On the Difficulty of Validating Voting Machine Software with Software*

Ryan Gardner† *ryan@cs.jhu.edu*
Sujata Garera† *sgarera@cs.jhu.edu*
Aviel D. Rubin† *rubin@jhu.edu*

Abstract

We studied the notion of human verification of software-based attestation, which we base on the Pioneer framework. We demonstrate that the current state of the art in software-based attestation is not sufficiently robust to provide humanly verifiable voting machine integrity in practice. We design and implement a self-attesting machine based on Pioneer and modify, and in some cases, correct the Pioneer code to make it functional and more secure. We then implement it into the GRUB bootloader [1], along with several other modifications, to produce a voting machine that authenticates and loads both the Diebold AccuVote-TS voting software as well as its underlying operating system. Finally, we implement an attack on the system that indicates that it is currently impractical for use and argue that as technology advances, the attack will likely become more effective.

1 Introduction

The Florida 2000 debacle in the United States resulted in passage of the Help America Vote Act, which provided billions of dollars in funding to the states to invest in electronic voting systems. As a result, the use of the Direct Recording Electronic (DRE) became widespread. These software-based systems have come under fire from security experts and activists claiming, among other things, that there is no way to verify that the machines do not contain malicious code. This is a valid concern. In this paper, we address a subset of this problem.

We examine the possibility of a mechanism by which a poll worker, on election day, could validate that the software in a voting machine is the software that was produced by the vendor, without modification. Our contribution highlights the difficulties in achieving voting software integrity and demonstrates that it is extremely unlikely that current software attestation techniques can provide security for electronic voting.

Validating that software has not changed is not the same as ensuring that it does not contain malicious code. Although making strong guarantees about the security or quality of the software itself may, in many ways, present more challenges than ensuring authenticity, our work does not make any claims in this regard. Other research is developing improvements in this area [6, 25, 35].

The current solution to the problem of validating that correct, unchanged software is on a DRE is inadequate. While voting machine vendors are encouraged to submit hashes of their DREs' code to the National Software Reference Library (NSRL) [2], there is unfortunately no process in place for verifying these hashes.

Even if there were a method to check the hash of the executing binary on the voting machines in the precincts, it would require complete trust in the hash verification function. Thus, the approach reduces back to the unsolved problem of ensuring that software is authentic. Potential solutions using trusted hardware, such as a processor with Intel Lagrande [19] or AMD Pacifica [4] technology and a Trusted Platform Module (TPM) [32] are similarly problematic. Although they could be used to provide securely signed hashes of the software, one would still require some completely trusted mechanism to verify these signatures.

The goal of our work is to examine the possibility of a solution whereby a human poll worker could validate that the software running on a voting machine has not been modified, without the assistance of any computing device. We describe our implementation of a voting machine verification system designed to allow for the attestation and verification of its software at boot time. Our framework employs the Diebold voting machine software that was found on the Internet in 2003 [16].

The core of our system is Pioneer [26], a software based tool for verifying the integrity of code execution. Pioneer provides a mechanism for validating that correct

*This work was supported by NSF grant CNS-0524252.
†Johns Hopkins University, Baltimore, Maryland

code is running based on some of the performance characteristics of the computer. In particular, it relies on the fact that certain operations would require a noticeable increase in computation time if code were modified. We attempt to build on this concept by extending the Pioneer primitive so that the time difference between a run with legitimate software and one with modified software is easily discernible by a poll worker.

We implement a version of the best known attack on our system, and show that to achieve the desired human verifiability, Pioneer must run uninterrupted for roughly 31 minutes on each machine before every election and the operation must be timed to within several seconds. Every voting machine in the precinct needs to undergo this procedure. We believe that our research demonstrates that this solution is not practical.

Our data and analysis differs considerably from that presented in [26]. We provide evidence that technological advances in processor architecture reduce the security of current software based attestation techniques. In particular, the increased parallelization of execution makes it difficult to achieve uniqueness of a run-time optimal implementation of a function.

2 Threat Model

Every system designed to provide security features needs to be evaluated within a threat model. As such, our work does not address all of the security problems in electronic voting. Rather, we are concerned with the issue of determining whether the software running on a voting machine on election day is the same software that was produced by the manufacturer. In our design, we make several assumptions about the capabilities of the adversary and point out that in the real world, an adversary is likely to be more powerful.

We assume that an adversary will not replace any of the hardware components in the voting system or their firmware. We limit attacks to modifications in the software. Such modifications could occur, for example, if an insider at the manufacturer changes the code after the hash values are computed and before the system is released. The software is also vulnerable to changes when the machines are in storage. It is common practice prior to elections to store voting machines overnight in the churches, synagogues and schools where the election will take place. In many instances, there are multiple people with physical access to the machines, and it is our experience (based on working at the polls) that the tamper seals on these machines are not very effective against a determined and resourceful adversary.

We assume that an adversary may be able to modify and control all of the DRE software, including the BIOS, bootloader, operating system, and voting application. In our specific implementation, we also assume that an attacker does not modify the BIOS. But, this assumption is based on our specific implementation and is not a design constraint.

Step	Poll Worker	Voting Machine
a	Enter Challenge →	
b	Start Timer	
c		Compute Self Checksum
d		← Report Checksum
e	Stop Timer	
f		Hash Voting Software
g		← Report Hash
h	Verify Time Difference	
i	Verify Checksum	
j	Verify Hash of Software	

Figure 1: Poll worker election day procedure

3 Our Approach

In this section, we describe a high level overview of our framework, without delving much into the implementation details, which are covered in a later section. One of the primary goals of this architecture is to provide human verifiable attestation.

3.1 Poll Worker Procedures

In our approach, each election day begins with a poll worker verifying that each voting machine produces a correct checksum, given a challenge, and that its software matches an authentic hash. To that end, the poll worker must be provided in advance with a trusted copy of the correct hash value. Furthermore, she must be able to produce a random, fixed-length challenge for each machine that cannot be anticipated by an adversary and for which she can verify the response. One way we envision that these values could be available to the poll worker is to provide a card with values that are hidden under a gray coating that could be scratched off by the poll worker when each machine is set up.

The hash value could be printed on the card in visible fashion, and the random challenge and corresponding response would be kept secret with the scratch-off in the same way numbers on lottery tickets are protected. To further assure randomness and unpredictability, perhaps the poll worker could pick such a card at random out of a box full of cards. There are many ways that a random challenge with a valid response could be chosen by a poll worker, but all of them require that the poll worker be given the values in advance and that the values be kept secret.

At the start of an election day, the poll worker would follow the procedures outlined in Figure 1. As illustrated, the poll worker enters a random challenge using a

key pad.[1] In response to this challenge, the voting machine displays a checksum value and a hash value as depicted in Steps d and g. The poll worker verifies that the checksum is identical to the expected response, based on the value on the scratch-off card. The poll worker also verifies that the time taken by the voting machine to compute the checksum is below a preset threshold[2] and that the hash of the software is correct. If all verification steps succeed, the machine's software would be considered authentic.

3.2 The Attestation Process

We now explain the process by which our system securely attests itself. The attestation occurs in two distinct phases:

Self-Checksumming. Shortly after the machine is powered on, the bootloader computes a checksum over all its running code, and reports the result to the user. If the response is verified to be correct, this dynamically establishes a root of trust in the bootloader.

Hashing. The now trusted bootloader computes a standard cryptographic hash over the disk image about to be booted. It then reports this hash to the user, who can verify it and thus validate that the software being booted is authentic.

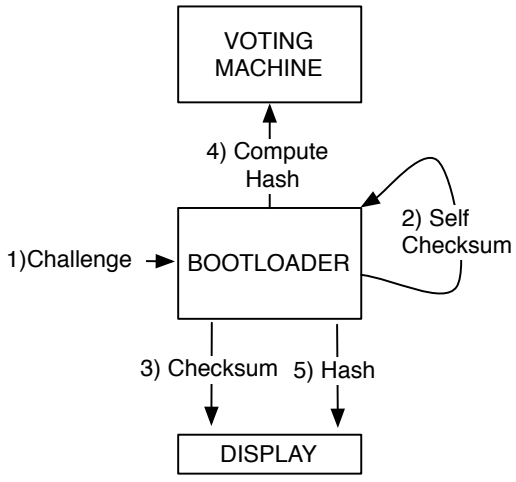

Figure 2: Proposed architecture

The attestation process is depicted in Figure 2, and we explain it in more detail below. Since the process of hashing is straightforward, we limit our discussion to the self-checksum and the attestation process as a whole.

To compute the self-checksum, we leverage Pioneer [26]. Pioneer is the work of Seshadri, Luk, Shi, Perrig, van Doorn, and Khosla. It is a challenge-response based tool originally designed for enforcing untampered code execution and produces checksums intended for verification by remote computers. Although we refer the reader to [26] for its details, we briefly describe Pioneer in the context of our work.

Pioneer can simply be thought of as an *optimally implemented* function over a challenge and the state of a specific range of memory that contains its code. It reads from pseudo-random addresses within the specified memory range and incorporates their values into its output (the checksum) and the pseudo-randomness itself. Notice that in this process, Pioneer reads and attests its own code.

The general idea behind Pioneer is that some authority will run the authentic code of interest (in our case, the honest bootloader), and can provide the correct function outputs corresponding to the "good" state of memory. These outputs can then be verified against the outputs of executions on other machines, and in this way, it operates like a standard checksum.

The critical property of Pioneer that distinguishes it from a regular checksum is its *optimality*. It was designed with the intention of preventing any code from computing the Pioneer function faster than it is currently implemented, i.e. when it is computing a legitimate checksum. Thus, an adversary may be able to write malicious code other than Pioneer that gives the same checksums for the same challenges, but that code should not run as fast as Pioneer. The verification process then makes use of this fact by requiring the verification of the Pioneer execution time as well as the correctness of its checksum. It is in this way that Pioneer can dynamically establish a root of trust in itself.

The major difference between the original Pioneer [26] and our use of Pioneer, is that we designed our version to be human verifiable. The original Pioneer provides a means of ensuring code execution on remote machines and is thus designed to run very quickly. As a result, the malicious and honest Pioneer execution times were distinguishable only by machines. In our work, we drastically increase the number of checksum iterations executed by Pioneer to force the best known attack version to run approximately 3 seconds slower than the legitimate one. With the aid of a clock, a human challenging our modified Pioneer can independently verify its integrity. We describe the implementation details of this modification in Section 4.1.1 and the resulting benchmarks from our honest and malicious implementations of Pioneer in Section 4.4.4.

3.3 Integrity Checking Location

There are several layers in the software stack where Pioneer can be placed including the BIOS, bootloader, and kernel. Note that, at higher layers, the complexity of the implementation increases. Table 1 shows the approximate size of the image that would need to be self-checksummed for each level (we approximated the size of the BIOS using the memory layout described in [3]).

Level	Image	Size
BIOS	System BIOS	8 KB
Bootloader	GRUB Stage 2	100 KB
Kernel	Linux 2.6	2.5 MB
	Windows CE	16 MB

Table 1: Approximate sizes of software images at different levels of the software stack

We place Pioneer in the GRand Unified Bootloader (GRUB) [1] both due to the small size of the bootloader image and the consistency of memory state early on in the boot process. Furthermore, the bootloader relies only on the BIOS for input and output events. Thus, by placing Pioneer in the bootloader, we avoid dealing with the complexity of the much larger input/output drivers of the kernel. This design decision fostered a clean and convenient implementation.

We compute the checksum and hash in the chainloader section of GRUB, which is the last thing to execute before GRUB loads the OS. Placing the code here minimizes the time between hash computation and voting machine use. We point out that, for our prototype, we could not place the integrity computation in the BIOS, primarily due to a lack of open source BIOS software. There is no fundamental limitation, however, that would prevent our proposed solution from being applied at the BIOS level. This may, in fact, be an ideal location for it.

Finally, we note that the time difference between the hashing of the disk image and its loading into memory does not present any immediate security threats, as such "check and use" scenarios often do. A successful verification of Pioneer guarantees that we only have a single thread of execution during this boot process, so no other code can alter the disk image between these operations. Similarly, our adversarial model excludes the addition of any hardware that might directly write to the disk.

4 Implementation

The implementation and analysis of our voting machine consists of three main pieces of software: the GRUB bootloader augmented with Pioneer, a Microsoft Windows CE operating system running the Diebold voting software, and a malicious bootloader that we built to analyze and test our implementation. We describe the considerations and challenges involved in implementing each as well as some corrections and modifications to the Pioneer code.

4.1 Self Attesting Bootloader

The bootloader is the heart of our design. It first establishes a root of trust in itself by computing a challenge-based checksum over its own code using Pioneer. It then attests and loads the voting machine. Although we use the bootloader to hash software for a voting system, our implementation could attest, load, and execute any bootable piece of code without modification.[3] We chose to use GRUB [1] as the underlying bootloader of our system due to its open source availability and high flexibility, and we explain the details and challenges of our modifications to it below.

4.1.1 User Interactions

Pioneer was originally designed to generate attestations that would be challenged and verified by a machine. Because such a trusted machine is unavailable at the polling place, we have made several modifications to Pioneer's functionality and interface so that it can interact meaningfully with humans.

The first critical change we needed to make to Pioneer concerns its execution time. According to [26], the difference in time between a checksum computed by a benign version of the original Pioneer and that of a malicious version is approximately .6 ms. We increase the number of iterations of the checksum from 2.5 million to 8.3 billion in order to increase this difference to 3 seconds, so that it is easily discernible by a poll worker. Consequently, the wait time of the poll worker increased to 31 minutes.

Our second, slight augmentation to Pioneer addresses the challenge input. The original Pioneer uses an 832 bit challenge. This is unnecessarily large and very tedious for a human to enter. Furthermore, human input is slow and no analysis has been conducted on Pioneer to determine whether having early partial knowledge of the challenge can enable an attacker to speed up its computation. To address both of these problems, we simply use the string entered by the poll worker as a key to RC4 [23], and after discarding the first 2048 bytes, the 832 bit challenge is set directly from the cipher stream. Although we envision scratch-off challenges providing approximately 64 bits of entropy, this approach allows for challenges of any size less than 256 characters.

In addition to modifying the input method, we try to simplify the output as much as possible. Like the challenge, the standard Pioneer checksum response is 832

```
...
challenge> 72944829210352831634

checksum:  MLDLFJUCSI48F

hash of booting partition:
   1UIRC2BVEMGN7G7A86SKM7F7QH89D594

Computer locked for several seconds.  Please
verify checksum and hash.
```

Figure 3: The GRUB bootloader output as displayed on the screen. The challenge was entered by a user.

bits long. To avoid clouding the verification process with unnecessary complications and possibly deterring poll workers from strictly adhering to procedure, we reduce the checksum output to 65 bits. We do this by computing a hash over the full checksum response. Because Pioneer checksums have an approximately uniform distribution over the challenge and the hash function weakly randomizes[4] these outputs, this reduction is very unlikely to produce any security threats. Finally, to minimize the number of characters that must be verified, we convert all values output, including those of the checksum and the hash, to a 5 bit ASCII character encoding consisting of characters [0-9A-NP-W]. (We omit the letter O to avoid confusion with zero.)

A sample interaction with our final bootloader is depicted in Figure 3.

4.1.2 Time of Attestation

Because our bootloader design only computes one checksum and hash, the time at which these computations are conducted is critical. We must ensure that attested code is unable to alter itself after the attestations are made. Thus, it makes the most sense to attest at a point after which, all remaining booting actions are deterministic.

In our implementation, we chose to compute the checksum and hash immediately before GRUB permanently passes control onto the voting machine bootloader. Although attesting this late in the booting process is not necessary, it allows for very easy verification that the remaining boot process is deterministic.

4.1.3 Ensuring a Closed Execution Path

Another critical aspect of our bootloader implementation concerns the possible paths of execution following attestation. We want to ensure that all such paths contain only attested code. (Of course, if the code is vulnerable, there is little we can do to prevent an adversary from executing code of her choice, so we reiterate that we are attempting to address this problem for secure software.) We approach this problem from two angles: attesting an appropriate range of memory, and strategically designing bootloader code.

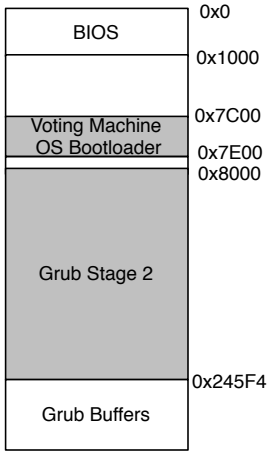

Figure 4: The layout of memory when the bootloader computes its self checksum. The shaded regions are the ones that we currently attest.

One seemingly straightforward means of achieving our goal is to simply attest all of memory. This solution is problematic in practice, however, because several memory values depend on time and other non-static quantities. Without special consideration, their attestation will yield inconsistent checksums. For this reason, we attempt to limit our attestation to a sufficient range of memory to achieve our desired goal. Furthermore, minimizing the range of attested memory also has significant advantages in allowing for the verification of software security properties although we leave focusing on this issue to future work.

The memory layout of our machine at the time of attestation is illustrated in Figure 4. Our implementation currently computes a checksum over all of GRUB stage 2, including the data within it, and the voting machine OS bootloader. From the point of attestation, GRUB stage 2 directly executes only within itself until it passes control to the OS bootloader, which will begin loading the previously hashed OS image. Hence, assuming direct execution and software without vulnerabilities, a valid checksum computation implies that any malicious code that would be executed must also be incorporated into one of the two attestations. Our implementation falls slightly short of our desired goals, however, with respect to the BIOS interrupt handlers, which may execute after attestation. We decided to leave the checksumming of the interrupt handlers to future work because we had very limited visibility of the BIOS internals and its code contained several non-static values. We emphasize, however, that this is not a limitation of our design, and with further study of the BIOS, this code could easily be included in

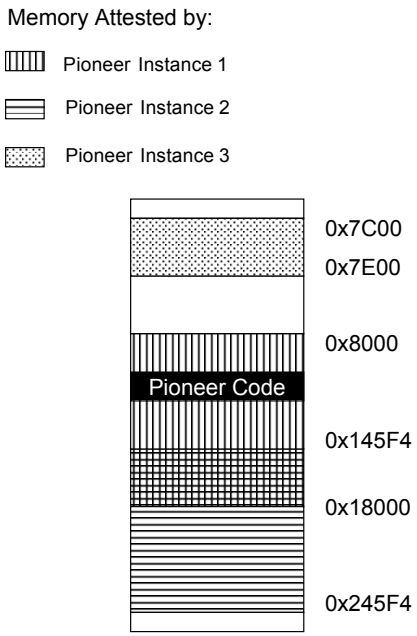

Figure 5: The memory regions attested by each instance of Pioneer. The first section of contiguous memory being checked represents the voting machine OS bootloader and the second one represents GRUB stage 2. The three instances of Pioneer checksumming code reside in the black box and attest the three marked regions in the order that they are numbered.

the attestation.

Although it seems like a reasonable solution, attesting the proper range of code is not sufficient to prevent undesired execution on its own. We must also protect the stack. For example, an adversary could still put a malicious return pointer on the stack and then jump to the attestation code in the middle of the correct bootloader. After computing the correct checksum, the bootloader would then return to the adversary's malicious code. Since achieving consistent checksums on the stack is quite challenging, the second way we ensure a closed execution path is with code design. We explicitly modify the bootloader and write our code to avoid accessing old data from the stack after attestation. We replace calls and returns with pairs of jumps or sometimes manually push correct return addresses before returning. Finally, we move necessary stack variables to the data segment, where they are incorporated into the checksum.

4.1.4 Checksum Chaining

One fundamental limitation of Pioneer is that it can only directly compute a checksum over a continuous memory range of n bytes, where n is a power of 2. However, as noted in Section 4.1.3, we want our system to attest all of GRUB stage 2 and the voting machine OS bootloader, which, in total, does not have such nice form. To attest the desired memory range, we run Pioneer several times in succession, feeding the output from one instance to the challenge input of another. (These are the same size.) The individual memory ranges we attest, as well as the order in which they are attested, are illustrated in Figure 5.

A slight complication occurs where Pioneer's security properties only hold when it computes a checksum over its own executing code. For this reason, of the three Pioneer instances we run, only the first securely establishes a root of trust in itself. We mitigate this fact, however, by using the first instance to attest all three sections of Pioneer code, establishing trust in each. From this point, the second two instances of Pioneer can reliably attest their target memory regions.[5]

As a last point of clarification, there are also no guarantees about the run time of malicious versions of Pioneer when it does not attest itself. Thus, to ensure that any attack on our system will still be detectable by humans, we only increase the number of checksum iterations executed by the first instance of Pioneer. We leave the other two to compute the original 2.5 million iterations, so they constitute a negligible .004% of the total Pioneer runtime.

4.2 Pioneer Code

Some of the most notable aspects of our design are actually modifications and results regarding the Pioneer code itself. Below we discuss how, in our implementation, we make it functional and more secure.

We used the Pioneer source code downloaded from [27], and confirmed with one of the authors that it was the final code they used for obtaining the results in [26].

4.2.1 Functionality

One of the initial sources of inconsistencies in our checksums was not caused by differences of values stored in memory, but by a subtle result of an instruction sequence in Pioneer itself. This sequence follows:

```
xor %r14, %r12
pushfq
```

It computes an `xor` of two values and pushes the `rflags` register onto the stack, which is later incorporated into the checksum. The *Intel Architecture Software Developer's Manual* [18], states that the value of the auxiliary carry flag (`AF`) (in the `rflags` register) is undefined after an `xor` instruction. It could have been possible that although undefined, `AF` is still deterministically

set after the execution of an `xor`. However, we found that this was not case in practice on our Intel Core 2 Duo processor. It may be the case that this same phenomenon does not occur on the processor used by the Pioneer authors, an Intel Pentium IV Xeon with EM64T extensions, but Intel manuals define the result the same [18].[6]

As with most of the challenges of our implementation, fixing this bug was much easier than finding it. We simply precede the `pushfq` instruction by an `add`, which leaves no flags undefined.

4.2.2 Security

The Pioneer code that was released is not quite as secure as Pioneer in design because it does not use code blocks of exactly 128 bytes to prevent simple insertion of code. The authors noted this fact [26].

We found that this lack of perfect alignment allows for a significant attack on the Pioneer code as it was released, and thus on our voting machine: After the `rflags` register is pushed onto the stack, it is added to the next memory address to be incorporated into the checksum. Thus, an adversary who can force the `rflags` register to have a substantially higher value than what is used by an honest Pioneer execution could cause the checksum to be computed over an unintended, high memory range. Ensuring that `rflags` consistently contains a higher value than was intended can be done by executing Pioneer at the lowest privilege level, for example. Doing so sets the IO Privilege Level (`IOPL`) bits of `rflags`, increasing the attested address range by 0x3000 bytes.

This increase in the checksummed address range comes at no cost in execution time on its own. The value of the address is, however, incorporated into the checksum later, so we must add one instruction to reduce it back to its expected value. This instruction could be an `xor`. Hence, the total cost of the attack is one `xor` computation per checksum iteration, approximately twice as fast as the two shift instructions required by the fastest attack described in [26].

To address this attack, we modify the Pioneer checksum code to simply incorporate the value of the `rflags` register directly into the checksum, rather than using it to calculate a memory address. The specific changes we made to the code are illustrated in Figure 6.

4.3 Windows CE Voting Kiosk

In order to develop a complete prototype, we use the Diebold source code analyzed by Kohno *et. al* [21] along with Windows CE 6.0 to build a voting machine kiosk.

```
1:  xor %r14, %r12
3:  pushfq
4:  add (%rsp), %rbx
      …
5:  add (%rbx), %r15
```

(a) Original code.

```
1:  xor %r14, %r12
2:  add %r12, %r15
3:  pushfq
4:  xor (%rsp), %r14
      …
5:  add (%rbx), %r15
```

(b) Our modified code.

Figure 6: Our modifications to a snippet of Pioneer code. We insert an `add` instruction at step 2 to define the flags before they are pushed. In step 4, we `xor` the flags into part of the checksum rather than adding them to the address of memory about to be read.

4.3.1 Integrating Diebold with Windows CE 6.0

The Diebold source was written to be compatible with an older version of Windows CE. We modify the code slightly to ensure compatibility with CE 6.0. A key advantage of using Windows CE, as opposed to Windows XP, is that Windows CE enables the entire operating system to be built into a read only image. This feature specifically enables the consistency of the boot process as discussed in the following section. The revised Diebold executable was incorporated into this read only image at compile time.

We further modify the CE platform infrastructure to provide for a kiosk environment. Specifically, we only incorporate into the CE image those features that are required by the voting machine, and we ensure that the OS image boots into the voting software rather than the CE shell.

One point to note is that while the executable and the necessary DLL files are compiled into the Windows CE image, the ballot definition file is passed externally. This design decision is discussed in the following section.

4.3.2 Achieving Read Only and Consistent Boot

The implementation of Diebold we worked with requires the administrator to configure the location of the ballot definition file. This file is usually distributed on election day via removable media such as storage cards. In addition, the software writes intermediate voting results, to a directory specified by the administrator. This ensures that if a machine shuts down during an election, it can resume tabulating votes by restoring the correct totals from persistent storage.

Given that we take a hash of the disk before it boots into the voting machine, it is essential that the disk image is consistent on all reboots. While the read only

nature of the OS image achieves this consistency, writing the votes to that image's partition would result in inconsistent hashes. To deal with this issue, we pass the ballot definition externally through a USB flash disk and configure the voting machine to write the votes to that location.[7] This infrastructure not only ensures integrity of the Windows CE image but also addresses the issue of providing a consistent hash on boot.

4.4 Malicious Bootloader

In order to demonstrate the human verifiability of our system and yet its overall impracticality, we also implement an attack against our self attesting bootloader, and show that its effects can be easily detected by humans but only after executing for an excessive amount of time. When run, our malicious bootloader reports the same checksum and hash values as the honest bootloader for a given challenge, but does not boot the voting machine. As a result, it takes slightly more time to report the checksum than the honest version. To our knowledge, this is the first full implementation of a malicious version of Pioneer.[8]

4.4.1 Attack Overview

Our attack represents a variation of the memory copy attacks described in [26], and its methodology is quite simple. We load an image of our honest bootloader into memory one megabyte above where it would normally reside and we execute a malicious version of Pioneer from its normal location. Figure 7 shows the corresponding memory layout for this attack. To compute the checksum over the memory of the honest bootloader rather than the malicious one, we `xor` the next checksum target address by 0x100000 (1 megabyte) immediately prior to memory read. Once the memory has been accessed, we `xor` the address by the value again to undo the change before the address itself is also incorporated into the checksum.

The execution overhead incurred by our malicious bootloader is two `xor` instructions per checksum iteration. This can be compared to the attack, described by the Pioneer authors as the best known attack, that alternatively requires the addition of two shifts. (The Pioneer authors did not neglect a possible `xor` attack. This comparable attack does not work against the unimplemented Pioneer design without modification, but its methodology is analogous to that of the shift attack.)

Both `xor` and shift have an execution latency of one CPU cycle [17], but the `xor` instruction is 7 bytes longer than that of shift [18]. Since the Pioneer code is only 1024 bytes, however, it should be loaded entirely into the CPU cache after the first checksum iteration, so the difference in instruction length is insignificant.

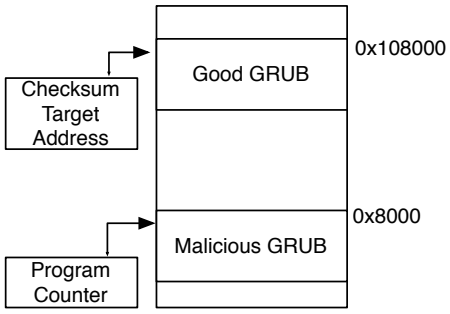

Figure 7: Layout of memory as used by the malicious bootloader.

4.4.2 Obtaining an Image of "Good" Memory

In order to implement the attack on our system, we needed an image of the honest bootloader's memory at the time of attestation. Obtaining such an image was difficult for several reasons. We could not use a debugger at the low, bootloader level,[9] and we did not have any specialty, memory reading hardware. We considered the approach of manually constructing the desired image from the bootloaders' binary files, but we found that such an approach was impractical. Specifically, in the time between when the bootloaders' binaries are loaded into memory and when Pioneer is executed, a number of variables in the data segment have been altered as well as many others that are embedded throughout bootloader assembly code. Furthermore, adding code to the bootloader to read the memory would substantially alter the image we were trying to obtain. Code directly appended to the binary was overwritten before it could be used.

To circumvent these problems, we decided to embed the code for reading the memory image into the `menu.lst` GRUB configuration file. `menu.lst` is used by GRUB to generate a user menu, and several parts of it are loaded into dedicated locations in unattested memory.[10] We inserted a small piece of code into the file, and manually edited one of the GRUB binaries to call the code in a way such that we could easily undo this minor change in the final memory image. Running the resulting bootloader with the modified `menu.lst` file successfully read an image of the desired memory and wrote it to some unused sectors of our hard disk.

4.4.3 Mirroring Memory Changes

The last challenge we faced in completing our attack was ensuring that the image of the bootloader's memory always replicated that of the real, executing bootloader at

Bootloader Version	Mean	Std. Dev.
Good	1873.230323 s	.110441
Malicious	1876.191313 s	.000232

Table 2: Checksum computation time statistics for the good and malicious versions of the bootloader. Ten measurements were taken for each bootloader version.

corresponding points in time.

Despite the fact that it seemed we had a perfect image of what was attested by the good bootloader, that image still differed from the memory that was actually executed, corrupting the malicious checksum. Determining the causes of these discrepancies was a rather challenging task. As before, we had no way of observing memory without severely altering its state. Fortunately, one thing we could do was alter the range of memory being attested without significantly changing its contents. To do this meaningfully, the new attested range must exclude the code constants that define it. So our best approach to locate memory inconsistencies was to adopt a binary search technique.

We began by attesting a small memory range where both the good and malicious bootloader would yield the same checksum and then increased this range in a binary search fashion, checking, at each step, whether the checksums remained the same or differed. Once we honed in on the location of a specific discrepancy, we analyzed the disassembled binaries and bootloader source code to try to determine the cause of the difference.[11]

We briefly point out two such discrepancies as examples:

- GRUB saves the location of the top of the protected mode stack in line with part of its code when switching to real mode. The function we called to write the good memory image to disk executes partially in real mode, and as a result altered this value before it was read.
- A dynamic piece of the Pioneer checksum is stored in the data segment, which is an attested portion of memory. Without specific modification, the malicious Pioneer alters this value in its own data segment rather than that of the good Pioneer image it is attesting.

After addressing all such discrepancies, we were able to perfectly mirror the execution of the honest bootloader in the memory image used by our malicious version.

4.4.4 Benchmarks

We measured the checksum computation time of both of our bootloaders in order to analyze the human verifiability and practicality of our attestations. All of our timing measurements were conducted on a 1.86 GHz, Intel Core 2 Duo processor with hyperthreading disabled and only one running CPU core. To obtain our numbers, we used slightly modified versions of the bootloaders, which we augmented to report timing information.

We obtained our measurements using the CPU clock cycle counter since the bootloader does not maintain a system clock. The reported times were computed as the number of clock cycles that executed during the computation of a checksum divided by the CPU clock cycle speed (1862.054 MHz). Although this method does not account for minor variations in the clock speed, these are minimal and are certainly undetectable by humans.

We recorded the computation time of 10 malicious and 10 good checksum computations. Our results are presented in Table 2. Each execution takes approximately 31 minutes. As can be seen by the minimal standard deviations, the times for each type of checksum are quite consistent. They differ by 358.251 milliseconds in the worst case. We observe that the standard deviation for the good run-times is significantly higher than that of the malicious run-times. We attribute this discrepancy to the fact that our modified benchmarking bootloader stores the execution start time in attested memory, bringing inconsistency to each checksum computation. Furthermore, the exit time from the Pioneer main loop is actually slightly probabilistic based on the state of memory being attested. Recall that the malicious benchmarking Pioneer, on the other hand, always attests the same saved copy of the good Pioneer's memory.

We presume that the slight variation between the run-times of the malicious executions is caused by the fact that there are instances during each timing where interrupts are enabled (for instance when the checksum is printed to the screen), and pending interrupts may consume a small number of clock cycles. The difference between the two average execution times is 2.96099 seconds, approximately the 3 seconds we targeted. Although this difference is discernible by a human, the roughly 31 minute wait time is unfortunately long and probably prohibitive for practical use.

Overall, the execution times are quite surprising. The results of the original Pioneer paper [26] show an approximate 1.23% overhead in execution time for a malicious version of Pioneer. Hence, when we began this work, we hypothesized that we could allow the bootloader to compute its checksum for approximately 8 minutes to create a 5.7 second difference between honest and malicious executions. However, we observe a malicious computation time overhead of only about 0.16%, requiring the approximately 31 minute execution time. The exact source of this difference in results is unknown to us with absolute certainty.

The most probable explanation, however, concerns

CPU architecture. The original Pioneer study [26] employs an Intel Pentium IV Xeon while we run a newer generation Intel Core 2 Duo. As Moore's Law nears the end of its applicability, microprocessor manufacturers and researchers have increasing incentives to maximize the parallelization of instruction execution. Continual advances in this area, will make unique code optimality more difficult to achieve and increase the effectiveness of attacks that require the insertion of instructions. We hypothesize that as technology advances the already minute overhead of the attack on our current implementation will diminish further, demanding longer run times. If attack instructions can be parallelized completely, the attack overhead on Pioneer may even reach 0. Because we cannot expect or require voting machine vendors to use legacy hardware, the current Pioneer implementation is not suitable for providing human verifiable integrity of voting machines.

5 Related Work

The notion of a *trusted computer system* was introduced in the early 1980s by Tasker [31]. Since then, significant research has been conducted in the areas of software attestation, verifiable code execution and the application of these techniques to voting integrity in general. We discuss both hardware and software based techniques here.

Hardware Based Techniques. Tygar and Yee were amongst the first to explore how secure coprocessors could be applied to protect workstation integrity and to remotely establish trust [33, 34]. Similarly, Sailer, Zhang, Jaeger, and van Doorn [24] describe a remote load time integrity checking technique using Trusted Platform Modules (TPMs) [32].

Gasser, Goldstein, Kaufman and Lampson were the first to fully authenticate the platform's software stack [15]. They proposed a distributed architecture where each machine is equipped with trusted hardware containing a public/private key pair. This is very similar to the later proposed TPM [32]. The key pair is used to sign attestations (hashes) of all code that the system boots. This attestation chain continues at higher levels in a recursive manner.

Terra followed with a very similar approach to attestation, by Garfinkel, Pfaff, Chow, Rosenblum, and Boneh [14]. It also uses chains of attestations up the software stack and attempts to isolate individual virtual systems by running each in a separate virtual machine. While Microsoft NGSCB, [7, 10, 11] is similar in functionality to Terra, it only attests software at the application level rather than the whole stack.

Petri, Fraser, Molina, and Arbaugh take another approach through Copilot [22]. This system uses an additional PCI card to perform periodic integrity measurements of the Linux kernel in memory. Unfortunately, the PCI card is unable to observe internal CPU state, so it becomes unaware of memory mappings in use by hardware, and is thus vulnerable to a kernel relocation attack [26].

Like Pioneer, Cerium [8] and BIND [30] are two techniques designed to guarantee the execution of attested code to a remote party. BIND emphasizes the ability to attest code at a "fine granularity" and relies on secure hardware to bootstrap trust in a machine. Cerium's security, on the other hand, is centered around a physically tamper resistant CPU with an embedded public/private key pair and a μ-kernel that runs within CPU cache.

One problem of all the attestation solutions mentioned above is that they require an external source of verification. In the voting setting, this source must also be trusted.

AEGIS, an architecture proposed by Arbaugh, Farber, and Smith, unlike other hard ware solutions, provides secure boot instead of trusted boot [5]. By adding a PROM card and slightly modifying the system BIOS, the authors ensure that successful boot of an AEGIS machine implies that it has run only correct, expected software. While AEGIS blends well into the voting setting, its most notable setback is that it requires trust of a very small portion of code in the BIOS. Some of the software based techniques discussed below can help address this issue.

Software Based Techniques. Kennel and Jamieson proposed a software based technique, Genuity, to establish the authenticity of a machine to a remote party [20]. Genuity relies on self-checksumming code and attempts to detect if the code runs in the correct location and if it is running in a simulator. Unfortunately, Shankar, Chew, and Tygar implement a successful attack on Genuity, and argue that the authors' approach of relying on machine specific computations to detect execution in a simulator is flawed [29].

Subsequently, Sheshadri, Perrig, van Doorn, and Khosla proposed another software based attestation technique, SWATT, primarily intended for use in embedded devices [28]. In SWATT, attesting devices conduct pseudo-random memory walks based on challenges from verifiers. The verifiers rely on the time of the memory walk as a means of ensuring that a correct walk was not simulated by malicious code. SWATT is very similar to Pioneer [26] and may also provide a possible foundation for human verifiable voting machine software attestation with some modifications. However, we chose Pioneer due to its more immediate application to a PC and guaranteed code execution.

We point out that the idea of deploying Pioneer in

voting has been suggested before. In an online PowerPoint presentation, many of the authors of Pioneer itself also suggest increasing the time of the Pioneer checksum computation to enable it to conduct human verifiable attestations of voting software [12]. They also claim to have implemented Pioneer on an Intel XScale-PXA255 processor where they report an attack computation overhead of approximately 40% in contrast with the reported 2% incurred on the original 64-bit x86.

Similarly, along with Wallach, in his comp527 computer systems security class, Eakin and Smith propose a project using Pioneer to attest software on voting machines [9]. They propose to re-engineer Pioneer to make it platform independent or run on 32-bit x86 and maintain the original Pioneer model with a remote verifier.

Garay and Huelsbergen propose a final software based integrity checking scheme [13]. Similar to Pioneer and SWATT, it uses a challenge, response protocol between the attesting system and the verifier where the time of response is critical to verification. However, the Garay-Huelsbergen technique uses executable programs (TEAS) as the challenges, which the attesting system must then execute. Because execution of downloaded code clearly presents a blaring security threat without establishing the certain integrity of the executable, this approach is not immediately applicable to voting.

Voting Machine Integrity. Along with work on verifying general software integrity, research has also been conducted towards ensuring the integrity of voting machines specifically. Sastry, Kohno, and Wagner describe methods of designing voting machines for verification in [25]. They focus on reducing the trusted computing base of a voting machine by structuring code in a way such that small sections of it can ensure the achievement of certain properties. Yee, Wagner, Hearst, and Bellovin propose a design for prerendering voting machine user interfaces so that each election's user interface may be verified prior to the election [35]. This work again aims to minimize the trusted code base and complexity of the voting machine software, motivated by the fact that a large portion of the machine software is typically the user interface. Finally, the "frogs" voting architecture, proposed by Bruck, Jefferson, and Rivest [6], separates the voting process into two physically independent stages: vote generation and vote casting. This approach, once again, drastically reduces the amount of code that must be trusted to ensure election integrity, since no trust is required of the vote generation module.

6 Conclusion and Future Work

We present a candidate architecture for verifying the validity of the software running on voting machines and implement it using Pioneer [26]. We implement an attack on our system and demonstrate that Pioneer [26] does not provide a practical solution to establishing self trust in the voting context. As CPU technology advances, we hypothesize that execution parallelization will increase, making unique code optimality harder to achieve.

As future work, we aim to further explore the possibility of software based attestation with hopes of improving these technologies and examining them in applications that may prove more practical than that of electronic voting machines. We believe that by exploring methods of forcing heavy interdependence between instructions, we may be able to prevent successful parallel execution of attack code. Some preliminary considerations indicate that by designing a self-checksumming function that modifies its own code, we may be able weaken many of the assumptions required by Pioneer [26]. One challenge that this approach must overcome, however, is that invalidation of the CPU instruction cache, caused by code modifications, could cause significant slowdowns in legitimate execution overhead. A valid solution must ensure that an attacker could not implement the function in a manner that avoids invalidating the instruction cache to gain an overall advantage in run-time. We have yet to determine the possibility of such a solution. Finally, we hope to examine the possible benefits of a self checksumming function that is bound by memory accesses rather than CPU instructions. This type of function may also be able to avoid many of the attack strengths derived from parallelization by pushing the execution bottleneck to the asynchronously accessed memory bus.

Acknowledgments

This work was supported by the National Science Foundation grant CNS-0524252. We thank Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla for developing Pioneer, providing its code, and replying to our questions. We also thank the EVT program committee and specifically the anonymous reviewers and Andrew Appel for providing us with critical insights to our work. And we thank Fabian Monrose for his helpful discussions.

References

[1] Grub: The grand unified bootloader. `http://www.gnu.org/software/grub/`.

[2] The national software reference library. `http://www.nsrl.nist.gov/vote.html`.

[3] Physical memory layout of the PC. http://my.execpc.com/~geezer/osd/ram/index.htm.

[4] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual*, volume 2: System Programming. September 2006.

[5] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, 1997.

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

[7] A. Carroll, M. Juarez, J. Polk, and T. Leininger. Microsoft palladium: A business overview, August 2002.

[8] B. Chen and R. Morris. Certifying program execution with secure processors. In *USENIX HotOS Workshop*, 2003.

[9] B. Eakin, B. Smith, and D. Wallach. Code verification for electronic voting machines. Rice University, comp527. https://sys.cs.rice.edu/course/comp527/PioneerVotingProjectProposal, Fall 2006.

[10] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *IEEE Spectrum*, 36(7):55–62, 2003.

[11] P. England and M. Peinado. Authenticated operation of open computing devices. In *ACISP '02: Australian Conference on Information Security and Privacy*, 2002.

[12] J. Franklin, M. Luk, A. Seshadri, and A. Perrig. Securely using untrusted terminals and compromised machines with human-verifiable code execution. http://www.cs.cmu.edu/~jfrankli/talks/human-verifiable-code-execution_stanford.ppt.

[13] J. A. Garay and L. Huelsbergen. Software integrity protection using timed executable agents. In *ASIACCS '06: ACM Symposium on Information, Computer, and Communications Security*, 2006.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles*, 2003.

[15] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *NIST/NCSC National Computer Security Conference*, 1989.

[16] B. Harris. *Black Box Voting: Vote Tampering in the 21st Century*. Elon House/Plan Nine, July 2003.

[17] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. November 2006.

[18] Intel Corporation. *Intel 64 and IA-32 Architectures Sofware Developer's Manual*, volume 2B: Instruction Set Reference, N-Z. November 2006.

[19] Intel Corporation. Intel trusted execution technology - preliminary architecture specification, November 2006.

[20] R. Kennell and L. H. Jamieson. Establishing the genuity of remote computer systems. In *USENIX Security Symposium*, 2003.

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

[22] J. Nick L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.

[23] R. L. Rivest. The RC4 encryption algorithm, 1992. RSA Data Security Inc.

[24] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. In *CCS '04: ACM Conference on Computer and Communications Security*, 2004.

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

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

[27] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems – codebase, 2005. http://www.cs.cmu.edu/~arvinds/pioneer.html.

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

[29] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *USENIX Security Symposium*, 2004.

[30] E. Shi, A. Perrig, and L. van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *IEEE Symposium on Security and Privacy*, 2005.

[31] P. S. Tasker. Trusted computer systems. In *IEEE Symposium on Security and Privacy*, 1981.

[32] Trusted Computing Group. TPM main part 1 - design principles, specification version 1.2, revision 94, March 2006.

[33] J. D. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proceedings*, 1994.

[34] B. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *USENIX Workshop on Electronic Commerce*, 1995.

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

Notes

[1]The Diebold DRE comes with a keypad that can be connected to the machine for accessibility. We use this keypad for the validation process to enter the challenge.

[2]To do this most easily, the poll worker could set a simple alarm to go off at the moment by which the checksum should have been displayed.

[3]All of the code for our bootloader as well as the malicious version is available at `http://cs.jhu.edu/~ryan/pioneer_voting/index.html`.

[4]In a final release of this code, we will use a pseudo-random function here, rather than a hash.

[5]We could have used hash functions in place of the second two instances of Pioneer but chose Pioneer for its short code size and to maintain the clean separation between checksumming the bootloader and hashing the disk.

[6]In their paper [26], the authors of Pioneer proceed the `pushfq` with an `or` instruction rather than `xor`. It leaves the `AF` flag undefined similarly.

[7]As an alternative approach, we could have written to a separate partition of the built-in hard disk. However, one of the main challenges we faced with regard to Windows CE was trying to get it to recognize this disk. The hard disk drivers provided by CE 6.0 were incompatible with our machine's chipset, so the USB stick provided a comparable option.

[8]Although the authors of Pioneer implemented a malicious version [26], the best evidence we have from their code [27] indicates that they probably added the two instructions necessary to make Pioneer successfully compute a correct checksum if inserted into the right memory configuration without actually setting up the full attack to do so.

[9]One approach that may have been beneficial here is to run our bootloader inside a virtual machine or emulator although most virtual machine monitors we are aware of use their own booting process and bypass the standard bootloader.

[10]This memory region does not pose a security threat since an attacker must modify the GRUB code to execute from it.

[11]This technique was also used to find inconsistencies in the checksums produced by the good bootloader when we were developing it.