An extended abstract of this paper appears in *IEEE Symposium on Security and Privacy 2006*, IEEE Computer Society Press, May 2006. This is the full version.

Tamper-Evident, History-Independent, Subliminal-Free Data Structures on PROM Storage -or- How to Store Ballots on a Voting Machine

David Molnar[*] Tadayoshi Kohno[†] Naveen Sastry[‡] David Wagner[§]

February 28, 2006

Abstract

We enumerate requirements and give constructions for the vote storage unit of an electronic voting machine. In this application, the record of votes must survive even an unexpected failure of the machine; hence the data structure should be *durable*. At the same time, the order in which votes are cast must be hidden to protect the privacy of voters, so the data structure should be *history-independent*. Adversaries may try to surreptitiously add or delete votes from the storage unit after the election has concluded, so the storage should be *tamper-evident*. Finally, we must guard against an adversarial voting machine's attempts to mark ballots through the representation of the data structure, so we desire a *subliminal-free representation*. We leverage the properties of Programmable Read Only Memory (PROM), a special kind of write-once storage medium, to meet these requirements. We give constructions for data structures on PROM storage that simultaneously satisfy all our desired properties. Our techniques can significantly reduce the need to verify code running on a voting machine.

[*] E-Mail: `dmolnar@cs.berkeley.edu` . URL: `http://www.cs.berkeley.edu/~dmolnar`. Supported in part by NSF CCR-0325311 and by a NSF Graduate Research Fellowship.

[†] Dept. of Computer Science & Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. E-mail: `tkohno@cs.ucsd.edu`. URL: `http://www-cse.ucsd.edu/users/tkohno`. Supported by NSF CCR-0208842, NSF ANR-0129617, and NSF CCR-0093337. Part of this research was performed while visiting the University of California at Berkeley.

[‡] E-Mail: `nks@cs.berkeley.edu`. URL: `http://www.cs.berkeley.edu/~nks`. Supported by NSF CNS-0524252 and by the Knight Foundation under a subcontract through the Caltech/MIT Voting Technology Project.

[§] E-Mail: `daw@cs.berkeley.edu`. URL: `http://www.cs.berkeley.edu/~daw`. Supported by NSF CCR-0093337 and CNS-0524252.

1 Introduction

"Good" vote storage mechanisms are a critical enabling technology for reliable and secure electronic voting. Without a "good" vote storage mechanism, votes might be lost during power failures, an adversary might be able to undetectably tamper with the voting record post-election, or an adversary with access to the voting record might be able to compromise voter privacy. Unfortunately, "good" vote storage in practice has traditionally not been easy to obtain, either because of a lack of understanding about what the appropriate goals should be, as in the case of Diebold [9], or because of genuine subtleties in the design of a "good" vote storage mechanism; the latter is the case of our observations in Section 5 about a recent and seemingly sufficient proposal for achieving one desirable aspect of "good" vote storage [17].

Because of the subtleties with and importance of "good" vote storage mechanisms, we consider it an important research objective to thoroughly (A) investigate what "good" vote storage actually means, hence the quotes around "good" above, and (B) investigate how to build "good" vote storage mechanisms in practice. We initiate such an investigation here.

Defining "good." As our first contribution we uncover seven key properties that we believe any "good" vote storage mechanism should have. Informally, these seven properties are:

1. *Simple* : We desire a vote storage mechanism that is simple to implement, analyze, and verify.
2. *Reliable* : The vote storage mechanism should not rely on fragile moving parts or other components that might fail during use.
3. *Durable* : The record of votes should survive unexpected crashes of the vote storage mechanism.
4. *Tamper-evident* : Anyone with read access to the voting record should be able to detect post-election tampering.
5. *History-independent* : Assuming a *non-malicious* vote storage mechanism, the contents of the voting record should not reveal information about the order in which ballots were cast.
6. *Subliminal-free* : A *malicious* vote storage mechanism should not be able to undetectably embed covert information into the voting record.
7. *Cost effective* : Election officials may only deploy these solutions if the cost per voter is not significantly more expensive than alternative technologies.

The third property is important because, even if the vote storage mechanism meets the second property and is reliable, catastrophic events like power loss and battery failure might cause a machine to crash. The fifth property is important since it might otherwise be possible to compromise voter privacy if one also knows the order in which people voted [9]. The sixth property is important because, if the property is not met, a malicious vote storage device could use the covert channel to leak information about who voted for whom [7]. Although a voter-verifiable paper audit trail (VVPAT) might alleviate some of the need for these properties, we still consider history-independence and the absence of subliminal channels to be very important if VVPAT-enabled electronic voting machines also maintain digital copies of the voting record.

Our proposals for simultaneously achieving these properties use a combination of hardware and algorithmic techniques. We survey our proposals for simplicity, reliability, tamper-evidence, history-independence, and subliminal-freeness next, and defer a discussion of durability to Section 7. We evaluate the cost effectiveness of our techniques in Section 9.

Scope of our work. Our work focuses on the interaction between voting machine *software* and voting storage. As we describe below in our discussion of an architecture for an electronic voting machine, our adversarial model includes a malicious piece of software that wishes to undetectably tamper with or leak information about already cast votes. Our key insight is that by properly designing the vote storage unit, we can reduce the strength of the properties that must be verified for the main code of the electronic voting machine. For example, if the vote storage unit provides tamper-evidence, we need not verify that the rest of the software respects the integrity of previously cast ballots. We believe this will significantly reduce the cost to verify voting machine software.

Of course, a voting machine used in a real election must deal with many other threats besides software. For example, the physical security of the vote storage unit itself is an issue. If the storage unit can be physically swapped for an "identical-looking" unit with different vote totals, then the integrity of the entire election process is compromised. Tamper-resistant physical security devices, as surveyed by Weingart [19], may assist here, although Anderson and Kuhn point out several such systems can be undetectably tampered with in practice [1]. These threats, while important, are not addressed by our work. Nor do we address behavior-based covert channels such as timing or power consumption attacks. Instead, we focus on providing properties that will simplify our analysis of voting machine software.

Tamper-evident vote storage. Our approach for providing tamper-evidence involves what we believe to be a novel application of Manchester codes and programmable read-only memory (PROM). At a high level, we apply a Manchester code to votes before writing the votes to a PROM. Exploiting the fact that bits on a PROM can only transition from 1s to 0s and not vice versa, we can then argue that any modification to the data on the PROM (which by definition would involve flipping 1 bits to 0 bits) would yield an incorrect Manchester code. The presence of an incorrect Manchester code on a PROM indicates tampering. We consider our method of using PROMs to achieve tamper-evidence to be one of our main contributions and we believe that our technique may have broader applicability beyond electronic voting. See Section 4 for details, as well as for a discussion of how and why we in some cases incorporate digital signatures into our construction.

We use PROMs and avoid other forms of write-once media, such as CD-Rs, because of our simplicity and reliability goals.

History-independent and subliminal-free vote storage. Our solutions for history-independent and subliminal-free vote storage build on our underlying approach for tamper-evidence. In particular, although our approaches for providing history-independence and subliminal-free vote storage are mostly algorithmic in nature, we are careful to ensure that one can implement our history-independent and subliminal-free data structures on a PROM, which is *not* the case with other proposals for history-independence that assume erasable storage [3, 5, 13, 15]. Even under the restriction of non-erasable storage, there are several natural approaches for achieving history-independence that use random coins when storing each vote, e.g., to store a vote, select at random an unused location on the PROM. However, because a malicious vote storage device could use its choice of "random" coins to leak information to a conspirator [7], and because we prefer not to assume the presence of a secure random number generator on even non-malicious vote storage devices, it is also natural to look for history-independent data structures that are deterministic.

Suppose Ω is the set of items which one may insert into our data structure. If Ω is finite and fairly small, one can create a history-independent data structure by preallocating a region of a PROM for each element in Ω and, within each region, counting in unary the number of times that element is inserted. When Ω is large, however, the unary counter approach becomes very space inefficient. Leveraging an observation by Naor and Teague [15] that the lexicographic ordering of a set of elements must by definition be history-independent, one can adopt the following "copyover list" data structure for large Ω: to insert an element into the data structure, sort the new element along with the existing elements, write that sorted list to the next unwritten portion of the PROM, and then erase the previous sorted list by setting all the bits to zero. Notice that the length of the just-erased list depends only on the number of items in the data structure, not their content or the order in which they were inserted.

We can improve our space efficiency by either combining the above two approaches or by employing a hash table, though in the latter case we must take special care to ensure history-independence even when two elements hash to the same bucket. The key insight we use to ensure this property is to implement each bucket using another history-independent data structure, such as the previous "copyover list" proposal, the main difference in space efficiency being that copying only occurs when there is a hash collision.

The resulting structure is history-independent. Just as in the case of a single list, the length and position of each bucket's copyover list depends only on the number of items in the list. This leaves the issue of whether different amounts of deleted material in each bucket leak information about the order of votes. To see that this is not the case, observe that the amount of deleted material depends only on the number of items in each bucket, which is equal to the number of hash collisions for that bucket. The adversary, however, can compute the number of collisions itself given only the abstract contents of the data structure and the hash function. Therefore, this does not leak information about the order in which votes were cast. Details and

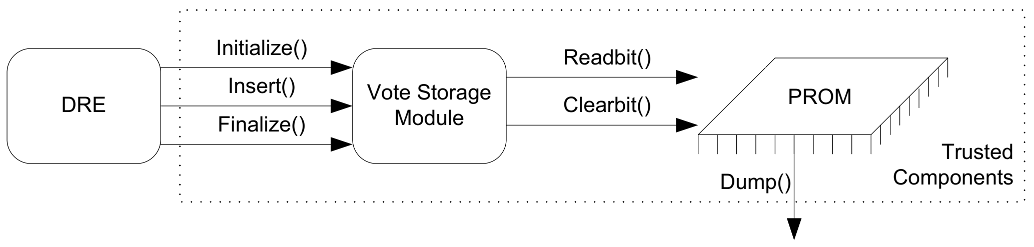

Figure 1. Our DRE architecture. The main part of the DRE has no access to writable media except through the vote storage module. After each vote is cast, the completed ballot is passed to a special vote storage module that arranges for it to be recorded on permanent storage.

extensions are in Sections 5 and 6.

Engineering issues with PROM storage. We acknowledge that there are engineering challenges with implementing our solutions on PROMs. For example, the atomicity properties of PROMs vary widely; writes may occur out of order or may only partially complete, especially in cases where loss of power or a crash occurrs during a write. We do not consider these issues in detail here, but do note these issues have been addressed by others in other contexts. For example, Niijima [16] reports on a log-structured file system designed to achieve good performance and durability even in the face of atomicity limitations. Further afield, but also illustrating both the limitations of and work-arounds for similar technologies, Gal and Toledo survey a wide variety of file systems designed for flash memories that use "wear-levelling" to avoid writing too many times to a single storage sub-unit between full resets [4].

In general, this work does not cover these engineering issues, focusing instead on the algorithmic design of data structures for vote storage. We consider PROM storage as an arbitrary-length array addressable at the bit granularity, with atomic operations to read and set each bit. Furthermore, we assume that all operations complete one at a time, in the order in which they are issued. Dealing with the engineering issues of PROM devices requires future work. We note, however, that our data structures do not require the same flexibility as a general-purpose file system. In addition, the performance requirements of the vote storage application are fairly modest.

Caveat. While we envision a simple, reliable, durable, tamper-evident, history-independent, and subliminal-free vote storage mechanism being an important component of future electronic voting machines, we acknowledge that vote storage is only one of many components that need to be made reliable and secure. Having addressed vote storage, we hope that the research community can now focus attention on improving the reliability and security of other portions of electronic voting machines. We note also the remaining engineering issues required to implement our ideas with PROM storage.

2 Our Architecture for an Electronic Voting System

We now describe our proposed architecture for a Direct Recording Electronic (DRE) voting machine (see Figure 1). We decompose the machine into three parts: a main *DRE* component, a *vote storage module*, and a removable *storage* device, represented in the figure by a PROM.

At the beginning of the election, the DRE component sends an Initialize command to the vote storage module to prepare for polling. Thereafter, the DRE component handles all interaction with the voter and produces a cast ballot. The ballot is in turn passed to the vote storage module by calling Insert. The vote storage module writes the ballot to the storage device, using the Clearbit and Readbit interfaces. At the end of the election, the DRE invokes the Finalize operation, which tells the vote storage module to modify the storage device in such a way as to preclude further votes from being recorded. The storage device might then be removed from the voting machine and transported securely to election headquarters, and the stored ballots will be available for tallying through the storage device's Dump interface.

2.1 Security Assumptions

Recall that our goal is to prevent untrusted software running on a DRE from tampering with or marking votes cast during an election, and so reduce the verification requirements for such software. We now lay out our threat model and security assumptions. We assume that the DRE has no access to writable media except through the vote storage module. Furthermore, we assume that none of the DRE's state survives between voting sessions. We also assume that no information can pass from the storage device to the stateless DRE component. This ensures that the only long-term storage available to the DRE is managed by the vote storage module. Our intent is that the vote storage module is a small and easily-verified piece of code that is isolated from the main part of the DRE. We assume for the purposes of this paper that the ballot reflects exactly the intent of the voter; for example, a voter-verified paper audit trail may be used to check the machine's result. Other than these restrictions, however, the DRE component may behave arbitrarily and adversarially.

The capabilities of the vote storage module will have an effect on the possible range of constructions for a ballot data structure. We consider three specific variations. First, we consider a vote storage component that shares a pre-set secret with an election verification authority. Second, we consider a vote storage component that does not share such secrets, but which has access to a random number generator. Finally, we consider components that have neither secrets nor randomness. This last variant is the most challenging because many cryptographic protocols require random bits. We will see, however, that our requirements can be met even in this deterministic case.

3 Proposed Security Goals

We would like the voting storage module, and the data structures that it maintains on the storage medium, to satisfy several properties:

- **Append-only multiset semantics.** The data structure should provide a multiset abstraction. A multiset is a collection that behaves like a set, except that each element of the collection may occur more than once. Thus, each element of a multiset has a multiplicity indicating how many times it occurs in the collection[1].

  The multiset data structure should provide a single operation, Insert. When a ballot is cast, it is added to the multiset, so that each element of the multiset represents a ballot that has been successfully cast. The data structure should provide append-only semantics: once a ballot is cast (i.e., inserted into the data structure), there should be no way to "uncast" it (e.g., to modify or delete it from the data structure).

- **Durability.** Ballots should be stored on an indelible, non-volatile storage medium. Ballots, once stored on the storage device, must never be lost (in the absence of an adversary): if n ballots are successfully cast and then the machine crashes, it should be possible to reconstruct the first n ballots. Reconstruction should be possible even if the machine crashes while in the middle of recording the $n + 1$st ballot.

- **Tamper-evidence.** Stored ballots should be protected from tampering, so that any unauthorized modifications made to the data structure can be detected. In particular, modifications made by the voting storage module are presumed to be authorized; all other modifications to the storage medium are unauthorized, and must be detectable. We distinguish between two flavors of tamper-evidence:

  - **Weak tamper-evidence.** We first focus on detecting modifications made after the close of the election. When polls are closed, the vote storage module will execute a Finalize algorithm. This operation should irreversibly freeze the contents of the data structure to ensure that it cannot be subsequently modified without detection. This prevents any additional ballots from being added after the polls close ("ballot stuffing"), and prevents poll workers from tampering with the contents of the data structure while the storage device is in transit ("ballot box tampering").

[1]Formally, a multiset m is a map $m : \Omega \to \mathbb{N}$, where Ω denotes the universe of possible element values.

When the votes are tallied from the storage device, election officials will invoke a Check algorithm to determine whether the data structure has been tampered with.

- **Strong tamper-evidence.** In some settings, we may also wish to prevent undetected modifications even if they occur while the machine is in operation. We assume that before the polls are opened, the storage medium will be initialized via an Initialize algorithm. Once initialized, we must detect any changes to the storage medium by any entity other than the vote storage module, no matter when the change occurs. Later we will see how to accomplish this by providing the vote storage module with a secret key; the security requirement is that anyone who does not know the key should be unable to undetectably modify the state of the data structure.

We will show later that it is possible to achieve both forms of tamper-evidence without any form of key management and without any out-of-band channels.

- **History-independence.** To preserve voter privacy, we must ensure that the data structure does not reveal any information about the order in which ballots were cast. Because polling places are public, the order in which people vote must be assumed to be public. Revealing the order in which votes were cast can thus lead to a serious violation of voter privacy. For example, the Diebold AccuVote-TS machine stored ballots sequentially, in the order they were cast [9]; thus, an adversary with read access to the storage device might be able to tell how individual voters voted, violating the secrecy of the ballot. This motivates our requirement for history-independent ballot storage.

  We distinguish between two kinds of history-independence [15], depending on when the adversary is able to view the contents of the data structure:

  - **Weak history-independence.** Once the election is over, the adversary should learn nothing about the order in which ballots were added to the data structure. In other words, the Finalize operation should eradicate any ordering information that may have been present.
  - **Strong history-independence.** This property asks that the state of the data structure never reveal any order information at any time. In such a scheme, Finalize is not needed to preserve history-independence. This version provides strong privacy protection: ballots are anonymized as they are cast (analogous to what occurs in a paper ballot election), so that no information that could compromise privacy is ever written to permanent storage in any form.

    Strong history-independence also ensures that privacy will be protected even if the machine is taken out of service without calling Finalize. For instance, if the machine crashes in the middle of the election, it might be necessary for officials to inspect the state of the storage device before Finalize has been called. Strong history-independence will ensure that such inspection cannot reveal anything about how previous voters have voted.

    Our notion of history-independence assumes that each abstract operation completes correctly. We will see that our constructions are not fully history-independent if the machine is interrupted midway through an operation. This appears to be necessary if our constructions are to achieve the durability properties we desire. In all cases, however, our constructions leak at most the state of the data structure immediately prior to the interrupted operation, plus (possibly) what the state of the data structure would be after the update. That is, if the machine crashes, an adversary could learn information about how the last voter voted, but not more.

  We focus primarily on perfect history-independence. Formally, the abstract state of the data structure is the mathematical multiset that it is intended to represent. History-independence requires that the data structure representation, as stored on the storage device, must depend only on the abstract state of the data structure and not on anything else (e.g., not on the order in which ballots were added to the multiset). Thus, for deterministic schemes, we require that there be some mathematical function f so that the concrete representation is precisely f(abstract state). For example, Insert(x) followed by Insert(y) should yield the bit-level representation as Insert(y) followed by Insert(x), since in both cases the abstract state of the multiset is $\{x, y\}$. For randomized schemes, the function f maps an abstract state to a distribution on concrete representations; thus the actual value of the concrete representation must not depend on anything other than f(abstract state).

- **Verifiable subliminal-freedom.** The voting system should be verifiably free of subliminal channels. If there are multiple valid representations for the same abstract state, it may be possible for a malicious vote storage mechanism to secretly embed some information into the vote file by choosing among these representations. If we are not careful, this might provide a covert channel by which malicious logic in the voting machine could leak information that violates voter privacy. For example, as several authors have noted, malicious logic might try to leak the time each ballot was cast so that a co-conspirator who later gains access to the storage device can correlate the ballot with the voter who cast it [7, 8]. Therefore, we require that it be possible to verify that the representation of the data structure does not conceal any hidden information[2].

  In some cases, our data structures are deterministic and have a canonical representation, thereby rendering such verification trivial, since the notion of subliminal-freedom coincides with history-independence.

  In other cases, we may wish to use a randomized scheme. One approach to verifiable subliminal-freedom for randomized data structures is to use hardware or software verification techniques to verify the implementation of the random number generator available to the DRE. If we prove that the RNG cannot be influenced by the DRE, and that the concrete representation depends only on the abstract state of the data structure and the values obtained from the RNG, then we can rule out subliminal channels. However, this has the drawback that such verification may be costly. We shall see later an algorithmic technique that can, in some cases, eliminate the need for such verification.

- **Predictable capacity.** Voting machines must be provisioned with sufficient capacity to store as many ballots as we might reasonably expect to be cast during the election. Therefore, it must be possible to predict the worst-case space usage of our data structure, as a function of the number of votes, so that we can choose a large enough storage device. Equivalently, given a fixed-size storage device, we should be able to predict a hard lower bound on the number of votes it will be able to store.

4 Tamper-Evident Write-Once Storage

Write-once storage media. We will exploit the properties of write-once storage media, i.e., devices whose physics ensure that data can only be written once. Perhaps the most familiar example is CD-R media. However, CD-R is poorly suited to voting because CD writers contain fragile moving parts that would constrain the reliability of the voting machine, and because they require complex software drivers that would make program verification hard.

Instead, we use Programmable Read Only Memory (PROM). A PROM is a solid state storage device in which 1 bits may be flipped to 0 bits, but not vice versa. A blank PROM device comes initialized to the all-ones state. Clearing a bit is an irreversible operation: once a bit position is cleared, it will never be 1 again. This irreversibility is the key property that permits us to achieve tamper-evidence. If traditional antifuse [20] PROMs are not readily available, we suggest that it might be possible to relax the write-once property and instead use One-Time Programmable Electrically Programmable Read Only Memory (OTP EPROMs). An EPROM is a device in which bits may be set electronically, but may only be cleared by exposure to ultraviolet light. An OTP EPROM is an EPROM device in an opaque housing that prevents such erasure. Using an OTP EPROM is a relaxation of the write-once property because OTP EPROMs can still be reset under some circumstances, e.g., if their protective coating is removed and if they are exposed to ultraviolet light.

As noted above, we will assume PROMs can typically be addressed at the bit level. Consequently, the PROM's externally visible interface exports two operations: $\mathsf{Clearbit}(i)$ and $\mathsf{Readbit}(i)$, where i designates a particular bit position. The physical properties of PROM restrict what the adversary can do. For two strings $x, y \in \{0,1\}^n$, we say that $x \preceq y$ if y can be obtained from x by flipping 1 bits to 0 bits. The $\preceq$ relation encapsulates exactly the restrictions on modifying a PROM: the string x can be replaced only with strings

[2]If there are multiple equivalent representations for the ballot that will counted in the same way, then a malicious DRE could use that to embed covert information in those ballots no matter what data structure we use [7]. Likewise, if the DRE is free to provide a ballot that does not reflect the voter's intent, the DRE can also embed covert information in a write-in not requested by the candidate. Therefore, we assume that the ballot itself, as generated by the DRE, is in some canonical form and accurately represents the voter's intent.

y such that $x \preceq y$. PROMs are reasonably inexpensive, so we can afford to use each PROM only once. For instance, the ST Microelectronics M27C4001-10B1 is a 4MBit OTP EPROM chip that costs \$2.75 [14].

Weak tamper-evidence through Manchester encoding. We observe that writing a *Manchester encoding* of data provides tamper-evidence properties. The Manchester encoding of a n-bit string x is a $2n$-bit codeword $M(x)$ obtained by applying the mapping $0 \mapsto 01$, $1 \mapsto 10$ to each bit of x. For example, the Manchester encoding of the string 101 is 100110.

The key property of the Manchester encoding is that if any set of 1 bits in a Manchester codeword are flipped to 0s, then the result is no longer a valid codeword. For example, if the first 1 bit is cleared in 100110, the resulting string 000110 is no longer a valid Manchester codeword. This gives us a simple test to determine whether an adversary has tampered with data stored on a PROM: we encode the data using the Manchester encoding, and then later check that the PROM still contains a valid codeword. By the properties of the PROM, any modification by the adversary will result in an invalid codeword and so can be easily detected. Consequently, Manchester encoding ensures that data, once written to the PROM, cannot be undetectably changed.

Zero-fragile encodings. We can generalize this technique by introducing the notion of *zero-fragile encodings*, where taking any valid codeword x and clearing some bits in x can never lead to another valid codeword[3]. This captures exactly the property that allows us to easily check for evidence of tampering on a PROM. Note that the Manchester encoding is a special case of a zero-fragile encoding.

The Manchester encoding is easy to understand, but it doubles the space usage. In fact, it is possible to improve upon the Manchester encoding. Let $\mathrm{wt}(x)$ denote the Hamming weight of the string x and $\overline{x}$ denote the complement of x. Consider the encoding $E(x) = x \,||\, \mathrm{wt}(\overline{x})_2$ obtained by concatenating the string x with the binary representation of $\mathrm{wt}(\overline{x})$. Of course, $0 \leq \mathrm{wt}(\overline{x}) \leq n$, so $E(x)$ is $n + \lceil \lg(n+1) \rceil$ bits long. The Manchester encoding corresponds to the special case where $n = 1$.

It is not hard to see that E is a zero-fragile encoding [12]. Clearing any bit(s) of x increases the value of $\mathrm{wt}(\overline{x})$. Increasing the value of any non-negative integer causes at least one bit position in its binary representation to change from 0 to 1. Consequently, there is no way to modify a E-codeword stored on a PROM into another valid E-codeword.

The encoding E has overhead low enough for any practical application, but asymptotically requires overhead logarithmic in x for storing its output. We note that if we are willing to assume the existence of a collision-resistant hash function family, we can obtain a "computationally" zero-fragile encoding with constant overhead. The encoding consists of x followed by a zero-fragile (e.g., Manchester) encoding of the hash of x. One can show that if there exists an adversary that breaks the zero-fragility of this encoding, then there exists an adversary that finds collisions in the hash function. In practice, however, this construction requires more space than our other constructions for reasonable input sizes. In the case of a keyed hash function, the device may also generate the key at initialization time, write a commitment to the key to the "zero tape," and finally reveal the key after the election is over.

Accounting for random bit errors. In practice, PROM devices may accumulate random bit errors from environmental or other factors. If we store just one copy of our data, it is susceptible to these bit errors. If we use an error correcting code, on the other hand, and accept results even if errors are corrected, then we open a possible subliminal channel. Specifically, a malicious DRE can use the location of corrected bit errors to communicate information about voters. We can address this by a procedural constraint: if any bit errors do occur, then the voting storage device is not considered to be in "canonical form" and the entirety of its contents must be handled only by election officials, not released to the general public. In this case, a malicious DRE can leak information only to election officials, who must then be trusted not to collude with the DRE to violate the privacy of the votes. To others, the malicious DRE can leak less than a bit of information, specifically $(1 - e)$, where e is the natural bit error rate of the PROM.

Weakly tamper-evident write-once storage. Putting it together, we can build a storage abstraction that provides tamper-evident storage consisting of fixed-length blocks, where each block can be written only once. Each block is initially in the all-ones state. To write a block with some data value x, we write the value x (in unencoded form) to the corresponding portion of the PROM. Also, a block can be permanently

[3]Formally, a zero-fragile encoding is one where there does not exist any pair of valid codewords x, y such that $x \prec y$ and $x \neq y$.

erased by setting it to the all-zeros value; such a block will no longer be able to store any value ever again. We reserve the first block for use during finalization. When it comes time to freeze the data structure, the Finalize algorithm writes the value $\mathrm{wt}(\overline{s})_2$ to the first block, where s denotes the state of the entire PROM (apart from the first block). The Check algorithm can simply check that this first block contains the correct count of 0 bits in the rest of the DRE. Notice that this guarantees weak tamper-evidence: once Finalize is called, it is impossible to change the contents of the PROM without detection. Thus, weak tamper-evidence can be achieved at a cost of only $\lg n$ bits, where n denotes the size of the PROM in bits. Our later schemes will build on this abstraction to provide additional security properties.

Strong tamper-evidence. We can also provide strong tamper-evidence by using a public-key signature scheme. We set aside the first block of storage to hold the public key. The Initialize algorithm checks that the PROM is empty (all-ones), generates a new keypair, and finally writes an encoding of the public key to the first block. To write the string x to block i, we write $E(x) \mid\mid \mathrm{Sign}(x, i)$ to the PROM, where $E(x) \mid\mid \mathrm{Sign}(x, i)$ consumes exactly one block. We can arrange that the private key is known only to the vote storage module, so that no one else can write validly signed blocks. The Check algorithm can use the public key found in the first block to verify each signature; invalid codewords or invalid signatures indicate tampering.

5 History Independence

Building incremental history-independent data structures on top of write-once storage is not entirely trivial. We were initially inspired by an elegant scheme of Shamos, proposed to a Voting System Performance Rating working group mailing list [17]. The data structure is organized as a hash table, stored directly on a write-once device such as a CD-R. We fix a hash function H and store each ballot x at position $H(x)$ in the hash table.

At first glance, this proposal appears to provide the history-independence property we need. By using a hash table instead of a sequential list, entries in the hash table appear to be placed independently of the order of arrival. If x and y are two ballots, then regardless of which is inserted first, it would seem to be the case that x is always stored at location $H(x)$ and y is always stored at location $H(y)$. Shamos suggested keeping a unary counter at each location to record the number of times each ballot has been inserted, which does not leak any information.

The issue we raise is: what if $H(x) = H(y)$ where x and y represent different ballots? Although Shamos suggests that such collisions might be handled in any of the usual ways, we disagree. Specifically, the CD can store only one of x or y at location $H(x) = H(y)$, so whichever ballot was created first will be the one stored at that location.

The usual methods for resolving collisions have the same problem. For instance, suppose we use multiple probing, so that ballot x is stored at position $H(i, x)$, where i is the smallest positive integer such that slot $H(i, x)$ is empty. However, if $H(1, x) = H(1, y)$, then whichever ballot was cast first will appear at that location, and the other one will appear elsewhere. This shows that when collisions occur, information is leaked about the order in which ballots were cast. Consequently, simple hash tables do not provide perfect history-independence.

While Shamos suggested using a table size large enough that collisions would be reasonably rare, we would prefer a scheme that leaks no information. Furthermore, as we will see, extra storage will directly impact the cost of our methods for providing tamper-evidence; therefore, we would prefer to use smaller tables and tolerate hash collisions while retaining history-independence. Accordingly, we describe four techniques for perfectly history-independent multi-sets. Each of these schemes will be layered on top of a weakly tamper-evident write-once storage abstraction to obtain a weakly tamper-evident append-only history-independent multiset.

Unary counters. A multiset m is a function $m : \Omega \to \mathbb{N}$, where Ω is the universe of possible element values. Suppose Ω is finite and fairly small, say $\Omega = \{w_1, \ldots, w_k\}$. Then the multiset m corresponds to a k-tuple $(m(w_1), \ldots, m(w_k)) \in \mathbb{N}^k$. Select an upper bound b, so that $0 \leq m(\cdot) < b$. Storing each integer in the k-tuple in unary provides a representation as a string of bk (encoded) bits. The multiset m is thus represented by $m(w_1)_1 || \cdots || m(w_k)_1$, where j_1 denotes the encoding of j in unary (i.e., $j_1 = 1^j || 0^{b-j}$). The Insert operation requires us to increment one of the unary counters, which is as simple as setting a bit.

Strong history-independence follows, since the representation is a deterministic function of m: there is a single canonical representation for each possible multiset.

Copyover lists. If the universe Ω is large, the unary counter method becomes very inefficient. An alternative is to store the elements of the multiset in lexicographically sorted order. Our starting point is an observation of Naor and Teague: if a list is sorted after every insertion, the resulting data structure is history-independent. In our setting, however, we cannot simply sort in-place when new elements are inserted. Instead, on every insertion, we compute the sorted list including the new item, copy over the contents of the list to the next unwritten portion on the storage medium, and finally erase the previous list. We call this data structure a *copyover list*. Pseudocode for the copyover list is shown in the Appendix.

The copyover list is strongly history-independent. Each set has a single canonical representation. The representation of a set containing n distinct elements consists of $n(n-1)/2$ erased blocks, followed by n blocks, each containing a zero-fragile encoding of an element, with elements appearing in sorted order, with all remaining blocks holding a zero-fragile encoding of the distinguished value for unused.

As an optimization, we may combine copyover lists with unary counters. Each record in the sorted list will be a pair of the form $(w, m(w)_1)$, where w represents the element and $m(w)_1$ its multiplicity encoded as a b-bit unary counter. This encoding preserves strong history-dependence, and the Insert operation is very efficient when inserting an element that already appears in the list. The main drawback of the copyover list is that it requires a full copy on every insertion. Therefore, for a multiset containing n distinct items (not counting multiplicities), the copyover list requires $\Theta(n^2)$ space.

Lexicographic chain tables. We can improve our average-case space usage by a hash table with chaining. The hash table is an array of buckets, and the ith bucket contains all elements that hash to i. Each bucket itself should be stored via some history-independent multiset data structure. Storing each bucket as a separate copyover list yields a data structure we call a *lexicographic chain table*. Pseudocode for a lexicographic chain table that uses copyover lists for buckets is shown in the Appendix.

Given a multiset and a choice of a hash function, it is easy to compute the canonical representation of the multiset as a lexicographic chain table. Therefore, this data structure is strongly history-independent, by a similar argument as used for the copyover list. The key difference between the lexicographic chaining table and the copyover list, however, is that copyovers are necessary only when collisions occur in the hash function. Even when collisions do occur, the size of each bucket is likely to be considerably smaller than the size of the multiset as a whole; since the space complexity of copyover lists is quadratic in their size, this is a significant improvement. This construction may be vulnerable, however, to a a side channel attack that measures the time to insert an element into the data structure and, thereby, detects the number of collisions.

Random placement tables. If the machine has a high-quality random number generator, we can further improve the space complexity with a simple technique. We allocate an array A of ℓ slots, each of which is initially empty. To insert the element x, we pick a random slot; if it is empty, we place x there, and otherwise we continuing probing slots at random until we find one that is empty. The algorithm is as follows:

```
Insert($x$):
1. Pick $i \in_R \{0, 1, \ldots, \ell - 1\}$ uniformly at random.
2. If $A[i]$ is non-empty, then go to step 1.
3. Set $A[i] := x$.
```

This scheme is strongly history independent, as long as the random number generator is flawless. Insertions take $O(1)$ time so long as we do not try to insert more than $\ell/2$ elements or so. Associating a unary counter to each slot yields an append-only multiset data structure.

6 Verifiably Subliminal-Free Data Structures

Unary counters and copyover lists are deterministic data structures, where each abstract state has a canonical concrete representation, and consequently they are verifiably subliminal-free: anyone can inspect the representation, verify that it is in canonical form, and thereby verify that there was no opportunity to covertly hide secret information in the way the data was represented. However, randomized schemes do not have this property. For example, if the random number generator (RNG) facility contains malicious logic, the malicious logic might be able to select its random numbers to embed secret information in the concrete representation of the data on the storage device.

Random placement tables. Random placement tables are vulnerable to subliminal channel attacks, if the RNG is not trustworthy. For instance, a malicious RNG could return the sequence $0, 1, 2, \ldots$ of "random" numbers, causing items to be recorded sequentially and possibly compromising voter privacy. Or, a malicious RNG could return the sequence $\pi(0), \pi(1), \ldots$, where π is a secret permutation known only to the attacker. There is no way to verify, merely by inspecting its concrete representation, that the random placement table is free of subliminal channels. Consequently, we must prove (e.g., using program verification techniques) that the RNG behaves correctly and cannot be influenced in any way by the DRE or by any other unverified code.

Lexicographic chain tables. A hash table with a pre-fixed hash function is a deterministic data structure, and hence verifiably subliminal-free. However, with a fixed hash function the worst-case behavior of a hash table is very poor, so in some settings we may wish to randomly choose the hash function to ensure that every sequence of Insert operations has reasonable space complexity (with high probability). For instance, the vote storage module might choose a random key k and use $F_k(\cdot)$ as the hash function, where F denotes some pseudorandom function (PRF). The problem with this naive scheme is that it is not subliminal-free: a malicious RNG could choose k to leak secret information.

We fix this vulnerability using an idea from collusion-free protocols [10]. Let $\mathcal{H}$ denote a cryptographic hash function such as SHA256; we will model $\mathcal{H}$ as a random oracle [2][4]. In our scheme, Initialize chooses a random key k and commits to k by storing $\mathcal{H}(k)$ at a location set aside for this purpose (using our tamper-evident storage abstraction). We must verify that this field will always be written before the Initialize operation completes. This ensures that the machine must commit to the randomness to be used *before* any votes are known to the machine. Finalize then stores k at another location set aside for it (again, encoded in a tamper-evident way). During tallying it is easy to confirm that the k is a valid opening of the commitment $\mathcal{H}(k)$ and that all items have been stored in the right location, as specified by k. Since the only randomness used in the representation of the data structure was committed to before the machine learned any secret information, this demonstrates that the randomness must be independent of all secrets that entered the machine during the election, and hence the randomness cannot subliminally leak any information that would violate voter privacy.

We are able to apply this technique to lexicographic chain tables, because there the randomness is only used to improve space complexity and is not needed for history-independence. In contrast, we cannot apply this "commit-use-reveal" method to random placement tables, because the randomness is essential for their history-independence property.

Strong tamper-evidence. Our construction for strongly tamper-evident storage (see Section 4) uses a public-key signature function. To avoid subliminal channels, this should be instantiated by a unique signature scheme [11], where for every public key pk and every message m there is a unique signature s that makes $\mathrm{Verify}(pk, m, s) = 1$. Fortunately, many standard signature schemes already provide this property.

7 Durability

The durability goal requires us to provide some protection in case the voting machine crashes while in the middle of clearing some sequence of bit positions. We propose a simple two-phase commit protocol. Let us illustrate its application to copyover lists. Leave space for a single (unencoded) "commit bit" at the beginning of each copy of the list. The commit bit is initially 1, which represents a list that has not yet been committed. The Insert operation copies the list (leaving the commit bit for the new copy set to 1); clears the commit bit for the new copy; and finally erases the old copy. We can recognize valid lists by the fact that their commit bit has been cleared. Even if a crash happens at any point in the middle of this operation, atomicity and durability will still be guaranteed. Ignoring all uncommitted lists and deferring erasure until after the new copy has been committed ensures that crashes cannot cause data corruption. Similar methods

[4]Standard commitment schemes use some additional random coins, which are revealed when the commitment is opened. However, there may be multiple ways to open the commitment using different values for these random coins, which enables a subliminal channel. The random oracle eliminates this vulnerability.

Data Structure	Requirements	HI	TE	VSF	Space
Unary Counter	$\lvert\Omega\rvert$ small	Yes	Yes	Yes	$O(n)$
Copyover List	None	Yes	Yes	Yes	$O(n^2)$
Lexicographic Table	None	Yes	Yes	Yes	$O(n \log^2 n)$ w.h.p
Random Table	RNG	Yes	Yes	(1)	$O(n)$

Figure 2. Summary of our data structures and their properties. Here RNG stands for "random number generator." HI, TE, and VSF stand for history-independence, tamper-evidence, and verifiable subliminal-freedom, respectively. Random placement tables are subliminal-free only if the RNG has been verified to be perfect.

can be applied to all of the data structures in this paper. As noted above, a crash in the middle of an update causes a failure of complete history-independence: it is possible for an adversary to then learn information about how the last voter voted, but not more. This appears to be necessary if we are to make a guarantee that votes will not be lost.

8 Constructions

We now show how to put these techniques together and apply them to the voting context. A summary of our constructions is in Figure 2.

A typical ballot consists of many races, and in each race there are several candidates (or "Yes"/"No" options, for a proposition or ballot measure) one may choose among. Votes for the standard candidates are stored in a *primary multiset* where the universe Ω is given by the set of names of standard candidates. The most efficient implementation is to use one unary counter for each candidate. Casting a vote for candidate c corresponds to invoking $\mathsf{Insert}(c)$ on the primary multiset, or equivalently, to incrementing the unary counter for candidate c. In the absence of write-ins, each ballot to be cast contains a set C of votes for various candidates; casting a ballot thus corresponds to executing $\mathsf{Insert}(c)$ once for each $c \in C$.

We store write-in votes in a *secondary multiset* data structure. There are many possibilities available to us. Write-in votes may be stored using any of the four multiset schemes suggested above (Section 5). In addition, we can either (a) use a single data structure to store all the write-ins, pooled across all races, or (b) use a separate data structure for each race to hold just the write-ins for that race. We suggest three candidate constructions for how to organize storage of write-in votes:

- A separate copyover list, one for each race, to hold the write-ins for that race. Casting a vote for write-in candidate w in race r corresponds to inserting w into the copyover list for r.
- A single pooled lexicographic chain table, used to store the multiset of all write-in votes. Casting a vote for write-in candidate w in race r corresponds to inserting the pair (w, r) into the chain table.
- A single pooled random placement table, used to store the multiset of all write-in votes. Casting a vote for write-in candidate w in race r corresponds to inserting the pair (w, r) into the random placement table.

The asymptotic space complexity of these constructions are given in Figure 2, and analyzed in more detail in Section 9. We use n to denote the number of distinct write-in candidates. In the lexicographic chain table construction, we suggest to use a PRF or 2-universal hash as our hash function and to provision the hash table with $2n$ buckets. With these parameter settings, we expect to use only $O(n \log^2 n)$ space for the lexicographic chain table, as we expect each bucket to have less than $O(\log n)$ entries with high probability. We discuss the exact probabilities further in Section 9. If the hash key is not revealed to the adversary until after the election, this analysis holds even in the presence of malicious insertions.

We propose using a weakly tamper-evident encoding. Then, all three of these constructions will provide a vote storage mechanism that is weakly tamper-evident, strongly history-independent, and durable. The copyover list and lexicographic chain table constructions are verifiably subliminal-free; in contrast, the random placement table requires a RNG that has been proven correct.

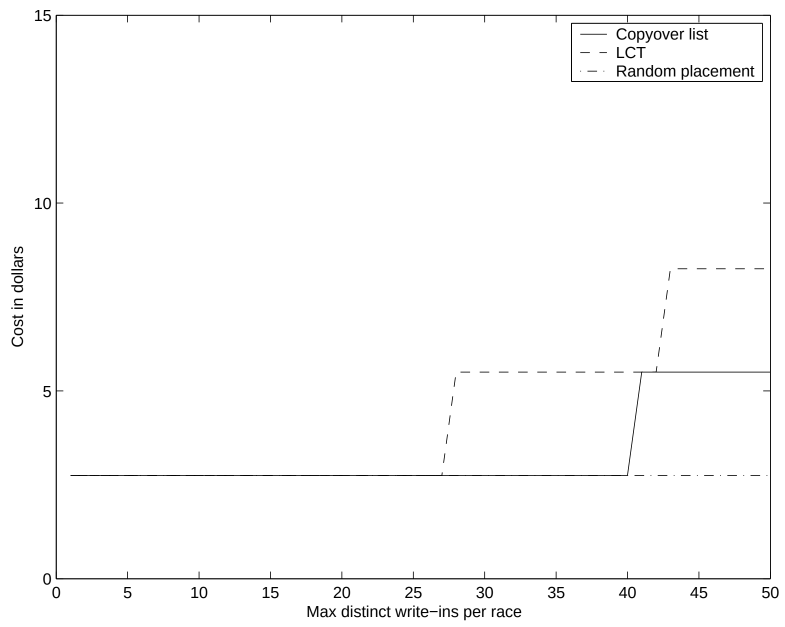

Figure 3. Cost estimates for a single voting machine using our ballot constructions for a PROM. Cost is a step function because we assume that PROM must be purchased in increments of a 4 megabit chip. Note the expected number of distinct write-ins is less than 1 per race, so costs remain reasonable even if the actual number of write-ins far exceeds the expected number.

9 Cost Analysis

We now calculate the cost for implementations of our ballot data structure with several choices of parameters. We will base our calculations and estimations on the November 2, 2004 election for precinct 213100001 in Alameda County, California[5]. We use the dataset from the Smart Voter website at `http://www.smartvoter.org`. The storage requirements, and hence cost of storage, depends on the number of voters, number of candidates, and number of write-ins.

There were 9 races with a total of 31 candidates and 20 (yes-no) state propositions or local ballot measures. In the absence of accurate polling place data, let us derive an estimate for the number of voters using each voting machine. This particular polling place was open for 13 hours, and if we assume it takes 15 seconds to consider each race or ballot measure, this allows for a maximum of 104 voters. Let us, for safety's sake, triple that guess, and assume there are a maximum of 300 voters per voting machine. This appears to be a safe estimate; anecdotal evidence suggests that a busy machine might receive 100–150 votes at most.

We use a unary counter to represent each race or option in a measure. For the example ballot, we will use $31 + 2 \times 20 = 71$ unary counters, one for every choice. Recall that incrementing a unary counter uses one bit. We must size the unary counter to be larger than the maximum number of expected voters; in our case, a total of $71 \times 300/8 \approx 2.60$ KB suffices.

Of course, for each of the 9 races, the voter is free to write in the candidate of their choice. A write-in consists of the candidate's name (30 characters long) as well as a unary counter for a total of 68 bytes per distinct write-in. Note that the number of distinct write-in candidate names (ignoring multiplicity) is likely to be significantly less than the total number of write-in votes (including multiplicity), since several people may write in the same candidate. Obtaining data on the number of distinct write-ins in actual elections is tricky, though. Published data for this election indicates a 0.6% write-in rate for each of the local races. The data, however, is incomplete. In particular, no data was available on write-ins in national races; also, the data available to us did not provide the number of distinct write-ins. If we make the assumption that the

[5]Precinct 213100001 is the precinct for the Claremont Hotel, site of the IEEE Symposium on Security and Privacy.

write-in rate is uniform across all DREs, we would expect each race to have fewer than 0.6% distinct write-ins. Given the uncertainty in elections, however, election administrators must be prepared for significantly more write-ins.

In Figure 3 we display a graph comparing the three constructions from Section 8[6]. The cost is almost entirely dependent on the maximum number of distinct write-ins per race that we wish to be prepared for: the greater the desired write-in capacity, the more PROM chips we need. In comparison, the cost for storing regular candidates (or repeated write-in votes for a single candidate) is negligible. Using random placement tables, the entire election can fit on a single 4MBit PROM chip, even with 50 distinct write-ins per race. If election workers use a copyover list instead, only two PROM chips are required to achieve the same capacity. Notice that this provides a tremendous degree of over-capacity: minus a denial of service attack, it would be rare to encounter a race that received 50 write-ins (which corresponds to half the expected number of voters), let alone 50 distinct write-in names.

It is interesting to note that even though the copyover list exhibits quadratic asymptotic behavior, it performs well in the voting application. Lexicographic hash chaining, even though it features better asymptotics, must over-provision space for many distinct values hashing to the same value; to obtain a suitably low chance of exhausting the space in one hash bucket induces a high constant value that is hidden in the asymptotic behavior.

The costs for using the random placement tables or a copyover list compare favorably with the cost of an optical scan ballot. Optical scan ballots can cost \$0.10 to \$0.30 per voter [18]; securely storing ballots on a DRE using our secure vote storage techniques cost less than \$0.05 per voter.

10 Related Work

Micciancio defined the basic problem of privacy for data structures and gave a construction of "oblivious trees" [13]. Naor and Teague extended the notion and gave several constructions, including the priority hash chaining construction that we build on [15]. Hartline et al. improved several data structures and studied relaxations of the history independence requirement [5]. Irani, Naor, and Rubinfeld showed that a Turing Machine with write-once polynomial space decides exactly the languages in the class P [6]; our PROM model differs in that multiple writes are allowed, so long as the new value can be obtained by flipping 1s to 0s. Shamos independently proposed using a hash table with unary counters for storing votes in a history-independent manner [17], although Shamos's scheme does not achieve perfect history independence. There is also a large body of work on implementing file systems with solid state storage, such as flash memory. Gal and Toledo survey this work, and Niijima gives an in-depth report on a flash memory file system that addresses engineering issues we discuss above [16, 4].

11 Conclusions

We have described constructions for data structures suitable for vote storage on electronic voting machines. These techniques may be useful for other applications as well. For example, "black box" recorders in airplanes have tamper-evidence requirements, as do odometers used to monitor truck drivers' compliance with rest laws. Our techniques exploit cheap commodity hardware, so that the consumables cost for an election is very reasonable. Our vote storage module provides a history-independent, tamper-evident, durable, and subliminal-free representation method to securely record ballots in an isolated module. We believe that a vote storage module with these properties will simplify the design of DRE voting systems and ultimately enhance the public's confidence in them.

12 Acknowledgments

We thank David Jefferson, Susan Hohenberger, abhi shelat, and Adam Stubblefield for helpful discussions and suggestions. We also thank the anonymous reviewers of the IEEE Symposium on Security and Privacy

[6]With the lexicographic chaining table, there is a small chance of bucket overflow. Therefore, for this scheme, we choose table and bucket sizes so that the probability of bucket overflow is no more than 2^{-30}.

and Philippe Golle. D. Molnar was supported in part by NSF CCR-0325311 and by a NSF Graduate Research Fellowship. T. Kohno was supported by NSF CCR-0208842, NSF ANR-0129617, and NSF CCR-0093337, and performed part of this research while visiting the University of California at Berkeley. N. Sastry was supported by NSF CNS-0524252 and by the Knight Foundation under a subcontract through the Caltech/MIT Voting Technology Project. D. Wagner was supported by NSF CCR-0093337 and CNS-0524252. We finally thank They Might Be Giants for their song entitled "Subliminal."

References

[1] R. Anderson and M. Kuhn. Tamper resistance - a cautionary note. In *The Second USENIX Workshop on Electronic Commerce*, pages 1–11, November 1996.

[2] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73. ACM Press, Nov. 3–5, 1993.

[3] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history independence. In D. Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 445–462. Springer-Verlag, Berlin, Germany, 2003.

[4] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[5] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. Rocke. Characterizing history independent data structures. In *ISAAC '02: Proceedings of the 13th International Symposium on Algorithms and Computation*, pages 229–240. Springer-Verlag, 2002.

[6] S. Irani, M. Naor, and R. Rubinfeld. On the time and space complexity of computation using write-once memory or is pen really much worse than pencil? *Mathematical Systems Theory*, 25(2):141–159, 1992.

[7] C. Karlof, N. Sastry, and D. Wagner. Cryptographic voting protocols: A systems perspective. In *Proceedings of the 14th USENIX Security Symposium*, pages 33–49. USENIX Association, Aug. 2005.

[8] A. M. Keller, D. Mertz, J. L. Hall, and A. Urken. Privacy issues in an electronic voting machine. In *WPES '04: Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, pages 33–34. ACM Press, 2004.

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

[10] M. Lepinksi, S. Micali, and a. shelat. Collusion-free protocols. In *STOC '05: Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, pages 543–552. ACM Press, 2005.

[11] A. Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 597–612. Springer-Verlag, Berlin, Germany, 2002.

[12] R. C. Merkle. A digital signature based on a conventional encryption function. In C. Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer-Verlag, Berlin, Germany, Aug. 16–20, 1988.

[13] D. Micciancio. Oblivious data structures: applications to cryptography. In *STOC '97: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 456–464. ACM Press, 1997.

[14] S. Microelectronics. M27C4001-10B1 OTP EPROM 4MBit data sheet, 2005.

[15] M. Naor and V. Teague. Anti-presistence: history independent data structures. In *STOC '01: Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, pages 492–501. ACM Press, 2001.

[16] H. Niijima. Algorithms and data structures for flash memories. *IBM Journal of Research and Development*, 39(5), September 1995.

[17] M. Shamos, March 2005. Post to VSPR mailing list. `https://lists.csail.mit.edu/pipermail/vspr-wg-ut/2005-March/000061.html`.

[18] VotersUnite.org. Comparing annual costs of DRE and optical scan systems, 2005. `http://www.votersunite.org/info/ComparingAnnualCostsDREvPBOS.pdf`.

[19] S. Weingart. Physical security devices for computer subsystems: A survey of attacks and defenses. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2000.

[20] Wikipedia. Antifuse. Visited February 17, 2006. `http://en.wikipedia.org/wiki/Antifuse`.

```
Unary Counter (U-CTR)
State (stored on PROM): $b$-bit bitvector $T[]$ (initialized to all 1 bits).
Implicit state: an integer $j$ (can be computed from $T$).

U-CTR.Initialize():
1. Set $j \leftarrow 0$.

U-CTR.Insert():
1. Clear bit $T[j]$.
2. Set $j \leftarrow j + 1$.

U-CTR.Finalize():
1. Return.
```

Figure 4. Algorithms for a durable, strongly history-independent unary counter. This data structure represents a multiset $m : \Omega \to \mathbb{N}$ where $|\Omega| = 1$. Note that to make the description clearer, we use j to refer to the implicit value of the unary counter; an actual implementation might compute j every time it is needed by scanning to find the first 1 bit in T. The tally can be easily reconstructed from the contents of the bitvector T.

A Data Structure Pseudocode

We now give pseudocode for each of the data structures described in Section 8.

```
Copyover List (CL)
State (stored on PROM): Table $T[][]$, containing size $n(n+1)/2$ blocks; $n$-bit bitvector commit[].
Implicit state: an integer $j$ (can be computed from commit).

CL.Initialize():
1. Set $j \leftarrow 0$.

CL.Insert($m$):
1. Let $m' = m$.
2. Let $(m'_1, \ldots, m'_{j+1}) = \mathrm{sort}(T[j] \cup \{m'\})$ be the result of lexicographically ordering the
   contents of the list $T[j]$ together with $m'$.
3. For $b = 0$ to $j$, do:
4.     Write $m'_b$ to $T[j+1][b]$.
5. Clear bit commit[$j+1$].
6. For $b = 0$ to $j-1$, do:
7.     Erase $T[j][b]$ by overwriting it with all-0s.
8. Set $j \leftarrow j+1$.

CL.Finalize():
1. For $a = j+1$ to $n$, do:
2.     For $b = 0$ to $a-1$, do:
3.         Erase $T[a][b]$ by overwriting it with all-0s.
4.         Clear bit commit[$a$].
```

Figure 5. State and algorithms for a durable, strongly history-independent copyover list of at most n items. We assume that every item in Ω fits in exactly one block. We use the notation $T[a]$ to refer to the ath "sublist" of the table T; the ath sublist is a blocks long. The notation $T[a][b]$ refers to the bth block in the ath sublist, for $1 \leq a \leq n$ and $0 \leq b < a$. The tally can be easily reconstructed from the contents of the PROM by simply reading the list entries.

```
Lexicographic Chain Table (LCT)
State (stored on PROM): Copyover lists $CL_1, \ldots, CL_n$; bitvector $D$; bitvector $K$.
Secret state (stored by vote storage module): PRF key $k$

LCT.Initialize():
1. Choose a PRF key $k$ at random.
2. Write $\mathcal{H}(k)$ to $D$.
3. for $i = 1$ to $n$, do
4.     Call $CL_i$.Initialize().

LCT.Insert($m$):
1. Let $i \leftarrow F_k(m)$.
2. Call $CL_i$.Insert($m$).

LCT.Finalize():
1. Write $k$ to $K$.
2. Securely delete $k$ from the vote storage module's state.
3. For $i = 1$ to $n$, do
4.     Call $CL_i$.Finalize()

LCT.Check():
1. If $\mathcal{H}(K) \neq D$, return false.
2. For $i = 1$ to $n$, do
3.     If $CL_i$.Check() returns false, return false.
4. return true.
```

Figure 6. State and algorithms for a durable, strongly history-independent lexicographic chain table. This construction uses a random oracle $\mathcal{H}$ and a PRF $F_{\cdot}(\cdot)$. The tally value can be easily reconstructed from the contents of the PROM by reconstructing the tallies from each copyover list.