RICE UNIVERSITY

VoteBox Nano:
A Smaller, Stronger
FPGA-based Voting Machine

by

Ersin Öksüzoğlu

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dan S. Wallach, Chair
Associate Professor
Department of Computer Science

Kartik Mohanram
Assistant Professor
Department of Electrical and Computer
Engineering

Peter Varman
Professor
Department of Electrical and Computer
Engineering

Houston, Texas

December, 2010
ABSTRACT

VoteBox Nano:
A Smaller, Stronger
FPGA-based Voting Machine

by

Ersin Öksüzoglu

This thesis describes a minimal implementation of a cryptographically secure direct recording electronic (DRE) voting system, built with a low-cost Xilinx FPGA board. Our system, called VoteBox Nano, follows the same design principles as the VoteBox, a full-featured electronic voting system. The votes are encrypted using Elgamal homomorphic encryption and the correctness of the system can be challenged by real voters during an ongoing election. In order to fit within the limits of a minimal FPGA, VoteBox Nano eliminates VoteBox’s sophisticated network replication mechanism and full-color bitmap graphics system. In return, VoteBox Nano runs without any operating or language runtime system and interacts with the voter using simple character graphics, radically shrinking the implementation complexity. VoteBox Nano also integrates a true random number generator (TRNG), providing improved security. In order to deter hardware tampering, we used FPGA’s native JTAG interface coupled with TRNG. At boot-time, the proper FPGA configuration displays a random number on the built-in display. Any interaction with the JTAG interface will change this random number, allowing the poll workers to detect election-day tampering, simply by observing whether the number has changed.
Acknowledgments

First, I thank my advisor, Professor Dan Wallach, for his continuous support throughout my M.S. program. He helped me a lot during the implementation and presentation of “VoteBox Nano” by correcting my speech and slides, teaching the basics of academic writing and literature survey. Although we were coming from different backgrounds, he accepted me to his security group and gave me a means to continue on my research.

Besides my advisor, I would like to thank the rest of my thesis committee: Professor Peter Varman and Professor Kartik Mohanram for their time and valuable feedback on my thesis. I also want to thank Professor Farinaz Koushanfar for her support, innovative research ideas and her role in my admittance to Rice University.

Last, but not least, I thank my family: my mother, Fatma Oksuzoglu; my father, Gazi Oksuzoglu and my sister, Meltem Oksuzoglu; for their limitless and unconditional support and encouragement to pursue my interests, even when the interests went beyond boundaries of language, field and geography, for listening to my complaints and frustrations, and maybe most importantly for believing in me.

Some portions of this thesis appeared in Reconfig’08 and EVT/WOTE’09 conferences. This work was supported, in part, by NSF grants CNS-0524211 and CNS-0509297.
# Contents

Abstract ii  
Acknowledgments iii  
List of Illustrations vi  
List of Tables vii  

1 Introduction 1  

2 Background 5  
2.1 Commercial voting machines .............................. 5  
2.1.1 Hart InterCivic ........................................ 5  
2.1.2 Diebold AccuVote-TSX ................................. 8  
2.1.3 Sequoia Edge ............................................ 9  
2.2 Proof of concept e-voting machines ..................... 11  
2.2.1 Sastry’s voting machine .................................. 11  
2.2.2 VoteBox .................................................. 14  
2.2.3 Pvote ..................................................... 18  
2.2.4 Sturton’s voting machine ............................... 19  
2.3 FPGA security .............................................. 20  

3 Implementation 22  
3.1 Selecting a platform ....................................... 22  
3.2 Implementation details ................................. 23  
3.2.1 Computation ........................................... 24  
3.2.2 User interface ......................................... 30
Illustrations

2.1 Hart InterCivic eSlate [22] .......................... 6
2.2 Hart InterCivic Judge’s Booth Controller (JBC) [22] ................. 7
2.3 Diebold AccuVote-TSX DRE voting machine .............................. 8
2.4 Sequoia Edge [21] ...................................... 10
2.5 Screenshot of the 3 Gumstix boards (taken from Sastry et al. [42]) .... 12
2.6 The voting machine implementation of Sastry et al. [42] ............... 12
2.7 Minimalistic user interface of Sastry et al. [42] ........................... 13
2.8 SWATT’s attack model .................................... 14

3.1 Processing element (PE) ................................ 27
3.2 Execution graph of the crypto engine ................................. 29
3.3 Screenshot of the VoteBox Nano user interface ...................... 31
3.4 Screenshot of the original VoteBox user interface .................. 31
3.5 Rotary dial and buttons are used for user inputs ..................... 32
3.6 TRNG with ring oscillators. ................................ 35
3.7 TRNG post-processing unit. ................................... 35

4.1 JTAG data input is used as a sealing mechanism ...................... 44
4.2 Dot-Matrix Display showing the session ID in the second line ........ 45

B.1 Xilinx Spartan-3E FPGA motherboard and chip structure. ........... 55
B.2 The Xilinx Spartan-3E 500 motherboard running VoteBox Nano. .... 56
Tables

2.1 Source code length of each e-voting software ........................................ 20

3.1 Measured frequencies with different oscillator lengths. ....................... 34
3.2 Slice count and source code length of each FPGA module. ..................... 38
3.3 FPGA resource utilization. ................................................................. 40
3.4 C code size. ......................................................................................... 40

B.1 Xilinx Spartan-3E FPGA resources. .................................................... 56
List of Algorithms

1 Left to right binary exponentiation .................................................. 25
2 CIOS Montgomery multiplication ....................................................... 26
Chapter 1

Introduction

Voters seem to prefer electronic voting systems [15], and administrators like the speed inherent in electronic tallies. Direct recording electronic (DRE) voting systems offer many advantages for both the voters and election administrators over precinct-based optical scanner systems such as:

- **Faster tally**: Official results can be ready on the even same day with much less human effort.

- **Lower cost**: Although, investing money into DRE voting machines may seem to raise the costs initially, following reasons make the DRE machines very affordable in the long run:
  
  - The lifespan of most DRE machines are quite long and they can be reused for many elections.
  
  - Total number of the personnel (poll workers) required to carry on elections are lower.
  
  - There is no ink cost and paper waste at the end.

- **Security and reliability**: Well designed DRE systems can take the advantage of the cryptographic functions, so that they can preserve voter’s privacy and vote’s integrity; therefore it is much harder to cheat or expose the votes during or after the elections. Additionally, they are more resistant to the attacks by
the biased poll workers or errors induced by human factor, because most DRE systems are autonomous. Moreover, they can issue real time alarms or warnings in case of:

- An attack to the system,
- Incorrect/unexpected checksum values,
- Problems with the user interface or input devices,
- Connection or power issues,
- Unhandled exceptions or similar programming errors, etc.

• **Accessibility:** DRE systems are easier to use for a large portion of the general population because:

  - Many different ballot layouts (e.g. higher contrast schemes, larger fonts) and different languages can be adapted very easily.
  - Blind voters can take the advantage of audio support of the DRE voting machines.
  - Most DRE systems can warn the voters when they skip a race accidentally.
  - DRE voting systems are able to give visual and/or audible confirmation at the end of the voting process, so that the number of the invalid votes (e.g. multiple selections in a single selection race) or incomplete votes are substantially reduced.

• **Fairness:** Most undecided voters are inclined to select the first candidate in a given list, so the ordering of the candidates can create an inequality in some races; however the DRE machines can shuffle the order of the candidates for each
voting session, minimizing this unfair advantage. Moreover, different racing schemes (approval voting, range voting, or ranked voting, etc.) can be applied according to needs for a more balanced election.

Unfortunately, present-day commercial electronic voting systems have well documented security flaws (see, e.g., the California Top-to-Bottom Reports [23, 6, 4]), leading many states to consider dumping their electronic systems for paper-based voting, often with precinct-based optical scanners; therefore before going into the details of various voting systems, we should highlight the desired properties of a reliable voting system:

- **Eligibility**: Only legitimate voters should vote and they should be allowed to vote only for once.

- **Universal verifiability**: The tallying process and final results should be public and everyone should be able to verify the results of the election.

- **Privacy**: In a healthy democratic election, the privacy of the voter should be preserved by paying attention to the following:
  
  - **Anonymity**: It should be impossible to trace the votes back to the voter so that the voter will be in absolute freedom.
  
  - **Receipt&coercion freeness**: The voter should not be able to prove to someone that he voted for a specific candidate. This is required so that no one can buy votes or force someone to vote for his candidate.

- **Fairness**: The voting system must treat every candidate absolutely the same.

- **Robustness**: The voting system should be reliable against denial of service
attacks, power and network problems, partial component failures and it should handle invalid inputs appropriately.

Our research in this thesis is an extension of VoteBox [40], one of many electronic voting systems that aim to offer a paperless electronic voting experience, desired by many voters and election administrators, while using end-to-end cryptographic techniques to verify the correct operation of the voting system. To that end, we built a simplified VoteBox-like system, which we call “VoteBox Nano”, using a Xilinx Spartan-3E 500 Starter Kit.

The thesis is organized as follows: In Chapter 2, we discuss previous electronic voting technologies, followed by a discussion of VoteBox and how it differs from VoteBox Nano. In Chapter 3, we describe the implementation of VoteBox Nano, with particular attention paid to the implementation of crypto module and true random number generator. Chapter 4 considers threats against VoteBox Nano, particularly via its JTAG interface, and describes how we provide tamper-detection. We conclude and present the future work in Chapter 5. Interested reader can find the details of our FPGA platform and the details of Elgamal encryption in the appendices.
Chapter 2

Background

2.1 Commercial voting machines

In this section, we will introduce three leading commercial e-voting systems and discuss their vulnerabilities. We will take only the DRE equivalent components into account, that is, optical scanner based devices are not going to be considered.

2.1.1 Hart InterCivic

Harts DRE voting system can be divided into two groups:

- **On-site device**: eSlate (Figure 2.1) is the component which the voter interacts with.

- **Election management component**: Judges Booth Controller (JBC) (Figure 2.2) is the central controller of the eSlate networks.

Both devices cost around $3000 [22] and the software written for the whole system consists of approximately 360k lines of source code [23] (mostly C++ and C). The eSlates and JBCs are designed to run on an embedded operating system (whose source code is not publicly available) on custom embedded hardware.

A recent independent review [23] lists some of the vulnerabilities of the Hart e-voting systems as follows:

- **Unsecured network interfaces**: The network interface of Hart InterCivic is
very susceptible to attacks. The communication line can be eavesdropped and new votes can be inserted. It is allowed by the design that the poll workers are able to connect to JBCs and change the software on the fly using this interface, however the same method can be used to take over any eSlate voting device, so that any result can be produced regardless of voters’ actual intentions.

- **Vulnerability to malicious inputs:** The Hart software does not check the integrity of the data coming from other devices and uses them “as is”, leading to unsafe executions. An attacker could easily compromise the whole voting system by injecting malicious inputs to the device.

- **Lack of cryptography:** In Hart e-voting system, the communication channel between devices are generally not encrypted at all, which is quite unexpected; as a result the security of the system is drastically reduced. Some components
communicate indirectly through PCMCIA memory cards called Mobile Ballot Boxes (MBBs) which are encrypted with a single symmetric key*, which is kept in vulnerable polling-place devices, making it easy for an attacker to get the key and change the election results for the entire county by forging votes.

- **Failure to protect ballot secrecy:** The ballot’s secrecy and the voter’s privacy are not well protected in Harts e-voting system. A poll worker or an election official can rebuild the order in which the votes were cast and authorized; therefore “who voted for whom” can be revealed easily. Moreover, a voter who has temporary access to an eSlate device, can extract all the votes cast from the beginning of the voting session. Similarly, he can know the votes in the other eSlate devices, if they are connected to the same JBC.

---

*only one key for the whole county*
2.1.2 Diebold AccuVote-TSX

Diebold is a well known company with their ATMs and other security products such as electronic safes, paper shredders, alarms, security cameras, etc. besides e-voting systems. Their DRE voting system is called “AccuVote-TSX” (see Figure 2.3) which is used together with the GEMS election management system. Although we could not find the exact price for AccuVote-TSX; we found a similar product’s price “AccuVote-TS” which is $3,500 per unit (see [8]). AccuVote-TSX ballot station software contains around 64k lines of C++, whereas GEMS management application has 116k lines of C++ code [6].

Documented flaws [6] of the AccuVote-TSX voting system can be listed as follows:

- **Vulnerability to malicious software**: An attacker can install malicious programs on voting machines or GEMS election management system, disrupting
the recording process and as a result he can change the outcome of the elections.

- **Susceptibility to viruses:** The Diebold system is also vulnerable to viruses, which can easily pass to the networked devices including the GEMS system. In the worst condition, one affected machine could spread the virus to all the voting machines in a county.

- **Failure to protect ballot secrecy:** Diebold AV-TSX records the votes in the order they are cast and the poll workers can easily know the order of the voters. These two pieces of information together will compromise the secrecy of the ballot.

- **Vulnerability to malicious insiders:** The Diebold system gives almost infinite authority to the users of GEMS management server; therefore anyone with access permit to county's GEMS server can change the ballot definitions or even alter the election results.

### 2.1.3 Sequoia Edge

Our last commercial e-voting system is Sequoia Edge (see Figure 2.4). It entered to the market in 1999, and used for the first time in 2000 presidential elections [21]. The software system contains over 124k of C code.

Sequoia Edge also suffers from critical vulnerabilities [4] that can change the results of the election such as:

- **Lack of data integrity check:** The Sequoia system has removable media input, however the data on the media is not checked against corruption especially for the devices entrusted to poll workers. Some possible dangers of using corrupted input data can be listed as follows:
The election results are stored on "Results Cartridge" or "MemoryPack" and these are not protected against tampering. A corrupted poll worker, who has access to those, can change recorded votes or lead to denial of service.

- A corruption in the installed firmware can cause incorrectly recorded votes and paper trails.

- **Cryptography**: Although Sequoia e-voting system uses crypto functions to protect the election results, these functions are often implemented in an ineffective manner. Most cryptographic functions are weak with known vulnerabilities. Moreover, all cryptographic keys are permanently hardcoded in the software, so even if only one device is compromised, the extracted key can be used for other devices, ruining the whole election.

- **Access Control**: The access control mechanism can be easily circumvented,
using the weaknesses in WinEDS\textsuperscript{1}.

- **Software Engineering:** The software is reported to have a lot of programming errors. Some of these include buffer overflows, format string vulnerabilities, and type mismatch errors. The code does not show defensive software engineering practices needed for high assurance systems. As a result, these errors decrease the overall security and reliability of the system.

2.2 **Proof of concept e-voting machines**

In this section, we will discuss various DRE voting projects, each trying to solve the most common problems that the commercial DRE voting machines have.

2.2.1 **Sastry’s voting machine**

Sastry et al. [42], having similar goals to our project, built a minimal voting device using Gumstix (connex 400xm) computing devices (Figure 2.5) which have following features:

- They are considerably cheaper than a commercial electronic voting machine ($144 for each).

- They are really small (2 cm by 8 cm).

- They have an Intel XScale PXA255 processor with a 400 MHz StrongARM core, running on 64 MB of RAM and 16 MB of flash.

\textsuperscript{1}WinEDS is used on Windows PCs for entering, editing, collecting, and reporting on election information stored in a Microsoft SQL Server database.
Figure 2.5: Screenshot of the 3 Gumstix boards (taken from Sastry et al. [42])

Figure 2.6: The voting machine implementation of Sastry et al. [42]
This architecture allowed Sastry to enforce a number of important properties. Because distinct hardware components were responsible for different aspects of the voting machine, the wires between them could be hand-traced and debugged (see Figure 2.6). For example, the vote cast and cancel buttons are hard-wired to the computing module responsible for casting a vote. A user cannot be fooled into believing they have cast a vote, such as by drawing a “cast vote” button on the screen that actually does nothing. Likewise, Sastry leverages the ability to reset a piece of hardware back to its original boot state. Once a vote is cast, a dedicated reset module will blast all the other modules, ensuring that no module can retain state across votes. Although Sastry’s user interface is minimalistic (Figure 2.7), it still relies on an embedded Linux kernel and offers no particular mechanism to verify that the running code is authentic.

To address code tampering, Sastry suggests the use of SWATT [46], which implements a challenge-response protocol between an external verifier and an embedded
device. The challenges are a function of the contents of the memory. If the embedded device had different code running, even if it keeps the proper code in a backup location (see Figure 2.8), then the time it would take to compute the response would vary due to variation in CPU effects such as cache hit rates, or in the amount of time it would take to shuffle the contents of memory back to their proper configuration.

In an FPGA with a soft-CPU, unfortunately, techniques such as SWATT or other techniques based around timing computations are easier to defeat because the attacker could just build a switch into the FPGA’s memory controller, allowing the memory to be instantly rearranged to its proper state, exclusively to respond to the challenge. Instead, we would rather pursue techniques that leverage the structure of the FPGA itself.

2.2.2 VoteBox

VoteBox is an end-to-end cryptographically secure e-voting platform developed for experimenting with voting security technologies [40]. VoteBox is implemented in

\footnote{In this case, the challenge will fail and we will detect tampering.}
Java and runs on any PC, MAC, or Linux computer.

While a complete description of VoteBox is beyond the scope of this work, it’s important to describe several of the features of the system and explain how we adapted or eliminated these features to fit into the limitations of the VoteBox Nano platform.

The key technical insights in VoteBox are:

- **Pre-rendered user interfaces** simplify the graphics subsystem [58]. VoteBox does not use a general-purpose graphical widget system or require the use of a general-purpose font rendering system. Instead, a separate tool prepares PNG files to be copied to the screen along with an XML description of the ballot. Due to limited resources in VoteBox Nano’s FPGA, we can only support character graphics. We still pre-render the user interface as a series of text-drawing commands.

- **Network ballot replication** increases the availability of voting records [41] by gossiping every message to every machine on the precinct-local network. Messages are all digitally signed, so bogus messages can be trivially ignored. Messages include hashes of earlier messages, creating an entangled timeline, which makes it difficult for an adversary to modify the past [29, 28]. Even if a voting machine has been tampered or destroyed, its records will survive in copies on other local voting machines. VoteBox’s replication features do not require machines to reach any sort of consensus on the proper value of any given vote. Instead, any inconsistencies in the ballots, should there be tampering, are resolved after the fact using a general-purpose “Querifier” tool [39].

In VoteBox Nano, we could not afford a general-purpose network stack and data replication scheme. Instead, VoteBox Nano systems communicate point-
to-point with a precinct controller. VoteBox Nano thus does not have the fault tolerance of VoteBox, but it does have the same cryptographic integrity properties.

- **Homomorphic ballot encryption** allows external observers to tally votes independently and ultimately validate the decrypted totals published by election officials. VoteBox uses Elgamal encryption⁸ [14], a public-key cryptosystem. Each voting machine knows the public key of the election authority and can encrypt ballots for the authority to decrypt. The homomorphic property for Elgamal, as in any homomorphic cipher, means that we can define an “addition” function ⊕ that allows any party who knows two encrypted values $E_k(x)$ and $E_k(y)$ to compute $E_k(x + y) = E_k(x) ⊕ E_k(y)$ without knowing the private key corresponding to $k$ or being otherwise required to derive the plaintext of $x$ or $y$.

For a ballot with $n$ candidates, there must then be $n$ homomorphic counters. In any given ballot, these must be the encryption of either 0 or 1. The entire ballot will then be signed by the voting machine, using a conventional digital signature, before being transmitted on the network. In this respect, VoteBox Nano faithfully implements the same cryptosystem as VoteBox, and thus helps guarantee that votes will be counted as they were cast.

- **Ballot challenges** solve the concern that the homomorphic counters for any given ballot may not represent the intent of the voter, perhaps as a consequent of malicious code running on the voting machine. VoteBox adapts a technique from Benaloh [3], where the voting process follows the usual series of dialogs. After the voter accepts the summary screen, two things happen. First, the machine

---

⁸See Appendix A for more information on Elgamal encryption scheme.
computes the encrypted ballot, as above, and transmits it on the network. Second, the voter is asked whether he or she wishes to “challenge” the ballot or “cast” it. If the voter challenges, the machine must reveal the plaintext on the network, where everybody can see and verify\(^\star\). (In the case of Elgamal encryption, the actual encryption operation includes a random number. When challenged, VoteBox publishes the random numbers used, which is sufficient to verify the encryption was correct.) If the voter “casts”, then the machine announces this fact and erases its internal plaintext.

Ballot challenges force the voting machine to commit to the ciphertext of the vote without knowing whether the voter will actually cast the vote or may be deliberately auditing the machine for correctness. If the machine cheats, it can then be caught in a legally convincing fashion (e.g., an auditor may have witnesses and video cameras). If a normal voter accidentally challenges a ballot, or if a malicious VoteBox were to deliberately challenge ballots that the voter wanted to cast, those specific ballot will not be counted, this fact will be observed by the poll workers who can offer the voter a chance to vote again. In this respect, VoteBox Nano faithfully implements the same challenge system as VoteBox, and thus helps guarantee that votes will be recorded they were intended.

- **Voter’s privacy** depends on the strength of the cipher and the election authority’s key management, as the ciphertext ballots are recorded (and timeline entangled) in the order that they were cast. If the election authority’s secret key got compromised, then individual votes could be decrypted and voter’s identity

\(^\star\)This challenged vote is not included in any election tally.
could be revealed. Furthermore, the random numbers used as part of the Elgamal cryptosystem may offer a subliminal channel in which a malicious voting machine might leak information about a voter’s plaintext preferences. VoteBox offers no particular protection against such attacks. VoteBox Nano, however, uses a combination of attestations as the platform’s authenticity along with a hardware-based true random number generator. Ultimately, our system in this work relies on “true” randomness from our FPGA’s configuration. An alternate approach is to construct a protocol where multiple untrusted machines can collaboratively derive good random numbers [18], which could fit in the networked communication model of VoteBox, but doesn’t match as well to VoteBox Nano.

This thesis addresses several weaknesses with the VoteBox approach. First, while VoteBox’s security model protects the integrity of a voter’s vote, it does nothing to protect the voter’s privacy if a VoteBox has been compromised with malicious software. Such a VoteBox could simply record the plaintext votes, in the order cast. Alternatively, a malicious VoteBox could use the random numbers that are required for the cryptographic operations as a subliminal channel to leak information about the plaintext. Second, VoteBox has a substantial amount of code, both in its Java implementation as well as in the language runtime system and the operating system that support it; a smaller system might be less likely to have bugs. This project aims to design a VoteBox-like system, with a minimal implementation along with improved security properties.

### 2.2.3 Pvote

Pvote [57] is currently the smallest software (see Table 2.1) written for electronic voting machine prototyping. It has only 460 lines of Python which implements the
bare minimums of an e-voting application. Pvote tries to reduce its code size as much as possible, because "a small program is easier to write correctly and easier to review for correctness and it is harder for backdoors and security flaws to go unnoticed" [56]. Pvote aims to be the user interface component of a larger system (maybe coupled with some end-to-end cryptographic verification mechanism).

The primary idea that makes Pvote tiny is the "prerendered user interface" which is supposed to offer the following advantages:

- **Less security-critical code:** As there is a clear separation between voting machine software and other less critical components such as ballot preperation tool, it is simpler and easier to verify the security-critical part of the code.

- **Less updates to security-critical code:** The user interface can be completely redesigned while keeping the voting machine software the same, which increases the reliability of the whole system.

- **Ballot design by professionals:** Professional designers (not programmers) can prepare the ballots with usability and accessibility in mind; without compromising the reliability of the code.

- **Easier to test:** Because the "new ballot definitions" are kept in platform-independent format such as image files, they can be tested by anyone for correctness before elections.

### 2.2.4 Sturton’s voting machine

This is another FPGA based electronic voting machine project by Sturton et al [48]. This voting machine implements the bare minimum required for an election using
<table>
<thead>
<tr>
<th>Voting machine</th>
<th>Lines of code</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pvote</td>
<td>460</td>
<td>Python</td>
</tr>
<tr>
<td>VoteBox Nano</td>
<td>2759</td>
<td>C, Verilog</td>
</tr>
<tr>
<td>Sastry et al.</td>
<td>5085</td>
<td>C, Java</td>
</tr>
<tr>
<td>VoteBox</td>
<td>14500</td>
<td>Java</td>
</tr>
<tr>
<td>Diebold AccuVote-TSX</td>
<td>64k</td>
<td>C++</td>
</tr>
<tr>
<td>Sequoia Edge</td>
<td>124k</td>
<td>C</td>
</tr>
<tr>
<td>Hart InterCivic</td>
<td>360k</td>
<td>C++, C</td>
</tr>
</tbody>
</table>

Table 2.1: Source code length of each e-voting software

...a finite-state transducer whose correctness is proved using formal verification and systemic testing methods.

In computer science terms, formal verification is proving (or disproving) the correctness of the intended algorithms used in a system with respect to a certain formal specification (or a property) by mathematical methods. In this work, Sturton et al. first defined how each part of the finite-state machine should work and formally verified its Verilog implementation. They had "design for verification and testability" properties in mind throughout their progress; therefore they first derived the needed properties to be proved; then designed the system accordingly, so that it would be easier to verify.

2.3 FPGA security

Wollinger et al. [51] provide a summary of security issues while doing cryptography on an FPGA, with a focus on how to maintain cryptographic secrets within the FPGA in the face of attacks such as attempts to read out the FPGA’s bitstream. If the bitstream of the FPGA, itself, is a trade secret, then the ability to read it out could well be sufficient to reverse-engineer the logic within it.

Xilinx and other FPGA manufacturers offer features aimed at preventing these
reverse-engineering attacks [27]. To prevent “IP theft,” some FPGA chips allow the bitstreams that define the FPGA configuration to be encrypted. When the FPGA boots, it can access an internal key and use this to decrypt the bitstream. An attacker reading the ciphertext would learn nothing and no queries are available to allow an attacker to read the decryption key from the FPGA. Alternatively, Alkabani and Koushanfar [1] show how to leverage chip-to-chip variations in their behavior to achieve “active hardware metering.” The FPGA configuration will now be unique to a given chip; moving it to another chip would not yield a functioning implementation.

These techniques are aimed at protecting the secrecy of the FPGA’s bitstream. For our voting machine, secrecy is a non-issue. We need to detect tampering, which is a different problem. Dutt and Li [12] propose adding “parity groups” to the logic blocks within the FPGAs, so changes in any one logic block will cause parity failures without corresponding changes elsewhere, which the randomization makes it difficult to defeat. Drimer and Kuhn [10] describe a protocol to enable an FPGA to reject configuration updates that are undesirable.

We actually want some form of externally verifiable attestation that validates the internal state of the FPGA. Chaves et al. [7] proposes to leverage the “partial reconfiguration” modes allowed in modern FPGAs to effectively lock down an attestation module which can speak for the contents of the rest of the FPGA. Similar approaches are taken by other authors [19, 13]. All of these techniques rely on external (computational) verifiers. We desire unskilled election observers to detect tampering without needing computers. Our approach to attestation is described in Chapter 4.
Chapter 3

Implementation

This section begins with providing reasons for using an FPGA platform and then a brief description of our FPGA board is given. Following parts explain each component of our design.

3.1 Selecting a platform

For any hardware design to survive in today’s highly competitive economy, the total cost should be as low as possible while satisfying all the design requirements. Modern general-purpose CPUs are available at a wide variety of price points, but a computer is more than just a CPU. Designers must often decide whether to use off-the-shelf special purpose chips, such as graphics processors, whether to emulate such devices in software on a general-purpose CPU or engineer custom application-specific integrated circuits (ASICs).

A common rule of thumb is that the ASICs are only economically viable if over a million will be manufactured. In the case of a voting machine design, it’s unlikely that there will ever be sufficient demand for such a large volume of custom parts. In such circumstances, FPGAs have grown in popularity for a variety of reasons. They contain a variety of hardware resources and allow the designer to connect them together with great design flexibility. Modifying an FPGA’s firmware configuration is no more difficult than recompiling and reloading software on a traditional computer. Modern FPGAs are sufficiently large and fast as to be able to implement “soft CPUs”
and a variety of other resources. They are also valuable for prototyping designs that will later be moved to custom ASICs, allowing designers to get a design right before investing the resources to produce the ASIC (Kuon and Rose [26] discuss these tradeoffs in more detail).

To implement our design, we used a Xilinx Spartan-3E 500 Starter Kit (see Appendix B), which is widely available and one of the cheapest platforms. Currently, the development board is around $150 and the chip (XC3S500E-4FGG320C) is around $31 [53, 2]. When purchased in larger quantities, prices will be significantly lower.

3.2 Implementation details

In the process of shoehorning VoteBox into our Xilinx Spartan-3E 500 starter kit, we had to make a number of design decisions to simplify the system. Of course, the resources available on a Xilinx Spartan-3E are far fewer than on a general-purpose computer; therefore we cannot afford the logic for a general-purpose graphics frame buffer, and instead, we used character graphics. Likewise, we have limited on-board storage, so we could not implement the replication features of VoteBox. Instead, a VoteBox Nano client would be tethered to its supervisor console, which would then record the votes. We initially considered implementing the VoteBox application purely on the hardware. This would have been error-prone and unwieldy, instead our implementation combines off-the-shelf modules, such as Xilinx’s “MicroBlaze” soft-CPU core, with custom logic for fast cryptography and for generating truly random numbers. The VoteBox Nano application, itself, is written in C and runs on the MicroBlaze processor.

Much like writing software, an FPGA designer does not need to start from scratch, but can instead use libraries provided by third parties. In particular, we leverage the
MicroBlaze [52] 32-bit soft processor core into our design which is a RISC-based engine with 32 registers (implemented in slice RAM), with separate instructions for data and memory access. It supports both on-chip BlockRAM and/or external memory. Moreover, it can be customized for particular operations by adding a barrel shifter, a faster divider, 32 bit or 64 bit hardwired multiplier, a floating point unit, and so forth. The sizes of instruction and data cache can be adjusted independently.

MicroBlaze with hooks make it easy to access custom logic elsewhere in the FPGA. This allowed us to re-implement a simplified VoteBox in a straightforward manner, which allowed faster compile and easier debug.

3.2.1 Computation

We didn’t want to rely on the soft-CPU for the heavy-weight modular exponentiation required for Elgamal crypto system. With a ballot having 30 or more issues, each requiring the encryption of two or more counters, slow cryptography could well be noticeable by the user, therefore we decided to use our own crypto engine which runs purely on hardware and performs all modular multiplication and exponentiation operations. This crypto engine is a modified version of our previous work [35]. In the original design, we implemented features (such as modulus blinding, fixing the execution time regardless of the inputs, checking for weak keys) against advanced side channel attacks, however for this project, we tried to make the crypto engine as small as possible, while still providing enough performance.
3.2.1.1 Modular Exponentiation

For modular exponentiation, we used the “left to right binary method” (also called square and multiply algorithm) which is described in Algorithm 1*. This algorithm requires thousands of modular multiplications, which are the speed bottleneck of many Public key Cryptosystems (PKC). In a direct implementation, this operation alone can be very demanding on hardware resources because of the prohibitively expensive multiplication and subsequent division operations; however we can decrease the design complexity by the use of Montgomery [32] multiplication algorithm, which avoids costly1 “trial division”.

Our chip has dedicated multipliers in hard logic; therefore we needed a “multi-precision” Montgomery algorithm to take the advantage of these fast multipliers. For that reason, we took a software optimized algorithm called CIOS [25] which is “coarsely integrated operand scanning” (see Algorithm 2) and parallelized it for efficient hardware implementation.

---

**Algorithm 1** Left to right binary exponentiation

**Require:** \( n \geq 0 \) and \( x \neq 0 \)

\( n = (n_{s-1}, n_{s-2}, \ldots, n_1, n_0) \)

**Ensure:** \( y = x^n \)

1: \( y \leftarrow 1 \)

2: for \( i = s - 1 \) to 0 do

3: \hspace{1em} if \( n_i == 1 \) then

4: \hspace{2em} \( y \leftarrow y \times y \)

5: \hspace{2em} \( y \leftarrow y \times x \)

6: \hspace{1em} else

7: \hspace{2em} \( y \leftarrow y \times y \)

8: \hspace{1em} end if

9: end for

---

*The algorithm works the same way for modular arithmetic.

1in terms of both chip area and execution time
Algorithm 2 CIOS Montgomery multiplication

Require: n \geq 0 and a \neq 0 and b \neq 0
\begin{itemize}
\item s = total number of words in each operand (a, b, n)
\item w = bit length of each word (i.e. 16 bits for this case)
\item a = (a_{s-1}, a_{s-2}, \ldots, a_1, a_0), b = (b_{s-1}, b_{s-2}, \ldots, b_1, b_0), n = (n_{s-1}, n_{s-2}, \ldots, n_1, n_0)
\item k = \lceil \log_2 n \rceil \text{ where } 2^{k-1} < n < 2^k
\item n_0^{-1} = \text{Least significant word of multiplicative inverse of n in mod } 2^k
\item C, S, m = w-bit registers and \{\} is used for bitwise concatenation.
\item t = (t_{s+1}, t_s, \ldots, t_1, t_0) = \text{results register (initialized to 0 at the beginning)}
\end{itemize}

Ensure: \( t = a \times b \times 2^{-k} \)

1: for \( i = 0 \) to \( s - 1 \) do
2: \hspace{1em} C \leftarrow 0
3: \hspace{1em} for \( j = 0 \) to \( s - 1 \) do
4: \hspace{2em} \{C, S\} \leftarrow t_j + a_j \times b_i + C \text{ (multiplication loop)}
5: \hspace{2em} t_j \leftarrow S
6: \hspace{1em} end for
7: \hspace{1em} \{t_{s+1}, t_s\} \leftarrow t_s + C
8: \hspace{1em} m \leftarrow t_0 \times (-n_0^{-1}) \mod 2^w
9: \hspace{1em} \{C, S\} \leftarrow t_0 + n_0 \times m
10: \hspace{1em} for \( j = 1 \) to \( s - 1 \) do
11: \hspace{2em} \{C, S\} \leftarrow t_j + n_j \times m + C \text{ (modular reduction loop)}
12: \hspace{2em} t_{j-1} \leftarrow S
13: \hspace{1em} end for
14: \hspace{1em} \{C, S\} \leftarrow t_s + C
15: \hspace{1em} t_{s-1} \leftarrow S
16: \hspace{1em} t_s \leftarrow t_{s+1} + C
17: end for
3.2.1.2 The details of the crypto engine

We have two major loops in Algorithm 2: the multiplication (step 4) and the reduction phase (step 11). As these loops can be parallelized, we decided to put both of them into the same processing element (PE); as a result, each PE is made out of two multipliers, two adders and six registers (Figure 3.1).

The execution graph of the crypto engine (with two PEs) is shown in Figure 3.2. Once PE\(_0\) generates the first word of the intermediate result \(t_0\) (i.e. the least significant word), the next processing unit (PE\(_1\)) concurrently starts the computation for the second iteration of the loop with the values it obtains from PE\(_0\). When a PE finishes the computation of an iteration, it is immediately assigned to the next available
iteration. The results of the last PE are captured in dual-port Block-RAM, which, at the same time, feeds the first PE in its assigned cycle, to ensure a continuous data flow and maintain high resource utilization. The level of parallelism can be increased by putting more PEs in a serial fashion.

Before the execution of each iteration of the loop (at each increment of the loop counter \( "i" \)), the value \( m \) must be calculated as shown in Step 8 in Algorithm 2. The multiplication with \( m \) makes the intermediate result’s least significant word zero, so that we can shift right “radix” bits without losing information. (The value of \( n_0^{-1} \) is calculated offline (only one word) and fixed as long as the modulus does not change). However, meanwhile, other PEs are still performing multiplication operation and to maintain a continuous data flow, we need to insert FIFO buffers (not shown in the execution graph) among the PEs and compensate for the time lost by this pre-calculation step. This will create an irregularity in the algorithm, which makes the control circuit more complex.

As only one word can be requested from each Block-RAM per clock cycle, only the first PE directly receives data from the Block-RAM; and similarly only the last PE writes the result words (\( t_i \)) to the Block-RAM. Therefore, the general data flow is cyclical, that is, all PEs forward their output (the sum \( S \)) to the next PE if available or to the Block-RAM and all PEs use their own carry values (\( C \)) in the next cycle. Additionally, PEs forward “used input variables” (i.e. \( a_j \) and \( n_j \)) to the next PE to exploit data reuse and simplify connection network. The remaining variable (\( b_i \)) is assigned to the PEs at the beginning of the each iteration of \( i \).

We could fit 4 PEs into our design and utilized eight 16x16 bit multipliers in parallel, which provides sufficient performance while having a low slice count. Our multiplier circuit runs at 100 MHz (whereas the MicroBlaze soft CPU and other mod-
rules in the design run at 50 MHz) and performs one 1024-bit modular exponentiation operation in 20 ms on average. As the one clock is an exact multiple of the other, the edges are aligned and there are no timing and communication issues.

Our crypto engine is designed to be parametric, so we can make adjustments to meet the application requirements and utilize a given FPGA efficiently by changing the following three parameters at the compile time:

1. **Number of PEs**: Total number of PEs is the main area vs. performance tradeoff metric. The engine must have at least two processing elements since the first and last processing elements are hardwired to RAM. There is diminishing rate of returns in using more PE's, that is, one cannot make the design any faster by adding more PE's after a certain point, which changes with the operand size.
2. **Radix (w):** This parameter determines the bit length of the hardwired multipliers and adders shown in Figure 3.1. As the radix is closely related with the maximum combinational path delay, it has a direct effect on the frequency. Radix is parameterized to take the full advantage of the block multipliers in a given device to achieve the best performance.

3. **Number of words (s):** The radix and the number of words in each operand together determine the bit-length of the operands; e.g. for 2048-bit operands and 16 bit radix, the number of words is 128. The number of words also determines the depth of the Block-RAM.

### 3.2.2 User interface

We initially wanted to implement a general-purpose, full-color frame buffer. It quickly became apparent that this would consume too much RAM and considerable chip real-estate, particularly if we wanted a reasonably high screen resolution. Instead, we adopted an off-the-shelf character-graphics module [37] which can display 80x60 characters at a time in any of 8 colors. This module outputs an analog VGA signal at 640x480 pixels. (There is no DVI output on our board, so there is no easy way to directly drive a digital monitor.) Figure 3.3 shows the VoteBox Nano in action. While certainly not as visually attractive as a color bitmap graphics system, particularly for supporting non-Latin character sets, this design eliminates the need for any graphics libraries.

For user input, we used the on-board rotary dial and buttons (see Figure 3.5). The dial gives us one-dimensional navigation through the user-interface. One button then allows the user to mark the currently selected item on the screen.

The VoteBox Nano is designed to be visually similar to the original VoteBox (see
Figure 3.3: Screenshot of the VoteBox Nano user interface.

Figure 3.4: Screenshot of the original VoteBox user interface.
Figure 3.5: Rotary dial and buttons are used for user inputs

Figure 3.4), although they use different ballot definitions. The ballot definition file used by VoteBox Nano has $X, Y$ coordinates and the color of the text shown on each screen, simplifying the GUI code substantially. Both the ballot definition file and the crypto setup are supplied to the FPGA immediately after it boots, making it difficult to cheat even with a compromised machine. The advantages of having pre-rendered GUI are covered by Yee et al. [57].

As discussed in Section 2.2.2, the progression of the VoteBox Nano user experience mimicks that of the full VoteBox. The voter is presented with one screen per race. After selecting a candidate, the voter can advance to the next page. At the end, the voter is shown a summary screen from which any particular race can be directly selected, giving the voter an opportunity to correct errors. If the voter indicates that the ballot is correct, then the encrypted values for every race are written to the serial port. The voter then receives one final question asking if they wish to challenge the ballot or cast it. If they cast the ballot, this is noted on the serial port and the voting session is complete. If they challenge the ballot, all of the random numbers used in the cryptography are written to the serial port. This allows an auditor to decrypt the votes and validate that the voting machine is not tampering with them. This process is very similar to the VoteBox's ballot challenge mechanism, based on
Benaloh's design [3].

3.2.3 Random number generation

Every Elgamal-encrypted ballot requires a distinct random number. If the algorithm for random number generation is weak, the numbers could be predicted (or forced to a certain value) by an adversary, compromising the ciphertexts. Clearly, a voter’s privacy relies on the unpredictability of the random numbers.

As we are using an FPGA, we can generate truly random numbers, not just pseudo-random sequences. To accomplish this:

- The implementation must be purely in digital domain for practicality and reliability. So external clocks or any analog components must be avoided. There is an analog-to-digital converter on our board which can be used as an entropy source, however its behavior will vary as the environment changes.

- There are several algorithms that utilize coupled oscillators for random number generation [47, 34]. However, they will run correctly only if the oscillators are implemented with phase locked loops (PLLs). Our target board, the Xilinx Spartan-3E 500, only has a Delay Locked Loop (DLL) based oscillator, so these algorithms will not work.

- The algorithm must provide random bits with a reasonable speed, while maintaining a very low slice count. We cannot afford to spend much chip real-estate on random number generation.
Table 3.1: Measured frequencies with different oscillator lengths.

<table>
<thead>
<tr>
<th>l</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>250 MHz</td>
</tr>
<tr>
<td>5</td>
<td>155 MHz</td>
</tr>
<tr>
<td>7</td>
<td>106 MHz</td>
</tr>
<tr>
<td>9</td>
<td>81 MHz</td>
</tr>
<tr>
<td>11</td>
<td>69 MHz</td>
</tr>
<tr>
<td>13</td>
<td>59 MHz</td>
</tr>
<tr>
<td>15</td>
<td>52 MHz</td>
</tr>
</tbody>
</table>

3.2.3.1 Random generator implementation

Given the constraints above, we chose to implement Schellekens et al. [43]'s circuit for true random number generation (TRNG). The circuit consists of ring oscillators, running at frequencies with small differences. Our entropy source is the jitter of each oscillator.

Figure 3.6 shows our noise source. Here $l$ denotes the number of the inverters in each ring, $k$ is the number of ring oscillators, $n[t]$ is the noise, $f_s$ is the sampling frequency, $s[t]$ is the digitized noise. The XOR gate at the end harvests the jitter entropy, so even if only one of the $k$ oscillators provide real random output, the final outcome will be random. Schellekens claims that using shorter oscillator rings (e.g., $l = 3$) will result in more jitter per period and will therefore have a higher entropy; moreover it will decrease the area requirements of the circuit. We also confirmed that shorter rings have less stable frequencies which we directly measured with an oscilloscope (see Table 3.1). In their minimal design, Schellekens uses 110 ring oscillators; our implementation has 128 to be safe while still maintaining a small area. The flip-flop latches the output of the XOR tree at 25 MHz.

Subsequently, every TRNG needs a post processing unit (see Figure 3.7) to increase the entropy by decreasing the bias in the random bits. With two shift registers of different size (again borrowing from Schellekens, the first shift register is 240 bits and the second register is 16 bits), we compress the output of the random number generator to increase the entropy at the cost of decreasing the throughput. The XOR
Figure 3.6: TRNG with ring oscillators.

Figure 3.7: TRNG post-processing unit.
taps of the first register are selected according to [256,16,113] cyclic code [11]. For the first 240 cycles, the second register is disabled and the first register is filled entirely with the random bits from the source. In the next 16 cycles, we continue to feed the first register, while the second register is filled with the XOR output of the first register. At the end of the 256th cycle, the random word (16 bit long) $r[t]$ is ready. This means our TRNG outputs random bits at around 1.56 Mb/s. For each new random word, used bits will be replaced entirely with the new ones, ensuring a stateless machine.

In our implementation of Schellekens's TRNG, we have to violate two well known digital design rules: First, we create combinational loops and second, we insert redundant elements to the circuit (having multiple inverters in the same path). To overcome these problems in the regular design flow, we have to instantiate Look-Up Tables (LUT primitives) as inverters in the Verilog source code, and manually place them into the FPGA in a pre-defined fashion using the user constraints file (UCF), so that each ring oscillator has similar path delays. Moreover, we have to prevent the Xilinx synthesis tool from optimizing away the seemingly redundant gates.

3.2.3.2 Randomness evaluation

As the TRNG is a critical component in our design, we want to make sure that it is unbiased and unpredictable. To evaluate the strength of our TRNG implementation, we captured 860 MB of its output for subsequent analysis.

We first analyzed our random data with the DIEHARD [30] random test suite, which has 15 internal tests. The output of each test is normalized into one or more $p$-values that should be distributed uniformly between 0 and 1. If any of them yield
a p-value that’s very close* to extremes (i.e. 0 or 1), then that would indicate of a problem. In practice, our TRNG passed all of these tests.

DIEHARD has not been updated since 1997. We then used Dieharder [5], which is more comprehensive and up-to-date. The Dieharder suite is composed of 107 tests and provides four different scores for each test (passed, possibly weak, poor, failed). We got 102 “passed,” 3 “possible weak” and 2 “poor” from the test suite with the default parameters. The reason why our extracted random data could not pass all the tests is the larger data size requirement of these tests. In one case, a test rewound our sample file 20 times, which of course affects the outcome. When we changed the parameters to avoid this rewinding, we passed every test.

We then used ENT [50] application, which conducts a variety of statistical analysis. Its results can be summarized as follows:

- Entropy = 8.000000 bits per byte.
- Optimum compression would reduce the size of this 880477629 byte file by 0 percent.
- Chi square distribution for 880477629 samples is 255.63, and randomly would exceed this value 47.71 percent of the times (numbers near 50% are very random, while numbers close to 0% or 100% are not random).
- Arithmetic mean value of data bytes is 127.5016 (127.5 = random).
- Monte Carlo value for Pi is 3.141483357 (error 0.00 percent).
- Serial correlation coefficient is -0.000028 (totally uncorrelated= 0.0).

*up to six digits of accuracy
### Table 3.2: Slice count and source code length of each FPGA module.

<table>
<thead>
<tr>
<th>Module</th>
<th>Slices</th>
<th>HDL lines</th>
<th>Custom lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypto accelerator</td>
<td>2119</td>
<td>760</td>
<td>760</td>
</tr>
<tr>
<td>MicroBlaze CPU</td>
<td>1390</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>DDR-RAM interface</td>
<td>1103</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>Random numbers</td>
<td>637</td>
<td>132</td>
<td>132 (HDL) + 388 (UCF)</td>
</tr>
<tr>
<td>VGA</td>
<td>352</td>
<td>2297</td>
<td>281</td>
</tr>
<tr>
<td>RS232</td>
<td>151</td>
<td>1228</td>
<td>0</td>
</tr>
<tr>
<td>Debug</td>
<td>142</td>
<td>1177</td>
<td>0</td>
</tr>
<tr>
<td>Dot-matrix display</td>
<td>115</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>Push buttons</td>
<td>64</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>Rotary knob</td>
<td>35</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>Other modules</td>
<td>1687</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>7795</td>
<td>5831</td>
<td>1763</td>
</tr>
</tbody>
</table>

As a final test, we attempted to compress the output of our random number generator with the gzip and bzip2 compression utilities, using the "-9" flag to get the best possible compression. Both utilities yielded output larger than the input (0.016% larger for gzip and 0.44% larger for bzip2).

All these tests suggest that our TRNG is doing a good job of generating random numbers.

#### 3.2.4 Modules and design complexity

For the design of VoteBox Nano, we took advantage of the off-the-shelf modules provided by the Xilinx Platform Studio (XPS) tool and the OpenCores collection. For more specialized operations, we wrote our own modules and attached them to the system.

Table 3.2 describes the FPGA space requirements of each module and how many lines of hardware description language (HDL) code we had to change to adapt the
module for VoteBox Nano. (Modules we implemented ourselves will have the same number of lines modified as present in total.) We did not need to modify the source code of standard modules like the MicroBlaze CPU and its debugger, the DDR-RAM controller, RS232 communication module and push button controllers. However we needed to remove the CPU’s instruction and data caches to fit the entire design into our FPGA chip. For the security and performance critical components, such as the modular exponentiator and TRNG, we wrote our own code from scratch. We similarly needed to write our own drivers for the rotary controller and the LCD dot matrix display. For VGA output, we modified an off-the-shelf module [37] by changing its bus structure to make it compatible with our design and fine-tuning it to reduce its area requirements. For our TRNG, as discussed in Section 3.2.3.1, we had to design the logic (with HDL) and create a user constraints file (UCF) to defeat the place and route (PAR) tools.

The overall device usage is shown in Table 3.3. Note that the total slice count is less than the sum of the slices that each module requires (see Table 3.2), because the modules do not always fully utilize the slices. The Xilinx synthesis tools will allow separate modules to share resources within a given slice.

Aside from the FPGA configuration, we needed to write C code for the MicroBlaze CPU to navigate through the user interface and orchestrate the steps of the voting machine. The MicroBlaze CPU, even without caches, is sufficiently fast for our performance needs, particularly given that the slow cryptographic operations are handled in custom hardware.

Table 3.4 shows the amount of C source code written for each major function of VoteBox Nano. The GUI functions are used to interact with the VGA display. We similarly needed a wrapper to operate our modular exponentiation (crypto) unit.
<table>
<thead>
<tr>
<th>Resource</th>
<th>Used</th>
<th>Total</th>
<th>Used %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice</td>
<td>4482</td>
<td>4656</td>
<td>96</td>
</tr>
<tr>
<td>Slice Register</td>
<td>5060</td>
<td>9312</td>
<td>54</td>
</tr>
<tr>
<td>Slice LUT</td>
<td>6760</td>
<td>9312</td>
<td>72</td>
</tr>
<tr>
<td>Hardwired Multiplier</td>
<td>12</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>Block RAM</td>
<td>15</td>
<td>20</td>
<td>75</td>
</tr>
</tbody>
</table>

*Table 3.3*: FPGA resource utilization.

<table>
<thead>
<tr>
<th>Code Segment</th>
<th>LOC</th>
<th>Semicolons</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI functions</td>
<td>86</td>
<td>47</td>
</tr>
<tr>
<td>Ballot read/write</td>
<td>169</td>
<td>99</td>
</tr>
<tr>
<td>Crypto</td>
<td>215</td>
<td>155</td>
</tr>
<tr>
<td>DSA</td>
<td>205</td>
<td>159</td>
</tr>
<tr>
<td>State machine</td>
<td>321</td>
<td>220</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>996</strong></td>
<td><strong>680</strong></td>
</tr>
</tbody>
</table>

*Table 3.4*: C code size.

We wrote our own code for DSA, in which we used an off-the-shelf SHA1 function written by Niyaz [33], and an MPI (multi precision integer) library implemented by Fromberger [16] for performing operations beyond the modular multiplication, which we support in hardware. A modest 321 lines of code implements the bulk of the VoteBox Nano state machine. The resulting machine code is approximately 122 KB, including all the necessary library support. The motherboard includes 32 MB of DRAM, which provides ample room for our heap and stack segments, which will grow linearly in total with respect to the number of races.
3.2.5 Programming the FPGA

There are two methods for loading a bitstream, using the JTAG\textsuperscript{§} port to directly program the chip and uploading the bitstream to a Flash RAM chip, also through JTAG, and setting the on-board jumpers such that the FPGA boots from the onboard Flash.

One useful property of our Spartan-3E chip is that we can issue JTAG commands while the chip is still running. This allows us to stop and restart the chip and to read and modify the configuration of both the FPGA and surrounding memory chips. The “debugger” inside the MicroBlaze soft CPU [52] uses this functionality to remotely inspect and modify the machine’s state.

Some more advanced FPGA chips (such as the Xilinx Virtex-5 series, but not the chip we use in this work) can alter a part of their configuration, while the chip is still active, so that a hardware module (which is not needed anymore) can be substituted with another, which results in a dynamic configuration increasing the efficiency of the chip usage. This property is called module-based partial reconfigurability. If Vote-Box Nano were to be ported to such an FPGA, this functionality would potentially complicate the security of the design. We will discuss our approach to security in Chapter 4.

\textsuperscript{§}Joint Test Action Group (JTAG) is the common name used for the IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture, used for test access ports on printed circuit boards to talk to the individual chips. More details at Wikipedia: http://en.wikipedia.org/wiki/Joint_Test_Action_Group
Chapter 4

Modeling possible attacks

Threat modeling for a system like VoteBox Nano is an unusual task. Given the end-to-end cryptographic mechanisms, we’re confident that we can detect a corrupted machine that is trying to attack the integrity of the votes. The threat analysis that appears in the original VoteBox paper [40] applies to VoteBox Nano as well.

Consequently, this work will consider the threats that the original VoteBox made no attempt to address: the attacks on the privacy of the voter. These could involve tampering with the random number generation, so that an adversary could predict the random numbers and decrypt the ballots. Other possible attacks are encoding voter’s preferences directly into the random number itself, or leaking information about the vote by other means (e.g., by flashing the LEDs). We could also imagine that a clever attacker might try to modify the user interface behavior in an attempt to confuse the voter. Any such attack would require tampering with the FPGA configuration or the software running on the soft CPU. As discussed in Chapter 2, we fundamentally need a mechanism that can attest to the correctness of our system. Unlike other attestation architectures, however, we want our attestation mechanism to be directly visible to the voters and poll workers.

4.1 JTAG

Rather than using a dedicated Trusted Platform Module (TPM [49]) circuit (although one could certainly be used here as well), we arrived at a much simpler solution that
works perfectly for VoteBox Nano, even though it may not be generally applicable to other FPGA attestation problems.

We observe that any attempt to reconfigure the FPGA fundamentally requires using the JTAG interface, either directly from the JTAG pins on the motherboard or through the USB management port. JTAG commands are used to initialize the FPGA's configuration and to load the software for the soft CPU. In short, JTAG can do just about anything, including being a vector for security attacks [24]. Rather than trying to disable the JTAG interface and lock down the FPGA configuration, either in whole or in part, we instead want to ensure that we can detect whether any JTAG commands have been issued during the election day, and we want to be able to use JTAG's ability to extract the state of the FPGA as a mechanism to validate that its state is correct. In fact, we could imagine a commercial VoteBox Nano system extending the JTAG pins outside the box, to where they could be accessed without requiring the case to be opened.

If we allow that our attacker may access the JTAG pins, then we clearly must be able to detect when this has occurred. At that point, why not have our threat model allow for the attacker to modify the hardware arbitrarily? It's certainly the case that an attacker could substitute a different board inside the voting machine that looks like the original with an evil FPGA chip; such an altered system could externally appear unmodified, yet it could ignore or emulate the JTAG commands it receives. An alternative attack can target the off-chip memory that stores plaintext vote array, because it is pretty much vulnerable to any external probing attacks.

For purposes of this work, we're willing to posit that an attacker is only capable of soft attacks. Our attackers may well connect to any external connector and issue commands or eavesdrop on serial port traffic, but they cannot eavesdrop on internal
chip buses, desolder and replace chips, or physically damage the hardware. This is probably a reasonable assumption, since attackers want to make sure they don't leave behind any physical evidence of their attacks. Any detection of hardware modifications would undermine the effectiveness of an attack. (Also, we note that every chip on our board is surface mounted; so replacing a chip requires specialized equipment.)

4.2 JTAG tamper detection

Our Xilinx motherboard has an onboard LCD display which can show two rows of 16 characters at a time. In a production VoteBox Nano, this secondary display could be mounted such that it's visible to the voter. When the system is reset, our configuration will generate a random number (session ID) and place it on the LCD display. When the system is reset, our configuration will generate a random number (session ID) and place it on the LCD display. Similarly, every time a JTAG command is processed, we will get a new session ID shown on the LCD display, because we were able to hook into the JTAG input pins, triggering our own logic when commands are sent (see Figure 4.1). The random number appears on the bottom line of the display in Figure 4.2.

When the VoteBox Nano is powered on for the day, there may be some JTAG commands sent by the supervisor to initialize the voting machine, but after the ini-

Figure 4.1: JTAG data input is used as a sealing mechanism
initialization is complete, the machine should be powered up and running by itself all day with the same, exact random number displayed. Poll workers can periodically inspect the machines to verify that, in fact, the same number is being displayed as was there in the morning. (A production system would also include some kind of battery backup to ensure that power failure does not compromise the system.)

Naturally, an attacker using the JTAG commands could reconfigure the FPGA and break the link between JTAG commands and the random number display. However, if the FPGA was left with this non-standard configuration at the end of the day, then JTAG commands to extract the FPGA’s state would return proof of the compromise. If, on the other hand, the attacker returned before the day ended to reinstall the proper FPGA configuration, then a fresh random number would again be assigned to the display, and the attacker would be unable to control its value. As such, there is no way for an attacker to compromise the state of the voting machine, then subsequently return it to its proper state without being detected. The only requirement is that poll workers be diligent in recording the random number at the beginning of the voting day and verifying it at the end. Also, at the end of the day, prior to powering-off the machines, poll workers should use a tool to validate the FPGA configuration (see Section 4.3).

To throw off suspicion, an attacker might try to issue a JTAG command that

---

**Figure 4.2**: Dot-Matrix Display showing the session ID in the second line
addresses the random number display directly, leaving everything else in the FPGA alone, perhaps after returning a compromised VoteBox Nano back to its proper configuration. In order to do this, the attacker would first need to pause the FPGA, then modify the display, then resume the FPGA. This final command will trigger our logic to sample the random number generator again, thus overwriting the attacker's desired value with a random one.

While we are generally excluding physical attacks against the hardware, such as desoldering chips or replacing the motherboard in its entirety, the simplest attack against our system would be to simply cut the data pin between the motherboard and the LCD display. The display would continue showing the same number but would not receive commands to update it. The simplest defense is to continuously play an animation of some sort. If the line is cut, the animation will stop.

The only remaining JTAG attack, then, is a denial of service attack. An attacker could simply hit the reset button or pull the power. When the VoteBox Nano returns to its operational state, it will have a fresh random number on the display. The poll workers could take the voting machine out of operation or they could audit the FPGA configuration. Again, since this requires issuing JTAG commands, this will change the random number.

If a production VoteBox Nano’s JTAG interface was externally available but kept under a tamper-sealed or key-locked door of some kind, the process of sealing and unsealing the door would be analogous to procedures used to manage present-day DRE voting systems. Also, by keeping the JTAG pins away from normal voters, this would help defeat simplistic denial of service attacks.
4.3 Verification and other attacks

We use Xilinx’s iMPACT tool to verify our FPGA configuration by examining the contents of the lookup tables (LUTs) and the inter-connection matrix. iMPACT ignores changes in the flip-flops (e.g., the CPU’s registers), because they change state while the system is running.

Do these limitations leave room for an attacker to hide modifications? Recall that the random number display is tied to the JTAG interface in VoteBox Nano. Even if an attacker were to issue a JTAG command to modify the FPGA’s state (kept in the flip-flops or the external DRAM) through iMPACT, it would still change the random number displayed on the LCD.

One final attack possibility might be a buffer overflow against the code running on the soft CPU. Since VoteBox Nano is implemented in C, it may well have buffer overflow vulnerabilities. Perhaps the attacker could inject malformed packets into the protocol spoken between the VoteBox Nano and the supervisor console and be able to compromise the software running on the soft CPU without triggering the random counter. While we did not explicitly engineer VoteBox Nano to be robust against such attacks, the codebase is small and simple enough to be amenable to either mechanized or manual code auditing.
Chapter 5

Conclusion and the future work

We have presented a minimalist FPGA-based electronic voting system with cryptographically strong, end-to-end guarantees that protect the integrity of the votes. Our system, VoteBox Nano, is the first FPGA-based direct recording electronic (DRE) voting machine, which is radically different than the previous designs.

VoteBox Nano leverages the inherent properties of an FPGA chip to generate truly random numbers for its cryptographic operations; therefore it offers improved security. Moreover, we use the JTAG interface to audit the FPGA’s configuration for correctness at any time. Our voting machine shows a “session ID” captured from the true random number generator which is triggered by the JTAG data line, making a very simple yet effective tamper evidence mechanism. Any external observer can just check the session ID and make sure the design is authentic.

Our design utilizes a parametric crypto engine (a previous work of ours) which runs purely on hardware and performs the heavy weight calculations such as modular exponentiation, without a noticeable delay. We use an off-the-shelf MicroBlaze soft CPU to accelerate the development process by making the debugging easier. MicroBlaze talks to every other module and manages the whole system; as a result, our design is a hybrid of software and hardware modules.

We tried to make our source code as short as possible (so it will be easier to audit and less likely to have bugs); therefore we needed to make some compromises: first, we used a lightweight RS232 serial port as the communication module instead
of ethernet; second, we dropped the replication features of VoteBox, because we don’t have enough space on our board and third, we utilized character graphics, instead of full-color frame buffer. Although our user interface does not seem very sharp when compared to the other e-voting machines and our communication protocol is slower than ethernet, we are confident that our system is both very reliable and secure. Our source code consists of 1763 lines of code for hardware and a mere 996 lines of code for software.

Future work could go in many directions. FPGAs are often used to prototype designs before building custom ASICs. VoteBox Nano could well be implemented with a custom ASIC, eliminating the risks of JTAG tampering altogether. By prototyping the system first in an FPGA, we can convince ourselves that we have the right feature set before embarking on an ASIC design project (assuming it was economically feasible, in the first place).

Alternatively, we could consider using a larger FPGA with more resources and a faster clock rate, allowing us to use full-color bitmap graphics rather than characters, and also allowing us to implement the networking and replication aspects of VoteBox that were omitted in order to fit within the smaller FPGA. Similarly, we can increase the encryption grade (e.g. we can use 2048-bit keys) to make the design even secure and/or use a faster (and inevitably larger) crypto engine for shorting the execution time.
Appendix A

Cryptosystem details

Public key cryptosystems (PKC) such as RSA [38], Elgamal [14], Diffie Hellman [9], Paillier [36] are used to communicate through an untrusted medium by two parties. In all PKC’s, there are two keys, one public key (known by anybody) and one private key (only known by the owner). Public keys are used to encrypt the messages and verifying the signatures; on the other hand, private keys are used to decrypt the ciphertext and signing messages. Private and public keys are related to each other, however one cannot derive the private key by knowing only public key in the practical computation limits for adequate key lengths (1024 bits or longer). PKCs are like one way functions, where anyone can encrypt a message (or verify a signature), but no one can decrypt a ciphertext (or sign a message) without the private key.

We choose to implement Elgamal [14] public key cryptosystem in our design, because of its probabilistic encryption and homomorphic properties, so that we can add the encrypted votes without decryption.

A.1 Elgamal encryption scheme

Let’s assume that Bob wants to send a secret message to Alice over an untrusted medium using Elgamal cryptosystem. We can divide the calculations into three parts:

1. Key generation (by Alice)

2. Encryption (by Bob)
3. Decryption (by Alice)

A.1.1 Key generation

- Alice needs a multiplicative cyclic group $G$ of order $q$ with generator $g$ (finding a generator in mod $q$ is depicted in Chapter 4 of [31]).
- She picks a random $x$ which is smaller than $q$.
- Alice calculates $h = g^x$.
- $(G, q, g, h)$ are published as the public key and $x$ is kept secret.

A.1.2 Encryption

Bob has a secret message $m$ to be sent to Alice.

- Bob picks a random $y$ smaller than $q$.
- Bob calculates $c_1 = g^y$.
- Bob also calculates "shared secret" $s = h^y$. ($s$ is only used for one encryption)
- Bob converts the secret message $m$ into an element of $m'$ of $G$.
- Bob calculates $c_2 = m'.s$
- Finally, he sends $(c_1, c_2)$ ciphertext pair to Alice.

A.1.3 Decryption

Alice has received $(c_1, c_2)$ and she is going to use her private key $x$ for decryption.

- Alice computes the shared secret $s = c_1^x = (g^y)^x$. 
• Alice calculates $m' = c_2.s^{-1}$ and converts $m'$ to $m$.

The random variables $(x, y)$ are canceled because:

$c_2.s^{-1} = m'.h^r.(g^{xy})^{-1} = m'.g^{xy}.g^{-xy} = m'$

To perform the decryption, we need either the secret key $y$:

$$\frac{x \times h^r}{(g^y)^r} = \frac{x \times g^{yr}}{g^{ry}} = x$$

or random variable $r$:

$$\frac{x \times h^r}{(g^y)^r} = \frac{x \times g^{yr}}{g^{ry}} = x$$

As can be seen from the equations above, the random variable $r$ is as important as the secret key $y$, therefore $r$ should be a strong (i.e. hard to predict) random number and must be discarded right after each session.

### A.2 Elgamal homomorphism

Homomorphism is a mathematical property of a certain function. In PKCs, we can utilize homomorphism over modular arithmetic such that we can multiply or add the encrypted secret with a constant or with another encrypted secret, without decryption.

Homomorphism has two types: additive and multiplicative:

- In multiplicative homomorphic systems, the product of two ciphertexts is the encryption of the product of the plaintexts.

- In additive homomorphic systems, the product of two ciphertexts is the encryption of the sum of the plaintexts.
We can have additive or multiplicative homomorphism according to our needs by placing the secret in the exponent or the base:

\[
E(x).E(y) = (g^{r_1}, x.h^{r_1}).(g^{r_2}, y.h^{r_2}) = (g^{(r_1+r_2)}, x.y.h^{(r_1+r_2)})
\]

\[= E(x.y \mod q) \quad \text{(multiplicative homomorphism)} \]

\[
E(g_1^x).E(g_1^y) = (g^{r_1}, g_1^x h^{r_1}).(g^{r_2}, g_1^y h^{r_2}) = (g^{(r_1+r_2)}, g_1^{(x+y)} h^{(r_1+r_2)})
\]

\[= E(g^{(x+y)} \mod q) \quad \text{(additive homomorphism)} \]

g_1 is another generator in multiplicative cyclic group G and \((x,y)\) are the plaintexts. In our protocol, we make use of the additive homomorphic version.
Appendix B

Xilinx Spartan-3E FPGA Structure

We used a Xilinx Spartan-3E 500 (Figure B.2) Starter Kit to implement our design. Our board has following components:

- 32 MB of DRAM and 16 MB of Flash RAM
- 16 character by 2 lines dot-matrix LCD display
- 4 slide switches, 4 push buttons and a rotary encoder (knob)
- 8 LEDs

The board has following ports: a USB port (for programming purposes only), an ethernet port, two RS232 ports, a standard VGA output and a PS/2 port. Simplified structure of an FPGA (Spartan-3E series) motherboard is depicted in Figure B.1.

The essential elements of a modern FPGA chip are configurable logic blocks (CLBs). In our chip, each CLB has 2 slice-Ms (slice-memory) and 2 slice-Ls (slice-logic). Every slice contains 16-bit SRAM which behaves like a 4-bit lookup table (i.e. four bits in, one bit out). Slice-M’s can be utilized as 16-bit shift-register or 16-bit RAM. All the important resources are shown in Table B.1. More information can be found in Xilinx Spartan-3E user guide and data sheets [55, 54].

The edges of the FPGA chip are covered with IOBs (Input-Output Blocks), while there are several other blocks inside. The Digital Clock Manager (DCM) provides the clock input needed by flip-flops. There are twenty 18 bit signed hardwired multipliers
and 18 kbit block RAMs which are placed to side by side locations. The rest of the die is roughly made up with interconnection matrices (not shown) that allow CLBs and other chip resources to be wired together. Larger logical structures can be synthesized by connecting multiple slices together.
<table>
<thead>
<tr>
<th>Resource</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent gates</td>
<td>500k</td>
</tr>
<tr>
<td>CLBs</td>
<td>1164</td>
</tr>
<tr>
<td>Slice-M</td>
<td>2328</td>
</tr>
<tr>
<td>Slice-L</td>
<td>2328</td>
</tr>
<tr>
<td>Total slices</td>
<td>4656</td>
</tr>
<tr>
<td>Digital Clock Managers</td>
<td>4</td>
</tr>
<tr>
<td>LUTs and FF</td>
<td>9312</td>
</tr>
<tr>
<td>18 bit signed multipliers</td>
<td>20</td>
</tr>
<tr>
<td>Total Distributed RAM</td>
<td>74 kbits</td>
</tr>
<tr>
<td>Total Block RAM</td>
<td>360 kbits</td>
</tr>
</tbody>
</table>

Table B.1: Xilinx Spartan-3E FPGA resources.

Figure B.2: The Xilinx Spartan-3E 500 motherboard running VoteBox Nano.
Bibliography


