element that are already present. Using such a serial number has the advantage of avoiding the additional subliminal channel that nonces would provide, but requires the append algorithm to be aware of which messages the signature validates.

3 A Simple Construction

We now turn to constructing history-hiding append-only signatures. We start with a simple construction that builds an HHAOS from any digital signature system. This construction stores large state on the voting terminal and also assumes that an upper bound on the total number of ballots cast is known ahead of time.

The system works as follows. Let L be an upper bound on the number of messages to be signed. At setup time we prepare L signature key pairs. Then to sign the ith message x_i, we pick at random an available key pair, sign x_i with it, and delete the private signing key. The signature contains the list of message-signature pairs plus all the unused signing keys.

[1] We assume "_finalize_" $\parallel n$ is a special message that cannot appear in X for any $n \in \mathbb{Z}$. We also assume the number of elements in X is stored in Φ.

More precisely, let (G, S, V) be a digital signature system. Here G generates signature key pairs, S signs messages, and V verifies signatures. We also need a collision resistant hash function H (such as SHA-256). The generic HHAOS, denoted HHAOS_S, for signing up to L messages works as follows:

$\mathsf{KeyGen}(\kappa) \rightarrow \mathrm{PK}, \Phi$

Run $G(\kappa)$ L times to generate L signature key pairs $(\mathrm{PK}_1, \mathrm{SK}_1), \ldots, (\mathrm{PK}_L, \mathrm{SK}_L)$. Output:

$$\mathrm{PK} \leftarrow H(\mathrm{PK}_1, \ldots, \mathrm{PK}_L) \quad \text{and}$$
$$\Phi \leftarrow \{\ (\mathrm{PK}_1, \mathrm{SK}_1, \mathsf{null}), \ldots, (\mathrm{PK}_L, \mathrm{SK}_L, \mathsf{null})\ \}$$

$\mathsf{Append}(\Phi, x) \rightarrow \Phi'$

Let $\Phi = \{(\mathrm{PK}_1, Y_1, Z_1), \ldots, (\mathrm{PK}_L, Y_L, Z_L)\ \}$ and let $x \in \{0,1\}^*$ be a string. To generate the new signature Φ' do:

- Pick a random $r \in \{1, \ldots, L\}$ for which $Y_r \neq \mathsf{null}$. This Y_r is a signing key for the public key PK_r.
- Generate a signature $\sigma \leftarrow S(Y_r, x)$ and output:

$$\Phi' \leftarrow \{(\mathrm{PK}_1, Y_1, Z_1),\ \ldots \\ (\mathrm{PK}_r,\ \mathsf{null},\ (x, \sigma)\ ),\ \ldots \\ (\mathrm{PK}_L, Y_L, Z_L)\ \}$$

The net effect on Φ is that the secret key SK_r is deleted from tuple r and replaced by a message-signature pair (x, σ).

$\mathsf{Verify}(\mathrm{PK}, X, \Phi) \rightarrow \{\mathrm{True}, \mathrm{False}\}$

Given a public key PK, a set of strings $X = \{\hat{x}_1, \hat{x}_2, \ldots \hat{x}_n\}$, and a signature $\Phi = \{\ (\mathrm{PK}_1, Y_1, Z_1), \ldots, (\mathrm{PK}_L, Y_L, Z_L)\ \}$ do the following:

- If $\mathrm{PK} \neq H(\mathrm{PK}_1, \ldots, \mathrm{PK}_L)$, output False and stop.
- For all $i = 1, 2 \ldots, L$ do:
  - If $Y_i \neq \mathsf{null}$ and Y_i is not a valid signing key for PK_i, output False and stop. $(*)$
  - If $Y_i = \mathsf{null}$ then Z_i is a message-signature pair $Z_i = (x, \sigma)$.
    - If $V(\mathrm{PK}_i, x, \sigma) = \mathrm{False}$, output False and stop.
- If $X = \{\ x \mid \exists i, \sigma :\ Z_i = (x, \sigma)\ \}$ output True, otherwise output False.

The test on line $(*)$ is crucial for security — without it there is a trivial attack on the system. To test that Y_i is a valid signing key for PK_i one can test that $V(\mathrm{PK}_i, m, S(Y_i, m))$ outputs True for some arbitrary message m.

The system clearly satisfies the correctness property for HHAOS as long as Append is activated no more than L times. Security follows from the security of the underlying signature system (G, S, V) and the collision resistance of H.

Theorem 1. *If (G, S, V) is existentially unforgeable under a chosen message attack and H is collision resistant then HHAOS_S is* append only unforgeable *and* history-hiding.

The proof is only outlined as the next construction is our focus. First, the history-hiding property is trivial since the final content of Φ is independent of the order in which messages were inserted. The append only property follows from the security of (G, S, V) by a straightforward argument. We note that during the append-only unforgeability game the simulator issues a single chosen-message query for PK. Hence, it suffices that (G, S, V) is existentially unforgeable against a single-query chosen message attack. In particular, (G, S, V) can be a one-time signature.

Performance. The size of Φ is always $O(L)$. The time to verify a signature is $O(L)$ no matter how many messages were appended to Φ.

Subliminal-freeness. We point out that this system is *not* subliminal-free. In particular, the machine running the Append algorithm could choose the random r pseudo-randomly so as to leak the order in which messages were added. For example, let k be a secret key embedded in the voting terminal. When appending the ith message, the voting terminal can choose the randomness r as $r \leftarrow F(k, i)$ where F is a pseudo-random permutation such as AES. The final signature Φ will appear to be properly generated. However, anyone who knows k can recover the exact order in which messages were appended.

Bounding the number of messages. The system needs an a-priori upper bound on the number of messages to be signed. For voting machines this is easily provided; a generous estimate suggests that less than 3,000 votes across all individual races may be cast in one day at a particular voting machine based on the time necessary to cast each [15]. Tripling this for safety, we may assume that well under 9,000 messages will need to be signed at each machine, a relatively small number.

4 An Efficient Construction

Our second construction, HHAOS_E, reduces the size of Φ so that its size at any moment depends only upon the number of messages signed so far. Also, the amount of data per message is far less than in the previous system. More

importantly, a further benefit of this construction is that it can evade the subliminal attack on the first system.

Recall that the system of Section 3 stores in Φ a list of public-keys plus a list of signatures. At a high level, our second system improves upon the previous scheme using two ideas. First, we plan to use an aggregate signature system to aggregate all the signatures in Φ into a single short signature. Recall that an aggregate signature system can compress a set of signatures from different public keys and on different messages into a single signature. We will use the BGLS aggregate signature system [2, 3] for this purpose.

Second, and more importantly, we use the fact that a BGLS aggregate signature cannot be de-aggregated. That is, given an aggregate signature it is not possible to remove any signature from the aggregate. This in turn means that we do not need to pre-generate all the public keys as we did in the previous section. Instead, the Append algorithm can generate public / private key pairs on the fly and simply append the resulting public-key to Φ. As a result, Φ now grows by one public-key per message signed.

4.1 Background

The second construction uses bilinear maps, which we now briefly review. For further background see [2]. Let $\mathbb{G}$ and $\mathbb{G}_T$ be multiplicative groups of prime order p. Let g be a generator of $\mathbb{G}$. Then a computable map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is a bilinear map if

$$\forall x, y \in \mathbb{G}, \ \ \forall a, b \in \mathbb{Z}, \quad e(x^a, y^b) = e(x, y)^{ab} \quad \text{(bilinearity)}$$

and $e(g, g) \neq 1$ (non-degeneracy). Several efficient implementations of bilinear maps (e.g., the Weil pairing) are currently available [13]. We also assume a hash function $H : \{0,1\}^* \to \mathbb{G}$ that we model as a random oracle.

4.2 Algorithms

$\mathsf{KeyGen}(1^\kappa) \to \mathrm{PK}, \Phi$

Fix groups $\mathbb{G}$ and $\mathbb{G}_T$ of order p, where the size of p is a determined by the security parameter κ. Pick a generator g of the group $\mathbb{G}$ and a random exponent $\alpha \xleftarrow{R} \mathbb{Z}_p$. Output

$$\mathrm{PK} \leftarrow (g,\ e(g,g)^\alpha) \qquad \text{and} \qquad \Phi \leftarrow (g^\alpha,\ \{\})$$

Here Φ is a signature on the empty set. The exponent α is discarded.

$\mathsf{Append}(\Phi, x) \to \Phi'$

Given a signature $\Phi = (S_1, S_2)$ and a new string $x \in \{0,1\}^*$, randomly select $r \xleftarrow{R} \mathbb{Z}_p$ and output the following as the new signature.

$$\Phi' \leftarrow \big(\ S_1 \cdot H(x)^r, \quad S_2 \cup \{(x,\ g^r)\}\ \big)$$

$\mathsf{Verify}(\mathrm{PK}, X, \Phi) \to \{\mathrm{True}, \mathrm{False}\}$

Let $\mathrm{PK} = (g,\ u = e(g,g)^\alpha)$ be a public key, $X = \{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n\}$ a set of strings, and $\Phi = (S_1, S_2)$ a signature. Then given $S_1 \in \mathbb{G}$ and $S_2 = \{(x_1, z_1),\ (x_2, z_2),\ \ldots,\ (x_\ell, z_\ell)\}$, compute

$$v \leftarrow e(g, S_1) \cdot \Big(\prod_{i=1}^{\ell} e(H(x_i), z_i)\Big)^{-1}.$$

If $u = v$ and $X = \{x_1, \ldots, x_\ell\}$ output true, otherwise output false.

We note that if the set S_2 contained in Φ is represented as an ordered list then Append must randomly permute the ordering of the elements before outputting Φ'. This is crucial for history-hiding.

4.3 Properties

The following three theorems correspond to the correctness and security properties given in Section 2. Correctness is a matter of simple algebra, append only unforgeability follows from the computational Diffie-Hellman assumption, and history-hiding may be proven with no assumptions. The proofs are given in Appendix A. The history-hiding proof also demonstrates that HHAOS_E (like HHAOS_S) is actually fully history-independent in addition to being history-hiding.

Theorem 2. HHAOS_E *is* correct.

Theorem 3. *If the computational Diffie-Hellman assumption holds in* $\mathbb{G}$*, then* HHAOS_E *is* append only unforgeable *in the random oracle model.*

Theorem 4. HHAOS_E *is* history-hiding.

The proof of Theorem 3 uses the fact that a BGLS aggregate signature cannot be de-aggregated. That is, given an aggregate signature on a set of messages X it is difficult to recover an aggregate for a subset of X. This property was already discussed in [2]. Coron and Naccache [6] later showed that de-aggregation is as hard as the Computational Diffie-Hellman problem.

The append-only requirement (Game 1), however, is more strict than de-aggregation — we require that the adversary not be able to produce an aggregate signature on any set Y where $X \not\subseteq Y$. Hence, append-only security is not directly implied by the difficulty of de-aggregation in BGLS. Our proof of Theorem 4.3 shows that the system has the append-only property. The proof is a little simpler than the proof in [6] since our settings are more flexible.

4.4 Performance

The algorithms KeyGen and Append have very modest computation requirements; Verify is somewhat more expensive. The KeyGen algorithm requires two modular exponentiations (the pairing can be precomputed). The Append algorithm requires two modular exponentiations, one modular multiplication, and one evaluation of the hash function H. The Verify algorithm requires $|X|+1$ pairings, $|X|$ modular multiplications, and $|X|$ evaluations of H. The space (in bits) required to store a history-hiding append only signature Φ for a set X is $\ell_1 + (|X|+1) \cdot \ell_2$, where ℓ_1 is the number of bits required to store the strings in X and ℓ_2 is the length of a group element from $\mathbb{G}$.

4.5 Subliminal Free Rerandomization

As described, the construction of Section 4.2 contains subliminal channels that could be used by a malicious implementation of the Append algorithm to violate the history-hiding property. As in the previous section, the values r_i can be used to leak the order in which votes were added.

This situation can be remedied by adding the following Rerandomize operation.

Rerandomize$(\Phi) \rightarrow \Phi'$

Given a signature $\Phi = (S_1, S_2)$, where $S_2 = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$, select $s_1, s_2, \ldots s_n \xleftarrow{R} \mathbb{Z}_p$ and compute

$$y'_i = y_i \cdot g^{s_i} \quad \text{for all } i \in \{1, \ldots n\}$$
$$S'_1 = S_1 \cdot H(x_1)^{s_1} \cdot H(x_2)^{s_2} \cdots H(x_n)^{s_n}$$
$$S'_2 = \{(x_1, y'_1), (x_2, y'_2), \ldots (x_n, y'_n)\} \ .$$

Output $\Phi' = (S'_1, S'_2)$.

The signature Φ' is then another correct signature for the same set, but with rerandomized values $r_1 + s_1, r_2 + s_2$, etc. As in the Append algorithm, if the set S'_2 within Φ' is produced as a list, the elements should first be randomly permuted.

If a signature Φ is produced by a potentially malicious server, its subliminal channels may be cleaned by having several parties run the Rerandomize algorithm on it. If any one of those parties is honest, then the subliminal channels will not contain any of the original information from the malicious server. This re-randomization can take place when the election is closed and before Φ is made public.

5 Secure Vote Storage

Now that we have introduced the HHAOS cryptographic primitive and given two constructions realizing it, we further consider its practical use in a Direct Recording Electronic (DRE) voting machine. We tailor our description to the use of the more efficient construction, HHAOS$_E$. First we will lay out our general assumptions regarding the hardware architecture of an electronic voting machine. Having established a reference platform, we will then describe each of several isolated modules and their relationships. These may be software modules on the same hardware, or hardware isolation may be employed [17]. Finally we will consider the operational procedures that should be carried out by poll workers and election officials to initialize the voting machines, provide access to voters, and verify results.

5.1 Hardware

Although the HHAOS scheme may be used with a wide range of potential DRE equipment, we base our discussion on commodity PC machines such as those suggested by the Open Voting Consortium (OVC) as a part of their architecture for an open, verifiable electronic voting system [10]. Specifically, the OVC recommends the use of a commodity PC with a locked case. The machine would most likely not have a hard drive, but instead boot from a publicly reviewed CD distributed before the election which contains the operating system (e.g., a stripped down Linux distribution), the voting machine software, and lists of candidates. Each machine would include a printer and a removable flash memory (i.e., a USB drive or a Secure Digital memory card) on which to record the electronic ballots. Input may be obtained through a touch screen or key pad.

In addition, we require that each machine have a small amount of internal non-volatile memory (e.g. flash) in which to store the initial history-hiding append only signature when the machine is initialized. We also assume the availability of a reasonably secure random number generator, such as the `/dev/urandom` device provided by the Linux kernel. The hardware assumptions of this PC-based architecture are consistent with recent work on high-assurance voting machines [15, 18] in addition to the OVC proposals, although the previously proposed PROM-based vote storage method only requires a random number generator if the "random placement table" technique is used. The HHAOS scheme for vote storage could also be employed within a system far less capable than a PC, such as the gum stick sized motherboards produced by Gumstix, Inc. and used in the prototype system of Sastry, et al. [17].

5.2 Modules

User interface module. Figure 1 depicts the relationship between several isolated modules, the first of which is the user interface module. The user interface module is the component of the electronic voting machine that interacts

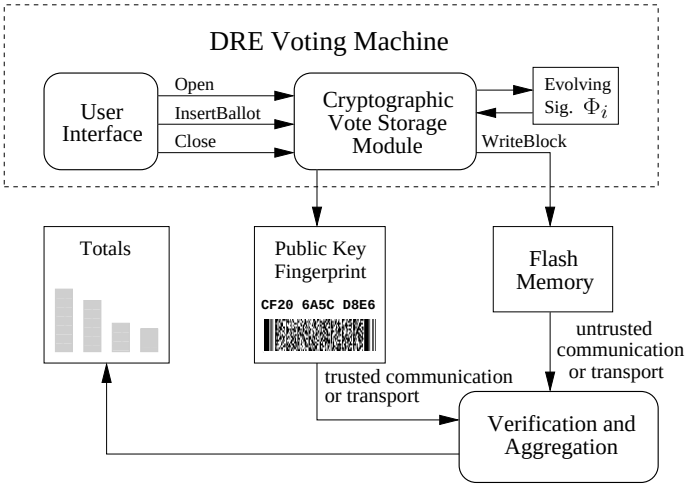

Figure 1. Relationships between modules in a DRE voting machine architecture.

with a voter to present them with the election choices and produce their completed ballot. Ideally, its source code should be small and easy to publicly verify [18]. After interacting with the voter, it invokes the InsertBallot procedure of the cryptographic vote storage module (CVSM). In describing the CVSM, we consider the ballots received from the user interface module to be simple bitstrings which are accumulated in a multiset. Each string which corresponds to a vote in a single electoral race.[2] Additionally, the user interface module provides poll workers with a means to set up the machine before polling begins and close the polls at the end of the polling period. These features invoke the Open and Close procedures of the CVSM.

Cryptographic vote storage module. The CVSM employs the HHAOS scheme of Section 4 to store the multiset of ballots on the removable flash memory while providing tamper evidence and maintaining history-independence. Here we give a high level description of the values stored in the CVSM and how they are updated. For concreteness, we give a more detailed description in Appendix B.

When the Open procedure is invoked by the user interface module, the CVSM uses the KeyGen algorithm of the HHAOS scheme to create a public key and initial signature. The public key is saved on the removable memory and a fingerprint (i.e., collision resistant hash) is printed using the printer attached to the machine. The handling of this sheet is described in Section 5.3. The initial signature is stored on non-volatile memory within the machine. When the InsertBallot procedure is invoked with a ballot b, the Append algorithm is used to update the internal signature with b (overwriting the previous value). The ballot is saved to the removable memory (taking care to ensure that every ordering of the ballots currently stored is equally likely). When the Close procedure is invoked, the CVSM uses Finalize to prevent any further additions to the signature and copies it to the removable memory.

[2] Grouping all the choices made by a voter into a single ballot string would provide subliminal channels which could be used by a voter under coercion [10].

Verification and aggregation module. To verify the signature on a set of ballots stored on a removable flash memory, we simply check that the public key fingerprint provided matches the public key stored on the memory and use the Verify algorithm to check that the signature matches the key and the stored ballots.

5.3 Operational Procedures

Initialization and polling. We assume the electronic voting machines are stored at central municipal facilities before being taken to the individual polling places on election day. Immediately prior to transport, the election officials should invoke the Open procedure on each machine, thus storing the initial history-hiding append only signature on the internal flash and printing out a sheet with the public key fingerprint. These sheets are collected for later verification of the electronic ballots. Ideally, the fingerprints should be immediately sent to the county canvassing facilities where they will be needed; this can be accomplished by simply reading the hex fingerprint over the phone. To minimize the possibility of the persons at the canvassing facility being tricked into using the wrong key fingerprints, they may be transmitted in additional ways such publicly posting them on the Internet and bringing the sheets to the canvassing facilities in hard copy. Note that from this point until the close of polling the machines should not be left unattended. If someone were to boot a machine with their own software and read the initial history-hiding append only signature stored internally, they may later be capable of replacing all the results reported by that machine. This and other threat scenarios are considered in detail in Section 6. Once the electronic voting machines are at the polling places and the polls have opened, voters may visit the machines one by one and have their votes recorded. After the polling period has ended, poll workers activate the Close procedure on each electronic voting machine and collect the removable flash memories containing the ballots.

Canvassing. The removable memories are transported to canvassing facilities where the contents are read. Using the public key fingerprints received from the staging facility, the contents of each memory are checked. The ballots may then be totaled and the results publicly announced. Note that if

we assume the public key fingerprints reach the canvassing facility securely, the integrity of the election does not depend on the integrity of the contents of the flash memories. It is therefore reasonable to transmit the signed electronic ballots over the Internet from the polling places to the canvassing facility rather than physically transporting the memories. This may somewhat reduce expenses. The history-hiding append only signatures should be rerandomized as described in Section 4.5; this may be performed once at the polling place before sending the electronic ballots to the canvassing facility and again at the canvassing facility before making the signed ballots publicly available.

6 Comparisons

The use of history-hiding append only signatures for secure vote storage in a DRE voting machine serves primarily as an alternative to the PROM system. While the PROM system ensures any illicit writes will be detected, it does not address the threat of one PROM being replaced with another. Ensuring the integrity of the election requires physical tracking and monitored transport of the PROM memories. The same considerations apply to the use of other write-once media such as recordable CD's in storing electronic ballots.

Essentially, the use of an HHAOS scheme replaces the physical tracking requirement by requiring secure communication of a public key fingerprint. A more simplistic approach to gain this effect would be to use a normal digital signature scheme to sign ballots stored by the vote storage module. However, it is likely necessary to save the signing key on non-volatile memory within the machine in order to transport it to the polling place and for fault tolerance, leaving it vulnerable to compromise. The append only property of an HHAOS scheme limits this threat by ensuring at least the integrity of ballots cast before the point of compromise.

We now detail a threat model in which to evaluate the cryptographic vote storage module of Section 5 and the PROM-based vote storage module. After explaining the model, we will highlight the improvements offered by the new techniques. Finally, we will compare the efficiency and robustness of the two approaches.

6.1 Threat Model

DRE voting machines face a wide variety of threats; however, we will restrict our attention to the types of attacks relevant to the new and previously proposed systems for vote storage. We focus on illicit read and write compromises to the memories involved in vote storage along with key management issues. In particular, we do not consider the issue of software verification. That said, the algorithms proposed in Sections 4 and 5 are simple enough to be verified by hand, with some effort. Assuming correct software, the three different components that will be considered in our threat model are the removable storage on which the electronic ballots are recorded (either a flash memory or a PROM), the internal flash memory on which the initial history-hiding append only signature is stored, and the public key fingerprint (these last two components only exist in the newly proposed system).

An adversary may gain read-only or read / write access to the removable or internal memory within a voting machine either between machine initialization and finalization or after finalization (a compromise prior to initialization will have no effect). Note that we may consider replacements of PROM's and writes to removable flash memories to be equivalent operations, since the contents of a PROM being replaced may first be read and partly copied over to the new PROM, gaining the effect of general purpose memory. Additionally, we consider the effect of the public key fingerprint printed during machine initialization being incorrectly communicated to the canvassing facility (e.g., as a result of social engineering attacks).

6.2 Threat Evaluation

Integrity. Given this threat model, we now evaluate the integrity properties of the new cryptographic vote storage module and the previous PROM vote storage module. In Table 1 we list all combinations of the previously described compromises and the resulting effects on election integrity.[3] The column for the PROM VSM depends only on the compromise to the removable memory, since that system does not include the internal memory or a public key fingerprint. Dashes in the table denote the collapse of several rows where the outcome is the same for all values of that variable.

The key security improvements offered by the CVSM over the PROM VSM manifest in scenarios B and E. In these cases the removable memory is swapped or illicitly written either before or after finalization, and the internal memory of the CVSM and the public key fingerprint are secure.[4] In both cases, any ballot tampering will be detected if the CVSM is used, but if the PROM VSM is used, the ballots currently stored at the point of compromise may be arbitrarily modified.

A lesser improvement is obtained if the internal memory of the CVSM is also compromised. In scenario C, if the adversary is able to write the internal memory when they write the removable memory, they may insert ballots undetected.

[3] Reads of the removable memory are not considered here since they affect only privacy, not integrity.

[4] A read of the internal memory at the time of compromise of the removable memory is also acceptable in scenario B.

	Removable Memory (Electronic Ballots)	Internal Memory (Evolving Sig.)	Public Key	Crypto	PROM
A	secure	—	—	✔	✔
B	swapped / written during polling	secure or read compromise	secure	✔	✘
C	swapped / written during polling	read / write compromise	secure	✔/✘	✘
D	swapped / written during polling	—	replaced	✘	✘
E	swapped / written after polling	secure	secure	✔	✘
F	swapped / written after polling	read or read / write compromise	secure	✔/✘	✘
G	swapped / written after polling	—	replaced	✘	✘

Key to symbols:

✔: No tampering possible without detection.

✔/✘: Possible to insert ballots undetected, but ballots already present at point of compromise may not be removed without detection.

✘: Arbitrary, undetected tampering with ballots present at point of compromise possible.

Table 1. Results of various threat scenarios on election integrity using the cryptographic and PROM vote storage modules.

They may not, however, remove or modify ballots already present without detection. Similarly, in scenario F, if the adversary gains read-only or read / write access to the internal memory after the first k ballots have been cast, then they may alter the set of ballots when they compromise the removable memory after finalization. However, the resulting set must include the first k ballots cast if it is to verify.

If the public key fingerprint does not correctly reach the canvassing facility, then the new system offers no improvements over the PROM-based system. It should be easier, however, to ensure the safe arrival of a public key fingerprint than a PROM.

An additional issue affecting election integrity is that of "vote disqualification" attacks, in which the attacker does not insert or delete votes, but instead attempts to prevent votes from being counted (presumably in a region supporting their opponent). An attacker who is able to replace the public key fingerprint or write the internal memory would be able cause the final signature check to fail, even if they do not have write access to the removable memory. This suggests the following policy. If the signature check fails, a recount should be performed based on a set of paper receipts or some other redundant source of information (if possible), but in no case should the votes be outright discounted.

Privacy. Having considered the improvements to election integrity offered by the use of the HHAOS scheme in the CVSM, we now compare the privacy properties of the CVSM and PROM VSM. Assuming a secure random number generator and a non-malicious implementation of the CVSM algorithms, the two systems offer the same privacy guarantees. The data structures in both the internal memory of the CVSM and its removable storage are history-independent. In either system, an illicit read of the removable storage during the polling process will reduce voter privacy by partitioning the ballots into those cast before the compromise and those cast after (but no further privacy will be lost). In the case of the CVSM, an illicit read of the value S_1 stored internally will reduce voter privacy in the same way, assuming the final contents of the removable storage are eventually made public.

However, in the case of a malicious random number generator or a malicious implementation of the CVSM algorithms, the new approach suffers from subliminal channels that may reveal a great deal of information about the ordering of ballots. The PROM VSM suffers the same problem when the random placement table technique for inserting ballots into the PROM is used with a malicious random number generator. This threat is mitigated when using the CVSM by employing the Rerandomize operation described in Section 4.5. If the contents of the removable memory are rerandomized once at the polling place after finaliza-

tion and once at the canvassing facility before the contents are publicly posted, then the subliminal channels will be publicly visible only if both the machines performing rerandomization are malicious. One point to be made regarding the process of rerandomization when using the CVSM is that the rerandomization operation may be performed by an untrusted entity. In the worst case, the subliminal channels will remain, but the machine performing rerandomization may not change the ballots without invalidating their signature. This is not the case if one were to rerandomize the output of the PROM VSM when using random placement tables. The ballots would need to be copied to a new PROM (or empty space on the original), and the machine performing rerandomization would need to be trusted to protect election integrity. When using the PROM VSM, however, subliminal channels may be avoided entirely by using a different (and less efficient) storage method, such as copyover lists or lexicographic chain tables [15].

6.3 Robustness and Efficiency

The cryptographic vote storage module described in Section 5 shares fault tolerance properties similar to those of the PROM-based vote storage module. All the information necessary for the CVSM to continue operation after a power failure or system crash is stored on non-volatile memory. When overwriting values on either the internal memory or the removable memory, simple two-phase commits may be used to allow recovery in the case of a crash in the midst of writing. In this case, a crash in the middle of an operation may reveal the last ballot stored, but there will be no further compromise of voter privacy. The unavailability of the public key fingerprint at verification time will prevent verifying the integrity of the electronic ballots, but will not prevent them from being counted.

The computational requirements placed on the voting machine by the CVSM algorithms are very modest. The voting machines need only compute modular exponentiations twice at initialization (the pairing may be precomputed) and twice for each ballot recorded (also evaluating a hash function for each ballot). This is well within the capabilities of low end commodity PC's or even much more limited embedded systems. If a commodity PC has already been chosen as the basic architecture for a DRE voting machine, the computational requirements of CVSM will not affect hardware selection. The necessary storage is also minimal. If we assume at most 9,000 votes across all races as in Section 3, 50-byte identifiers for each vote, and 160-bit group elements in $\mathbb{G}$ (typical of an elliptic curve), then less than 650KB are necessary on the removable storage (and only a single group element on the internal storage). The PROM-based VSM requires the purchase of new PROM's for each use of the machines. In contrast, a USB flash drive may be purchased (at consumer rates) for less than $9.00, a one time cost. If no internal non-volatile storage is otherwise available on the machines, 1Mbit flash memory chips may be purchased for less than $1.00 each.

7 Conclusions

We presented a new cryptographic tool for storing cast ballots on a voting terminal. The tool, called history-hiding append-only signatures (HHAOS), preserves all the benefits of a hardware-based solution, while preventing hardware replacement attacks. We presented an efficient realization of HHAOS using groups with a bilinear map. We also discussed a less efficient system that uses any standard signature scheme.

References

[1] M. Bellare and S. Miner. A forward-secure digital signature scheme. In *Proceedings of Crypto*, 1999.

[2] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Proceedings of Eurocrypt*, 2003.

[3] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. A survey of two signature aggregation techniques. *CryptoBytes*, 6(2), 2003.

[4] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history independence. In *Proceedings of Crypto*, 2003.

[5] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, 2004.

[6] J.-S. Coron and D. Naccache. Boneh et al.'s k-element aggregate extraction assumption is equivalent to the Diffie-Hellman assumption. In *Proceedings of Asiacrypt*, 2003.

[7] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the USENIX Security Symposium*, 1996.

[8] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[9] G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. In *Proceedings of Crypto*, 2001.

[10] A. M. Keller, D. Mertz, J. L. Hall, and A. Urken. Privacy issues in an electronic voting machine. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, 2004.

[11] E. Kiltz, A. Mityagin, S. Panjwani, and B. Raghavan. Append-only signatures. In *Proceedings of ICALP*, 2005.

[12] T. Kohno, A. Stubblefield, A. Rubin, and D. Wallach. Analysis of an electronic voting system. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 27–40, 2004.

[13] B. Lynn. The pairing-based cryptography (PBC) library. `http://crypto.stanford.edu/pbc`.

[14] D. Micciancio. Oblivious data structures: Applications to cryptography. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1997.

[15] D. Molnar, T. Kohno, N. Sastry, and D. Wagner. Tamper-evident, history-independent, subliminal-free data structures on PROM storage -or- how to store ballots on a voting machine. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[16] M. Naor and V. Teague. Anti-presistence: history independent data structures. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2001.

[17] N. Sastry, T. Kohno, and D. Wagner. Designing voting machines for verification. In *Proceedings of the USENIX Security Symposium*, 2006.

[18] K.-P. Yee, D. Wagner, M. Hearst, and S. M. Bellovin. Prerendered user interfaces for higher-assurance electronic voting. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT)*, 2006.

A Security and Correctness Proofs

Here we provide security and correctness proofs for the HHAOS scheme presented in Section 4.2.

A.1 Correctness

With a little algebra it is easy to verify that this scheme is correct according to Definition 1.

Theorem 2. HHAOS_E *is* correct.

Proof. Let $X = \{x_1, \dots x_n\} \subseteq \{0,1\}^*$ and assume $\mathrm{PK} = (g, e(g,g)^\alpha)$ and $\Phi_n = (S_1, S_2)$ are generated as in Definition 1. Let $r_1, r_2, \dots r_n \in \mathbb{Z}_p$ be the random values chosen in the successive invocations of Append. Let s_i denote the discrete log (base g) of $H(x_i)$ for each $i \in \{1, \dots n\}$. Then within Verify we compute

$$\begin{aligned} v &= e(g, S_1) \cdot \Big(\prod_{(x,y)\in S_2} e(H(x), y) \Big)^{-1} \\ &= e\Big(g, g^\alpha \prod_{i=1}^{n} H(x_i)^{r_i}\Big) \cdot \Big(\prod_{i=1}^{n} e(H(x_i), g^{r_i})\Big)^{-1} \\ &= e\Big(g, g^\alpha \prod_{i=1}^{n} g^{r_i s_i}\Big) \cdot \Big(\prod_{i=1}^{n} e(g^{s_i}, g^{r_i})\Big)^{-1} \\ &= e(g,g)^{\alpha + r_1 s_1 + r_2 s_2 + \cdots r_n s_n} \cdot e(g,g)^{-r_1 s_1 - r_2 s_2 - \cdots r_n s_n} \\ &= e(g,g)^\alpha \quad . \end{aligned}$$

Since we also have that $X = \{\, x \mid \exists y\ (x,y) \in S_2 \,\}$, Verify will return True and the scheme is correct. □

A.2 Append Only Unforgeability

We now prove HHAOS_E append only unforgeable in the random oracle model based on the computational Diffie-Hellman assumption.

Theorem 3. *If the computational Diffie-Hellman assumption holds in* $\mathbb{G}$*, then* HHAOS_E *is* append only unforgeable *in the random oracle model.*

Proof. Suppose the (t', ϵ')-CDH assumption holds in $\mathbb{G}$; that is, any probabilistic algorithm running in time at most t' solves CDH with probability at most ϵ'. Then we will show that HHAOS_E is (t, ϵ)-append only unforgeable with $t' = O(t \cdot \mathsf{poly}(\kappa))$ and $\epsilon' \geq \epsilon/(e(q+1))$, where $q \leq t$.

Assume a t time algorithm $\mathcal{A}$ wins Game 1 with probability at least ϵ while making at most q random oracle queries. We construct an algorithm $\mathcal{B}$ which solves CDH in time $O(t \cdot \mathsf{poly}(\kappa))$ with probability at least $\epsilon/(e(q+1))$.

Definition of $\mathcal{B}$. We receive a CDH instance $g, A = g^a$, $B = g^b$ and use it in Game 1 with $\mathcal{A}$. In order to answer random oracle queries, we maintain sets S and Γ and a map $f : \{0,1\}^* \to \mathbb{Z}_p$, all initially empty. The set S will contain all messages for which the random oracle has been called, and we will assign some of these to the set Γ. For convenience, we also define the function $H : \{0,1\}^* \to \mathbb{G}$ as

$$H(s) = \begin{cases} g^{f(s)} & \text{if } s \in \Gamma \\ B \cdot g^{f(s)} & \text{if } s \notin \Gamma \end{cases}$$

We carry out Game 1 with $\mathcal{A}$ as follows.

Setup Define $\alpha = ab$ and give $\mathrm{PK} = (g, e(A,B)) = (g, e(g,g)^\alpha)$ to $\mathcal{A}$.

Whenever $\mathcal{A}$ makes a random oracle query for $s \in \{0,1\}^*$ (in this phase or later), we answer as follows. First, check if $f(s)$ is defined (that is if $s \in S$). If so, return $H(s)$. If $f(s)$ is not defined, save a uniformly random value from $\mathbb{Z}_p$ as $f(s)$. Then we add s to S and add it to Γ with probability $\frac{q}{q+1}$. Then return $H(s)$.

Corrupt We receive $X = \{x_1, \dots x_n\}$ from $\mathcal{A}$. Without loss of generality we can assume that $X \subseteq S$, since if that is not the case we can just call the oracle for all $x_i \notin S$. If $X \subseteq \Gamma$, we abort the simulation. Otherwise, we may successfully produce a signature Φ_n for X.

Let x_k be an element of X that is not in Γ. We compute a signature Φ_n to return to $\mathcal{A}$ as follows. Select $r_1, \dots r_{k-1}, r_{k+1}, \dots r_n \xleftarrow{R} \mathbb{Z}_p$. Define $r_k = -a$. Compute

$$\begin{aligned} \Phi_n = (\{(x_1, g^{r_1}), \dots (x_k, A^{-1}), \dots (x_n, g^{r_n})\}, \\ H(x_1)^{r_1} \cdots A^{-f(x_k)} \cdots H(x_n)^{r_n}) \\ = (\{(x_1, g^{r_1}), \dots (x_k, g^{r_k}), \dots (x_n, g^{r_n})\}, \\ g^\alpha H(x_1)^{r_1} \cdots H(x_k)^{r_k} \cdots H(x_n)^{r_n}) \end{aligned}$$

and return Φ_n to $\mathcal{A}$. By the definition of r_k and $H(x_k)$, this is a well formed response.

Note that all our responses to $\mathcal{A}$ are properly distributed. The only values which have not been selected as in the regular scheme are $\alpha = ab$ and $r_k = -a$, which are independent and distributed identically to values selected as in the regular scheme. Also, the values given in response to random oracle queries are independent and distributed uniformly at random over $\mathbb{G}$.

Forge We receive a set $Y = \{y_1, \dots y_m\}$ and a signature $\Phi_Y = (S_1, S_2)$ from $\mathcal{A}$. If $\mathsf{Verify}(\mathrm{PK}, Y, \Phi_Y) = \mathrm{False}$ or $X \subseteq Y$, $\mathcal{A}$ has failed at Game 1 and we abort. Also, if $Y \not\subseteq \Gamma$, we abort.

Otherwise, we may use the forgery produced by $\mathcal{A}$ to solve our CDH instance. Denote the contents of S_2 in Φ_Y as $S_2 = \{(y_1, z_1), (y_2, z_2), \ldots (y_m, z_m)\}$. Note that $\{\, y \mid \exists z\ (y, z) \in S_2 \,\} = Y$ because $\mathsf{Verify}(\mathrm{PK}, Y, \Phi_Y) = \mathrm{True}$. Compute

$$C = S_1 \cdot z_1^{-f(y_1)} \cdot z_2^{-f(y_2)} \cdots z_m^{-f(y_m)}$$

and return C as the answer to the CDH instance.

We now demonstrate that $C = g^{ab}$. Since $\mathsf{Verify}(\mathrm{PK}, Y, \Phi_Y) = \mathrm{True}$, $\mathcal{A}$ must have queried for all $y \in Y$ at some point,[5] so $f(y)$ is defined for all $y \in Y$. Additionally, we have that

$$e(g,g)^{\alpha} = e(g, S_1) \cdot (e(H(y_1), z_1) \cdots e(H(y_m), z_m))^{-1}$$

so

$$e(g,g)^{ab} = e(g, S_1) \cdot e(g^{f(y_1)}, z_1)^{-1} \cdots e(g^{f(y_m)}, z_m)^{-1}$$

and

$$\begin{aligned} e(g, g^{ab}) &= e(g, S_1 \cdot z_1^{-f(y_1)} \cdots z_m^{-f(y_m)}) \\ \Rightarrow g^{ab} &= S_1 \cdot z_1^{-f(y_1)} \cdots z_m^{-f(y_m)} \ . \end{aligned}$$

Thus $C = g^{ab}$.

Analysis of $\mathcal{B}$. We now analyze the probability that $\mathcal{B}$ aborts before it can successfully solve its CDH instance. Let E_1 be the event of $\mathcal{A}$ succeeding at Game 1, and let E_2 be the event of $X \not\subseteq \Gamma$ and $Y \subseteq \Gamma$. The probability that $\mathcal{B}$ does not abort is then $\Pr[\, E_1 \wedge E_2 \,]$.

Since $\mathcal{B}$ produces well formed responses distributed identically to those of HHAOS_E in its interactions with $\mathcal{A}$, we have that $\Pr[\, E_1 \,] \geq \epsilon$. Now we compute $\Pr[\, E_2 | E_1 \,]$. Assume E_1. Let $\theta = \frac{q}{q+1}$. Then

$$\begin{aligned} \Pr[\, E_2 \,] &= \Pr[\, (X \not\subseteq \Gamma) \wedge (Y \subseteq \Gamma) \,] \\ &= \Pr[\, (X \setminus Y \not\subseteq \Gamma) \wedge (Y \subseteq \Gamma) \,] \\ &= \Pr[\, X \setminus Y \not\subseteq \Gamma \,] \Pr[\, Y \subseteq \Gamma \,] \\ &= (1 - \theta^{|X \setminus Y|}) \theta^{|Y|}. \end{aligned}$$

Since $\mathcal{A}$ succeeds, $X \not\subseteq Y$ and therefore $|X \setminus Y| \geq 1$. Also, $|Y| \leq q$. So

$$\begin{aligned} \Pr[\, E_2 \,] &\geq (1 - \theta)\theta^{|Y|} \\ &\geq (1 - \theta)\theta^{q} \\ &= \frac{1}{q+1} \cdot \left(\frac{q}{q+1}\right)^{q} \\ &\geq \frac{1}{q+1} \cdot \frac{1}{e} \ . \end{aligned}$$

[5] We neglect the possibility of $\mathcal{A}$ guessing the output of the random oracle, which may be made arbitrarily unlikely by increasing the output length of the random oracle.

Thus, $\Pr[\, E_2 | E_1 \,] \geq 1/(e(q+1))$, $\Pr[\, E_1 \,] \geq \epsilon$, and $\Pr[\, E_1 \wedge E_2 \,] \geq \epsilon/(e(q+1))$. So $\mathcal{B}$ does not abort and successfully solves the CDH instance with probability at least $\epsilon/(e(q+1))$. Furthermore, $\mathcal{B}$ takes time $O(t \cdot \mathrm{poly}(\kappa))$.

So if the (t', ϵ')-CDH assumption holds in $\mathbb{G}$, then HHAOS_E is (t, ϵ)-append only unforgeable, where $t' = O(t \cdot \mathrm{poly}(\kappa))$, $\epsilon' \geq \epsilon/(e(q+1))$, and $q \leq t$. In particular, if every PPT algorithm solves CDH in $\mathbb{G}$ with probability negligible in κ, then HHAOS_E is append only unforgeable. □

A.3 History-Hiding

It is straightforward to establish that the HHAOS scheme is also history-hiding.

Theorem 4. HHAOS_E *is* history-hiding.

Proof. Specifically, we show that any adversary has advantage exactly zero in Game 2. Run $\mathsf{KeyGen}(1^{\kappa})$ to compute PK and $\Phi_0 = (g^{\alpha}, \{\})$. Return PK to an adversary $\mathcal{A}$. After receiving a set $X = \{x_1, \ldots x_n\}$ and two permutations λ_0, λ_1 from $\mathcal{A}$, select

$$\begin{aligned} r_1, r_2, \ldots r_n &\xleftarrow{R} Z_p \\ r'_1, r'_2, \ldots r'_n &\xleftarrow{R} Z_p \end{aligned}$$

and compute

$$\begin{aligned} \Phi_n = \big(\ & g^{\alpha} H(\lambda_0(x_1))^{r_1} \cdots H(\lambda_0(x_n))^{r_n}, \\ & \{(\lambda_0(x_1),\, g^{r_1}), \ldots (\lambda_0(x_n),\, g^{r_n})\} \ \big) \\ \Phi'_n = \big(\ & g^{\alpha} H(\lambda_1(x_1))^{r'_1} \cdots H(\lambda_1(x_n))^{r'_n}, \\ & \{(\lambda_1(x_1),\, g^{r'_1}), \ldots (\lambda_1(x_n),\, g^{r'_n})\} \ \big) \end{aligned}$$

According to Game 2, if our coin b is 0 we must return Φ_n, otherwise we return Φ'_n. However, since $r_1, r'_1, r_2, r'_2, \ldots$ are selected independently and multiplication in $\mathbb{G}$ is commutative, Φ_n and Φ'_n are identically distributed random variables. So $\mathcal{A}$'s guess b' is independent of which of the two we return and thus independent of our coin flip b. We then have that $|\Pr[\, b' = b \,] - \frac{1}{2}| = 0$ and have shown that the scheme is history-hiding. □

Additionally, it is evident from the proof that HHAOS_E (like HHAOS_S) is not only history-hiding, but history-independent in the information theoretic sense.

B Implementation Details

Here we provide concrete details on efficiently and securely implementing the cryptographic vote storage module (CVSM) described in Section 5.2. We first detail the values stored by the CVSM, then the procedures for updating them.

The CVSM achieves multiset semantics by appending to a string the number of copies already present before inserting it into the set of stored strings, as described in Section 2.1. Specifically, the CVSM uses a hash table $C : \{0,1\}^* \to \mathbb{N}$ which keeps track of the number of copies of each string we have encountered. This may be stored in the main (volatile) memory of the CVSM process; its usage is further explained below. Referring to the HHAOS scheme described in Section 4, the history-hiding append only signature $\Phi = (S_1, S_2)$ is stored in two parts. During the polling process, we store the value $S_1 \in \mathbb{G}$ on the internal flash memory within the machine. The contents of S_2 are stored on the removable flash memory along with several other values. To refer to these locations on the removable memory, we denote the content of the removable memory with the structure given in C-like pseudocode in Figure 2. Here n is an upper bound on the number of ballots we will need to store and ℓ is the length of each ballot. These values on the removable storage along with the value S_1 on the internal storage are manipulated by the following procedures.

Open Select $\alpha \xleftarrow{R} \mathbb{Z}_p$ and compute $\mathrm{PK} = e(g,g)^\alpha$. Print a fingerprint of the public key PK. Save $S_1 \leftarrow g^\alpha$, $M \leftarrow 0$, and $P \leftarrow \mathrm{PK}$.

InsertBallot Upon receiving a ballot string $b \in \{0,1\}^\ell$, lookup b in the hash table C, incrementing the value $C(b)$ if it is found. If b is not found, insert 1 at $C(b)$. If b collides with another string $b' \neq b$ in C, use chaining and sort the strings at that location. Sorting collisions is necessary to maintain history independence. Next, randomly select $r \xleftarrow{R} \mathbb{Z}_p$, $i \xleftarrow{R} \{0, \ldots M\}$, and save $S_1 \leftarrow S_1 \cdot H(b||C(b))^r$. Then copy $S_2[M] \leftarrow S_2[i]$, store $S_2[i] \leftarrow (\, b||C(b),\ \ g^r\,)$, and save $M \leftarrow M + 1$. Note that this method of selecting a location for the new pair in S_2 ensures that every ordering of the current values in S_2 is equally likely.

Close Randomly select $r \xleftarrow{R} \mathbb{Z}_p$ and write $V_1 \leftarrow S_1 \cdot H(\text{“\_finalize\_”}||M)^r$ and $V_2 \leftarrow g^r$ on the removable storage. Save $S_1 \leftarrow 0$ on the internal storage.

To verify the ballots stored on a removable memory using a public key fingerprint, carry out the following operations. First check that the fingerprint provided matches the public key P stored on the memory. Next, scan through the first M entries in the array S_2 and compute the following.

$$z_1 \leftarrow \prod_{(x,y)\in S_2} e(H(x), y)$$
$$z_2 \leftarrow e(H(\text{“\_finalize\_”}||M), V_2)$$

If $P = e(g, V_1) \cdot z_1^{-1} \cdot z_2^{-1}$, report that the history-hiding append only signature on the recorded ballots has verified and proceed to total the ballots; otherwise, report an error.

```
struct {
    P;   // public key:  element of 𝔾_T

    M;   // number of ballots stored:
         // element of  {0,... n − 1}

    V_1; // final value of S_1:  element of 𝔾

    V_2; // finalization value:  element of 𝔾

    S_2[n]; // ballots:  array of n blocks,
         // each of which stores a pair (x, y)
         // where  x ∈ {0,1}^(ℓ+⌈log_2 n⌉) and y ∈ 𝔾
}
```

Figure 2. Values stored on the removable flash memory within the voting machine.