Designing for Audit: A Voting Machine with a Tiny TCB (short paper)

Ryan W. Gardner, Sujata Garera, and Aviel D. Rubin

Johns Hopkins University, Baltimore MD 21218, USA

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 and a perceived voter preference for touch-screen machines [16]. However, in the last 6 years, a multitude of studies have analyzed the security of electronic voting devices used in elections [10, 4], and each has found significant security flaws in every revision of their software. 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.

Several researchers have made significant progress toward designing voting systems that may provide increased assurances that the software is free of vulnerabilities and backdoors. Two of the most notable works are those of Yee *et al.* [20, 19] and Sastry *et al.* [13]. Yee *et al.* reduce the complexity of voting software by prerendering the electronic ballot design [20] and write a full voting machine application, Pvote, in only 460 lines of original Python code [19]. 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++ [4], or the Sequoia Edge, which consists of over 124,000 lines of C [1]. 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 [13]. 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.

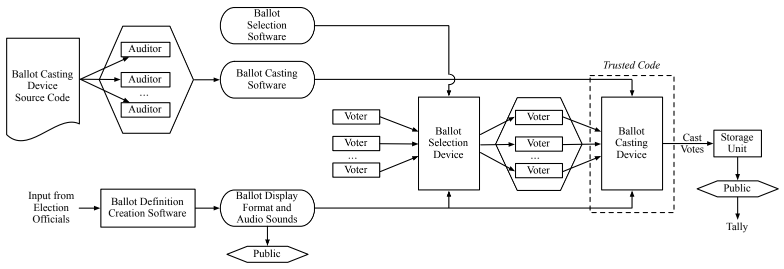

Fig. 1. Flow of data through an election. Hexagons represent reviews by a large number of parties.

Carefully auditing *all* components that contribute to the development and execution of a piece of software that casts votes is essential. Consider pygame, for example, a multimedia library used by Pvote. It contains 28,660 lines of code and numerous bugs have been reported on it including possible inappropriate memory accesses, which may allow exploitation of Pvote itself.[1] 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 [17].

In this work, we create a new vote casting system that drastically decreases the amount of software that must be trusted. Utilizing the idea of Frogs [3], we separate the device that voters use to confirm and cast their votes from the hardware they use to select the votes. Then, 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 pushing non-sensitive and verifiable functions to stages before and after the ballot is cast. 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. Here we summarize the architecture and a few aspects of the implementation of our vote casting system with a more complete description and details in the full version of the paper [7].

2 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 pub-

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

lished or, if preferable [8], made available to a large but select group of auditors. Then, auditing on a wide variety of systems is encouraged. 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. Finally, auditors or other individuals assemble the code. Because assembling is (mainly) a one-to-one process, 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 [3] and Yee *et al.* with prerendered user interfaces [20, 19] 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 and accurate. 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 selection device, and in aggregate, the review of large numbers of voters minimizes the probability of undetected malicious activity. 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.

3 Our Voting System

We now outline the functionality of our voting system and describe aspects of the implementation of our ballot casting device on an LPC2148 ARM microcontroller interfacing with additional hardware.[2]

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

3.1 Functionality Overview

The voting process involves 3 primary components: an authentication device, a ballot selection device, and a ballot casting device. Prior to the election, each political party generates a set of authentication and encryption keys. These keys are transferred to the ballot casting device via a smart card the morning of the election and stored in the device's RAM. An alternative approach could keep the keys exclusively on the smart cards and compute cryptographic operations on the cards themselves [3] although it requires more smart-card interfacing hardware. We compute all cryptographic operations on the ballot casting device to make the device's code easy to adapt to either approach.

On election day, each voter first obtains a voter card from a poll worker who authenticates the card using the authentication device. An authenticated card, which we implement using a memory smart card, contains a unique, encrypted authenticator that acts as the voter's ticket to vote. (Here, encryption prevents the voter from potentially obtaining her authenticator and selling her vote by identifying the authenticator and her corresponding vote during tallying.) Next, the voter takes her card to vote as described in Section 2. She selects her votes on the ballot selection device, which writes her ballot to her voter card. As she then transfers her card to the ballot casting device, that device presents visual and audio versions of the ballots and gives the option to cast. If the voter casts her ballot, the raw data that was presented is written to a randomly chosen, empty location on the device's non-volatile storage along with the voter's decrypted authenticator. The device also updates a counter of cast votes and signs all data written for the vote using a key from each political party. Unfortunately computing the decryptions and all the signatures takes several minutes on our device with its 60 MHz processor and 20 MHz memory bus, but faster or more specialized hardware could preclude this problem. Lastly, the device erases the voter's card, which is returned for reuse.

3.2 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 2. 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. 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

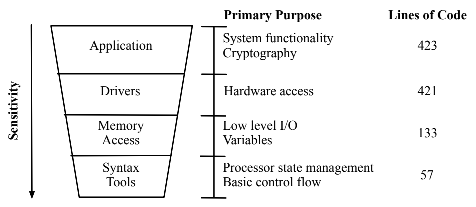

Fig. 2. Structure of our code.

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`

Fig. 3. 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.)

management at higher levels. It is the only layer allowed to access the stack or process status (`cpsr` or `spsr`). Constructs provided by this layer include tools for preserving and changing state during macro instantiations and interrupts and syntax for basic for-loops.

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, discussed in Section 3.3. 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. One example of code for computing a checksum that can be written above the syntax tools and memory access layers is given in Figure 3.

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. This layer provides tools for writing to the display, accessing flash memory cards, accessing smart cards, playing audio clips (using the timer and D/A converter), determining button state, and reading from the A/D converter. One of our primary techniques for simplifying drivers is to carefully limit error checking (discussed in the full version of the paper [7]).

Application: The highest layer implements the actual functionality of the vote casting device. A significant portion of the code at this layer implements cryptographic operations.

3.3 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" [12]. 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 3.2.[3] 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. 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. When an array is statically declared, our code defines metadata that describes its position and size, which is 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 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.

3.4 Cryptographic Operations

We utilize several cryptographic operations to help our vote casting device preserve voters' privacy and ensure the authenticity of cast votes. Implementations

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

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 [14] to authenticate data written to the flash card and ElGamal encryption [6] 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 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 [9]. The function is very simple, albeit inefficient, and is provably collision-resistant assuming the hardness of factoring [9]. To compress the result of the function, we xor blocks of 160 bits as suggested by Senderek [15].

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 [5] and use the Von Neumann transition mapping technique [18] followed by parity computation [5] to eliminate skew from the samples. 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 [11] to verify that the values obtained are statistically indistinguishable from random. We could use Blum Blum Shub [2] as a pseudo-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 2. This also increases election transparency.

4 Conclusion

The security-sensitive functions of a voting machine can be simple, and simplicity reduces oversight and error. We have presented a voting system on which one can cast her vote while trusting only 1,034 lines of code. Reducing this trusted code base eases the task of verification and may thereby provide higher voting assurances.

References

1. 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.
2. L. Blum, M. Blum, and M. Shub. Comparison of two pseudo-random number generators. In *CRYPTO '82: Advances in Cryptology*, 1982.
3. S. Bruck, D. Jefferson, and R. L. Rivest. A modular voting architecture ("Frogs"). In *WOTE '01: Workshop on Trustworthy Elections*, 2001.
4. 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.
5. D. E. Eastlake, S. D. Crocker, and J. I. Schiller. RFC1750 - randomness recommendations for security. Available at `http://www.faqs.org/rfcs/rfc1750.html`.
6. T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO '84: Advances in Cryptology*, 1984.
7. R. W. Gardner, S. Garera, and A. D. Rubin. Designing for audit: A voting machine with a tiny TCB (full version). Available at `http://cs.jhu.edu/~ryan/min_tcb_voting/`, 2009.
8. J. L. Hall. Transparency and access to source code in electronic voting. In *EVT '06: USENIX/ACCURATE Electronic Voting Technology Workshop*, 2006.
9. J. J.K. Gibson. Discrete logarithm hash function that is collision free and one way. *IET Computers and Digital Techniques*, 138(6), November 1991.
10. 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.
11. 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.
12. V. Saraswat. Java is not type-safe. Technical report, AT&T Research, August 1997.
13. N. Sastry, T. Kohno, and D. Wagner. Designing voting machines for verification. In *USENIX Security Symposium*, 2006.
14. C. P. Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO '89: Advances in Cryptology*, 1989.
15. R. Senderek. A discrete logarithm hash function for RSA signatures. Available at `http://senderek.com/SDLH/discrete-logarithm-hash-for-RSA-signatures.ps`.
16. 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.
17. K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8), 1984.
18. J. von Neumann. Various techniques used in connection with random digits. *National Bureau of Standards Applied Mathematics Series*, 12, 1951.
19. 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.
20. 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.