Replayable Voting Machine Audit Logs

Arel Cordero David Wagner
University of California, Berkeley

Abstract

Audit logs are an important tool for post-election investigations, in the event of an election dispute or problem. We propose a new approach to logging that is designed to provide a record of all interactions between each voter and the voting machine. Our audit logs provide a comprehensive, trustworthy, replayable record of essentially everything the voter saw and did in the voting booth, providing investigators a tool for reconstructing voter intent and diagnosing election problems. We show how our design preserves voter anonymity and protects against vote-buying and coercion. We implement a prototype logging subsystem, built on the Pvote voting platform, and demonstrate that the approach is feasible.

1 Introduction

Elections do not always go smoothly. When problems occur, or if an election is disputed, it is important that the voting system preserve forensic evidence that can be used to investigate reported problems. Many commercial e-voting systems generate and maintain audit logs—records that describe the operation of a voting machine in an election—for this purpose. Unfortunately, the audit logs in currently deployed voting systems fall short in several respects: they often record only minimal information, omitting vital information that might be needed in a post-election investigation [20]; there is not always a clear pattern to what is logged [11]; there are few protections to ensure the integrity of the logs, so any failure of the voting software can also corrupt the audit logs [4, 10, 17]; and there is no guarantee that the logs contain an accurate or complete record of what happened on election day.

We study how to improve this situation. We show that it is possible to design audit log mechanisms that remedy these shortcomings. Our audit logs are more complete: we expect that in many cases they may help investigators diagnose reported problems, test hypotheses about the cause of those problems, check that votes were recorded and interpreted correctly, reconstruct voter intent (to a certain extent), and possibly even correct problems. For instance, our audit logs provide useful evidence of a voter's interaction with the machine, which may be helpful in reconstructing her intent. Because audit logs can potentially weaken the degree of voter anonymity a voting machine provides, they must be carefully designed to ensure they protect ballot secrecy. We show how careful design of the logging data structures can protect the secrecy of the voter's ballot and protect against vote-buying and coercion.

This work is partially inspired by the second author's involvement in a post-election investigation of the Sarasota Congressional District 13 election [20], where over 13% of DRE ballots recorded no vote in the CD13 contest. In that election, a number of voters alleged that the DRE never gave them a chance to vote in the CD13 contest or that their initial selection in the CD13 contest was not displayed on the DRE summary screen. One of the challenges in that investigation was that, to protect voter privacy, the DREs used in that election did not retain any information that would enable investigators to recreate exactly what voters did or did not see on the DRE screen. As a result, those allegations could only be checked through indirect means. If there had been some way to replay the sequence of screen images that each voter saw and the user interface actions the voter took in response to each screen, this would have enhanced our ability to investigate the cause of the undervote and to ascertain voter intent: for instance, we could have checked whether the CD13 contest was always displayed to every voter, and we could have checked whether there was any evidence that some voters' selections in the CD13 contest were not reflected accurately on the final summary screen. In this paper, we design a way to retain this kind of information, without violating the secrecy of the ballot.

1.1 Problem Statement

We study how the audit logging mechanisms in electronic voting machines should be designed. We want to generate and preserve a comprehensive, trustworthy set of electronic records that can be used to detect and diagnose many kinds of equipment problems and failures. These logs should record useful evidence of voter intent, preserve voter anonymity, and avoid interfering with any other requirements of the voting machine.

We want election investigators to be able to use these audit logs after the election to reconstruct the interaction that each voter had with the voting machine. We focus on designing audit logs that enable investigators to reconstruct everything the voter saw on the voting machine's screen, everything that the voter did (e.g., every location on the screen that the voter touched, every button that the voter pressed), and every ballot that was cast as a result of these actions. Audit logs would not normally be used to count the votes. Instead, the idea is that, in case of problems with the election or election disputes, it should be possible for election investigators to use the logs to reconstruct and infer, as best as possible, the voter's intent. Achieving this goal requires recording far more than today's voting systems.

Ideally, these audit logs would provide an *independent* way for investigators to verify that the voter's intent was recorded accurately, to correct any errors the machine may have made, and to gather evidence and test hypotheses about the possible causes of these errors. In practice, we cannot fully achieve the ideal of full independence: we do not know how to ensure that the audit log mechanism will be truly independent of the voting machine software. However, we seek to minimize the likelihood of failures that simultaneously affect both the records of cast ballots and the audit logs. In particular, we would like to be able to correct or detect many common kinds of failures, such as errors in recording the voter's selections, user interface flaws, ballot design problems, configuration errors, and touchscreen miscalibration.

However, some kinds of failures are out of scope for this paper. We assume the software that executes on election day matches the certified version, and that this software is free of malicious logic and backdoors; and we assume that the hardware is trustworthy, correct, and free of tampering. We make no attempt to detect violations of these assumptions, so our audit logs can not solve all problems (especially security issues that involve malicious insiders)—but we hope they will be useful in practice nonetheless.

Audit logs must not compromise ballot secrecy. This poses a significant technical challenge: the more information we record, the greater the risk that this might provide a way to link voters to how they voted, or that it might provide a way for voters to prove how they voted, sell their votes, or be coerced. Many obvious schemes have serious ballot secrecy problems.

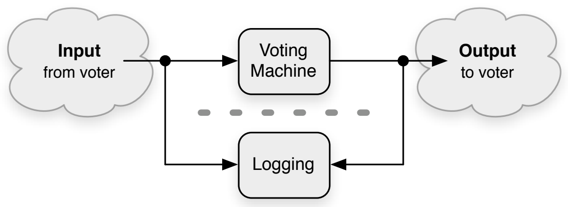

Figure 1: Conceptual problem statement. An audit log should record the I/O of a voting system as closely as possible to what the voter experiences.

We focus on building audit logs for machines that interact directly with a voter, namely, for DREs and electronic ballot markers (EBMs).

It would be useful if these audit log mechanisms could be deployed on existing voting systems, without hardware changes. Because many jurisdictions have recently deployed new e-voting systems, they may be reluctant or unable to replace their equipment any time soon. Therefore, we'd prefer a way to retrofit existing voting systems with better audit log mechanisms simply by upgrading their software. In practice, this restriction places limits on what we can accomplish, so we examine what is the best that can be done simply through software upgrades. We also investigate how future voting system platforms could better support trustworthy audit log mechanisms.

1.2 Our Approach

How do we improve the trustworthiness of the logging system? By isolating the logging subsystem from the rest of the voting machine code, we minimize the possible interactions between the logging code and the rest of the voting software. Also, we design the logging code to be as simple and robust as possible, in hopes that this will make it easier to get right, easier to verify, and easier to trust. As we shall see later, the logging subsystem can be dramatically simpler than the rest of the voting software.

How do we make audit logs more useful? Audit logs exist to record evidence of voting machines' operations. We propose that these logs should record the sequence of all I/O from the voting machine *as experienced by the voter*, or as close to that as possible. We record all outputs: if the voting machine has a LCD screen, we will record every image displayed on the screen; if it has a printer, we record all text that is printed. Similarly, we record all inputs: if the voting machine has a touchscreen input, we record the position of every touch the voter makes; if it has physical buttons, we record every press of these buttons; if it supports other kinds of input (e.g., for accessibility), we record those input events as well. Because this essentially captures all the ways the voting machine can communicate with the voter, and vice versa, and because the results of the voting machine (i.e., the final vote tallies) should be determined by the voters' interaction with the machine through this interface, I/O-based audit logs can capture useful evidence of voter intent.

We store these events in the order they occur, so that it is possible to replay the sequence of all interactions between the voter and the voting machine. We call these types of audit logs *replayable*, because the I/O can be replayed on an independent system without re-implementing any of the logic of the voting machine. As a result, election investigators can use this data to replay anonymized voting sessions and test hypotheses about the election results.

How do we address ballot secrecy? Any system that records votes in the order they are recorded endangers voter anonymity, because it allows an observer who notices the order in which voters cast their ballots and who has access to this electronic record to learn how each voter voted. We protect ballot secrecy by ensuring that the order in which voters vote is independent of the data associated with them. For each voter, we bundle up all of the audit log entries associated with that voter into a single record, so that there is one record per voter—and then we store these records in a random order. Moreover, we are careful to avoid recording the absolute time at which any voter interacts with the voting system or the length of time that it takes for a voter to vote, since these can also breach ballot secrecy.

How do we avoid interfering with the rest of the voting software? We must be sure that adding our logging subsystem to an existing voting machine will not disrupt or interfere with its operation. If the legacy voting software works correctly on its own, then adding our logging subsystem must not cause it to crash, deadlock, misbehave, or otherwise endanger the security, usability, reliability, or privacy of the voting machine. To achieve this goal, we rely upon hardware or software isolation mechanisms (e.g., memory protection or type-safe languages) to prevent the logging code from interfering with the code or state of the rest of the voting software; we expose only a narrow interface between these two software components; we strive to minimize the complexity of the logging subsystem; and we structure our logging code so we can demonstrate that its operations terminate successfully in bounded time.

2 Design

2.1 Simplicity

In our architecture, the voting machine contains two components: the *voting subsystem* (this includes the legacy voting software; it implements the user interface, voting logic, and vote-recording functionality, among other things) and the *logging subsystem* (which is responsible for storing log records in a private manner). Our logging subsystem presents a minimal interface to the rest of the voting machine. Essentially, we export only an *append record* function[1]. Compared to the complex requirements of a voting machine, this may represent a significant simplification.

2.2 Isolation

We assume that we have a trusted platform that provides some mechanism for isolation, and we use it to enforce the separation between the logging system and the voting machine. We have two requirements. First, to ensure that all logged events accurately represent what the voter experienced, the isolation mechanism must protect the logging subsystem in case the voting subsystem misbehaves or runs amok. Second, it must protect the voting subsystem from any negative effects of the logging subsystem. We do not want the introduction of our logging subsystem to reduce the overall security, privacy, or reliability of the voting machine.

Protecting the logging system. The isolation mechanism should prevent the voting subsystem from bypassing or fooling the logging subsystem. In particular, since we are interested in recording the I/O of the system, the logging system should accurately capture all voter-visible I/O, despite any efforts otherwise by the voting machine. This means that we must mediate the voting subsystem's access to I/O devices and ensure that a copy of all inputs and outputs visible to the voter are also sent to the logging subsystem. Also, the logging subsystem needs its own storage medium where the logs can be recorded, and the isolation mechanism must prevent the voting subsystem from reading or writing these logs. Finally, the voting subsystem must not be able to modify the private state or code of the logging subsystem.

Protecting the voting system. The isolation mechanism must prevent the logging subsystem from modifying the private state or the code of the voting subsystem. The voting subsystem needs its own storage medium

[1]Features for reading or deleting log entries should not be available to the voting machine's trusted election-day logic, and hence are omitted from its code base.

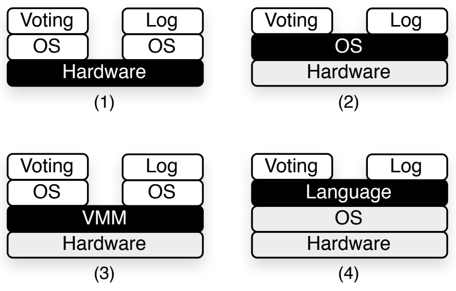

Figure 2: Four architectural choices for isolating the voting and logging subsystems from each other.

where electronic cast vote records and other data can be recorded, and we must prevent the logging subsystem from accessing that data.

Ways to implement isolation. We identify four different ways to meet these requirements (see Figure 2):

1. We could use hardware isolation, e.g., by running the voting subsystem on one microprocessor and the logging subsystem on another microprocessor and restricting their connectivity to each other and to I/O devices [16].

   The Prime III voting system works similarly. It physically routes the video and audio output of the voting machine through a VHS recorder [19, 5], and the video tape serves as a record or log of the voting session. Although now the physical security of the recorder and tapes must be trusted, this provides strong isolation. A misbehaving—even maliciously programmed—voting machine will have to struggle to avert the logging mechanism. Voter anonymity, however, remains an issue in Prime III, as we discuss below and in Section 5.

2. We could use OS isolation mechanisms. For instance, we could run the voting subsystem and the logging subsystem in two separate processes and rely upon OS memory protection to keep them separate.

3. We could use a virtual machine monitor, running the voting subsystem and logging subsystem in two different virtual machines [6]. This allows us to mediate all of their interactions with each other and with I/O devices using a small module integrated into the VMM [8].

4. We could use a type- and memory-safe programming language for isolation, relying upon these language features to ensure that one subsystem cannot tamper with the private state or code of the other subsystem.

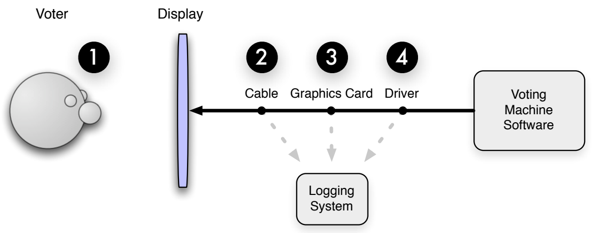

Figure 3: Four possible places where the video output signal could be recorded. All transformations to the video signal that occur after it is recorded and before it is seen by the voter can cause undetected inaccuracies in the audit log and hence must be trusted.

These approaches have different tradeoffs. For instance, hardware isolation may provide the greatest level of assurance, but it would require hardware changes to existing voting machines. Software isolation may be easier to retrofit, but it requires trust in the OS kernel, VMM, or language platform.

We note that the isolation mechanism does not need to be perfect to be useful. Even a suboptimal isolation mechanism can still be useful for detecting prevalent classes of important failures, like configuration or operator errors. Experience suggests that, for many election administrators, these problems are more palpable and urgent threats than the adversarial threats computer security experts often take for granted.

2.3 Crafting a replayable audit log

Capturing voter intent. As explained in Section 1.2, we are primarily interested in recording the I/O of the voting machine, *as experienced by the voter*. However, the extent to which we can do this is limited by where in the data path the I/O is recorded. Consider the example of video output. Figure 3 shows four places where the video signal could be recorded. With the exception of perhaps an external camera mounted near the eye-level of the voter (1), the video that is recorded will be transformed before the voter sees it. For instance, recording the analog video output from the video card (2), the digital bitmap rendered by the graphics card (3), or the screen images produced by the graphics device driver (4) would not differentiate between an image displayed on a clean screen, or one displayed on an obscured or damaged screen. Similarly, a signal recorded from the graphics card frame buffer (3) may not detect a disconnected cable or a fault in the hardware D/A converter, though it *could* detect bugs in the device driver or voting software that cause the wrong image to be displayed.

Voter anonymity. As one might imagine, what gets logged strongly affects the degree of voter anonymity of the system. Even without recording the voter's name, face, voice or any other identifying information, data from the voter's *interaction with the voting machine* may still be enough to identify her. For instance, if we store a timestamp with each log entry, and if the log entry can be linked to a record of the voter's votes, then this may be enough to link a voter with how they voted. This is especially unfortunate since common practices—and even the Voluntary Voting System Guidelines [1]—suggest timestamping all log entries. We protect voter anonymity by never associating timestamps with our log entries. A voting machine is always free to keep a separate supplemental log where all entries are timestamped as long as those entries reveal nothing about how the voter voted; we consider that out of scope for this paper.

More subtly, duration information can compromise voter anonymity. For instance, if each log entry records the amount of time it took for the voter to vote, and if each log entry can be linked to a record of that voter's votes, then an observer who notes how long it took for Alice to vote may be able to infer how Alice voted. Even partial information about the time the voter spends on a particular screen could potentially violate voter anonymity. We counter this threat by never storing duration information: the log record associated with a voter contains the sequence of screen images and input events but no information about the duration between these events. This does leave one residual risk: if the number of events is correlated to the time it takes for a voter to vote, this may endanger voter anonymity. We leave it to open work to evaluate this issue more thoroughly.

It is impossible to protect voter anonymity if the voting subsystem is malicious or buggy. Therefore, our anonymity goals are conditioned on the assumption that the voting subsystem is correct, non-malicious, and does not itself violate voter anonymity.

We address vote-buying and coercion in Section 2.5.

Video. Video—the sequence of screen images shown on the voting machine's display—is arguably the machine's primary channel of communicating with the voter. Ideally we would like to record this video in real time, so that election investigators can replay a "movie" of the video images that the voter saw. However, real-time video reveals vote duration, which is a problem for voter anonymity. Instead of recording real-time video, we record a frame of video only when the image on the screen changes. Because voting machines typically show static content that changes only in response to user input events—for instance, they normally do not display animations—this approach redacts timing information by concealing how long the voter spends on each screen.

This approach essentially consists of run-length encoding the video, and then omitting the run lengths. Unfortunately, this approach does remove some information that may be relevant to election investigators: for instance, if investigators are concerned that the voting machine took too long to respond to user inputs, our audit logs will not contain enough information for them to investigate this hypothesis.

We assume that the voting subsystem does not display the current time to the voter on the screen. Fortunately, it should be straightforward to modify legacy voting software to ensure that this is the case.

We do not record audio. Real-time audio recording has the same problem with duration as video. However, unlike a sequence of static screen images, audio output is inherently temporal. We have not found any clean solution to this problem. Consequently, in our design, we do not record audio outputs from the voting machine. We recognize that this may omit important information that would be useful in an election investigation, and we consider it an interesting open problem to eliminate this limitation.

Other issues. Other subtle issues can threaten voter anonymity. First, *input device preferences* may be correlated with voter identity, especially for less common accessibility devices. For instance, if an observer notices that Alice is the only voter to vote using a sip-and-puff input device, and if the input modality is recorded in the audit log, then it will be possible to identify the log entry that corresponds to Alice and hence learn how she voted. We currently do not have a good solution to this problem, so our approach currently fails to support accessibility.

Second, *externally controlled signals* might mark or identify audit logs. For instance, if the voting machine accepts speech input (presumably through voice recognition), external sounds or noise may be reflected in the audit log entries and thus may endanger voter privacy. We do not support this kind of voting interface; we assume that the user interface is such that all input events come from the voter, not from the environment.

2.4 Storing the logs anonymously

The way we store the logs—the data structure we use and the physical medium on which it is stored—also affects voter anonymity. To avoid revealing information about the order in which log entries were inserted into the log, we require the log storage unit to be *history independent* [13, 9, 12]. See Section 3.4 for the data structure and algorithms we use.

2.5 Vote-buying and coercion

It is important that we *prevent* voters from proving how they voted, even in the case where voters are colluding with a vote-buyer or coercer. While we cannot completely eliminate these risks, we can provide reasonable protection. In particular, we require that the logging subsystem avoid making the problem any worse than it already is with existing voting systems.

We only attempt to prevent vote-buying and coercion under the following threat model. We assume that the voting machine and all voting software is non-malicious (for a malicious voting machine can easily enable vote-buying). We are not concerned about attacks where the vote-buyer or coercer are in collusion with an election official or other insider. There are already many ways that a voter can prove how she voted to her election official: for instance, she can enter a pre-arranged unique string as a write-in in some contest, or she can enter an unusual pattern of votes in down-ballot races. Our audit logs do provide additional ways to do so, for instance by entering a special pre-arranged sequence of "next screen"/"previous screen" inputs, but this does not provide the voter with any extra power she does not already have. For these reasons, we assume that all election officials and other insiders are trusted not to cooperate in vote-buying or coercion schemes. Instead, we focus only on preventing vote-buying and coercion by outsiders, and we attempt to ensure that voters cannot prove how they voted to any outsider.

Despite the possibility of voter collusion, an audit logging system still must provide anonymity to the remaining voters. In other words, if a voter is not intentionally colluding *with an insider* to prove how they voted, we do not want their ballot or audit log to be linkable to their identity. We acknowledge that the increased information we record about a voting session reveals strictly more information that could be used to link it to a voter. For instance, an elderly voter with known poor motor reflexes may have a distinct pattern of voting that may indirectly appear in the audit logs. The extent to which this is a problem is a subject for future work, but we point out that this kind of information already exists in other forms of voting, such as the pressure and method of filling in bubbles, or the handwriting for a write-in candidate.

Ideally, we would prefer if all audit logs could be routinely released to the public, to enable any interested party to perform their own analysis on these logs. Unfortunately, a policy of releasing our audit logs to the public after every election introduces vote-buying and coercion risks.

One way to prevent vote-buying and coercion would be to treat our audit logs as privileged (not public) information. Under this model, election officials would protect the confidentiality of these logs and avoid disclosing them to untrusted individuals. In particular, audit logs would not be released to the public. This would suffice to ensure that voters cannot use the audit logs to mark their ballots and prove to an outsider how they voted, eliminating the incentive for vote-buyers or coercers to try to pressure voters. This strategy mirrors the requirement that electronic cast vote records must not be released in raw form to the public (lest they enable vote-buying or coercion via pattern voting or special write-ins); in other words, in this model, audit logs would be subject to the same confidentiality requirements as existing cast vote records. Somewhat surprisingly, this is sufficient to ensure that our audit logs do not make vote-buying and coercion any easier. In this model, our scheme would not harm the transparency of the voting system, but neither would it improve transparency: the logs would allow election officials to investigate election results, but not allow members of the public to perform their own investigation.

Another possibility would be to accept some risk of vote-buying and coercion in exchange for better transparency. For instance, officials might ordinarily treat audit logs as privileged, except that in the special case of a contested election, the logs might be released to the independent auditors, candidates, and their representatives. Ultimatey, the extent to which audit logs should be kept private depends on many factors, including the risk of vote buying and coercion in any particular region, so the handling of our audit logs may be a policy matter that is best left to local election officials who are familiar with local conditions.

3 Implementation

We prototyped our design by extending Pvote [21], a voting system written in Python. Pvote is already remarkably compact because of its use of a *pre-rendered* user interface. Unfortunately, a security review by five computer security experts suggested that, even for a system as small as Pvote, it is still difficult to be confident that voting software is free of bugs [22].

3.1 Isolation

Pvote is written in Pthin, a subset of Python that includes only a small number of language primitives. Our prototype relies on the memory- and type-safety of Pthin for isolation. In particular, this makes it easy to verify

```
INIT()
 1  $session\_id \leftarrow \{0,1\}^{128}$ uniformly at random
 2  $sequence\_id \leftarrow 0$

LOG($s_1, s_2, ..., s_k$)
 1  $s \leftarrow$ SERIALIZE($session\_id, sequence\_id, s_1, s_2, ..., s_k$)
 2  $sequence\_id \leftarrow sequence\_id + 1$
 3  ADD-RECORD($s$)
```

Figure 4: Algorithms to initialize a log record, and append entries to it, utilizing the ADD-RECORD($\cdot$) method of the HIDS (see Figure 9). SERIALIZE may use any method for serializing its inputs to a uniquely decodable string, e.g., using length-prepending.

that no function in Pvote can access or modify any part of the logging component's memory or instructions except through the logger's public interface, and vice versa. This interface supports only three publicly accessible operations[2]: INIT(), which begins a new log record for a voting session; LOG($\cdot$), which appends an event to the record; and COMPRESS($\cdot$), an optional helper function we discuss in Section 3.3. Algorithms for the first two operations are described in Figure 4.

3.2 The logging subsystem

We implemented the logging subsystem in 110 lines of Pthin code. An additional 100 lines of code support reading and reconstructing the logs so they can be replayed after the election (see Figure 5).

In our system, a log is a set of log records, each corresponding to a single voter. A log record is an ordered sequence of events. Each event is an ordered sequence of arbitrary-length byte strings. (See Figure 6.) Thus, we execute INIT() once for each voter at the beginning of their voting session to create a new log record, and we invoke LOG($\cdot$) at every voter-visible I/O event to append the event to that log record. It is up to the programmer to log events consistently. Fortunately, we have found that it is easy to manually inspect the code to verify that the logging functions are called at every relevant place in the Pvote code.

In our implementation, the first string of an event represents the type of event that occurred. The remaining parameters include data for that event. The events we log in our prototype are summarized in Figure 7. For instance, the place in the code that reads a touch event calls

```
Log.log( "touch", x, y )
```

where `x` and `y` are string representations of the integer coordinates (pixel offsets) where the screen was touched. Likewise, whenever the screen is updated, we record the new image that will be displayed on the screen, in raw bitmap form.

Video. As we discuss in Section 2.3, we do not record real-time video. Instead we record a screenshot of the display whenever the display image changes. Our implementation uses the Pygame [14] library to record the bitmap displayed to the voter. In particular, in the one place in the Pvote code where the display update method is called, we log the raw RGB pixel data of the complete displayed image.

3.3 Compression

Recording bitmap images can be memory and bandwidth intensive. For better efficiency, we experimented with compressing the bitmap data before logging it. We were interested in capturing this data exactly, so we considered two lossless compression schemes: run-length encoding, an encoding that replaces runs of identical bytes with a single byte followed by the length of the run; and the standard `zlib` encoding. For run-length encoding, we tested implementations in Python and C. For the zlib encoding, we used the standard zlib library of Python (written in C). While we do not rule out other forms of compression, we found run-length encoding particularly appealing for its simplicity, and its suitability for compressing screenshots containing large regions of the same color.

A drawback of compression is the complexity it adds to the logging subsystem. Libraries written in C additionally lack the memory- and type-safety properties of a pure Python implementation. In this case, we must treat compression as a trusted operation, another reason to prefer a simple algorithm like run-length encoding. Despite the drawbacks, our evaluation of compression found it very useful for improving the performance of logging video keyframes in our prototype (see Section 4).

3.4 History independent data structure

As we explain in Section 2.4, the order of the voting session log records must be anonymized. We design our log data format to be history independent:

> "A data structure implementation is *history independent* if any two sequences S_1 and S_2 that yield the same content induce the same distribution on the memory representation." [13]

[2]Python does not itself distinguish between public and private or protected methods, but we found it easy to verify by inspection of the code that only the "public" methods are ever called within Pvote.

	Election-day mission-critical functions	Supporting functions
`Log`	*Init. & logging:* 23 lines	*Reconstructing records:* 41 lines
`HIDS`	*Adding records:* 87 lines	*Listing records:* 39 + *Resetting:* 20 lines
Total	**110 lines**	**100 lines**

Figure 5: Lines of Python code, as counted by `sloccount`[18].

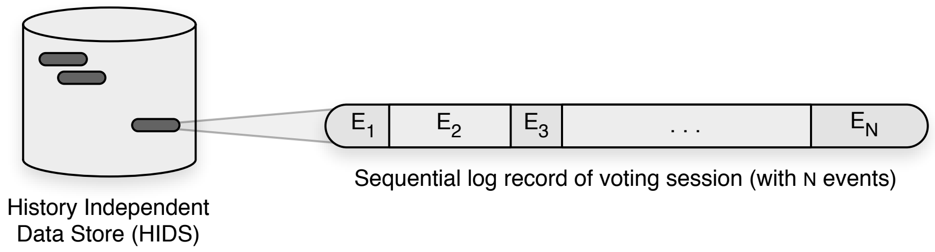
E_1
E_2
E_3
. . .
E_N
Sequential log record of voting session (with N events)
History Independent Data Store (HIDS)

Figure 6: A conceptual view of logging. A log record consists of an ordered sequence of log events. The log records are stored in the HIDS in a random order.

Event Type	Parameters	Explanation
`display`	*bitmap, width, height*	Records the raw RGB image whenever display changes.
`touch`	*x, y*	Logs coordinates whenever the voter presses the screen.
`key-press`	*key*	Logs any keypad button that was pressed.
`print`	*text*	Records data sent to ballot printer.
`ballot-definition-read`	*hash*	When ballot-definition is read, records its SHA-1 hash.
`reset-button`	-	Indicates a voting session is complete and will reset.

Figure 7: Pvote I/O events that our system logs.

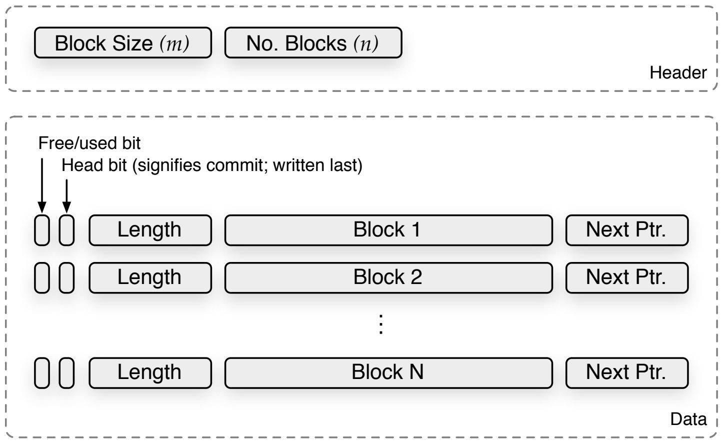

Figure 8: The basic data structure of the History Independent Data Structure (HIDS) used by our logging system. This structure is intended to map well to NAND flash memory, where each row of the data section maps to a page in flash.

We need our history independent data structure (HIDS) to provide an efficient insertion operation; support for large, variable-length records; and a simple implementation. However, the last requirement, *simplicity*, is our primary objective because we want to make the code easy to verify and to gain confidence in.

The HIDS we implement is shown in Figure 8. Given a block size m and the total number of blocks n, we initialize the data section of our HIDS to the known default value of the underlying memory (e.g., 0). To ensure that insertions will not fail, we require the HIDS to allocate a constant factor c more space than it expects to use. For expository purposes we assume $c = 2$. The data structure is a table of blocks plus metadata, which may be linked to other blocks to form a list. The HIDS is used to store the set of log records, and each record is inserted randomly into this structure as a linked list of blocks. By design, the only operations this data structure supports are ADD-RECORD$(\cdot)$, which adds a record randomly to the HIDS (see Figure 9), and LIST-RECORDS$()$, which reconstructs the contents of all the records in a random order. The latter operation is only done to retrieve the data after an election, so its performance is not as important. Deletion and lookup of records are intentionally not supported.

To insert records, we use a cryptographically-strong pseudo-random number generator (PRNG). Although a hash-based approach is also viable—and there are reasons it may be appealing[3]—we choose to work with a PRNG to minimize the complexity of the code.

Theorem 3.1. *Our HIDS is history independent.*

Proof. Our data structure is a table of blocks. Each record insertion writes data to one or more unused blocks. Each block is selected independently and uniformly at random from the remaining unused blocks. Suppose we insert r records (comprising k blocks) into a data structure of n total blocks. Let σ denote the state of the resulting data structure after these insertions. The probability of seeing the memory representation σ after these r insertions is

$$\Pr[\sigma] = \frac{1}{n} \times \frac{1}{n-1} \times \frac{1}{n-2} \times \cdots \times \frac{1}{n-(k-1)},$$

independent of the order in which in these records were inserted. Therefore, the data structure is history independent. □

Theorem 3.2. *If we allocate twice as much space as required ($c = 2$),* ADD-RECORD$(\cdot)$ *will fail with probability at most* 2^{-128}.

[3]Particularly, a hash-based approach with a unique, deterministic ordering of the records avoids possible subliminal channels [12].

```
ADD-RECORD($record$)
 1  $b \leftarrow \lceil \text{LENGTH}(record)/m \rceil$
 2  $data \leftarrow$ Split $record$ into list of $m$-sized blocks
 3  for $i \leftarrow 0$ to $b-1$
 4      do $loc[i] \leftarrow$ CHOOSE-FREE-BLOCK()
 5
 6  for $i \leftarrow 0$ to $b-1$
 7      do WRITE-SPECIFIED-BLOCK($i$)
 8
 9  $j \leftarrow loc[0]$
10  $length_j \leftarrow \text{LENGTH}(record)$
11  $head\_bit_j \leftarrow True$

WRITE-SPECIFIED-BLOCK($i$)
 1  $j \leftarrow loc[i]$
 2  $used\_bit_j \leftarrow True$
 3  $block_j \leftarrow data[i]$
 4  if $i < b-1$
 5     then $next\_ptr_j \leftarrow loc[i+1]$

CHOOSE-FREE-BLOCK()
 1  for $i \leftarrow 0$ to 127
 2       do $r \leftarrow \{0,1,...,n-1\}$ uniformly at random
 3           if not $used\_bit_r$
 4              then return $r$
 5  error "fail gracefully"
```

Figure 9: Algorithm for inserting records into our history independent data structure (HIDS). We assume the table is a constant factor larger than the maximum expected size, to keep the data structure sparse and allow for fast, simple insertion.

Proof. Assume that our HIDS contains space for n blocks of data, and that we insert records containing at most $n/2$ blocks in aggregate. At any time ADD-RECORD$(\cdot)$ is called, at least $n/2$ blocks are available, so the probability of selecting a non-empty block in lines 2–4 of CHOOSE-FREE-BLOCK$()$ is $p \leq 0.5$. The probability of failing to find a free block after 128 independent trials is $p^{128} \leq 2^{-128}$. □

We designed this data structure to be resilient in the face of hardware failures. In particular, consider a fail-stop model where at any point the storage device can fail, causing all subsequent reads and writes to fail with an error. We assume that all small writes are atomic and synchronous, with no re-ordering of reads or writes. As a reminder, we assume that the data storage device is originally initialized to a default value (e.g., 0), different from the encoding of *True*. With these conditions, we can prove:

Theorem 3.3. *Under this failure model, if* ADD-RECORD(·) *fails while trying to insert a record, the data structure will not be corrupted. Also, any partially inserted record(s) can be distinguished from complete records.*

Proof. The first thing that gets written to a block is its *used_bit*, which is flipped to *True* in line 2 of WRITE-SPECIFIED-BLOCK(·). In the failure model outlined above, this happens atomically. Once this bit is flipped, this block (even if not completely written) will not be selected by CHOOSE-FREE-BLOCK() again. Moreover, since we use the underlying storage medium in a write-once fashion, any failure that may occur during writing of this log record will not affect the other log records that have previously been successfully inserted into the data structure.

What we need to show then is that an incomplete block, or a block that is part of an incomplete record (i.e., part of a linked list of blocks that was never committed), will be distinguishable from a block from a complete record. The final line of ADD-RECORD(·) sets $head\_bit_j \leftarrow True$ for the first block of the record. Under the assumptions given earlier, this happens atomically, after the previous operations have completed. A complete record is therefore recognizable because it starts with a "head block", i.e., a block with *head_bit* set. Blocks that are part of an incomplete record (including incomplete blocks) will be "head-less" and therefore distinguishable. □

Physical storage. The history independence of the device that stores our HIDS must also be considered. For instance, even though our data structure is write-once, the file system on which it is stored may for its own reasons decide to rearrange the physical memory, while maintaining the appearance of a consistent and stationary logical address space. In fact, file systems optimized for flash, our target storage medium, will do this for the sake of reliability and performance [7]. We are not aware of any existing electronic voting system that takes this into account. For example, the Sequoia voting system attempts a history independent data structure for storing vote records. The storage device, however, is a commodity removable Compact Flash card running a standard file system not designed for history independence [4].

Our data structure is designed so it can be stored directly on NAND flash memory, in lieu of a file system, bypassing the flash translation layer (FTL) normally responsible for mapping logical addresses to physical ones to provide wear-leveling and other useful functions. We discuss below how we compensate for this.

Our data structure maps onto NAND flash in the following way. Each row of the data section (containing $used\_bit_j$, $head\_bit_j$, $length_j$, $block_j$, and $next\_ptr_j$) will be stored in its own page in flash (the block size m would be chosen accordingly). To make this work well, we have designed the HIDS to meet the following requirements: the data structure should modify storage in a write-once fashion (since re-writing is slow and creates the possibility that failures could corrupt data structure invariants); the data structure should take into account the reduced reliability of writing data in flash (the flash translation layer normally accounts for this with bad block mapping techniques [7]); and the data structure should account for wear-leveling (again the flash translation layer normally accounts for this).

It is easy to verify that our data structure is write-once in the sense that a bit that has been programmed to 1 or 0 is never modified. However, NAND flash is usually programmed a page at a time. By writing the first block of a record last (along with its *head_bit*), we get a data structure that is write-once with page granularity.

Writing directly to flash exposes us to common reliability problems normally handled by the flash translation layer. We can make up for this by keeping a list of bad blocks to avoid[4] that is fixed and static for the duration of an election, and by using error correcting codes (ECC). NAND flash usually provides a little extra memory per page specifically for ECC. Instead of maintaining a dynamic bad block map, and moving blocks around when they fail[5], we can use ECC to detect and correct single-bit errors in place. After the election, we recover the original stored values and update the static bad block list.

The final common function of the flash translation layer is to perform wear-leveling, a function that ensures the distribution of writes is spread out over many blocks and not concentrated on a few. Because our data structure is write-once, and writes are uniformly distributed over its memory space, our data structure implicitly wear-levels.

3.5 Authentication and integrity of the log

There is an orthogonal—but important—issue to our discussion of trustworthy replayable audit logs, and that is ensuring that the logs cannot be tampered with and that they are authentic. We consider this out of the scope of our paper, though we point out two possible approaches. First, one could imagine the voting machine digitally signing the entire data structure following the election. This would not protect against mid-day attacks, but it

[4]Defective blocks in NAND flash are typically identified at the time of manufacturing. Blocks may also fail in operation, for which a bad block map is usually maintained.

[5]Dynamic bad block mapping would typically copy bad blocks to new locations when an error is detected and update the bad block map. However, either of these actions could unwittingly reveal information about the history of the data structure.

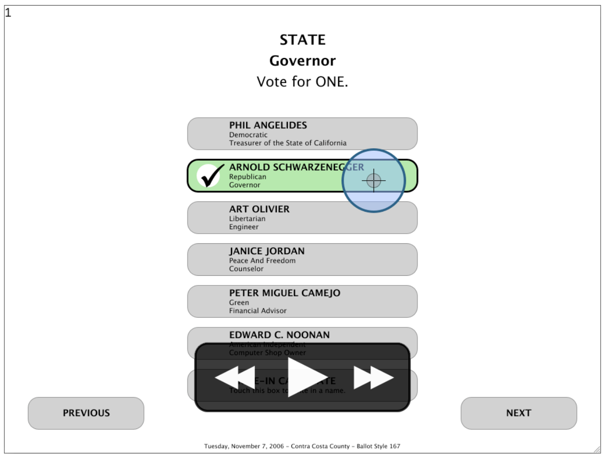

Figure 10: A screenshot of our replay mechanism showing a selection screen previously shown to a voter. The cross-hair indicates the location of a touch event in the box for Arnold Schwarzenegger.

```
SWITCHING TO RECORD 2 of 3.
Record 2 has 62 log entries.
0  ['hash', '\x9f\x81eJ\x05\xa9\xe8QN\xe7}#...']
1  ['display_changed', 'x\x9c\xec\xddu\xb8\x15...', '1024', '768']
2  ['key', '306']
3  ['key', '306']
4  ['touch', '382', '220']
5  ['touch', '573', '596']
6  ['touch', '805', '692']
7  ['display_changed', 'x\x9c\xec\x9du\x9cU...', '1024', '768']
8  ['touch', '90', '682']
9  ['display_changed', 'x\x9c\xec\xddu\xb8...', '1024', '768']
10 ['touch', '797', '723']
11 ['display_changed', 'x\x9c\xec\x9du\x9c...', '1024', '768']
12 ['key', '54']
13 ['touch', '621', '575']
14 ['touch', '857', '708']
15 ['display_changed', 'x\x9c\xec\xddyx\x14...', '1024', '768']
16 ['touch', '634', '300']
17 ['display_changed', 'x\x9c\xec\xdd\x07X...', '1024', '768']
18 ['touch', '624', '370']
 ⋮
```

Figure 11: An example of recorded events that can later be replayed. Within a `record` (a voting session), the `entries` are sequential. However, the order of records is independent of the actual order of voters. Note that the `display_changed` entry includes as a parameter a serialized bitmap image (truncated in the figure) of the entire screen as shown to the voter.

could help reduce chain-of-custody concerns. Alternatively, one could use history-hiding, append only signatures [3] to sign each block of data as it is entered into the HIDS, limiting the extent of a mid-day attack.

3.6 Replay Mechanism

The recorded I/O should be descriptive enough that it is possible to replay the logs without any knowledge of the voting machine's design and without re-implementing any of its logic. Our implementation is similar to a video player, in that the user may select any one of the voting sessions (not knowing when the session took place) and step or play through all the events (see Figures 10 and 11). The replay program then renders images on the screen and displays the location of every touch on the touchscreen.

4 Evaluation

4.1 Setup

We evaluated our implementation on a ballot definition based on the November 7, 2006 General Election of Contra Costa County, California[6]. This example ballot definition included only five contests from that election. The five contests are presented to the voter on 12 distinct screens, linked by "next" and "previous" buttons. There are also separate screens for entering write-in candidates.

We use a 1GB history independent data structure, allocated as 524,288 blocks, each 2KB long. We log the events shown in Figure 7, and we compress the video images using run-length encoding. We used a 1.83 GHz Intel Core Duo processor MacBook Pro running Mac OS 10.5 with 1.5GB 667MHz DDR2 SDRAM and an 80GB Toshiba Serial-ATA hard drive for our tests.

4.2 Performance measurements

We tested the performance of our prototype implementation on this ballot definition by casting a number of test votes using our code. We have not conducted a formal user study, but we expect these figures to be useful indicators of the viability of our approach.

For this ballot definition, our system records on the order of 100 separate I/O events per voting session. These events occupy on the order of 1500–2000 blocks in the HIDS (using run-length encoding), which amounts to 3–4MB of flash memory used per voting session. The majority of this data is image data because every change on the screen requires us to record a 2MB screenshot of the display. As a result, although most events use only a small fraction of the 2KB block they are written to, the total space consumption is dominated by the display events.

These measurements can be used to estimate the amount of flash memory that will be needed, as a function of the number of votes cast. We assume that the size of the storage medium will be conservatively provisioned to ensure that the HIDS does not become more

[6]This ballot definition is included with version 1.0b of the Pvote source code.

Average time to log one image

	CPU time to compress	Estimated insertion time	Resulting size after compression		Lines of C code to implement	Total estimated time
No compression	0 ms	384 ms	2,304 KB	100.00%	**0 lines**	**384 ms**
Zlib compression	73 ms	8 ms	44 KB	1.92%	**7734 lines**	**81 ms**
Run-length enc.	16 ms	19 ms	116 KB	5.05%	**101 lines**	**36 ms**

Figure 12: A comparison of average latencies incurred by compressing and storing one screenshot image in our prototype implementation. Images are 1024x768 pixel RGB images, about 2MB each uncompressed. The total time to log an image is estimated as the sum of the CPU time to compress and the time to insert the compressed image into the HIDS. The estimated insertion time is a function of the number of random-access reads and writes required to insert the image data into the HIDS. For evaluation we assume a cost of 25μs for random-access reads and 300μs for random-access programming to a 2KB block, representative of what might be expected in a NAND flash implementation. The third column shows the average size of a compressed screenshot, and the fourth column shows the compression ratio of the compression scheme. The number of lines of C code required to implement compression is also shown to give the reader a sense of the relative complexity of compression.

than 50% full. With these parameters, a 1GB HIDS can accommodate approximately 150 voters without exceeding the 50% capacity limit, for an election with a similar number of contests as the one we tested. Our experience is that it is rare for a single DRE to process more than 100–150 voters on election day, due to limits on how quickly voters can make it through the voting process. Consequently, we expect that 1GB of flash memory should suffice for an election with a similar, five-contest ballot definition.

We also measured the latency introduced by our logging subsystem, to evaluate whether it would affect the responsiveness of the user interface. Our measurements showed that the total latency is dominated by the time it takes to store screenshots, which even after compression are 50–100 times the size of other logged events. We found that the time to store a screenshot in the audit log can be attributed almost entirely to two factors: the CPU time to compress the raw bitmap image, and the time to read and write blocks on non-volatile storage. We measured the CPU time for compressing screenshots on our implementation; results are shown in the first column of Figure 12. Also, we found that inserting a RLE-compressed screenshot to the audit log requires about 65 random-access page reads and writes to flash memory, on average. Based on an expected latency of 25μs per random-access read and 300μs for programming a 2KB page, we estimated the flash-related latency that our implementation would incur. The second column of Figure 12 shows these flash latency estimates for each compression method[7], and the last column shows our prediction of the total latency introduced by our logging subsystem per user interface event. For instance, when using RLE compression, we expect that our logging subsystem will introduce less than 40ms of latency per user interface event. We do not expect this to noticeably affect the responsiveness of the voting machine's user interface.

To cross-check these estimates, we also validated these latency estimates by measuring the performance of our implementation on a laptop, using a hard disk instead of a flash device for nonvolatile storage. Of course, seeks are much slower on a hard disk, so one would expect this to be appreciably slower as a result. After correcting for the difference in seek times, the measured performance on our laptop is consistent with our estimates in Figure 12.

4.3 Practicality and cost

Our experimental setup includes only five contests. The actual election had 115 different contests with on the order of 40 contests shown to any single voter, or about eight times the size of our test ballot. If we extrapolate proportionally from the performance estimates above we would expect to need 8GB of storage for a machine servicing fewer than 150 voters. If we further assume NAND flash costs \$10 per GB, this comes out to a memory cost of \$80 per machine or around 53 cents per voter. Because the memory is reusable, this cost could be amortized over multiple elections. While this is just a rough estimate, it indicates that the cost of this scheme is not outright prohibitive. Given the declining cost of storage, we expect that replayable audit logs will become more affordable in the future.

While thinking of ways to reduce costs, we considered recording the complete voting sessions of only a random sample of voters. While the resulting audit logs could certainly be useful to detect and analyze many threats

[7]These estimates are measured in the best case, when the HIDS is empty. In worst-case conditions, where the HIDS is up to 50% full, the number of reads will increase by at most a factor of $2\times$, while all other contributions to latency remain unchanged.

and failure modes, this comes at the expense of a complete picture of the election day events. There is a qualitative difference between explaining an argument to the public based on complete data, versus an equally strong argument based on statistics.

4.4 Discussion of goals

We achieve a usable, replayable logging system in a small number of lines of code. The algorithms chosen—and our design choices in general—prioritize simplicity over efficiency and features, for the purpose of increasing the trustworthiness of the code and design. Our use of language-based isolation, however, is suboptimal even though it may still be useful for a large class of problems, particularly those involving misconfiguration or operator error. Hardware-based isolation, on the other hand, could result in a strong replayable auditing system.

Our system records all the relevant I/O Pvote provides to a voter except audio. We leave audio to future work because of our concern with duration and anonymity. Our approach of recording only screen images results in data that captures useful evidence of voter intent, while removing precise duration information. It is less clear how to do something similar for audio, because audio is inherently temporal.

5 Related Work

The two most closely related works we know of are Prime III [5] and the independent audit framework of Garera et al. [8]. Prime III is a voting system especially designed to accommodate voters with visual or aural impairments. In Prime III, all video and audio from the voting machine is copied to a VHS or DV recording device as it is displayed to the voter. This ensures that the audit logs are independent of the proper functioning of the software in the voting machine. However, Prime III's electronic records reveal significant information about the time, order and duration of votes cast and thus endanger voter anonymity. In addition, Prime III records only output from the voting machine, but not inputs from the voter, such as the location where voters touch the screen. Our work extends their approach by recording all I/O experienced by the voter, and by better protecting voter anonymity.

The independent audit framework approach of Garera et al. monitors the behavior of the voting machine by analyzing its video output in real time. They use computer vision techniques to infer important state transitions (such as a candidate being selected or a vote being cast) and then they log these inferred transitions. In comparison, our system does not try to analyze this data in real time, but rather logs all I/O so that this data can be analyzed after the election. They use a virtual machine monitor to isolate the monitoring system from the voting machine. While their approach is good for voter anonymity because it does not record I/O, it does not save as much evidence for post-election investigations.

Ptouch, a predecessor to Pvote, takes a step in the direction of replayable audit logs [21]. Ptouch does not record any I/O during the process of voting and does not record full screen images, but it does introduce the idea of recording data exactly as it is seen by the voter. For each candidate selected by the voter, Ptouch records a bitmap image of the candidate's name (as seen and selected by the voter during the voting process) in the electronic cast vote record.

The Auditorium project developed techniques for ensuring the robustness and integrity of event logs using locally networked voting machines [15]. Their work studies how to log data, while we examine the question of what to log, so their work is complementary.

Molnar et al. [12] introduced the notion of history independence to vote storage on voting machines.

A review [20] of voting machine firmware used in Sarasota County motivated our study of audit logs that capture user behavior in greater detail than found in current voting systems. In that election, some voters alleged that the CD13 contest was never displayed to them, that their initial selection did not appear on the confirmation screen, or that the voting machine did not recognize their attempts to select one candidate in that contest. Unfortunately, the audit logs available in that election were limited. If voting machines in Sarasota County had been equipped with the kind of audit log mechanism proposed in this paper, investigators would have had considerably more evidence to investigate these allegations and would have been able to replay voter sessions to see whether these allegations were accurate.

Bellovin first proposed the idea of full-interaction audit logs to us in private communication in 2004 [2]. Tyson later independently proposed a similar concept, motivated by his work in the Sarasota County voting review [20]. This paper explores this concept in greater detail.

6 Conclusion

We propose a method for recording reliable audit logs directly that record the interactive behavior of the voting machine *as it is experienced by the voter*. We call these *replayable* audit logs, and we show they can be generated by recording data directly from the I/O of the voting machine. We also show how to protect ballot secrecy by logging frames of video only when the image changes and by protecting the confidentiality of audit logs. As a result, this approach allows useful evidence of voter intent to be logged, while protecting the anonymity of the voter. Our prototype implementation demonstrates the practicality as well as limitations of the approach.

Acknowledgments

Steve Bellovin and Gary Tyson first suggested this research direction to us, and we gratefully acknowledge their contributions. We also thank Juan Gilbert, Prabal Dutta, Adrian Mettler, David Molnar, and the anonymous reviewers for their helpful feedback. Thank you also to Shereen Daly for her exceptional patience and kind moral support of the first author. This research was supported by NSF CNS-0524745.

References

[1] Voluntary voting system guidelines, 2005. `http://www.eac.gov/voting%20systems/voting-system-certification/2005-vvsg`.

[2] BELLOVIN, S. M. Personal communication, August 2004.

[3] BETHENCOURT, J., BONEH, D., AND WATERS, B. Cryptographic methods for storing ballots on a voting machine. In *In Proceedings of the 14th Network and Distributed System Security Symposium* (2007), pp. 209–222.

[4] BLAZE, M., CORDERO, A., ENGLE, S., KARLOF, C., SASTRY, N., SHERR, M., STEGERS, T., AND YEE, K.-P. Source Code Review of the Sequoia Voting System, July 2007. `http://www.sos.ca.gov/elections/voting_systems/ttbr/sequoia-source-public-jul26.pdf`.

[5] CROSS, E. V., ROGERS, G., MCCLENDON, J., MITCHELL, W., ROUSE, K., GUPTA, P., WILLIAMS, P., MKPONG-RUFFIN, I., MCMILLIAN, Y., NEELY, E., LANE, J., BLUNT, H., AND GILBERT, J. E. Prime III: One Machine, One Vote for Everyone. In *On-Line Proceedings of VoComp 2007* (July 2007). `http://www.vocomp.org/papers/primeIII.pdf`.

[6] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2003).

[7] GAL, E., AND TOLEDO, S. Algorithms and data structures for flash memories. *ACM Computing Surveys 37*, 2 (2005).

[8] GARERA, S., AND RUBIN, A. D. An independent audit framework for software dependent voting systems. In *14th ACM conference on Computer and Communications Security* (2007).

[9] HARTLINE, J., HONG, E., MOHR, A., PENTNEY, W., AND ROCKE, E. Characterizing history independent data structures. In *International Society for Analysis, its Applications and Computation* (2002).

[10] INGUVA, S., RESCORLA, E., SHACHAM, H., AND WALLACH, D. S. Source Code Review of the Hart InterCivic Voting System, July 2007. `http://www.sos.ca.gov/elections/voting_systems/ttbr/Hart-source-public.pdf`.

[11] KOHNO, T., STUBBLEFIELD, A., RUBIN, A., AND WALLACH, D. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy* (2004).

[12] MOLNAR, D., KOHNO, T., SASTRY, N., AND WAGNER, D. Tamper-Evident, History-Independent, Subliminal-Free Data Structures on PROM Storage -or- How to Store Ballots on a Voting Machine (Extended Abstract). In *IEEE Symposium on Security and Privacy* (2006).

[13] NAOR, M., AND TEAGUE, V. Anti-persistence: History independent data structures. In *Symposium Theory of Computing* (2001).

[14] Pygame. `http://www.pygame.org/`.

[15] SANDLER, D., AND WALLACH, D. S. Casting votes in the auditorium. In *USENIX/Accurate Electronic Voting Technology Workshop* (2007).

[16] SASTRY, N., KOHNO, T., AND WAGNER, D. Designing voting machines for verification. In *USENIX Security Symposium* (2006).

[17] WAGNER, D., CALANDRINO, J. A., FELDMAN, A. J., HALDERMAN, J. A., YU, H., AND ZELLER, W. P. Source Code Review of the Diebold Voting System, July 2007. `http://www.sos.ca.gov/elections/voting_systems/ttbr/diebold-source-public-jul29.pdf`.

[18] WHEELER, D. A. SLOCCount User's Guide, 2004. `http://www.dwheeler.com/sloccount/sloccount.html`.

[19] WILLIAMS, P., CROSS, E. V., MKPONG-RUFFIN, I., MCMILLIAN, Y., NOBLES, K., GUPTA, P., AND GILBERT, J. E. Prime III: Where Usable Security and Electronic Voting Meet, February 2007. `http://www.usablesecurity.org/papers/primeIII.pdf`.

[20] YASINSAC, A., WAGNER, D., BISHOP, M., DE MEDEIROS, T. B. B., TYSON, G., SHAMOS, M., AND BURMESTER, M. Software Review and Security Analysis of the ES&S iVotronic 8.0.1.2 Voting Machine Firmware, February 2007. `http://election.dos.state.fl.us/pdf/FinalAudRepSAIT.pdf`.

[21] YEE, K.-P. *Building Reliable Voting Machine Software*. PhD thesis, UC Berkeley, 2007.

[22] YEE, K.-P. Report on the Pvote security review. Tech. Rep. UCB/EECS-2007-136, EECS Department, University of California, Berkeley, Nov 2007.