VoteBox Nano: A Smaller, Stronger FPGA-based Voting Machine (Short Paper)

Ersin Öksüzoğlu[†] ersin@rice.edu
Dan S. Wallach[‡] dwallach@cs.rice.edu

[†]*Department of Electrical and Computer Engineering,* [‡]*Department of Computer Science, Rice University*

Abstract

This paper describes a minimal implementation of a cryptographically secure electronic voting system, built with a low-cost Xilinx FPGA board. This system, called VoteBox Nano, follows the same basic design principles as VoteBox, a full-featured electronic voting system. As with VoteBox, the votes are encrypted using Elgamal homomorphic encryption and the accuracy 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 and storage facilities. In return, VoteBox Nano runs without any operating systems or language runtime system, radically shrinking the implementation complexity. VoteBox Nano also integrates a hardware true random number generator, providing improved security for the ballot cryptography. In order to deter hardware tampering, which might be done to compromise the random number generator, the FPGA's native JTAG interface can be used to verify the FPGA's configuration. At boot-time, the proper FPGA configuration also displays a random number on the built-in display. Any interaction with the JTAG interface will replace the random number with another one, allowing poll workers to detect election-day tampering, simply by observing whether the number has changed.

1 Introduction

Electronic voting systems offer many advantages for both the voters and election administrators. Voters seem to prefer electronic voting systems [13], and administrators like the speed inherent in electronic tallies. Unfortunately, present-day commercial electronic voting systems have well-documented security flaws (see, e.g., the California Top-to-Bottom Reports [19, 6, 4]), leading many states to consider dumping their electronic systems for paper-based voting, often with precinct-based optical scanners.

Our research is an extension of VoteBox [33], 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. VoteBox integrates pre-rendered user interfaces, network ballot replication, homomorphic ballot encryption, and Benaloh-style ballot challenges.

This paper 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 operating system that support it; a smaller system might be less likely to have bugs. This project aims to product a VoteBox-like system, with a minimal implementation, that can nonetheless improve on VoteBox's security properties.

To that end, we built a simplified VoteBox-like system, which we call VoteBox Nano, using a Xilinx Spartan-3E 500 Starter Kit. 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. Of course, the resources available on a Xilinx Spartan-3E are far fewer than on a general-purpose computer. We cannot afford the logic for a general-purpose graphics frame buffer, so we instead use 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.

The paper is organized as follows: In Section 2, we discuss previous electronic voting technologies and research on FPGA security, followed by a discussion of VoteBox in Section 3 and how it differs from VoteBox Nano. In Section 4, we discuss how our FPGA platform works. In Section 5, we describe the implementation of VoteBox Nano, with particular attention paid to

the implementation and evaluation of our true random number generator. Section 6 considers threats against VoteBox Nano, particularly via its JTAG interface, and describes how we provide tamper-detection. We conclude and present future work in Section 7.

2 Background

Sastry et al. [35], having similar goals to our project, built a minimal voting device using Gumstix computing devices. 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. 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. Sastry's implementation, however, 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 [39], which implements a challenge-response protocol between an external *verifier* and an embedded device. The challenges are a function of the contents of the device's memory contents. If the embedded device had different code running, even if it keeps the proper code in a backup location, 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 [38, 37, 18, 15] are easier to defeat because the attacker could just build a switch into the FPGA 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.

Wollinger et al. [42] 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 (a "readback attack"), its internal SRAM, and so forth. 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 (see, e.g., Lesea [23]). To prevent "IP theft," FPGA chips allow the bitstreams that define the FPGA configuration to be encrypted. When the FPGA boots, it can access an internal key store 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 [10] propose adding "parity groups" to the logic blocks within the FPGAs, so changes to any one logic block will cause parity failures without corresponding changes elsewhere, which the randomization makes difficult to defeat. Drimer and Kuhn [8] describe a protocol to enable an FPGA to reject configuration updates that are undesirable.

What we really want, though, is some form of externally verifiable *attestation* that the internal state of the FPGA is correct. Chaves et al. [7] proposes to leverage the "partial reconfiguration" modes allowed in modern FPGA's to effectively lock down an attestation module which can then speak for the contents of the rest of the FPGA. Similar approaches are taken by other authors [17, 11]. All of these techniques rely on external (computational) verifiers. We would like, if possible, for unskilled election observers to be able to detect evidence of tampering without needing computers. We will discuss our approach to attestation in Section 6.

3 VoteBox

While a complete description of VoteBox is beyond the scope of this paper, 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.

VoteBox is an end-to-end cryptographically secure voting system platform developed for experimenting with voting security technologies [33]. VoteBox is implemented in Java and runs on any PC, Mac, or Linux computer. The key technical insights in VoteBox are:

Pre-rendered user interfaces simplify the graphics subsystem [46]. 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 [34] 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 [25, 24]. 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 [32].

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 encryption [12], 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 $\oplus$ that allows any party who knows two encrypted values $E_k(x)$ and $E_k(y)$ to compute $E_k(x + y) = E_k(x) \oplus E_k(y)$ 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 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 it, verify it, and know not to include it in any election tally. (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.

Privacy Voter privacy, with VoteBox, 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 was compromised, then individual votes could be decrypted and voter privacy could be compromised. Furthermore, the random numbers used as part of the Elgamal cryptosystem 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 to the platform's authenticity along with a hardware-based true random number generator. Ultimately, our system 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 [16], which could fit in the networked communication model of VoteBox, but doesn't match as well to VoteBox Nano. Randomness is described further in Section 5.

4 Implementation 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 whether to engineer custom application-specific integrated circuits (ASICs).

A common rule of thumb is that ASICs are only economically viable if over a million of any given ASIC 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, field-programmable gate arrays (FPGAs) have grown in popularity for a variety of reasons. FPGAs 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. FPGAs 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 [22] discuss these tradeoffs in more detail.)

For our work, we used a Xilinx Spartan-3E 500 Starter Kit, 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 ([44, 2]). When purchased in larger quantities, prices will be significantly lower. (Xilinx makes much larger, faster, and more expensive parts, some including an onboard CPU in "hard" logic as well.)

The essential elements of a modern FPGA are configurable logic blocks (CLBs). In our Xilinx chip, each CLB has four "slices." Each slice has two four-bit lookup tables (LUT), externally configurable, with one-bit of output. Half of the slices ("slice-M's") additionally have a 16-bit shift-register and a 16-bit RAM cell. Also on-chip are input-output blocks (IOBs), 18-bit hardware multipliers, blocks of 18 kbits of RAM, and the interconnections that allow CLBs and other chip resources to be wired together.

Resource	Amount
Equivalent gates	500k
CLBs	1164
Slices	4656
Total Distributed RAM	74 kbits
Total Block RAM	360 kbits
18 bit signed multipliers	20

Table 1: Xilinx Spartan-3E FPGA resources.

Just as software designers have high-level languages and compilers that abstract away many of the low-level details, so do FPGA designers. FPGA configurations are developed in various hardware description languages (HDLs) such as Verilog and VHDL. These are then compiled into bitstreams which the FPGA can load.

There are two methods for loading a bitstream, using the JTAG port[1] to directly program the chip and uploading the bitstream to a Flash RAM chip, also though JTAG, and setting the on-board jumpers such that the FPGA boots from the onboard Flash.

One useful property of the Xilinx chip we use is that it allows JTAG commands while the chip is running. This allows us to stop and restart the chip and to read and modify the configuration of both the FPGA and its surrounding memory chips. When using the MicroBlaze soft CPU [43], the debugger 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 VoteBox Nano were to be ported to such an FPGA, this functionality would potentially complicate the security design. (See Section 6 for more on this.)

[1] 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

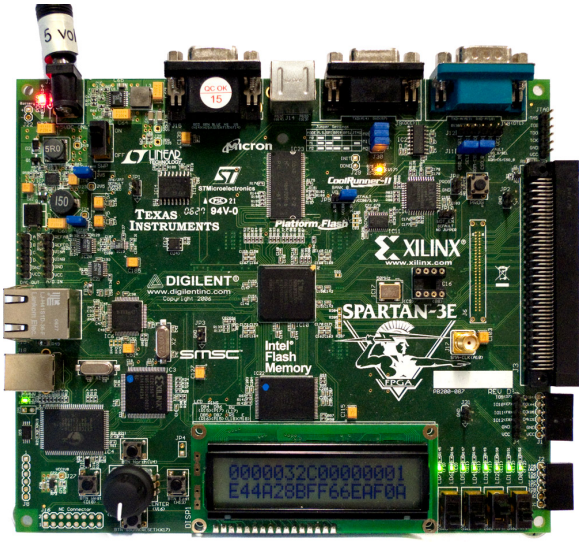

Figure 1: The Xilinx Spartan-3E 500 motherboard running VoteBox Nano. The left half of first line of the LCD display is showing how many seconds the FPGA is active in hex; while the current number of people who have voted is shown on the the right half. The second line is the random number, which is used to detect tampering (see Section 6.2).

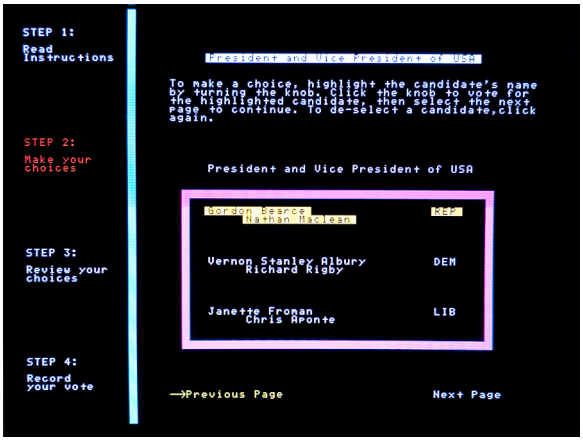

Figure 2: Screenshot of the VoteBox Nano user interface.

5 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.

5.1 Computation

We initially considered implementing the VoteBox application purely on the hardware. This would have been error-prone and unwieldy. Instead, we decided to use a MicroBlaze soft CPU (see Section 4) with hooks that make it easy to access custom logic elsewhere in the FPGA. This allowed us to reimplement a simplified VoteBox in a straightforward manner.

We didn't want to rely on the soft-CPU for the heavy-weight modular exponentiation of our 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. Luckily, our chip has dedicated multipliers in hard logic. Our design, based on earlier work [30], performs multi-precision Montgomery multiplication [27] using a technique called "coarsely integrated operand scanning [21]." We used 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 modules in the design run at 50 MHz) and performs one 1024-bit modular exponentiation operation in 20 ms on average.

5.2 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 which can display 80x60 characters at a time in any of 8 colors [31]. 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 2 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. 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, 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. The advantages of having pre-rendered GUI are covered by Yee et al. [45].

As discussed in Section 3, the progression of the VoteBox Nano user experience mimicks that of the full VoteBox. The voter is presented with one screen per race.

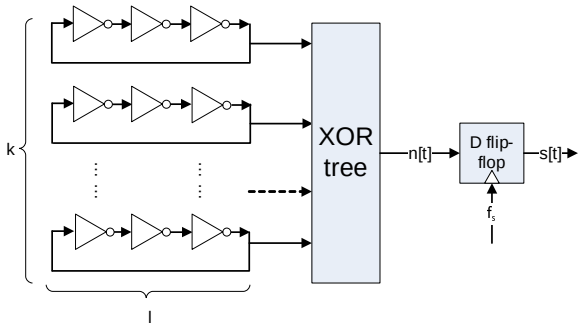

Figure 3: TRNG with ring oscillators.

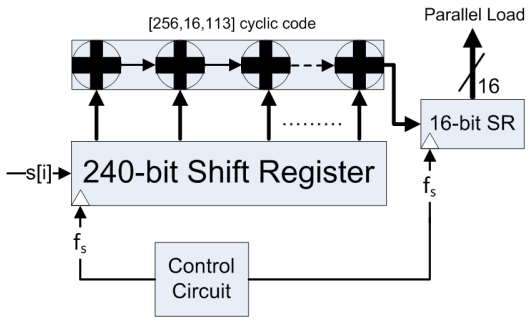

Figure 4: TRNG post-processing unit.

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].

5.3 Random number generation

Every Elgamal-encrypted value in our system requires a distinct random number. If the algorithm for random number generation is weak, the numbers could be predicted by an adversary, allowing the adversary to decrypt 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 digital for practicality. 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 around it changes.
- There are several algorithms that utilize coupled oscillators for random number generation [40, 29]. 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.

5.3.1 Random generator implementation

Given the above constraints, we chose to implement Schellekens et al. [36]'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 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 2). 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 4) 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, which increases entropy while decreasing the throughput. The XOR taps of the first register are selected according

l=3	l=5	l=7	l=9	l=11	l=13	l=15
250 MHz	155 MHz	106 MHz	81 MHz	69 MHz	59 MHz	52 MHz

Table 2: Measured frequencies with different oscillator lengths.

to a [256,16,113] cyclic code [9]. 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 256^{th} cycle, the random word ($r[t]$) of 16 bits is ready. This means our TRNG outputs random bits at around 1.56 Mb/s. For each new random word, all old bits will be replaced with the new ones, ensuring that this is a stateless machine.

In our implementation of Schellekens's TRNG, we have to violate two 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.

5.3.2 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 [26] 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 zero or very close to one (i.e., to six digits of accuracy), then that would be indicative 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 [41], which conducts a variety of statistical analyses. 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).

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.

5.4 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 from the OpenCores collection. For more specialized operations, we wrote our own modules and attached them to the system.

Table 3 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, or the RS232 and push button controllers. However we needed to remove

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

Table 3: Slice count and source code length of each FPGA module.

Resource	Used	Total	Used %
Slice	4482	4656	96
Slice Register	5060	9312	54
Slice LUT	6760	9312	72
Hardwired Multiplier	12	20	60
Block RAM	15	20	75

Table 4: FPGA resource utilization.

Code Segment	LOC	Semicolons
GUI functions	86	47
Ballot read/write	169	99
Crypto	215	155
DSA	205	159
State machine	321	220
Total	**996**	**680**

Table 5: C code size.

the CPU's instruction and data caches to fit the design into the minimum possible chip space. For the security 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. We used an off-the-shelf VGA module [31], only modifying 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 5.3.1, we had to define both the gate layout (with HDL) and create a user constraints file (UCF) to defeat the logic optimizer.

The overall device usage is shown in Table 4. Note that the total slice count is less than the sum of the slices that each module requires (see Table 3), 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 to be used by the MicroBlaze CPU to present the UI and orchestrate the steps of the voting machine. The MicroBlaze CPU, even without caches, is more than sufficiently fast for our performance needs, particularly given that the slow cryptographic operations are handled in custom hardware.

Table 5 shows the amount of C source code written for each of VoteBox Nano's major functions. The GUI functions are used to interact with the VGA display. We similarly needed a wrapper to operate our modular exponentiation unit, implementing the Elgamal cryptosystem. We wrote our own code for DSA, in which we used an off-the-shelf SHA1 function written by Niyaz [28], and an MPI (multi precision integer) library implemented by Fromberger [14] 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.

6 Tamper detection

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 [33] applies to VoteBox Nano as well.

Consequently, this paper will only consider threats that the original VoteBox made no attempt to address: attacks on the privacy of the voter. These could involve attempts to weaken the random number generation; if external observers can predict the random numbers, then they can decrypt the ballots. Attacks could also involve encoding voters' preferences directly into the random number itself, or otherwise leaking information about the plaintext values of the vote (e.g., by flashing the LEDs). We could also imagine that a clever attacker might try to modify the UI's 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 Section 2, what we fundamentally need is a mechanism whereby a VoteBox system can *attest* to the correctness of its internal state. Unlike other attestation architectures, however, we want some aspect of the attestation to be directly visible to the voters and poll workers.

6.1 JTAG

Rather than using a dedicated TPM 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 [20]. 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 the 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 paper, 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; replacing a chip requires specialized equipment.)

6.2 JTAG tamper detection

As mentioned in Section 4, our Xilinx motherboards have an onboard LCD display which can show two rows of 16 characters at a time. In a production VoteBox Nano, this secondary screen could be mounted such that it's visible to the voter. When the system is reset, our configuration will generate a random number and place it on the screen. Similarly, every time any JTAG command is processed, we will get a new random number and put it on the screen. (We were able to hook into the JTAG input pins, triggering our own logic when commands are sent.) The random number appears on the bottom line of the display in Figure 1.

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 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 6.3).

To throw off suspicion, an attacker might try to issue a JTAG command that 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 entirely, 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' procedures, once they detect this, could be to take the voting machine out of operation, or they could be to 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.

6.3 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, because this state (e.g., the CPU's registers) changes 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 use of the JTAG interface in the VoteBox Nano design. Even if an attacker were to issue a JTAG command to modify FPGA state which are passed over by iMPACT, such as flip-flops or the external DRAM, it would still change the random number display as above.

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 (whereas VoteBox is implemented in Java, and thus is robust against such attacks). 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.

7 Conclusions and Future Work

We have presented the design and prototype implementation of VoteBox Nano, an minimalist FPGA-based voting system with cryptographically strong, end-to-end guarantees that protect the integrity of votes. VoteBox Nano leverages the security properties of FPGAs to generate truly random numbers for its cryptographic operations and to safely enable the use of the JTAG interface to audit the FPGA configuration for correctness.

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 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 the use of 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.

Acknowledgements

The authors wish to thank the anonymous referees for their helpful comments and feedback. We also thank Scott Crosby and Daniel Sandler for their assistance, and we thank the program chairs for allowing us to expand our paper beyond its original length to better address the referees' concerns. This research was funded, in part, by NSF grants CNS-0524211 and CNS-0509297.

References

[1] Alkabani, Y. M., and Koushanfar, F. Active hardware metering for intellectual property protection and security. In *Proceedings of the 16th USENIX Security Symposium (USENIX Security 2007)* (Boston, MA, Aug. 2007).

[2] Avnet Electronics Marketing. *Part Orders*. http://avnetexpress.avnet.com/store/em/EMController?action=products&term=xc3s500e&N=0&langId=-1&storeId=500201&catalogId=500201&hbxSType=New+Search&x=15&y=14.

[3] Benaloh, J. Ballot casting assurance via voter-initiated poll station auditing. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)* (Boston, MA, Aug. 2007).

[4] Blaze, M., Cordero, A., Engle, S., Karlof, C., Sastry, N., Sherr, M., Stegers, T., and Yee, K.-P. *Source Code Review of the Sequoia Voting System*. California Secretary of State's "Top to Bottom" Review, July 2007. http://www.sos.ca.gov/elections/voting_systems/ttbr/sequoia-source-public-jul26.pdf.

[5] Brown, R. G. Dieharder: A random number test suite. http://www.phy.duke.edu/~rgb/General/dieharder.php.

[6] Calandrino, J. A., Feldman, A. J., Halderman, J. A., Wagner, D., Yu, H., and Zeller, W. P. *Source Code Review of the Diebold Voting System*. California Secretary of State's "Top to Bottom" Review, July 2007. http://www.sos.ca.gov/elections/voting_systems/ttbr/diebold-source-public-jul29.pdf.

[7] Chaves, R., Kuzmanov, G., and Sousa, L. On-the-fly attestation of reconfigurable hardware. In *International Conference on Field Programmable Logic and Applications (FPL 08)* (May 2008).

[8] Drimer, S., and Kuhn, M. A protocol for secure remote updates of FPGA configurations. In *5th International Workshop on Applied Reconfigurable Computing* (Karlsruhe, Germany, Mar. 2009).

[9] Duplichan, S. [256,16,113] cyclic code. ftp://ftp.win.tue.nl/pub/home/aeb/math/codes/genmats/2/2.257.17.113.

[10] Dutt, S., and Li, L. Trust-based design and check of FPGA circuits using two-level randomized ECC structures. *ACM Transactions on Reconfigurable Technology Systems 2*, 1 (2009), 1–36.

[11] Eisenbarth, T., Güneysu, T., Paar, C., Sadeghi, A.-R., Schellekens, D., and Wolf, M. Reconfigurable trusted computing in hardware. In *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing (STC 2007)* (Alexandria, VA, Nov. 2007).

[12] Elgamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory 31*, 4 (Jul 1985), 469–472.

[13] Everett, S., Greene, K., Byrne, M., Wallach, D., Derr, K., Sandler, D., and Torous, T. Is newer always better? The usability of electronic voting machines versus traditional methods. In *Proceedings of CHI 2008* (Florence, Italy, Apr. 2008).

[14] Fromberger, M. J. MPI: Arbitrary precision integer arithmetic. http://spinning-yarns.org/michael/mpi/.

[15] Gardner, R. W., Garera, S., and Rubin, A. D. On the difficulty of validating voting machine software with software. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)* (Boston, MA, Aug. 2007).

[16] Gardner, R. W., Garera, S., and Rubin, A. D. Coercion resistant end-to-end voting. In *13th International Conference on Financial Cryptography and Data Security* (Feb. 2009).

[17] Glas, B., Klimm, A., Schwab, D., Müller-Glaser, K., and Becker, J. A prototype of trusted platform functionality on reconfigurable hardware for bitstream updates. In *19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP '08)* (Monterey, CA, June 2008).

[18] Gratzer, V., and Naccache, D. Alien vs. Quine. *IEEE Security & Privaacy 5*, 2 (March–April 2007), 26–31.

[19] Inguva, S., Rescorla, E., Shacham, H., and Wallach, D. S. *Source Code Review of the Hart InterCivic Voting System*. California Secretary of State's "Top to Bottom" Review, July 2007. http://www.sos.ca.gov/elections/voting_systems/ttbr/Hart-source-public.pdf.

[20] Jack, B. Exploiting embedded systems. In *Blackhat Europe 2006* (Amsterdam, Netherlands, Mar. 2006). http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Jack.pdf.

[21] Koç, Ç. K., Acar, T., and Kaliski Jr., B. S. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro 16*, 3 (1996), 26–33.

[22] Kuon, I., and Rose, J. Measuring the gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays (FPGA '06)* (Monterey, California, 2006), pp. 21–30.

[23] Lesea, A. *IP Security in FPGAs*. Xilinx Inc., Feb. 2007. http://www.xilinx.com/support/documentation/white_papers/wp261.pdf.

[24] Maniatis, P. *Historic Integrity in Distributed Systems*. PhD thesis, Stanford University, Stanford, CA, Aug. 2003.

[25] Maniatis, P., and Baker, M. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium* (San Francisco, CA, Aug. 2002).

[26] Marsaglia, G. DIEHARD: Battery of tests of randomness. http://stat.fsu.edu/pub/diehard/.

[27] Montgomery, P. L. Modular multiplication without trial division. *Math. Computation 44* (1985), 519–521.

[28] Niyaz, P. K. Secure hash algorithm (SHA-1) reference implementation in C/C++. http://www.hoozi.com/Articles/SHA1_Source.htm.

[29] Norashikin, M. T., Illiasaak, A., and Hani, M. K. A true random number generator for crypto embedded systems. In *Regional Postgraduate Conference on Engineering and Science (RPCES)* (2006), pp. pp. 253–256.

[30] Öksüzoğlu, E., and Savaş, E. Parametric, secure and compact implementation of RSA on FPGA. In *Proceedings of the 2008 International Conference on Reconfigurable Computing and FPGAs (RECONFIG '08)* (Washington, DC, 2008), pp. 391–396.

[31] Pearson, T. OPB-compatible VGA character display, no DAC, Oct. 2007. OpenCores, http://www.opencores.org/?do=project&who=opb_vga_char_display_nodac.

[32] Sandler, D. R., Derr, K., Crosby, S., and Wallach, D. S. Finding the evidence in tamper-evident logs. In *Proceedings of the 3rd International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE '08)* (Oakland, CA, May 2008).

[33] Sandler, D. R., Derr, K., and Wallach, D. S. VoteBox: a tamper-evident, verifiable electronic voting system. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)* (San Jose, CA, 2008), pp. 349–364.

[34] Sandler, D. R., and Wallach, D. S. Casting votes in the Auditorium. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)* (Boston, MA, Aug. 2007).

[35] Sastry, N., Kohno, T., and Wagner, D. Designing voting machines for verification. In *Proceedings of the 15th USENIX Security Symposium* (Vancouver, B.C., Canada, Aug. 2006).

[36] Schellekens, D., Preneel, B., and Verbauwhede, I. FPGA vendor agnostic true random number generator. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on* (Aug. 2006), pp. 1–6.

[37] Seshadri, A., Luk, M., Perrig, A., van Doorn, L., and Khosla, P. Externally verifiable code execution. *Communications of the ACM 49*, 9 (Sept. 2006), 45–49.

[38] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM Symposium on Operating Systems Principles (SOSP 2005)* (Brighton, U.K., Dec. 2005).

[39] Seshadri, A., Perrig, A., van Doorn, L., and Khosla, P. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, CA, May 2004).

[40] Simka, M., Drutarovsky, M., Fischer, V., and Fayolle, J. Model of a true random number generator aimed at cryptographic applications. *2006 IEEE International Symposium on Circuits and Systems (ISCAS 2006)* (2006).

[41] Walker, J. ENT: A pseudorandom number sequence test program. http://www.fourmilab.ch/random.

[42] Wollinger, T., Guajardo, J., and Paar, C. Cryptography on FPGAs: State of the art implementations and attacks. *ACM Transactions on Embedded Computer Systems 3*, 3 (Aug. 2004).

[43] Xilinx. MicroBlaze soft processor core. http://www.xilinx.com/tools/microblaze.htm.

[44] Xilinx. Spartan generation boards and kits. http://www.xilinx.com/products/boards_kits/spartan.htm.

[45] Yee, K.-P. Extending prerendered-interface voting software to support accessibility and other ballot features. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)* (Boston, MA, Aug. 2007).

[46] Yee, K.-P., Wagner, D., Hearst, M., and Bellovin, S. M. Prerendered user interfaces for higher-assurance electronic voting. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '06)* (Vancouver, B.C., Canada, 2006).