Designing for Audit: A Voting Machine with a Tiny TCB

Ryan W. Gardner
Johns Hopkins University
Baltimore, Maryland
ryan@cs.jhu.edu

Sujata Garera
Johns Hopkins University
Baltimore, Maryland
sgarera@cs.jhu.edu

Aviel D. Rubin
Johns Hopkins University
Baltimore, Maryland
rubin@jhu.edu

ABSTRACT

Thoroughly auditing voting machine software has proved to be difficult, and even efforts to reduce its complexity have relied on significant amounts of external code. We design and implement a device that allows a voter to confirm and cast her vote while trusting only 1,034 lines of ARM assembly. The system, which we develop from scratch, supports visually (and hearing) impaired voters and ensures the privacy of the voter as well as the integrity of the tally under some common assumptions. We employ several techniques to increase the readability of our code and make it easier to audit.

1. INTRODUCTION

Electronic voting has become the instrument of democracy in many parts of the world as a result of historic dilemmas stemming from the paper ballot [8, 12] and a perceived voter preference for touch-screen machines [45]. However, in the last 6 years, a multitude of studies have analyzed the security of electronic voting devices used in elections [27, 23, 24, 17, 7, 25, 4, 33, 48, 9, 15, 35], and each has found significant security flaws in their software. Not only were security weaknesses found upon the first review of the software for each system, but new flaws have been found and existing vulnerabilities have persisted through patches, revisions, and attempted fixes. Consequently, it has become clear that it is extremely difficult to exhaustively audit or ensure security guarantees of the voting software used in current elections.

While the assurances regarding the machines' security are limited, the threats against voting systems are vast. Accidental flaws and vulnerabilities present serious hazards as with most systems, but intentionally inserted malicious functionality, backdoors, and bugs also represent dangerous possibilities in voting. Furthermore, the adversary has tremendous motivation. The president of the United States, for example, is often considered the most powerful person in the world [1], and the winners of many elections have much to be gained. For these reasons, we need voting systems that allow people to assure themselves that the systems are properly designed and constructed without vulnerabilities or backdoors.

Several researchers have made significant progress toward meeting this goal. Two of the most notable works are those of Yee *et al.* [51, 50, 49] and Sastry *et al.* [40]. Yee *et al.* reduce the complexity of voting software by prerendering the electronic ballot design [51] and write a full voting machine application, Pvote, in only 460 lines of original Python code [50, 49]. It is clearly more feasible to audit and ensure the robustness of such a small piece of software than that of the Diebold AccuVote-TSX, for example, which contains over 65,000 lines of C++ [7], or the Sequoia Edge, which consists of over 124,000 lines of C [4]. Sastry *et al.* physically separate modules of a voting system in hardware to allow security properties to be verified more easily by examining the individual components [40]. Their prototype contains only 5,085 lines of trusted code. While both of these studies make significant advances in the state of voting machine software, they also rely on the integrity of several large pieces of critical, external code, including libraries, operating systems, compilers, and interpreters, and 5085 lines remains a fairly large number for audit.

Carefully auditing *all* components that contribute to the development and execution of a piece of software that casts votes is essential. Consider pygame [34], for example, a game development library that provides the multimedia functions of Pvote. This library contains 28,660 lines of code and numerous bugs have been reported on it including segmentation faults and other crashes that may have resulted from inappropriate memory accesses.[1] At the same time, bugs in the pygame library may allow exploitation of the Pvote process the same as bugs in the code written specifically for Pvote itself. Similarly, along with operating systems and code interpreters, the use of a compiler enables potential attack vectors to the voting application as the compiler could create a vulnerable or malicious executable [46].

In this work, we create a new vote casting system that drastically decreases the amount of software that must be trusted. Our fundamental approach is twofold. First, we utilize the idea of Frogs [6] and create a separation between the device on which voters confirm and cast their votes and the hardware they use to select the votes. Secondly, we build the code for the confirmation and casting device from the foundation up. That is, we compose every byte of code or data that is written to the device, and we eliminate the need to rely on the integrity of a compiler by writing purely in assembly language. We significantly reduce the complexity of this confirmation device by taking an approach similar to that of prerendering [51, 50]. We push all ballot presentation processing to earlier stages of the voting process, before anything is finalized and malicious activity may be detected. Then, we defer all interpretation and tallying to later stages, which can be public and redundant. We also decrease the code needed for cryptography by using constructs that are built on the same arithmetic operations. As a result, we create a very simple "present, anonymize, and record" system that executes what seems to be the most consequential stage

[1] http://pygame.motherhamster.org/bugzilla/

of voting, the step where the vote is actually cast.

In addition to reducing the code for the voting system itself, writing exclusively in assembly is also advantageous as it eliminates the need to trust a compiler. In particular, assembling has two relevant differences from compiling: (1) Assembling is (mainly) a one-to-one function, and (2) it is simple. For these reasons, we can eliminate trust in any one assembler or system. People can create and use a variety of assemblers and compare their output for the voting software until a consensus is reached on a correct, executable binary. At the same time, assembly language also introduces challenges over higher-level languages in that it is generally more difficult to audit and review. We mitigate this risk by abstracting primitive actions with self-contained, individually verifiable macros that provide higher-level interfaces to data and functions. We also implement a simple form of data safe-typing so memory is accessed by variable name and addresses are checked at run-time.

Our currently implemented device is designed to work in conjunction with present-day voting hardware. It allows voters to confirm and cast their votes after selecting them using any machine that can write to a smart card, such as the Premier (formerly Diebold) AccuVote-TSX (touch screen). However, ultimately we envision that our work would be combined with that of Sastry *et al.* [40] in the creation of new voting devices where the separation of a selection and casting device would be transparent to the voter. Our device supports visually impaired voters, prevents tampering with stored ballots, and ensures that voters can only cast one ballot, under some common assumptions. In total, its code is written in 1,034 lines of ARM assembly.

2. RELATED WORK

Several papers have examined ways of making voting machines easier to audit and verify. Bruck *et al.* are the first to suggest separation of vote creation and vote casting systems with Frogs [6]. Their architecture increases modularity of the voting machine and reduces the complexity of the code that casts a ballot. Sastry *et al.* utilize hardware isolation techniques to separate different functions of a voting machine with distinct, physical components [40]. Each module, simpler than an entire voting machine, can then be verified and audited individually. Unlike the Frogs architecture [6], their system does not alter the traditional voter experience. The trusted code base (TCB) of their prototype implementation consists of 5085 lines of code, most of which is implemented in Java although the underlying operating system and standard libraries are not considered in this measurement, as the authors mention.

Yee *et al.* [51] propose separating the preparation of the user interface from the voting machine and implement Pvote, an application that presents the voter with a prerendered electronic ballot while accepting touchscreen input. In a subsequent paper, Yee extends this system with audio and input features that provide an accessible interface for voters with visual disabilities [50]. Pvote is written in 460 lines of original, Python code. However, it also utilizes external libraries for multimedia functions and hash computation and requires an operating system for hardware communication.

Previous research has also explored other techniques for increasing the security of voting machines. Sandler *et al.* design and implement VoteBox, a system that leverages a variety of research techniques to provide a verifiable voting platform [37]. In particular, it maintains highly redundant, tamper-evident audit logs, harnessing methods that the authors also introduced with the Auditorium framework [38]. VoteBox utilizes prerendered user interfaces [51, 50] to provide a DRE-like voter experience while reducing the amount of code necessary to do so. It further achieves end-to-end verifiability using cryptographic techniques and by providing the voter an option to challenge the voting system [2, 3].

Cordero and Wagner propose a method for logging all interactions between a voter and the voting machine to allow for post-election auditing [11]. Their system builds on Pvote and relies on the type and memory safety properties of Pthin for ensuring isolation of the logging and voting software. Garera and Rubin implement an independent audit framework for the Diebold DRE voting machine [19]. Unlike the system by Cordero and Wagner, theirs uses hardware virtualization to isolate software. In addition, it aims at maintaining an independent count of the vote totals and achieves this through real time image processing of the voting machine screens that the user interacts with.

To increase the assurances provided by an audit, in our implementation, we complement the ARM assembly language with a small framework that provides a weak form of type safety for variable accesses. The notion of types in assembly language has been explored previously by Morrisett *et al.* [31, 20, 30, 29] who extend standard assembly and create TAL to provide a variety of built-in types and a set of rules associated with each type. For example, TAL provides an abstraction for code labels and rules that check that control transfer only occurs to instances of the code label type. The TAL constructs are capable of translating several high-level programming features such as arrays, modules, exceptions, abstract data types and others. Although TAL is significantly more powerful and efficient than our framework, it is notably more complex than we require. It also does not allow for run-time address verification.

3. THREAT MODEL

In this study we are specifically interested in designing software for a voting device that can be rigorously audited for robustness and functionality without trusting any other code. As we focus on this problem, we note important, additional conditions that remain essential to the security of a complete voting system as well as assumptions made in developing our solution.

While our study examines software functionality, we do not address the issue of authenticity or ensuring that the correct software is actually put on the machine. Other studies have made progress toward a solution to this problem [44, 43, 16, 18]. We also assume that the hardware of the voting device operates correctly and non-maliciously.

We consider our work in a setting with multiple code auditors and multiple political parties. We assume that at least one auditor analyzes the code using a system with benign hardware and an uncompromised, honest operating system, compiler, etc. (at least with respect to the auditing tasks) and that this auditor is not persuaded that her results are inaccurate. Similarly, we rely on members from at least one political party to act only in the interest of that party. In particular, we require them to generate appropriate cryptographic keys and maintain their secrecy. We assume that poll workers do not authorize any voter to vote more than

once or allow anyone else to do so.

Lastly, our vote casting device stores secret keys in RAM, and we assume that the adversary cannot read these off of the device [21]. Additionally, we assume that the medium that stores the ballots is physically inaccessible to the adversary during the election. We discuss this requirement in more detail along with some alternatives in Section 6.2.

4. GOALS

Before discussing design and implementation, we outline desirable properties of a vote casting system.

4.1 Functionality

We first consider objectives that are important to the operation and behavior of a secure vote casting device and outline them below. Several have been mentioned in previous work [50, 40].

- The ballots input to the device and the user interactions should determine the operation of the system. The behavior of the system should be exactly the same during testing and during an actual election.
- To preserve privacy, each voter's voting session should be independent of previous sessions.
- Each voter should be able to cast one ballot.
- No voter should be able to cast more than one ballot.
- A ballot should be cast only when the voter has confirmed her choices and has instructed the system to cast them.
- No one should be able to add or remove ballots from the storage record.
- The system should be tolerant and informative of hardware error.
- The system should be tolerant and informative of user error.

4.2 Auditability

Several properties are also desirable to improve the effectiveness of auditing a system's software.

- The code base of the system should be as small as possible. Less code to audit allows careful examination of each line. Note that this property often conflicts with other properties. For example, our code has 87 lines of constant definitions, which could trivially be removed. However, they make the code easier to read, and thus simpler to audit overall.
- The code should be organized into logically separated components that can be individually verified. Modularity breaks the task of auditing down into pieces that are each more simple than a monolithic system.
- The code should utilize a limited set of instructions. This property is particularly important for code written in assembly language. Previous research shows that assembly programs with frequently used instructions are easier to understand than those with rarely used ones [10].
- The code should use a syntax that is intuitive to a reader. Common control flow structures and means of accessing variables are generally more familiar to auditors and thus likely to require less mental processing than lower-level, complex branching and memory references.

5. HIGH-LEVEL APPROACH

One of the primary objectives of our work is to reduce the amount of code that needs to be trusted for voters to electronically cast votes. We take an approach of reducing trust by redundantly dividing it among many entities.

The basic flow of data in our architecture is illustrated in Figure 1. Many months before the election, the source code for the vote casting device is published or, if preferable [22], made available to a large but select group of auditors. Then, auditing on a wide variety of systems is encouraged, including systems with different hardware, OSes, and any applications involved in auditing. The idea is that even if a number of systems have compromised components, at least one system will not and can perform an impartial audit. People can even develop new assemblers since they are relatively simple,[2] and they can be used along with many existing ones. Finally, auditors or other individuals assemble the code. Because assembling is (mainly) a one-to-one process,[3] binaries can be compared and analyzed until consensus is reached on a correct image for the device. Hence, we eliminate trust in any single system or component by distributing the auditing and binary generation in this way.

In addition to enabling verification of a correct binary, redundantly dividing trust also provides us with a means of drastically simplifying the sensitive vote casting process without significantly increasing threats to the other steps of an election. This is the general technique used by Bruck *et al.* with Frogs [6] and Yee *et al.* with prerendered user interfaces [51, 50] and allows us to remove the vast majority of vote processing from the ballot casting device.

Several months prior to the election, the display format for the ballot is standardized and publicized. This specification includes not only the exact visual appearance and sounds of the ballot as they are presented to the user but also their raw format as it is sent directly to the display and audio hardware of the casting device. The public can review the format and ensure that it is clear, simple, and free of any potentially problematic qualities. Then, on the day of the election, each voter is given a memory card for storing her vote when she enters the polls. She selects her votes using a ballot selection device, and her ballot is recorded to her card in its raw format. She then passes her card on to the ballot casting machine, and the card's data is sent directly to the display and audio hardware of the device so she may view it and confirm her selections. This step also allows the voter to detect any dishonest behavior of the ballot selec-

[2]The assembler output for our code is not a complicated elf file, but a simple binary that is mainly the result of translation from text instructions to binary opcodes and macro expansion.

[3]There are some minor exceptions to this. Namely, there are multiple ways of assembling some instructions. However, a simple diff of binaries will reveal locations of such differences and manual examination can confirm correct assembly. Alternatively, those assembling the binary can specify more rigorous standards for assembly.

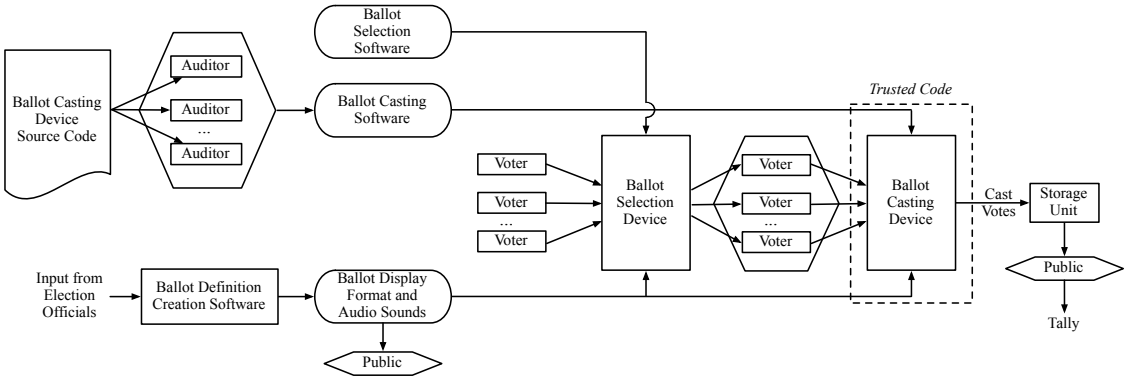


Figure 1: Flow of data through an election. Hexagons represent reviews by a large number of parties.

tion device, and in aggregate, the review of large numbers of voters minimizes the probability of undetected malicious activity.[4] From this point, if the voter chooses to cast, the same, unprocessed data that was displayed is anonymously recorded to non-volatile ballot storage along with some authentication information. Again, because an abundance of people can independently interpret and process the information, the anonymous ballots can be made public by disclosing the storage data exactly as it is, in raw form. Despite the data's increased complexity, the risk of undetected error or dishonesty is minimal due to the extensive number of people potentially reviewing it. By moving all the untrusted processing to before and after the point when the ballot is cast in this way, we greatly simplify this critical point in the election process.

6. SYSTEM FUNCTIONALITY

In this section we describe the operation of our system and the voting process without delving into the implementation specifics, which are discussed in the section that follows. In addition to our vote casting device, we have implemented an authentication device and a ballot selection device as illustrated in Figure 2 for testing and demonstration. However, we focus many of the details of our discussion on processes that directly interact with the sensitive vote casting device.

6.1 System Components

Figure 2 illustrates the voting process and the involved entities. As mentioned in the previous section, our architecture requires some small, memory hardware for temporarily storing voters' selections as they step through the polls. In our current system, we use memory smart cards because many current voting machines support writing to smart cards, making our device compatible with them as potential hardware for the ballot selection. However, any similarly inexpensive, portable storage device could be used with a slightly modified implementation. As illustrated, our ballot casting device has 5 primary interfaces:

- Smart card reader: can read and also write smart cards although we typically use the conventional phrase "smart card reader" when referring to it.
- Display: provides visual output for the voter.
- Audio output: a pair of headphones so the voter can optionally hear a description of her ballot and instructions.
- Storage unit: holds records of cast ballots and also stores audio files played by the device.
- Buttons: allow the voter to choose to cast or revise her ballot.

6.2 Voting Process

We now describe the voting process using our device and explain its precise functions.

Initialization. Prior to election day, each major political party generates a public/private key pair for encryption and a public/private key pair for signatures. Each party writes the secret components of both key pairs to a number of smart cards, which we refer to as key cards. They disclose the public components. In our implementation, we assume 2 such political parties. In addition, the parties append a list of valid audio file addresses to their key cards. These are locations on the storage unit where legitimate audio files are to reside. The lists should be standardized and publicly known as should everything about the audio files. Each party also randomly generates a secret share of a message authentication code (MAC) key and stores it on some separate, portable memory devices (possibly other smart cards). Lastly, the vote casting device's storage is cleared of all votes, and all the audio files relevant to the election are written to the specific, publicly known locations on it. Poll workers test the correctness of the audio files by viewing and listening to a comprehensive selection of ballots, without casting.

The morning before the election, the authentication device is provided with the MAC key shares via the appropriate memory device from each party, and it combines them into a final MAC key. The ballot casting device is turned on, and

[4]As an example of one way this might work, suppose a vote selection device is reported to be behaving erroneously. It may be withdrawn from use and tested. The machine should be given no indication that it is being tested and testing can be conducted during the election by simulating voters. This election-time, voter auditing of ballot selection machines parallels the end-to-end voting approach introduced by Benaloh [2, 3], and only a small percentage of voter auditors provides a high guarantee in the accuracy of the final tally. As an example using analysis by Neff [32], consider an election with 100,000 voters in which selection machines attempted to inaccurately "mark" 500 ballots. If a randomly selected 1% of the voters effectively reviewed their ballots (i.e., noticed any misrepresentations on them), then a voter would notice the machine's incorrect behavior with greater than 99% probability.

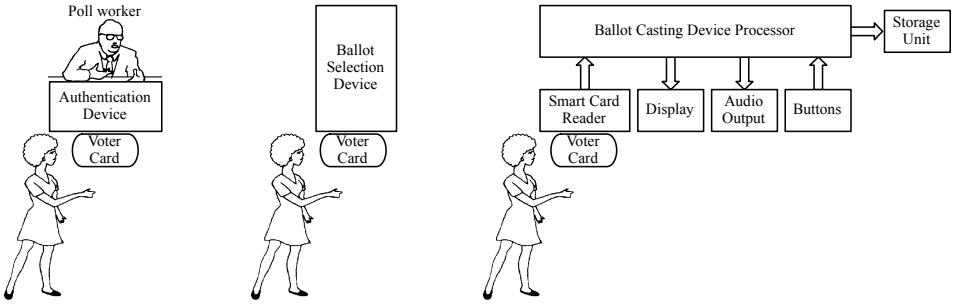


Figure 2: System components and voting process.

as it is currently implemented, it requests a key card from each of the parties (one at a time). As they are inserted, it makes some basic checks that each card is well-formed and reads the encryption key and the signature key from each and loads them into RAM. It also reads the list of valid audio file addresses from each key card. Once it has received all the key cards, it is ready for voter use.

Although we have implemented a key distribution process as described above, in practice there are several ways it might be performed. The option with the greatest security guarantees may be to have each party provide cryptographic smart cards that decrypt and sign internally [6].[5] Our primary reason for executing cryptographic operations on the device itself is twofold. First, we were unable to find commercial, off-the-shelf (COTS) hardware to support two or more smart card readers using our interfaces. Second, we wanted to maximize the flexibility of our techniques' application. In particular, since our current implementation provides a superset of the functionality needed by the cryptographic smart card approach, adapting it to that setting is straightforward.

Another option is to have the vote casting device randomly generate its own keys when it is turned on.[6] However, onboard generation would introduce some procedural complexities with respect to publicizing authentic public keys. All of these approaches have trade-offs in terms of software complexity, procedural complexity, security, and cost. We leave determining the details of the most desirable process to further study and evaluation by multiple disciplines. Our focus is to provide an implementation and techniques that could easily allow for any of them.

Authenticating a voter. The voting process begins with a poll worker providing the voter with an authenticated voter smart card. The primary purpose of this step is to ensure that each voter can only vote once. The poll worker first inserts a blank card into the authentication device. In turn, the device generates a 160-bit random number and computes a MAC over it using the MAC key generated during initialization.[7] These values constitute the voter's authenticator $\mathcal{A}$. Lastly the authenticator is encrypted using the public encryption keys generated by each political party, and the result $\mathcal{C}_{\mathcal{A}} = \mathsf{Enc}_{k1}(\mathsf{Enc}_{k2}(\mathcal{A}))$ is written to the voter's smart card.

As we describe below, once all the votes are published for tallying, only ballots with a *unique* authenticator $\mathcal{A}$ are counted. Thus, even adversaries who can read and write smart cards freely cannot cast more than one vote because only entities possessing the MAC key can generate new authenticators. We encrypt the authenticators so that voters with smart card readers cannot read the value of their authenticator off of their card and later identify their vote after the cast ballots (containing cleartext authenticators) are publicized.

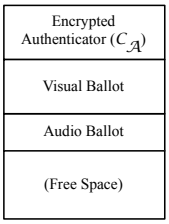
Vote selection. Once the voter has obtained an authenticated smart card from a poll worker, she takes it to the ballot selection device. Here, the voter selects her votes on a touchscreen machine, for example, and when she is finished, the device writes a visual description of her ballot to her smart card in raw form (visual ballot). It also writes a list of addresses of audio files that sequentially correspond to an audio description of the ballot (audio ballot). The complete layout of the voter smart card after this step is depicted in Figure 3(a).

Ballot confirmation. Next, the voter takes her smart card to the ballot casting device. Again, it makes some basic checks that the card is well formed and displays the ballot to the voter and also gives an audio description of it by reading and playing the audio files specified on the voter card. Prior to playing each audio file, the ballot casting device also checks that the provided file address is in the list of valid addresses provided by each party. This ensures that the device only tries to read legitimate audio files from the storage unit. Figure 4 illustrates an example formatted ballot as it is displayed to the voter. Recall that the presentation step requires very little data processing since the visual and audio ballot descriptions are stored in the format expected by the relevant output devices. We also clarify that once the data is read off the smart card, it remains in RAM on the device and nothing else is read from the card again for that confirmation session. Thus, the data displayed is the same data that

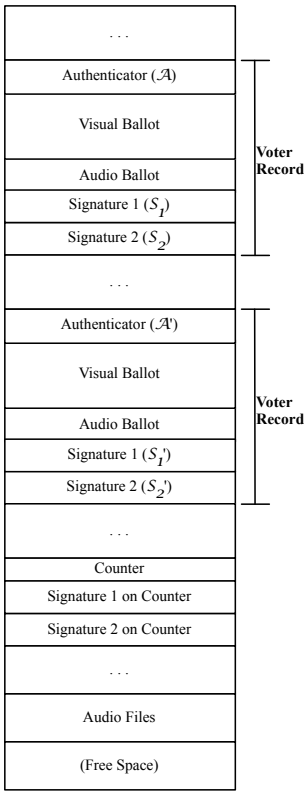
[5]Some smart cards are capable of storing keys that cannot be directly read from the card and computing cryptographic operations using those keys.

[6]Our device already supports all the operations necessary to do this, so making the modification requires minimal additional code.

[7]Rather than a MAC, we could also use a signature here. We chose MAC because it is slightly smaller and requires less processing power of the authentication device. Valid, cast votes, as we describe in the tallying phase, still require multiple signatures. Thus, even when we release the (symmetric) MAC key for authenticator verification, complete, cast votes cannot be computed. An alternate protocol may wish to use a signature rather than a MAC, as an extra precaution against leaked signature keys, however.



(a) Voter smart card.



(b) Storage unit.

Figure 3: Component data layouts.

is potentially recorded, and any changes that could occur in the smart card data (by a malicious smart card, for example) during the confirmation session are inconsequential. The voter is presented with visual and audio instructions indicating her to push one button to cast her ballot and another not to cast at that time. If she does not wish to cast, nothing is saved to the storage unit and she may return her voter card to a ballot selection device to alter her ballot.

Casting a ballot. If the voter chooses to cast a ballot, the ballot casting device decrypts the authenticator $\mathcal{A}$ using the private encryption keys that were provided on the parties' key cards. It also computes two signatures $\mathcal{S}_1$ and $\mathcal{S}_2$ over the voter's decrypted authenticator $\mathcal{A}$, her visual ballot, and her audio ballot. The signatures are computed using the parties' signature keys. Then all of these items including the 2 signatures are written to a randomly selected, blank record location on the storage unit. The general layout of the storage unit is illustrated in Figure 3(b). The signatures preserve the integrity of the ballots after they are cast. Unfortunately, computing the 4 signatures and 2 decryptions per vote takes several minutes and is impractical on our 60 MHz processor with 20 MHz memory bus. However, faster or more specialized hardware could preclude this problem.

Although we did not implement it this way in our prototype, we recommend that vote storage be implemented using a PROM chip following the technique introduced by Molnar

Figure 4: Our device displaying an example ballot.

et al. [28]. Their method allows for storage in which any change to a written bit is detectable and hence prevents cast votes from being erased.[8] Our implementation currently uses an SD flash memory card due to the greater availability and lower cost of the hardware. For this reason, the vote casting device also maintains and writes a signed counter of the total votes cast as illustrated in Figure 3(b). The counter prevents ballot erasure from a *single*, released copy of the storage unit. An adversary must still be prohibited from accessing the storage unit during the election. Otherwise she can roll the record of cast ballots back in time.

We have also implemented an alternate version of our device which ties cast ballots to their addresses in storage and maintains a bitmap expressing the locations of cast ballots rather than a counter simply keeping track of quantity. This offers slightly stronger guarantees because an adversary who reverts to an old bitmap cannot eliminate cast ballots of her *choice*. (However, she can still erase the newest votes.)

After the ballot has been cast, the ballot casting device erases the voter smart card, and it should be returned to the poll worker. The card is erased only after the cast ballot has been completely recorded. This way, if casting is interrupted, by power failure for example, the voter may use her card again to cast her ballot. On the other hand, even if voters remove their card before it is erased, they cannot effectively cast more than one vote due to the uniqueness of the authenticators.[9]

6.3 Tallying Process

At the end of the election, the storage unit is made public in its raw form. The shares of the MAC key from each party are publicized, and the authenticator in each voter record is validated after eliminating any duplicate authenticators. Signatures on the voter records are also verified along with

[8]The physical PROM chip also needs some type of authentication with this technique or it can be replaced entirely.

[9]Multiple casting of ballots does pose a potential vector for coercion attacks. However, such attacks that violate appropriate voting policies, can also be performed by video recording oneself casting a ballot, for instance.

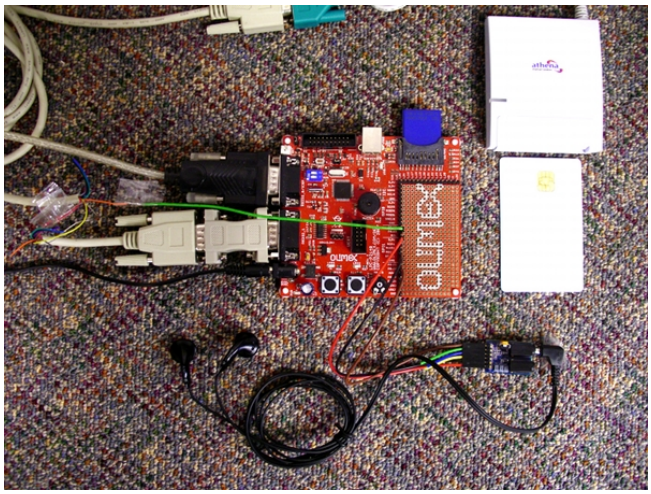


Figure 5: Prototype device hardware.

the correctness of the counter value. Furthermore, the cast ballots are checked for proper formatting and consistency. For example, people verify that the audio ballot corresponds to the visual ballot exactly and that both adhere to all strict specifications without variation. This verification ties the audio and visual ballots together so that a malicious vote selection machine cannot attempt to deceive a voter who uses only one interface. Finally, all interested parties interpret the ballot data and compute a tally of the votes that it represents.

7. IMPLEMENTATION

We now describe some aspects of our implementation and techniques we used to improve the auditability and robustness of our code.[10]

7.1 Hardware Overview

We implement our ballot casting device using an Olimex prototype board with an ARM7 compatible LPC 2148 microcontroller. We chose the board because it is inexpensive (76.95 USD at the time of writing) and has hardware interfaces for the components we require. ARM is also a fairly simple architecture. Our prototype device is pictured in Figure 5.

We use a 32MB SD memory card for storage of the cast votes and audio files although we recommend using a PROM chip in a finalized version [28]. The device's visual output is supplied through the serial cable connected at the upper-left of the board as pictured. In our development, we configured a standard PC to emulate a serial terminal, but for production, of course, separate, graphic hardware displays could be utilized. Using such a display requires no change in our code since ballot data is provided in raw form. For audio output there is a small amplifier module wired to the board with attached headphones.

Input is provided to the device by two buttons on the board (functioning as "CAST" and "REVISE"), and a serial smart-card reader/writer. One may notice that for the smart

[10]All of our code is available at `http://cs.jhu.edu/~ryan/min_tcb_voting/`.

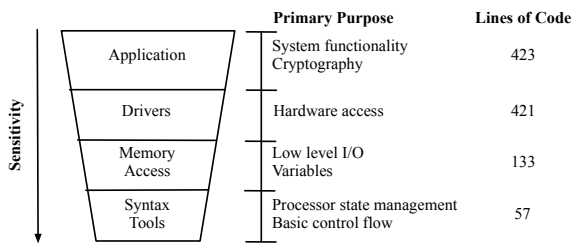


Figure 6: Structure of our code.

card reader to work properly, we had to manually connect the serial data ready to transmit (DTR) line from the smart card reader. This was necessary because the DTR pin on our board's serial connector simply dangles, unattached. We also had to switch the receive and transmit lines of our particular cable. The use of both a serial display and a serial reader for the smart cards simplifies code by reducing the number of interfaces that must be supported. Our device does not currently communicate with a printer, but a serial printer could also be attached and driven (using different hardware) with only a few additional lines of code.

In total, the hardware we purchased for the device cost 190.83 USD, not including the headphones or the display.

7.2 Structure and Readability

We organize our code into 4 distinct layers to simplify the task of auditing. Our basic approach is to compact security-sensitive functionality and common syntax abstractions and functions into small portions of code at the lowest layers. Then we write the larger, higher layers using these tools so that their implementation utilizes fewer, and more familiar constructs. This structure also contributes significantly to a shorter code length by maximizing code reuse.

The structure of our code is illustrated in Figure 6. Each layer is only permitted to use a subset of the instructions and functions that are used by the layer below it. In turn each layer exports functionality and syntax in the form of macros to those above it. The restrictions for each layer can be enforced by the auditors. Such enforcement involves simply checking that specified instructions and macros are not used at layers where they are prohibited and can be done almost entirely with text searching tools (e.g. `grep`). We attempt to make the code for the most privileged layers as small as possible since they are the most security-sensitive. The number of lines for each is listed in the figure. We explain the function and privileges of each layer briefly below.

Syntax tools. This is the lowest layer of code and is designed to provide syntax for control flow and functions to enable simpler macro design and processor-state management at higher levels. It is the only layer allowed to access the stack or process status (`cpsr` or `spsr`). This layer provides constructs for the following:

- preserving processor state and preparing parameters upon entering a macro
- restoring processor state and preparing a return value upon exiting a macro
- processor mode switching and state preservation for interrupt handling
- syntax for basic, counting for-loops

```
Our Code                                    Comment or Equivalent Java/C Style Syntax
declare_var_empty_array  buff, 256          @declare buff, 256 byte array or 256/4=64 int array
declare_var_empty_array  sum, 4             @declare sum, 4 byte array or 1 int array
...                                         @... (any code, changes buff and sum)
mov                      r2, #0             @r2=0
for checksum_buff,       r0, #V_intlen_buff @for(r0=0; r0<buff.length; r0++) {
    read_int_array       buff, r0, r1       @    r1=buff[r0]
    add                  r2, r2, r1         @    r2=r2+r1
end_for checksum_buff                       @}
write_int_array          sum, #0, r2        @sum[0]=r2
```

Figure 7: Example code for computing the 32-bit, 2's complement addition checksum over the data in `buff` and writing the result to `sum`. Uses constructs from the syntax tools and memory access layers. (`r0`, `r1`, and `r2` are ARM registers.)

As indicated in Figure 6, it consists of 57 lines of code.

Memory access. The primary function of the memory access layer is to provide user-friendly interfaces for declaring and accessing variables and conducting hardware I/O. It also implements a weak form of type safety, which we discuss in Section 7.3. No code above the memory access layer is allowed to use memory accessing or I/O instructions. Our code currently supports two basic types of variables, byte arrays and integer (32-bit element) arrays because these structures are flexible and all that we require. Simple integer-type variables, for example, are represented using a 1 element integer array. Similarly, large numbers, buffers, display data, lists, etc. are also easily expressed by the structure. The simplification in the code reduces the complexity of auditing as well.

Like the layer below, the constructs provided by the memory access layer are beneficial with respect to syntax. One example of code for computing a checksum that can be written above the syntax tools and memory access layers is given in Figure 7. Although the syntax may not be quite as appealing as C, for example, it is still quite clear and overwhelmingly more intuitive than raw assembly.

Drivers. Drivers provide application friendly interfaces to the hardware. Note that they do not actually directly access hardware themselves but rather use the I/O constructs provided by the memory access layer, and layers above the drivers are prohibited from accessing the hardware using anything but the functions provided by the drivers. The reader may notice that this layer is nearly the same size as the application layer. This is a result of the small amount of data processing required by our device. This layer provides tools for the following:

- writing to the display
- reading and writing flash memory cards (using an SPI interface)
- reading and writing smart cards
- playing audio clips (using the timer and digital to analog converter)
- determining button state
- reading from the analog to digital converter (for random number generation)

Aside from maximizing code reuse, which we achieve largely by using the lower-layer constructs for common functions, one of our primary techniques for simplifying drivers is to carefully limit error checking, as we discuss in Section 7.4.

Application. The highest layer implements the actual functionality of the vote casting device. It uses the drivers to determine the users' interactions and respond accordingly, carrying out the protocol outlined in Section 6. A significant portion of the code at this layer implements cryptographic operations.

7.3 Type Safety

One of the features we implement into our code is a weak form of type safety. The term "type safety" generally refers to the restriction that "the only operations that can be performed on data in a language are those sanctioned by the type of data" [39]. It helps reduce the risk of many common exploits such as stack and heap overflows by preventing data intended to be written to one variable from being written elsewhere. We do not have true type-safe code but rather achieve a *weak form of type safety* because we rely on auditors to enforce the simple restrictions of each code layer as discussed in Section 7.2.[11] Specifically, type safety has meaning only in the context of a language, and as we refer to it here, we consider it in the "language" allowed for the drivers and application layers of our code. (I.e. among other things, the code cannot use memory accessing instructions.) However, if we assume that the simple "language" allowed at that layer is enforced, we implement a form of what is often referred to as "dynamic type safety", where the code performs run-time checks to ensure that the operations performed on given data are allowed for that data.

Our code has only two essential data types, static variable arrays, and device I/O registers (for communicating with external hardware). We do not allow any dynamic variables as our code does not require them. As a result, enforcing type safety is fairly simple. When an array is statically declared, our code defines metadata that describes its position and size. These values are associated with its name. Then, whenever the variable is read from or written to using one of the data access macros, the assembler expands a series of code around the access to verify that the resulting memory address is actually associated with that variable before the access occurs. If, at run-time, the check determines otherwise, the device displays an error, "Data access violation" (assuming the display has been configured by this point), and halts. Reads and writes from hardware I/O registers are handled similarly. Although variable names are not associated with these registers, the constructs permitted for

[11] Recall that enforcing these restrictions only involves verifying that certain instructions are not used at specific layers.

accessing them verify that register addresses lie within the valid range prior to reading or writing.

In addition to memory for static data, code at the application and driver levels indirectly utilizes the stack for state preservation. The constructs used to do so (from the syntax tools layer) also check that the stack has not filled into any data variables every time something is pushed. If it has, the code similarly displays an error and halts.

One unfortunate side-effect of dynamic address checking, however, is that it is slow. This does not matter for the vast majority of our vote casting device's functions and hence, through nearly all of the code, every time data is accessed, the operation is first verified as described above. However, when it comes to public-key cryptographic operations, speed is important. For this reason, we also implement some specific type-safe arithmetic operations in the memory access layer of code. These include large number addition, subtraction, and shifting. Because everything about these operations and their parameters is known at assemble time, the loops that iterate through the large number arrays are actually unrolled. This way addresses can easily be checked *statically* and execution speed is also maximized. Such loop unrolling and static checking is always possible since we do not use (or even allow) any dynamic variables or pointers.

7.4 Error Handling

Our vote casting device consists of several, independent pieces of hardware, and our code must effectively use all of them despite the possibility of error. At the same time, errors result from a variety of conditions, and error handling has the potential to significantly increase complexity of device driver code. We attempt to avoid most of this complexity by taking an end-to-end approach to error checking when possible and omitting the majority of the intermediate checks.

Consider the SD flash card. There are a number of ways that negative hardware behavior might be indicated. As a few examples, the card might reply that a command is unrecognized, it might fail to respond to a signal at some point, it could declare that a checksum is bad (maybe as the result of a transmission error), a timeout might occur, or it might be busy. Accounting for such possibilities, which can occur on occasion, is essential. However, rather than checking for each of them, we simply check that the desired result is achieved.

The ultimate goal of the flash card is to store data. Thus, when the code needs to write to the card, it executes the entire, standard sequence of operations that complete the write on a working card in a ready state (omitting the majority of error checks). When it finishes, it reads back the block of data that it wrote and verifies that it was written correctly. If any bits differ, the code starts the write process back from the beginning and tries again. If it tries many times without success, the device displays "Flash card error" and halts. Poll workers and voters are indifferent to the cause of the error because the meaning is the same. Most likely, they need a new flash card or a new casting device.

When the ballot casting device is turned on or reset, we write a block of random data to the end of the flash card (verifying correctness as usual) after initializing it to ensure that initialization succeeded and it is functioning properly. This also allows us to omit the majority of error checking during initialization, further simplifying the code.

Like some of our other techniques, this approach is severely inefficient, but the speed reduction is insignificant for the application.

7.5 Cryptographic Operations

We utilize several cryptographic operations as described in Section 6 to help our vote casting device preserve voters' privacy and ensure the authenticity of cast votes. Implementations of cryptographic tools are often quite long and complex, however, so we take several approaches to keeping them as simple as possible.

Our primary approach to minimizing the complexity of our code's cryptography is to utilize constructs all based on the same operations, namely basic modular arithmetic and exponentiation. We use Schnorr signatures [41] to authenticate data written to the flash card and ElGamal encryption [14] for preserving the privacy of unique voter authenticators. With an appropriate hash function, both of these fit our criteria nicely and require a relatively minimal number of operations.

A hash function is required to compute Schnorr signatures[12] as is the case for nearly all other signatures. Since typical hash functions are rather complex, we chose to use a discrete logarithm hash, computed as $\mathsf{hash}(x) = g^x \mod n$ for $n = pq$ with large, unknown primes p, q [26]. The function is very simple, albeit inefficient, and is provably collision-resistant assuming the hardness of factoring [26]. To compress the result of the function, we xor blocks of 160 bits as suggested by Senderek [42].

Furthermore, the ElGamal decryption for a ciphertext (c_1, c_2) with secret key x and modulus m is classically described as $p = \frac{c_2}{c_1^x} \mod m$. We clarify that to avoid the need to include (and thus audit) code for the extended Euclidean algorithm, we send $z = -x \mod q$ to the vote casting device on the key smart cards (where q is the order of the relevant group). Then our code computes the ElGamal decryption as $p = c_2 c_1^z$.

Our ballot casting device requires random numbers for several operations. Random number generation is performed by sampling our microcontroller's analog to digital converter (ADC). We leave the ADC disconnected as suggested by Eastlake *et al.* to pick up electrical noise in the air [13] and use the Von Neumann transition mapping technique [47] followed by parity computation [13] to eliminate skew from the samples.[13] This method allows us to generate all the randomness needed for one voter very quickly. We used NIST's test suite for random number generation [36] to verify that the values obtained are statistically indistinguishable from random.[14] We could use Blum Blum Shub [5] as a pseudo-

[12] A Schnorr signature is computed as follows: Let $\mathbb{G}$ be a public group of prime order q with generator g. Suppose $\mathbb{G}$ is a subgroup of $\mathbb{Z}_p^*$. Let x be the secret key $0 < x < q$. To generate a signature on message M, choose a random k, $0 < k < q$, and compute $r = g^k \mod p$. Let $e = \mathsf{hash}(M||r)$ and $s = k - xe \mod q$. The signature is (e, s) [41].

[13] More explicitly, we consecutively sample the ADC and form pairs from the least significant bit of each sample. Then we generate a 1 for a (1,0), a 0 for a (0,1), and throw others away [47]. If we want n total bits, we repeat this process until we have $10n$. Lastly we xor every nth bit to obtain a final n bits of randomness.

[14] We collected 2MB of random bits from our random number generator (RNG) and performed the 16 tests from NIST's test suite [36] on the resulting data. Truly random data would behave "less randomly" than our data 12% of the time

random number generator (PRNG) with minimal additional code. However, since we would still need the ability to generate a seed and the hardware RNG is sufficiently fast, we use it for all of our random number generation.

We avoid authentication operations, such as authenticator or signature verifications, entirely by pushing them to the public, tallying phase of the election as outlined in Section 6, which also increases election transparency. As a result, xor and modular addition, subtraction, multiplication, and exponentiation are sufficient for all the cryptography leveraged by our device, keeping its software much simpler than it would be otherwise.

8. UNIFYING INTERFACES AND FUTURE WORK

The device we have implemented can be used with existing voting hardware and, in particular, any machines capable of writing to a smart card could be used for vote selection. A downside of this implementation, however, is that it alters the voter's experience as it requires interaction with an additional device.

Sastry *et al.* address this issue by designing an architecture that seamlessly joins distinct hardware modules in a way that is transparent to the voter [40]. Ultimately we envision that our techniques could be combined with theirs to merge our ballot casting device with separate hardware implementing the vote selection. The idea is summarized as follows: The voter selects her votes, unknowingly interacting with a hidden vote selection module. Once she finishes, the separate casting module claims exclusive access to the input and output devices of the apparent, unified voting machine. It achieves this using a hardware I/O multiplexer, over which the casting module always has complete control, and it effectively disconnects the vote selection hardware. The casting module presents the voter with her final ballot, as usual, and after the voter has had sufficient time to review her choices and approves them, it casts her ballot and returns I/O device access to the vote selection module. Depending on the input and output devices chosen, adapting our ballot casting device to this architecture could require very little additional code. Namely it would need functionality to drive and manage the multiplexer. Although we would want to omit the use of a smart card, of course, the ballot selection device could even be implemented to emulate the smart card reader and directly communicate with the ballot caster through a serial cable. This option would require no change to the code of our ballot casting device, but we do not recommend it since using another interface would be even simpler.

One item we would still like to explore is the exact trust required of a vote selection device. We have suggested that separate ballot casting devices (or hidden, internal modules) may provide a means of auditing and detecting malicious activity in ballot selection (Section 5). However, the optimal procedures and exact guarantees for doing so are unknown. We believe that exploring effective policies and gaining a more precise understanding of threats in this regard remains useful, important future work.

9. CONCLUSION

The security-sensitive functions of a voting machine can be simple, and simplicity reduces oversight and error. We have implemented a device that allows voters to review and cast their votes in only 1,034 lines of ARM assembly code. It utilizes no operating system or libraries, and as such has a significantly smaller trusted computing base (TCB) than current voting systems. It also eliminates the need to trust a compiler by being written in a simple language with well-defined output. Although the device is inexpensive and can be used with current voting machines that use smart cards, eventually it may be most useful in a slight adaptation. We imagine that we could apply the technique of Sastry *et al.* [40] and use it to create a complete, unified voting machine where the most sensitive components rely on only a very small amount of code. Reducing this trusted code base eases the task of verification and may thereby provide higher voting assurances.

in the worst of these tests and 43% of the time on average.

10. REFERENCES

[1] T. G. Ash. Who should be the world's most powerful person? *The Guardian*, 3 January 2008.

[2] J. Benaloh. Simple verifiable elections. In *EVT '06:USENIX/ACCURATE Electronic Voting Technology Workshop*, 2006.

[3] J. Benaloh. Ballot casting assurance via voter-initiated poll station auditing. In *EVT '07:USENIX/ACCURATE Electronic Voting Technology Workshop*, 2007.

[4] M. Blaze, A. Cordero, S. Engle, C. Karlof, N. Sastry, M. Sherr, T. Stegers, and K.-P. Yee. Source code review of the Sequoia voting system. Technical report, California Secretary of State, July 2007.

[5] L. Blum, M. Blum, and M. Shub. Comparison of two pseudo-random number generators. In *CRYPTO '82: Advances in Cryptology*, 1982.

[6] S. Bruck, D. Jefferson, and R. L. Rivest. A modular voting architecture ("Frogs"). In *WOTE '01: Workshop on Trustworthy Elections*, 2001.

[7] J. A. Calandrino, A. J. Feldman, J. A. Halderman, D. Wagner, H. Yu, and W. P. Zeller. Source code review of the Diebold voting system. Technical report, California Secretary of State, July 2007.

[8] D. Cauchon and J. Drinkard. Florida errors cost Gore the election. 5 May 2001. Available at `http://www.usatoday.com/news/washington/2001-05-10-recountmain.htm`.

[9] Compuware Corporation. Direct recording electronic (DRE), technical security assessment report. Technical report, Ohio Secretary of State, November 2003.

[10] C. R. Cook. Information theory metric for assembly language. *Software Engineering Strategies*, 1993.

[11] A. Cordero and D. Wagner. Replayable voting machine audit logs. In *EVT '08: USENIX/ACCURATE Electronic Voting Technology Workshop*, 2008.

[12] J. Dao, F. Fessenden, and T. Z. Jr. Voting problems in Ohio spur call for overhaul. 24 December 2004. Available at `http://www.nytimes.com/2004/12/24/national/24vote.html?pagewanted=print%&position=`.

[13] D. E. Eastlake, S. D. Crocker, and J. I. Schiller. RFC1750 - randomness recommendations for security. Available at `http://www.faqs.org/rfcs/rfc1750.html`.

[14] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO '84: Advances in Cryptology*, 1984.

[15] A. J. Feldman, J. A. Halderman, and E. W. Felten. Security analysis of the Diebold AccuVote-TS voting machine. In *EVT '07:USENIX/ACCURATE Electronic Voting Technology Workshop*, 2007.

[16] J. Franklin, M. Luk, A. Seshadri, and A. Perrig. PRISM: enabling personal verification of code integrity, untampered execution, and trusted i/o on legacy systems or human-verifiable code execution. Technical report, CMU Cylab, February 2007.

[17] R. Gardner, A. Yasinsac, M. Bishop, T. Kohno, Z. Hartley, J. Kerski, D. Gainey, R. Walega, E. Hollander, and M. Gerke. Software review and security analysis of the Diebold voting machine software. Technical report, Florida Department of State, July 2007.

[18] R. W. Gardner and S. Garera. Detecting code alteration by creating a temporary memory bottleneck. *IEEE Transactions on Information Forensics and Security*, December 2009.

[19] S. Garera and A. D. Rubin. An independent audit framework for software dependent voting systems. In *CCS '07: ACM Conference on Computer and Communications Security*, 2007.

[20] D. Grossman and G. Morrisett. Scalable certification for typed assembly language. In *TIC '00: Workshop on Types in Compilation*, 2001.

[21] A. J. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, 2008.

[22] J. L. Hall. Transparency and access to source code in electronic voting. In *EVT '06: USENIX/ACCURATE Electronic Voting Technology Workshop*, 2006.

[23] H. Hursti. Critical security issues the Diebold optical scan design, July 2005. Available at `http://www.blackboxvoting.org/BBVreport.pdf`.

[24] H. Hursti. Diebold TSx evaluation: Critical security issues with Diebold TSx, May 2006. Available at `http://www.blackboxvoting.org/BBVreportIIunredacted.pdf`.

[25] S. Inguva, E. Rescorla, H. Shacham, and D. S. Wallach. Source code review of the Hart InterCivic voting system. Technical report, California Secretary of State, July 2007.

[26] J. J.K. Gibson. Discrete logarithm hash function that is collision free and one way. *IET Computers and Digital Techniques*, 138(6), November 1991.

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

[28] D. Molnar, T. Kohno, N. Sastry, Naveen, and D. Wagner. 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.

[29] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talx86: A realistic typed assembly language. In *Workshop on Compiler Support for System Software*, 1999.

[30] G. Morrisett, K. Crary, D. Walker, and N. Glew. Stack-based typed assembly language. In *Journal of Functional Programming*, 1998.

[31] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3), 1999.

[32] A. Neff. Election confidence: A comparison of methodologies and their relative effectiveness at achieving it, 2003. Available at `http://www.votehere.com`.

[33] E. Proebstel, S. Riddle, F. Hsu, J. Cummins, F. Oakley, T. Stanionis, and M. Bishop. An analysis of the Hart Intercivic dau eslate. In *EVT '07: USENIX/ACCURATE Electronic Voting Technology Workshop*, 2007.

[34] pygame. Available at `http://www.pygame.org`.

[35] Raba Technologies LLC. Trusted agent report Diebold AccuVote-TS voting system. Technical report, January 2004.

[36] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. *NIST Special Publication 800-22*, 2001.

[37] D. Sandler, K. Derr, and D. S. Wallach. Votebox: a tamper-evident, verifiable electronic voting system. In *USENIX Security Symposium*, 2008.

[38] D. Sandler and D. S. Wallach. Casting votes in the Auditorium. In *EVT '07: USENIX/ACCURATE Electronic Voting Technology Workshop*, 2007.

[39] V. Saraswat. Java is not type-safe. Technical report, AT&T Research, August 1997.

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

[41] C. P. Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO '89: Advances in Cryptology*, 1989.

[42] R. Senderek. A discrete logarithm hash function for RSA signatures. Available at `http://senderek.com/SDLH/discrete-logarithm-hash-for-RSA-signatures.ps`.

[43] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: ACM Symposium on Operating Systems Principles*, 2005.

[44] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Sofware-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.

[45] R. M. Stein, G. Vonnahme, M. Byrne, and D. Wallach. Voting technology, election administration, and voter performance. *Election Law Journal: Rules, Politics and Policy*, 7(2), June 2008.

[46] K. Thompson. Reflections on trusting trust.

Communications of the ACM, 27(8), 1984.

[47] J. von Neumann. Various techniques used in connection with random digits. *National Bureau of Standards Applied Mathematics Series*, 12, 1951.

[48] D. Wagner, D. Jefferson, and M. Bishop. Security analysis of the Diebold AccuBasic interpreter. Technical report, Voting Systems Technology Assessment Advisory Board, July 2007.

[49] K.-P. Yee. Pvote. `http://pvote.org/`.

[50] K.-P. Yee. Extending prerendered-interface voting software to support accessibility and other ballot features. In *EVT'07: USENIX/ACCURATE Electronic Voting Technology Workshop*, 2007.

[51] K.-P. Yee, D. Wagner, M. Hearst, and S. M. Bellovin. Prerendered user interfaces for higher-assurance electronic voting. In *EVT '06: USENIX/ACCURATE Electronic Voting Technology Workshop*, 2006.