Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems*

Arvind Seshadri
CMU/CyLab

Mark Luk
CMU/CyLab

Elaine Shi
CMU/CyLab

Adrian Perrig
CMU/CyLab

Leendert van Doorn
IBM

Pradeep Khosla
CMU/CyLab

ABSTRACT

We propose a primitive, called Pioneer, as a first step towards verifiable code execution on untrusted legacy hosts. Pioneer does not require any hardware support such as secure co-processors or CPU-architecture extensions. We implement Pioneer on an Intel Pentium IV Xeon processor. Pioneer can be used as a basic building block to build security systems. We demonstrate this by building a kernel rootkit detector.

Categories and Subject Descriptors: Software, Operating Systems, Security and Protection, Verification.

General Terms: Security.

Keywords: Verifiable Code Execution, Software-based Code Attestation, Dynamic Root of Trust, Rootkit Detection, Self-checksumming Code.

1 INTRODUCTION

Obtaining a guarantee that a given code has executed untampered on an untrusted legacy computing platform has been an open research challenge. We refer to this as the problem of *verifiable code execution*. An untrusted computing platform can tamper with code execution in at least three ways: 1) by modifying the code before invoking it; 2) executing alternate code; or 3) modifying execution state such as memory or registers when the code is running.

In this paper, we propose a *software-based* primitive called Pioneer[1] as a first step towards addressing the problem of verifiable code execution on legacy computing platform without relying on secure co-processors or CPU architecture extensions such as virtualization support. Pioneer is based on a challenge-response protocol between an external trusted entity, called the *dispatcher*, and an untrusted computing platform, called the *untrusted platform*. The dispatcher communicates with the untrusted platform over a communication link, such as a network connection. After a successful invocation of Pioneer, the dispatcher obtains assurance that: 1) an arbitrary piece of code, called the *executable*, on the untrusted platform is unmodified; 2) the unmodified executable is invoked for execution on the untrusted platform; and 3) the executable is executed untampered, despite the presence of malicious software on the untrusted platform.

To provide these properties, we assume that the dispatcher knows the hardware configuration of the untrusted platform, and that the untrusted platform cannot collude with other devices during verification. We also assume that the communication channel between the dispatcher and the untrusted platform provides the property of *message-origin authentication*, i.e., the communication channel is configured so that the dispatcher obtains the guarantee that the Pioneer packets it receives originate from the untrusted platform. Furthermore, to provide the guarantee of untampered code execution, we assume that the executable is self-contained, not needing to invoke any other software on the untrusted platform, and that it can execute at the highest processor privilege level with interrupts turned off.

The dispatcher uses Pioneer to dynamically establish a trusted computing base on the untrusted platform, called the *dynamic root of trust*. All code contained in the dynamic root of trust is guaranteed to be unmodified and is guaranteed to execute in an untampered execution environment. Once established, the dynamic root of trust measures the integrity of the executable and invokes the executable. The executable is guaranteed to execute in the untampered execution environment of the dynamic root of trust. In Pioneer, the dynamic root of trust is instantiated through the *verification function,* a *self-checking* function that computes a checksum over its own instructions. The checksum computation slows down noticeably if the adversary tampers with the computation. Thus, if the dispatcher receives the correct checksum from the untrusted platform within the expected amount of time, it obtains the guarantee that the verification function code on the execution platform is unmodified.

Pioneer can be used as a basic primitive for developing security applications. We illustrate this by designing a kernel rootkit detector. Our rootkit detector uses a software-based kernel integrity monitor. Instead of using rootkit signatures or low level filesystem scans to find files hidden by a rootkit, our kernel integrity monitor computes periodic hashes of the kernel code segment and static data structures to detect unauthorized kernel changes. The trusted computer uses Pioneer to obtain a guarantee that the kernel integrity monitor is unmodified and runs untampered. When implemented on version 2.6 of the Linux kernel, our rootkit detector was able to detect all publically-known rootkits for this series of the Linux kernel.

*This research was supported in part by CyLab at the Carnegie Mellon University under grant DAAD19-02-1-0389 from the Army Research Office, by NSF under grant CNS-0509004, and by a gift from IBM, Intel and Microsoft. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, Carnegie Mellon University, IBM, Intel, Microsoft, NSF, or the U.S. Government or any of its agencies.

[1]We call our primitive Pioneer because it can be used to instantiate a trusted base on an untrusted platform.

SOSP'05, October 23—26, 2005, Brighton, United Kingdom.

An important property of Pioneer is that it enables software-based code attestation [21]. Code attestation allows a trusted entity, known as the *verifier*, to verify the software stack running on another entity, known as the *attestation platform*. The verifier and the attestation platform are usually different physical computing devices. A measurement agent on the attestation platform takes integrity measurements of the platform's software stack and sends them to the verifier. The verifier uses the integrity measurements obtained from the attestation platform to detect modifications in the attestation platform's software stack.

The Trusted Computing Group (TCG) has released standards for secure computing platforms, based on a tamper-resistant chip called the Trusted Platform Module (TPM) [24]. The verifier can use the TPM on the attestation platform to obtain the guarantee of load-time attestation, whereby the verifier obtains a guarantee of what code was loaded into the system memory initially. All code is measured before it is loaded and the measurements are stored inside the TPM. In response to an attestation request, the attestation platform sends the load-time measurements to the verifier.

The SHA-1 hash function is used as the measurement agent in TCG. The collision resistance property of SHA-1 has been compromised [25]. The adversary can exploit this vulnerability to create a good version and a malicious version of an executable with the same hash value. The adversary can then undetectably exchange the good copy of the executable with the malicious copy on the attestation platform. After obtaining the load-time measurements from the attestation platform, the verifier believes that the attestation platform loaded the good copy of the executable. In reality, the attestation platform has loaded the malicious copy. Hence, the load-time attestation guarantee provided by TCG does not hold anymore. Also, other systems that rely on the load-time attestation provided by TCG such as Terra, Intel's LaGrande Technology and AMD's Pacifica are compromised as well [4, 11, 12].

It is not possible to update the TCG measurement agent using software methods. The only way to update is to physically replace hardware. TCG is designed this way to prevent an adversary from replacing the measurement agent with a malicious version. However, this also means that whenever the cryptographic primitives used by TCG are compromised, the only way to re-secure already deployed systems is to physically replace their hardware.

The software-based code attestation provided by Pioneer does not require any hardware extensions to the attestation platform. The verifier depends on Pioneer to guarantee the verifiably correct execution of the measurement agent. Pioneer-based code attestation has three main advantages: 1) it can be updated using software methods if the underlying primitives are compromised, 2) it works on legacy systems that lack secure co-processors or other hardware enhancements to protect the measurement agent from a malicious attestation platform, and 3) it provides the property of *run-time attestation*, i.e., the verifier can verify the integrity of software running on the attestation platform at the present time. Run-time attestation provides a stronger guarantee than the TCG-based load-time attestation, since software can be compromised by dynamic attacks after loading.

The paper is organized as follows. Section 2 describes the problem we address, our assumptions, and attacker model. In Section 3, we give an overview of Pioneer. We then describe the design of the verification function and its implementation on the Intel Pentium IV Xeon processor in Sections 4 and 5, respectively. Section 6 describes our kernel rootkit detector. We discuss related work in Section 7 and conclude in Section 8.

2 Problem Definition, Assumptions & Attacker Model

In this section, we describe the problem we address, discuss the assumptions we make, and describe our attacker model.

2.1 Problem Definition

We define the problem of *verifiable code execution*, in which the dispatcher wants a guarantee that some arbitrary code has executed untampered on an untrusted external platform, even in the presence of malicious software on the external platform.

The untrusted platform has a self-checking function, called the verification function. The dispatcher invokes the verification function by sending a challenge to the untrusted platform. The verification function returns a checksum to the dispatcher. The dispatcher has a copy of the verification function and can independently verify the checksum. If the checksum returned by the untrusted platform is correct and is returned within the expected time, the dispatcher obtains the guarantee that a dynamic root of trust exists on the untrusted platform. The code in the dynamic root of trust measures the executable, sends the measurement to the dispatcher, and invokes the executable. The executable runs in an untampered execution environment, which was set up as part of instantiating the dynamic root of trust. The dispatcher can verify the measurement since it has a copy of the executable. Taken together, the correctness of the checksum and correctness of the executable measurement provide the guarantee of verifiable code execution to the dispatcher.

Even if malicious software runs on the untrusted platform, it cannot tamper with the execution of the executable. The adversary can perform an active DoS attack and thwart Pioneer from being run at all. However, the adversary cannot cheat by introducing a false negative, where the correct checksum value has been reported within the expected time to the dispatcher, without the correct code executing on the untrusted platform.

2.2 Assumptions

We assume that the dispatcher knows the exact hardware configuration of the untrusted platform, including the CPU model, the CPU clock speed, and the memory latency. We also assume that the CPU of the untrusted platform is not overclocked. In addition, the untrusted platform has a single CPU, that does not have support for Symmetric Multi-Threading (SMT). For the x86 architecture, we also assume that the adversary does not generate a System Management Interrupt (SMI) on the untrusted platform during the execution of Pioneer.

We assume the communication channel between the dispatcher and the untrusted platform provides message-origin authentication i.e., the dispatcher is guaranteed that all Pioneer packets it receives originate at the untrusted platform. Also, we assume that the untrusted platform can only communicate with the dispatcher during the time Pioneer runs. Equivalently, the dispatcher can detect the untrusted platform attempting to contact other computing platforms. We make this assumption to eliminate the *proxy attack*, where the untrusted platform asks a faster computing device (proxy), to compute the checksum on its behalf.

Assuming that the untrusted platform has only one wired communication interface, we can provide message-origin authentication and eliminate the proxy attack by physically connecting the untrusted platform to dispatcher with a cable. Also, if the untrusted platform can only communicate over a Local Area Network (LAN), the network administrators can configure the network switches such that any packets sent by the untrusted platform will reach only the dispatcher.

2.3 Attacker Model

We assume an adversary who has complete control over the software of the untrusted platform. In other words, the adversary has administrative privileges and can tamper with all software on the untrusted platform including the OS. However, we assume that the adversary does not modify the hardware on the untrusted platform. For example, the adversary does not load malicious firmware onto peripheral devices such as network cards or disk controllers, or replace the CPU with a faster one. In addition, the adversary does not perform DMA-based attacks like scheduling a DMA-write causing a benign peripheral device to overwrite the executable between the time of measurement and time of invocation.

3 Pioneer Overview

In this section, we give an overview of the verification function and describe the challenge-response protocol used to set up a dynamic root of trust on the execution platform and to obtain the guarantee of verifiable code execution.

3.1 The verification function

The verification function is the central component of the Pioneer system. It is responsible for performing an integrity measurement on the executable, setting up an execution environment for the executable that ensures untampered execution, and invoking the executable. As Figure 1 shows, the verification function has three parts: a checksum code, a hash function and a send function.

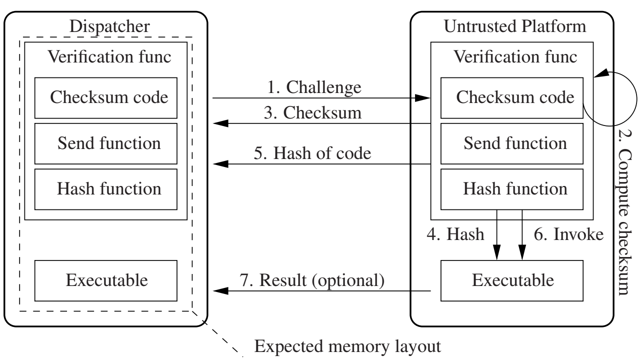


Figure 1: Overview of Pioneer. The numbers represent the temporal ordering of events.

Checksum code. The checksum code computes a checksum over the entire verification function, and sets up an execution environment in which the send function, the hash function and the executable are guaranteed to run untampered by any malicious software on the untrusted platform. The checksum code computes a fingerprint of the verification function, i.e., if even a single byte of the verification function code is different, the checksum will be different with a high probability. Thus, a correct checksum provides a guarantee to the dispatcher that the verification function code is unmodified. However, an adversary could attempt to manipulate the checksum computation to forge the correct checksum value in spite of having modified the verification function. For example, the adversary could detect when the checksum code reads the altered memory locations and redirect the read to other memory locations where the adversary has stored the correct values. To detect such manipulations, we construct the verification function such that if an adversary tries to manipulate the checksum computation, the computation time will noticeably increase. Thus, a correct checksum obtained within the expected amount of time is a guarantee to the dispatcher that the verification function code on the untrusted platform is unmodified and that there is an environment for untampered execution on the untrusted platform. In other words, the dispatcher obtains the guarantee that there is a dynamic root of trust on the untrusted platform.

Hash function. We use SHA-1 as the hash function to perform the integrity measurement of the executable. Although the collision resistance property of SHA-1 has been compromised, we rely on the second-preimage collision resistance property for which SHA-1 is still considered secure. To achieve this property, we design the hash function so that it computes the hash of the executable as a function of a nonce that is sent by the dispatcher. Thus, the adversary cannot take advantage of the compromised collision resistance property of SHA-1 to create to two different copies of the executable both of which have the same hash value. After the measurement, the hash function invokes the executable.

Send function. The send function returns the checksum and integrity measurement to the dispatcher over the communication link.

3.2 The Pioneer Protocol

The dispatcher uses a challenge-response protocol to obtain the guarantee of verifiable code execution on the untrusted platform. The protocol has two steps. First, the dispatcher obtains an assurance that there is a dynamic root of trust on the untrusted platform. Second, the dispatcher uses the dynamic root of trust to obtain the guarantee of verifiable code execution.

1.	$D:$	$t_1 \leftarrow$ current time, $nonce \xleftarrow{R} \{0,1\}^n$
	$D \rightarrow P:$	$\langle nonce \rangle$
2.	$P:$	$c \leftarrow \text{Checksum}(nonce, P)$
3.	$P \rightarrow D:$	$\langle c \rangle$
	$D:$	$t_2 \leftarrow$ current time
		if $(t_2 - t_1 > \Delta t)$ then exit with failure
		else verify checksum c
4.	$P:$	$h \leftarrow \text{Hash}(nonce, E)$
5.	$P \rightarrow D:$	$\langle h \rangle$
	$D:$	verify measurement result h
6.	$P:$	transfer control to E
7.	$E \rightarrow D:$	$\langle$result (optional)$\rangle$

Figure 2: The Pioneer protocol. The numbering of events is the same as in Figure 1. D is the dispatcher, P the verification function, and E is the executable.

We describe the challenge-response protocol in Figure 2. The dispatcher first sends a challenge containing a random nonce to the untrusted platform, initiating the checksum computation of the verification function. The untrusted platform uses the checksum code that is part of the verification function to compute the checksum. The checksum code also sets up an execution environment to ensure that the send function, the hash function and the executable can execute untampered. After computing the checksum, the checksum code invokes the send function to return the checksum to the dispatcher. The dispatcher has a copy of the verification function and can independently verify the checksum. Also, since the dispatcher knows the exact hardware configuration of the untrusted platform, the dispatcher knows the expected time duration of the checksum computation. After the send function returns the checksum to the dispatcher, it invokes the hash function. The hash function measures the executable by computing a hash over it as a function of the dispatcher's nonce and returns the hash of the executable to the dispatcher using the send function. The dispatcher also has a copy of the executable and can independently verify the hash value. The hash function then invokes the executable, which optionally returns the execution result to the dispatcher.

4 Design of the Checksum Code

In this section, we discuss the design of the checksum code that is part of the verification function. The design is presented in a CPU-architecture-independent manner. First, we discuss the properties of the checksum code, and explain how we achieve these properties and what attacks these properties can prevent or help detect. Then, we explain how we set up an execution environment in which the hash function, the send function and the executable execute untampered. In Section 5, we shall describe how to implement the checksum code on an Intel Pentium IV Xeon processor.

4.1 Required Properties of the Checksum Code

The checksum code has to be constructed such that adversarial tampering results in either a wrong checksum or a noticeable time delay. We now describe the required properties of the checksum code and explain how these properties achieve the goals mentioned above.

Time-optimal implementation. Our checksum code needs to be the checksum code sequence with the fastest running time; otherwise the adversary could use a faster implementation of the checksum code and use the time saved to forge the checksum. Unfortunately, it is an open problem to devise a proof of optimality for our checksum function. Promising research directions to achieve a proof of optimality are tools such as Denali [15] or superopt [10] that automatically generate the most optimal code sequence for basic code blocks in a program. However, Denali currently only optimizes simple code that can be represented by assignments, and superopt would not scale to the code size of our checksum function.

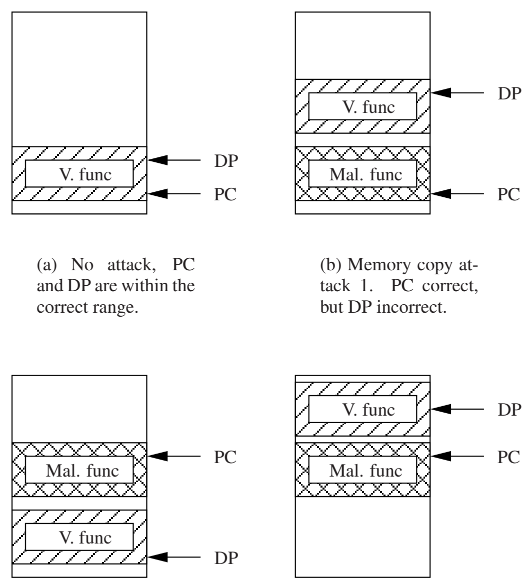
To achieve a time-optimal implementation, we use simple instructions such as `add` and `xor` that are challenging to implement faster or with fewer operations. Moreover, the checksum code is structured as code blocks such that operations in one code block are dependent on the result of operations in the previous code block. This prevents operation reordering optimizations across code blocks.

Instruction sequencing to eliminate empty issue slots. Most modern CPUs are superscalar, i.e., they issue multiple instructions in every clock cycle. If our checksum code does not have a sufficient number of issuable instructions every clock cycle, then one or more instruction issue slots will remain empty. An adversary could exploit an empty issue slot to execute additional instructions without overhead. To prevent such an attack, we need to arrange the instruction sequence of the checksum code so that the processor issue logic always has a sufficient number of issuable instructions for every clock cycle. Note that we cannot depend solely on the processor out-of-order issue logic for this since it is not guaranteed that the out-of-order issue logic will always be able to find a sufficient number of issuable instructions.

CPU state inputs. The checksum code is self-checksumming, i.e., it computes a checksum over its own instruction sequence. The adversary can modify the checksum code so that instead of checksumming its own instructions, the adversary's checksum code computes a checksum over a correct copy of the instructions that is stored elsewhere in memory. We call this attack a *memory copy attack*. This attack is also mentioned by Wurster et al. in connection with their attack on software tamperproofing [28]. The adversary can perform the memory copy attack in three different ways: 1) as shown in Figure 3(b), the adversary executes an altered checksum function from the correct location in memory, but computes the checksum over a correct copy of the checksum function elsewhere in memory; 2) as shown in Figure 3(c), the adversary does not move the correct checksum code, but executes its modified checksum code from other locations in memory; or 3) the adversary places both the correct checksum code and its modified checksum code in memory locations that are different from the memory locations where the correct checksum code originally resided, as shown in Figure 3(d).

It is obvious from the above description that when the adversary performs a memory copy attack, either the adversary's Program Counter (PC) value or the data pointer value or both will differ from the correct execution. We cause the adversary to suffer an execution time overhead for the memory copy attack by incorporating both the PC and the data pointer value into the checksum. In a memory copy attack, the adversary will be forced to forge one or both of these values in order to generate the correct checksum, leading to an increase in execution time.

Both the PC and the data pointer hold virtual addresses. The verification function is assumed to execute from a range of virtual addresses that is known to the dispatcher. As a result, the dispatcher knows the excepted value of the PC and the data pointer and can compute the checksum independently.



(a) No attack, PC and DP are within the correct range.

(b) Memory copy attack 1. PC correct, but DP incorrect.

(c) Memory copy attack 2. PC incorrect, DP correct.

(d) Memory copy attack 3. PC and DP incorrect.

Figure 3: Memory copy attacks. PC refers to the program counter, DP refers to the data pointer, V.func refers to the verification function, and Mal. func refers to the malicious verification function.

Iterative checksum code. As Figure 4 shows, the checksum code consists of three parts; the initialization code, the checksum loop and the epilog code. The most important part is the checksum loop. Each checksum loop reads one memory location of the verification function and updates the running value of the checksum with the memory value read, a pseudo-random value and some CPU state in-

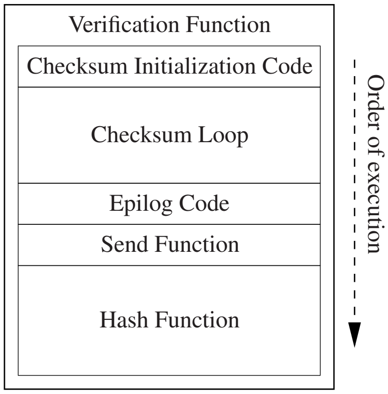


Figure 4: Functional structure of the verification function. The checksum code consists of an initialization code, the checksum loop which computes the checksum, and the epilog code that runs after the checksum loop but before the send function.

formation. If the adversary alters the checksum function but wants to forge a correct checksum output, it has to manipulate the values of one or more of the inputs in every iteration of the checksum code, causing a constant time overhead per iteration.

Strongly-ordered checksum function. A *strongly-ordered function* is a function whose output differs with high probability if the operations are evaluated in a different order. A strongly-ordered function requires an adversary to perform the same operations on the same data in the same sequence as the original function to obtain the correct result. For example, if a_1, a_2, a_3, a_4 and a_5 are random inputs, the function $a_1 \oplus a_2 + a_3 \oplus a_4 + a_5$ is strongly-ordered. We use a strongly ordered function consisting of alternate add and xor operations for two reasons. First, this prevents parallelization, as at any step of the computation the current value is needed to compute the succeeding values. For example, the correct order of evaluating the function $a_1 \oplus a_2 + a_3 \oplus a_4$ is $(((a_1 \oplus a_2) + a_3) \oplus a_4)$. If the adversary tries to parallelize the computation by computing the function in the order $((a_1 \oplus a_2) + (a_3 \oplus a_4))$, the output will be different with high probability. Second, the adversary cannot change the order of operations in the checksum code to try to speed up the checksum computation. For example, if the adversary evaluates $a_1 \oplus a_2 + a_3 \oplus a_4$ in the order $(a_1 \oplus (a_2 + (a_3 \oplus a_4)))$, the output will be different with high probability.

In addition to using a strongly ordered checksum function, we also rotate the checksum. Thus, the bits of the checksum change their positions from one iteration of the checksum loop to the next, which makes our checksum function immune to the attack against the Genuinity function that we point out in our earlier paper [21].

Small code size. The size of the checksum loop needs to be small for two main reasons. First, the code needs to fit into the processor cache to achieve a fast and deterministic execution time. Second, since the adversary usually has a constant overhead per iteration, the relative overhead increases with a smaller checksum loop.

Low variance of execution time. Code execution time on modern CPUs is non-deterministic for a number of reasons. We want a low variance for the execution time of the checksum code so that the dispatcher can easily find a threshold value for the correct execution time. We leverage three mechanisms to reduce the execution time variance of the checksum code. One, the checksum code executes at the highest privilege CPU privilege level with all maskable interrupts turned off, thus ensuring that no other code can run when the checksum code executes. Two, the checksum code is small enough to fit completely inside the CPU's L1 instruction cache. Also, the memory region containing the verification function is small enough to fit inside the CPU's L1 data cache. Thus, once the CPU caches are warmed up, no more cache misses occur. The time taken to warm up the CPU caches is a small fraction of the total execution time. As a result, the variance in execution time caused by cache misses during the cache warm-up period is small. Three, we sequence the instructions of the checksum code such that a sufficient number of issuable instructions are available at each clock cycle. This eliminates the non-determinism due to out-of-order execution. As we show in our results in Section 5.3, the combination of the above three factors leads to a checksum code with very low execution time variance.

Keyed-checksum. To prevent the adversary from pre-computing the checksum before making changes to the verification function, and to prevent the replaying of old checksum values, the checksum needs to depend on a unpredictable challenge sent by the dispatcher. We achieve this in two ways. First, the checksum code uses the challenge to seed a Pseudo-Random Number Generator (PRNG) that generates inputs for computing the checksum. Second, the challenge is also used to initialize the checksum variable to a deterministic yet unpredictable value.

We use a T-function as the PRNG [18]. A T-function is a function from n-bit words to n-bit words that has a single cycle length of 2^n. That is, starting from any n-bit value, the T-function is guaranteed to produce all the other $2^n - 1$ n-bit values before starting to repeat the values. The T-function we use is $x \leftarrow x + (x^2 \vee 5) \bmod 2^n$, where $\vee$ is the bitwise-or operator. Since every iteration of the checksum code uses one random number to avoid repetition of values from the T-function, we have to ensure that the number of iterations of the checksum code is less than 2^n when we use an n-bit T-function. We use $n = 64$ in our implementation to avoid repetition.

It would appear that we could use a Message Authentication Code (MAC) function instead of the simple checksum function we use. MAC functions derive their output as a function of their input and a secret key. We do not use a MAC function for two reasons. First, the code of current cryptographic MAC functions is typically large, which is against our goal of a small code size. Also, MAC functions have much stronger properties than what we require. MAC functions are constructed to be resilient to MAC-forgery attacks. In a MAC-forgery attack, the adversary knows a finite number of (data, MAC(data)) tuples, where each MAC value is generated using the same secret key. The task of the adversary is to generate a MAC for a new data item that will be valid under the unknown key. It is clear that we do not require resilience to the MAC forgery attack, as the nonce sent by the Pioneer dispatcher is not a secret but is sent in the clear. We only require that the adversary be unable to pre-compute the checksum or replay old checksum values.

Pseudo-random memory traversal. The adversary can keep a correct copy of any memory locations in the verification function it modifies. Then, at the time the checksum code tries to read one of the modified memory locations, the adversary can redirect the read to the location where the adversary has stored the correct copy. Thus, the adversary's final checksum will be correct. We call this attack the *data substitution attack*. To maximize the adversary's time overhead for the data substitution attack, the checksum code reads the memory region containing the verification function in a pseudo-random pattern. A pseudo-random access pattern prevents the adversary from predicting which memory read(s) will read the modified memory location(s). Thus, the adversary is forced to monitor every memory read by the checksum code. This approach is similar to our earlier work in SWATT [21].

We use the result of the Coupon Collector's Problem to guarantee that the checksum code will read every memory location of the verification function with high probability, despite the pseudo-random memory access pattern. If the size of the verification function is n words, the result of the Coupon Collector's Problem states: if X is the number of memory reads required to read each of the n words at least once, then $Pr[X > cn \ln n] \leq n^{-c+1}$. Thus, after $O(n \ln n)$ memory reads, each memory location is accessed at least once with high probability.

4.2 Execution Environment for Untampered Code Execution

We now explain how the checksum code sets up an untampered environment for the hash function, the send function, and the executable.

Execution at highest privilege level with maskable interrupts turned off. All CPUs have an instruction to disable maskable interrupts. Executing this instruction changes the state of the `interrupt enable/disable` bit in the CPU condition codes (flags) register. The `disable-maskable-interrupt` instruction can only be executed by code executing at the highest privilege level. The initialization code, which runs before the checksum loop (see Figure 4), executes the `disable-maskable-interrupt` instruction. If the checksum code is executing at the highest privilege level, the instruction execution proceeds normally and the `interrupt enable/disable` flag in the flags register is set to the `disable` state. If the checksum code is executing at lower privilege levels one of two things can happen: 1) the `disable-maskable-interrupts` instruction fails and the status of the `interrupt enable/disable` flag is not set to `disable`, or 2) the `disable-maskable-interrupt` instruction traps into software that runs at the highest privilege level. Case 2 occurs when the checksum code is running inside a virtual machine (VM). Since we assume a legacy computer system where the CPU does not have support for virtualization, the VM must be created using a software-based virtual machine monitor (VMM) such as VMware [2]. The VMM internally maintains a copy of the flags register for each VM. When the VMM gains control as a result of the checksum code executing the `disable-maskable-interrupt` instructions, the VMM changes the state of the `interrupt enable/disable` flag in the copy of the flags register it maintains for the VM and returns control to the VM. This way, the actual CPU flags register remains unmodified.

We incorporate the flags register into the checksum in each iteration of the checksum loop. Note that the adversary cannot replace the flags register with an immediate since the flags register contains status flags, such as the carry and zero flag, whose state changes as a result of arithmetic and logical operations. If the adversary directly tries to run the checksum code at privilege levels lower than the highest privilege level, the final checksum will be wrong since the `interrupt enable/disable` flag will not be set to the `disable` state. On the other hand, if the adversary tries to cheat by using a software VMM, then each read of the flags register will trap into the VMM or execute dynamically generated code, thereby increasing the adversary's checksum computation time. In this way, when the dispatcher receives the correct checksum within the expected time, it has the guarantee that the checksum code executed at the highest CPU privilege level with all maskable interrupts turned off. Since the checksum code transfers control to the hash function and the hash function in turn invokes the executable, the dispatcher also obtains the guarantee that both the hash function and executable will run at the highest CPU privilege level with all maskable interrupts turned off.

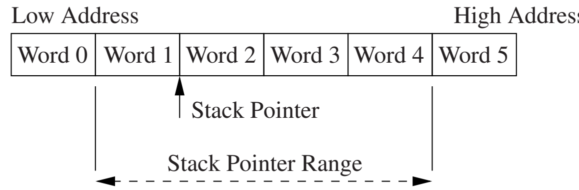


Figure 5: The stack trick. A part of the checksum (6 words long in the figure) is on the stack. The stack pointer is randomly moved to one of the locations between the markers by each iteration of the checksum code. Note that the stack pointer never points to either end of the checksum.

Replacing exception handlers and non-maskable interrupt handlers. Unlike maskable interrupts, exceptions and non-maskable interrupts cannot be temporarily turned off. To ensure that the hash function and executable will run untampered, we have to guarantee that the exception handlers and the handlers for non-maskable interrupts are non-malicious. We achieve this guarantee by replacing the existing exception handlers and the handlers for non-maskable interrupts with our own handlers in the checksum code. Since both the hash function and the executable operate at the highest privilege level, they should not cause any exceptions. Also, non-maskable interrupts normally indicate catastrophic conditions, such as hardware failures, which are low probability events. Hence, during normal execution of the hash function and the executable, neither non-maskable interrupts nor exceptions should occur. Therefore, we replace the existing exception handlers and handlers for non-maskable interrupts with code that consists only of an *interrupt return* instruction (e.g., `iret` on x86). Thus, our handler immediately returns control to whatever code was running before the interrupt or exception occurred.

An intriguing problem concerns where in the checksum code we should replace the exception and non-maskable interrupt handlers. We cannot do this in the checksum loop since the instructions that replace the exception and non-maskable interrupt handlers do not affect the value of the checksum. Thus, the adversary can remove these instructions and still compute the correct checksum within the expected time. Also, we cannot place the instructions to replace the exception and non-maskable interrupt handlers in the initialization code, since the adversary can skip these instructions and jump directly into the checksum loop.

We therefore place the instructions that replace the handlers for exceptions and non-maskable interrupts in the epilog code. The epilog code (see Figure 4) is executed after the checksum loop is finished. If the checksum is correct and is computed within the expected time, the dispatcher is guaranteed that the epilog code is unmodified, since the checksum is computed over the entire verification function. The adversary can, however, generate a non-maskable interrupt or exception when the epilog code tries to run, thereby gaining control. For example, the adversary can set an execution break-point in the epilog code. The processor will then generate a debug exception when it tries to execute the epilog code. The existing debug exception handler could be controlled by the adversary. This attack can be prevented by making use of the stack to store a part of the checksum. The key insight here is that all CPUs automatically save some state on the stack when an interrupt or exception occurs. If the stack pointer is pointing to the checksum that is on the stack, any interrupt or exception will cause the processor to overwrite the checksum. We ensure that the stack pointer always points to the middle of the checksum on the stack (see Figure 5) so that part of the checksum will always be overwritten regardless of whether the stack grows up or down in memory.

Each iteration of the checksum loop randomly picks a word of the stack-based checksum for updating. It does this by moving the stack pointer to a random location within the checksum on the stack, taking care to ensure that the stack pointer is never at either end of the checksum (see Figure 5). The new value of the stack pointer is generated using the current value of the checksum and the current value of the stack pointer, thereby preventing the adversary from predicting its value in advance.

The epilog code runs before the send function, which sends the checksum back to the dispatcher. Thereby, a valid piece of checksum is still on the stack when the epilog code executes. Thus, the adversary cannot use a non-maskable interrupt or exception to prevent the epilog code from running without destroying a part of the checksum. Once the epilog code finishes running, all the exception handlers and the handlers for non-maskable interrupts will have been replaced. In this manner, the dispatcher obtains the guarantee that any code that runs as a result of an exception or a non-maskable interrupt will be non-malicious.

5 Checksum Code Implementation on the Netburst Microarchitecture

In this section we describe our implementation of the checksum code on an Intel Pentium IV Xeon processor with EM64T extensions. First, we briefly describe the Netburst microarchitecture, which is implemented by all Intel Pentium IV processors, and the EM64T extensions. Next, we describe how we implement the checksum code on the Intel x86 architecture. Section 5.3 shows the results of our experiments measuring the time overhead of the different attacks. Finally, in Section 5.4 we discuss some points related to the practical deployment of Pioneer and extensions to the current implementation of Pioneer.

5.1 The Netburst Microarchitecture and EM64T Extensions

In this section, we present a simplified overview of the Intel Netburst microarchitecture that is implemented in the Pentium IV family of CPUs. We also describe the EM64T extensions that add support for 64-bit addresses and data to the 32-bit x86 architecture.

Figure 6 shows a simplified view of the front-end and execution units in the Netburst architecture. The figure and our subsequent description are based on a description of the Netburst microarchitecture by Boggs et al. [7].

The instruction decoder in Pentium IV CPUs can only decode one instruction every clock cycle. To prevent the instruction decoder from creating a performance bottleneck, the Netburst microarchitecture uses a trace cache instead of a regular L1 instructions cache. The trace cache holds decoded x86 instructions in the form of μops. μops are RISC-style instructions that are generated by the instruction decoder when it decodes the x86 instructions. Every x86 instruction breaks down into one or more dependent μops. The trace cache can hold up to 12000 μops and can issue up to three μops to the execution core per clock cycle. Thus, the Netburst microarchitecture is a 3-way issue superscalar microarchitecture.

The Netburst microarchitecture employs seven execution units. The load and store units have dedicated Arithmetic Logic Units (ALU) called Address Generation Units (AGU) to generate addresses for memory access. Two double-speed integer ALUs execute two μops every clock cycle. The double speed ALUs handle simple arithmetic operations like add, subtract and logical operations.

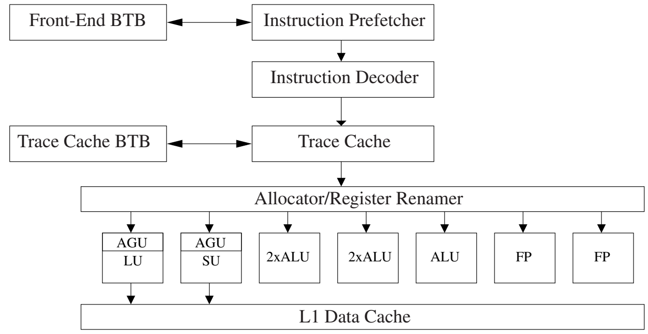


Figure 6: The Intel Netburst Microarchitecture. The execution units are LU: Load Unit; SU: Store Unit; AGU: Address Generation Unit; 2xALU: Double-speed Integer ALUs that execute two μops each per cycle; ALU: Complex Integer ALU; FP: Floating Point, MMX, and SSE unit.

The L1-data cache is 16KB in size, 8-way set associative and has a 64 byte line size. The L2 cache is unified (holds both instructions and data). Its size varies depending on the processor family. The L2 cache is 8 way set associative and has a 64 byte line size.

The EM64T extensions add support for a 64-bit address space and 64-bit operands to the 32-bit x86 architecture. The general purpose registers are all extended to 64 bits and eight new general purpose registers are added by the EM64T extensions. In addition, a feature called segmentation[2] allows a process to divide up its data segment into multiple logical address spaces called *segments*. Two special CPU registers (`fs` and `gs`) hold pointers to segment descriptors that provide the base address and the size of a segment as well as segment access rights. To refer to data in a particular segment, the process annotates the pointer to the data with the segment register that contains the pointer to the descriptor of the segment. The processor adds the base address of the segment to the pointer to generate the full address of the reference. Thus, `fs:0000` would refer to the first byte of the segment whose descriptor is pointed to by `fs`.

5.2 Implementation of Pioneer on x86

We now discuss how we implement the checksum code so that it has all the properties we describe in Section 4.1. Then we describe how the checksum code sets up the execution environment described in Section 4.2 on the x86 architecture.

Every iteration of the checksum code performs these five actions: 1) deriving the next pseudo-random number from the T-function, 2) reading the memory word for checksum computation, 3) updating the checksum, 4) rotating the checksum using a `rotate` instruction, and 5) updating some program state such as the data pointer. Except for reading the CPU state and our defense against the memory copy attack, all properties are implemented on the x86 architecture exactly as we describe in Section 4.1. Below, we describe the techniques we employ to obtain the CPU state on the x86 architecture. We also describe how we design our defense against the memory copy attacks.

CPU state inputs. The CPU state inputs, namely the Program Counter (PC) and the data pointer, are included in the checksum to detect the three memory copy attacks. On the x86 architecture with EM64T extensions, the PC cannot be used as an operand for

[2]Unlike the IA32 architecture, the EM64T extensions do not use code or stack segments. So, the `cs` and `ss` segment registers are ignored by the processor. Also, the `ds` and `es` segment registers are not used by the processor for accessing data segments.

any instruction other than the `lea` instruction. So, if we want to include the value of the PC in the checksum, the fastest way to do it is to use the following two instructions: first, the `lea` instruction moves the current value of PC into a general purpose register, and next, we incorporate the value in the general purpose register into the checksum. Since the value of the PC is known in advance, the adversary can directly incorporate the corresponding value into the checksum as an immediate. Doing so makes the adversary's checksum computation faster since it does not need the `lea` instruction. Hence, on the x86 platform we cannot directly include the PC in the checksum.

Instead of directly including the PC in the checksum, we construct the checksum code so that correctness of the checksum depends on executing a sequence of absolute jumps. By including the jump target of each jump into the checksum, we indirectly access the value of the PC.

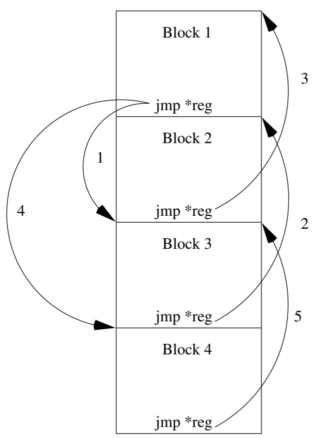


Figure 7: Structure of the checksum code. There are 4 code blocks. Each block is 128 bytes in size. The arrows show one possible sequence of control transfers between the blocks.

As Figure 7 shows, we construct the checksum code as a sequence of four code blocks. Each code block generates the absolute address of the entry point of any of the four code blocks using the current value of the checksum as a parameter. Both the code block we are jumping from and the code block we are jumping to incorporate the jump address in the checksum. The last instruction of code block jumps to the absolute address that was generated earlier.

All of the code blocks execute the same set of instructions to update the checksum but have a different ordering of the instructions. Since the checksum function is strongly ordered, the final value of the checksum depends on executing the checksum code blocks in the correct sequence, which is determined by the sequence of jumps between the blocks.

The checksum code blocks are contiguously placed in memory. Each block is 128 bytes in size. The blocks are aligned in memory so that the first instruction of each block is at an address that is a multiple of 128. This simplifies the jump target address generation since the jump targets can be generated by appropriately masking the current value of the checksum.

Memory copy attacks. Memory copy attacks are the most difficult attacks to defend against on the x86 architecture, mainly for of three reasons: 1) the adversary can use segmentation to have the processor automatically add a displacement to the data pointer without incurring a time overhead; 2) the adversary can utilize memory addressing with an immediate or register displacement, without incurring a time overhead because of the presence of dedicated AGUs in the load and the store execution units; and 3) the PC cannot be used like a general purpose register in instructions, which limits our flexibility in designing defenses for the memory copy attacks.

We now describe how the adversary can implement the three memory copy attacks on the x86 architecture and how we construct the checksum code so that the memory copy attacks increase the adversary's checksum computation time.

In the first memory copy attack shown in Figure 3(b), the adversary runs a modified checksum code from the correct memory location and computes the checksum over a copy of the unmodified verification function placed elsewhere in memory. This attack requires the adversary to add a constant displacement to the data pointer. There are two ways the adversary can do this efficiently: 1) it can annotate all instructions that use the data pointer with one of the segment registers, `fs` or `gs`, and the processor automatically adds the segment base address to the data pointer, or 2) the adversary can use an addressing mode that adds an immediate or a register value to the data pointer, and the AGU in the load execution unit will add the corresponding value to the data pointer. However, our checksum code uses all sixteen general purpose registers, so the adversary can only use an immediate to displace the data pointer.

Neither of these techniques adds any time overhead to the adversary's checksum computation. Also, both techniques retain the correct value of the data pointer. Thus, this memory copy attack cannot be detected by including the data pointer in the checksum. However, both these techniques increase the instruction length. We leverage this fact in designing our defense against this memory copy attack. The segment register annotation adds one byte to the length of any instruction that accesses memory, whereas addressing with immediate displacement increases the instruction length by the size of the immediate. Thus, in this memory copy attack, the adversary's memory reference instructions increase in length by a minimum of one byte. An instruction that reads memory without a segment register annotation or an immediate displacement is 3 bytes long on the x86 architecture with EM64T extensions. We place an instruction having a memory reference, such as `add mem, reg`, as the first instruction of each of the four checksum code blocks. In each checksum code block, we construct the jump target address so that, the jump lands with equal probability on either the first instruction of a checksum code block or at an offset of 3 bytes from the start of a code block. In an unmodified code block, the second instruction is at an offset of 3 bytes from the start of the block. When the adversary modifies the code blocks to do a memory copy attack, the second instruction of the block cannot begin before the 4th byte of the block. Thus, 50% of the jumps would land in the middle of the first instruction, causing the processor to generate an `illegal opcode` exception.

To accommodate the longer first instruction, the adversary would move its code blocks farther apart, as Figure 8 shows. The adversary can generate its jump target addresses efficiently by aligning its checksum code blocks in memory in the following way. The adversary places its code blocks on 256 byte boundaries and separates its first and second instruction by 8 bytes. Then, the adversary can generate its jump addresses by left-shifting the correct jump address by 1. We incorporate the jump address into the checksum both before and after the jump. So, the adversary has to left-shift the correct jump address by 1 before the jump instruction is executed and restore the correct jump address by right-shifting after the jump is complete. Thus, the adversary's overhead for the first memory copy attack is the execution latency of one left-shift instruction and one right-shift instruction.

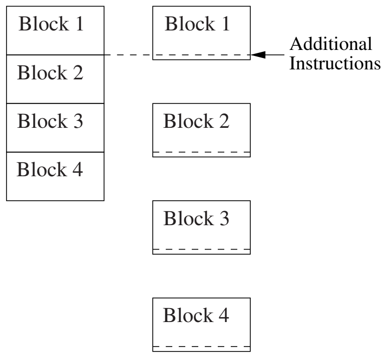


Figure 8: Comparison of the code block lengths in the original verification function and an adversary-modified verification function. The adversary moves its code blocks in memory so that the entry points of its code blocks are at addresses that are a power of two.

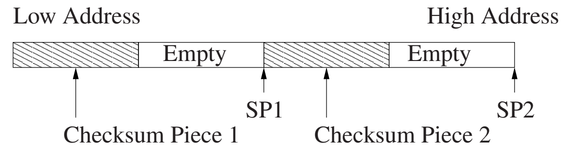


Figure 9: The layout of the stack on an x86 processor with EM64T extensions. Both checksum pieces are 8 bytes long and are aligned on 16-byte boundaries. The empty regions are also 8 bytes long. The stack pointer is assigned at random to one of the two locations SP1 or SP2.

In the second memory copy attack shown in Figure 3(c), the adversary keeps the unmodified verification function at the correct memory location, but computes the checksum using a modified checksum code that runs at different memory locations. In this case, the entry points of the adversary's code blocks will be different, so the adversary would have to generate different jump addresses. Since we include the jump addresses in the checksum, the adversary would also have to generate the correct jump addresses. Hence, the adversary's checksum code blocks would be larger than 128 bytes. As before, to accommodate the larger blocks, the adversary would move its code blocks apart and align the entry points at 256 byte boundaries (Figure 8). Then, the adversary can generate its jump address by left-shifting the correct jump address and by changing one or more bits in the resulting value using a logical operation. To restore the correct jump address, the adversary has to undo the changes either by loading an immediate value or by using a right-shift by 1 and a logical operation. In any case, the adversary's time overhead for this memory copy attack is greater than the time overhead for first memory copy attack.

In the third memory copy attack shown in Figure 3(d), both the unmodified verification function and the adversary's checksum code are not present at the correct memory locations. Thus, this attack is a combination of the first and the second memory copy attacks. The adversary's time overhead for this memory copy attack is the same as the time overhead for the second memory copy attack.

Variable instruction length. The x86 Instruction Set Architecture (ISA) supports variable length instructions. Hence, the adversary can reduce the size of the checksum code blocks by replacing one or more instructions with shorter variants that implement the same operation with the same or shorter latency. The adversary can use the space saved in this manner to implement the memory copy attacks without its code block size exceeding 128 bytes. To prevent this attack, we carefully select the instructions used in the checksum code blocks so that they are the smallest instructions able to perform a given operation with minimum latency.

Execution environment for untampered code execution. In order to get the guarantee of execution at the highest privilege level with maskable interrupts turned off, the checksum code incorporates the CPU flags in the checksum. The flags register on the x86 architecture, `rflags`, can only be accessed if it is pushed onto the stack. Since we use to the stack to hold a part of the checksum, we need to ensure that pushing the `rflags` onto the stack does not overwrite the part of the checksum that is on the stack. Also, a processor with EM64T extensions always pushes the processor state starting at a 16-byte boundary on receiving interrupts or exceptions. Thus, we need to make sure that the checksum pieces on the stack are aligned on 16-byte boundaries so they will be overwritten when an interrupt or exception occurs.

Figure 9 shows the stack layout we use for x86 processors with EM64T extensions. Our stack layout has checksum pieces alternating with empty slots. All four elements are eight bytes in size. The checksum code moves the stack pointer so that the stack pointer points either to location SP1 or to location SP2. On the x86 architecture, the stack grows downwards from high addresses to low addresses. To push an item onto the stack, the processor first decrements the stack pointer and then writes the item to the memory location pointed to by the stack pointer. With EM64T extensions, pushes and pops normally operate on 8-byte data. Since the stack pointer is always initialized to either SP1 or to SP2, a push of the `rflags` register will always write the flags to one of the empty 8-byte regions. If an interrupt or exception were to occur, the processor would push 40 bytes of data onto the stack, thereby overwriting either checksum piece 1 or both checksum pieces.

We keep checksum pieces on the stack to prevent the adversary from getting control through an exception or a non-maskable interrupt. However, the x86 architecture has a special non-maskable interrupt called System Management Interrupt (SMI), which switches the processor into the System Management Mode (SMM). The purpose of SMM is to fix chipset bugs and for hardware control.

The SMI does not save the processor state on the stack. So, it is not possible to prevent the SMI by keeping checksum pieces on the stack. Since the SMI is a special-purpose interrupt, we assume that it never occurs when the verification function runs. During our experiments, we found this assumption to be true all the time. In Section 5.4, we discuss how we can extend the current implementation of Pioneer to handle the SMI.

Description of verification function code. Figure 10 shows the pseudocode of one code block of the verification function. The block performs six actions: 1) deriving the next pseudo-random value from the T-function; 2) generating the jump address, the stack pointer, and the data pointer using the current value of the checksum, 3) pushing `rflags` onto the stack, 4) reading a memory location containing the verification function, 5) updating the checksum using the memory read value, previous value of the checksum, the output of the T-function, the `rflags` register, and the jump address, and 6) rotating the checksum using the rotate instruction.

The checksum is made up of twelve 64-bit pieces, ten in the registers and two on the stack. The checksum code uses all sixteen general purpose registers.

Figure 11 shows the assembler code of one block of the verification function. The code shown is not the optimized version but a verbose version to aid readability.

//Input: y number of iterations of the verification procedure
//Output: Checksum C, (10 segments in registers C_0 to C_9,
// and 2 on stack C_{stk_1}, C_{stk_2}, each being 64 bits)
//Variables: $[code\_start, code\_end]$ - bounds of memory address under verification
// $daddr$ - address of current memory access
// x - value of T function
// l - counter of iterations
// $rflags$ - flags register
// $jump\_target[1:0]$ - determines which code block to execute
// $temp$ - temp register used to compute checksum
$daddr \leftarrow code\_start$
for $l = y$ to 0 **do**
Checksum 1
//T function updates x where $0 \leq x \leq 2^n$
$x \leftarrow x + (x^2 \vee 5) \bmod 2^n$
//Read $rflags$ and incorporate into $daddr$
$daddr \leftarrow daddr + rflags$
//Read from memory address $daddr$, calculate checksum.Let C be the checksum vector and j be the current index.
$jump\_target \leftarrow not(jump\_target) + loop\_ctr \oplus x$
$temp \leftarrow x \oplus C_{j-1} + daddr \oplus C_j$
if $jump\_target[1] == 0 and jump\_target[0] == 0$ **then**
$C_j \leftarrow C_j + mem[daddr + 8] + jump\_target$
else
$C_j \leftarrow C_j + jump\_target$
end if
$C_{j-1} \leftarrow C_{j-1} + temp$
$C_{stk} \leftarrow C_{stk} \oplus jump\_target$
$C_{j-2} \leftarrow C_{j-2} + C_j$
$C_{j-3} \leftarrow C_{j-3} + C_{j-1}$
$C_j \leftarrow rotate\_right(C_j)$
//Update $daddr$ to perform pseudo-random memory traversal
$daddr \leftarrow daddr + x$
//Update rsp and jump_target
$rsp[1] \leftarrow C_j[1]$
$j \leftarrow (j+1) \mod 11$
$jump\_target[8:7] \leftarrow C_j[8:7]$
$jump\_target[1:0] \leftarrow temp[0], temp[0]$
if $jump\_target[8:7] = 0$ **then**
goto Checksum 1
else if $jump\_target[8:7] = 1$ **then**
goto Checksum 2
else if $jump\_target[8:7] = 2$ **then**
goto Checksum 3
else if $jump\_target[8:7] = 3$ **then**
goto Checksum 4
end if
Checksum 2
...
Checksum 3
...
Checksum 4
...
end for

Figure 10: Verification Function Pseudocode

5.3 Experiments and Results

Any attack that the adversary uses has to be combined with a memory copy attack because the adversary's checksum code will be different from the correct checksum code. Hence, the memory copy attack is the attack with the lowest overhead. Of the three memory copy attacks, the first has the lowest time overhead for the adversary. Hence, we implemented two versions of the checksum code using x86 assembly: a legitimate version and a malicious version that implements the first memory copy attack (the correct code plus two extra shift instructions).

Experimental setup. The dispatcher is a PC with a 2.2 GHz Intel Pentium IV processor and a 3Com 3c905C network card, running Linux kernel version 2.6.11-8. The untrusted platform is a PC with a 2.8 GHz Intel Pentium IV Xeon processor with EM64T extensions and an Intel 82545GM Gigabit Ethernet Controller, running Linux kernel version 2.6.7. The dispatcher code and the verification function are implemented inside the respective network card interrupt handlers. Implementing code inside the network card interrupt handler enables both the dispatcher and the untrusted platform to receive the Pioneer packets as early as possible. The dispatcher and the untrusted platform are on the same LAN segment.

Assembly Instruction	**Explanation**
//Read memory	
add (rbx), r15	*memory read*
sub 1, ecx	decrement loop counter
add rdi, rax	$x \leftarrow (x * x)$ *OR* $5 + x$
//modifies jump_target register rdx *and* rdi	
xor r14, rdi	$rdi \leftarrow rdi \oplus C_{j-1}$
add rcx, rdx	$rdx \leftarrow rdx + loopctr$
add rbx, rdi	$rdi \leftarrow rdi + daddr$
xor rax, rdx	*input x (from T function)*
xor r15, rdi	$rdi \leftarrow rdi \oplus c_j$
//modifies checksum with rdx *and* rdi	
add rdx, r15	*modify checksum* C_j
add rdi, r14	*modify checksum* C_{j-1}
xor rdx, -8(rsp)	*modify checksum on stack*
xor r15, r13	$C_{j-2} \leftarrow C_{j-2} \oplus C_j$
add r14, r12	$C_{j-3} \leftarrow C_{j-3} + C_{j-1}$
rol r15	$r15 \leftarrow rotate[r15]$
//Pseudorandom memory access	
xor rdi, rbx	$daddr \leftarrow daddr \oplus random_bits$
and mask1, ebx	*modify daddr*
or mask2, rbx	*modify daddr*
//Modify stack pointer and target jump address	
xor rdx, rsp	*Modify rsp*
and mask3, esp	*create rsp*
or mask4, rsp	*create rsp*
and 0x180, edx	$jump\_target \leftarrow r15$
and 0x1, rdi	$rdi \leftarrow rdiAND0x1$
add rdi, rdx	$rdx \leftarrow rdx + rdi$
add rdi, rdi	*shift rdi*
add rdi, rdx	$rdx \leftarrow rdx + rdi$
or mask, rdx	*create jump_target address*
xor rdx, r15	*add jump target address into checksum*
//T function updates x, at rax	
mov rax, rdi	*save value of T function*
imul rax, rax	x = x*x
or 0x5, rax	$x \leftarrow x * x$ *OR* 5
//Read flags	
pushfq	*push rflags*
add (rsp), rbx	$daddr \leftarrow daddr + rflags$
jmp *rdx	*jump to 1 of the 4 blocks*

Figure 11: Checksum Assembly Code

Empty instruction issue slots. In Section 4.1, we mentioned that the checksum code instruction sequence has to be carefully arranged to eliminate empty instruction issue slots. The Netburst Microarchitecture issues μops, which are derived from decoding x86 instructions. Hence, to properly sequence the instructions, we need to know what μops are generated by the instructions we use in the checksum code. This information is not publically available. In the absence of this information, we try to sequence the instructions through trial-and-error. To detect the presence of empty instruction issue slots we place `no-op` instructions at different places in the code. If there are no empty instruction issue slots, placing `no-op` instructions should always increase the execution time of the checksum code. We found this assertion to be only partially true in our experiments. There are places in our code where `no-op` instructions can be placed without increasing the execution time, indicating the presence of empty instruction issue slots.

Determining number of verification function iterations. The adversary can try to minimize the Network Round-Trip Time (RTT) between the untrusted platform and dispatcher. Also, the adversary can pre-load its checksum code and the verification function into the CPU's L1 instruction and data caches respectively to ensure that it does not suffer any cache misses during execution. We prevent the adversary from using the time gained by these two methods to forge the checksum.

The theoretically best adversary has zero RTT and no cache misses, which is a constant gain over the execution time of the correct checksum code. We call this constant time gain as the *adversary* *time advantage*. However, the time overhead of the adversary's checksum code increases linearly with the number of iterations of the checksum loop. Thus, the dispatcher can ask the untrusted platform to perform a sufficient number of iterations so that the adversary's time overhead is at least greater than the adversary time advantage.

The expression for the number of iterations of the checksum loop to be performed by the untrusted platform can be derived as follows. Let c be the clock speed of the CPU, a be the time advantage of the theoretically best adversary, o be the adversary's overhead per iteration of the checksum loop represented in CPU cycles, and n is the number of iterations. Then $n > \frac{c*a}{o}$ to prevent false negatives[3] in the case of the theoretically best adversary.

Experimental results. To calculate the time advantage of the theoretically best adversary, we need to know the upper bound on the RTT and the time saved by pre-warming the caches. We determine the RTT upper bound by observing the `ping` latency for different hosts on our LAN segment. This gives us an RTT upper bound of 0.25ms since all ping latencies are smaller than this value. Also, we calculate the amount of time that cache pre-warming saves the adversary by running the checksum code with and without pre-warming the caches and observing the running times using the CPU's `rdtsc` instruction. The upper bound on the cache pre-warming time is 0.0016ms. Therefore, for our experiments we fix the theoretically best adversary's time advantage to be 0.2516ms. The attack that has the least time overhead is the first memory copy attack, which has an overhead of 0.6 CPU cycles per iteration of the checksum loop. The untrusted platform has a 2.8 GHz CPU. Using these values, we determine the required number of checksum loop iterations to be 1,250,000. To prevent false positives due to RTT variations, we double the number of iterations to 2,500,000.

The dispatcher knows, r, the time taken by the correct checksum code to carry out 2,500,000 iterations. It also knows that the upper bound on the RTT, rtt. Therefore, the dispatcher considers any checksum result that is received after time $r + rtt$ to be late. This threshold is the *adversary detection threshold*.

We place the dispatcher at two different physical locations on our LAN segment. We run our experiments for 2 hours at each location. Every 2 minutes, the dispatcher sends a challenge to the untrusted platform. The untrusted platform returns a checksum computed using the correct checksum code. On receiving the response, the dispatcher sends another challenge. The untrusted platform returns a checksum computed using the adversary's checksum code, in response to this challenge. Both the dispatcher and the untrusted platform measure the time taken to compute the two checksums using the CPU's `rdtsc` instruction. The time measured on the untrusted platform for the adversary's checksum computation is the checksum computation time of the theoretically best adversary.

Figures 12 and 13 show the results of our experiments at the two physical locations on the LAN segment. Based on the results, we observe the following points: 1) even the running time of the theoretically best adversary is greater than the Adversary Detection Threshold, yielding a false negative rate of 0%; 2) the checksum computation time shows a very low variance, that we have a fairly deterministic runtime; 3) we observe some false positives (5 out of 60) at location 2, which we can avoid by better estimating the RTT.

We suggest two methods for RTT estimation. First, the dispatcher measures the RTT to the untrusted platform just before it sends the challenge and assumes that the RTT will not significantly increase in the few tens of milliseconds between the time it measures the RTT and the time it receives the checksum packet from the untrusted platform. Second, the dispatcher can take RTT measurements at coarser time granularity, say every few seconds, and use these measurements to update its current value of the RTT.

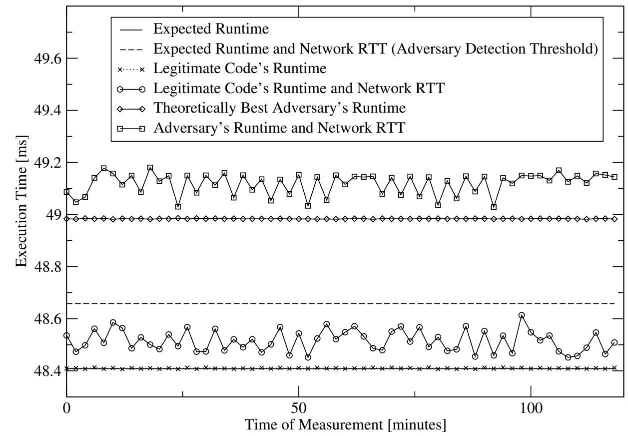


Figure 12: Results from Location 1.

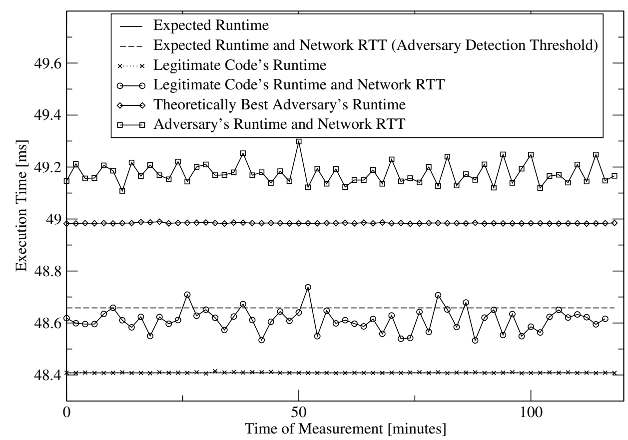


Figure 13: Result from Location 2.

5.4 Discussion

We now discuss virtual-memory-based attacks, issues concerning the practical deployment of Pioneer, and potential extensions to the current implementation of Pioneer to achieve better properties.

Implementing the verification function as SMM module. The System Management Mode (SMM) is a special operating mode present on all x86 CPUs. Code running in the SMM mode runs at the highest CPU privilege level. The execution environment provided by SMM has the following properties that are useful for implementing Pioneer: 1) all interrupts, including the Non-Maskable Interrupt (NMI) and the System Management Interrupt (SMI), and all exceptions are disabled by the processor, 2) paging and virtual memory are disabled in SMM, which precludes virtual-memory-based attacks, and 3) real-mode style segmentation is used, making it easier to defend against the segmentation-based memory copy attack.

Virtual-memory-based attacks. There are two ways in which the adversary might use virtual memory to attack the verification func-

[3]A false negative occurs when Pioneer claims that the untrusted platform is uncompromised when the untrusted platform is actually compromised.

tion: 1) the adversary could create memory protection exceptions by manipulating the page table entries and obtain control through the exception handler, or 2) the adversary could perform a memory copy attack by loading the instruction and data Translation Lookaside Buffer (TLB) entries that correspond to the same virtual address with different physical addresses. Since we use the stack to hold checksum pieces during checksum computation and later replace the exception handlers, the adversary cannot use memory protection exceptions to gain control.

The adversary can, however, use the CPU TLBs to perform a memory copy attack. Wurster et al. discuss how the second attack can be implemented on the UltraSparc processor [28]. Their attack can be adapted to the Intel x86 architecture in the context of Pioneer as follows: 1) the adversary loads the page table entry corresponding to the virtual address of the verification function with the address of the physical page where the adversary keeps an unmodified copy of the verification function, 2) the adversary does data accesses to virtual addresses of the verification function, thereby loading the its mapping into the CPU's D-TLB, and 3) the adversary replaces the page table entry corresponding to the virtual address of the verification function with the address of the physical page where the adversary keeps the modified checksum code is kept. When the CPU starts to execute the adversary's checksum code, it will load its I-TLB entry with the mapping the adversary set up in step 3. Thus, the CPU's I-TLB and D-TLB will have different physical addresses corresponding to the same virtual address and the adversary will be able to perform the memory copy attack.

The current implementation of Pioneer does not defend against this memory copy attack. However, a promising idea to defend against the attack is as follows. We create virtual address aliases to the physical pages contaning the verification function so that the number of aliases is greater than the number of entries in the CPU's TLB. Each iteration of the checksum code loads the PC and the data pointer with two of the virtual address aliases, selected in a pseudo-random manner. If the checksum loop performs a sufficient number of iterations so that with high probability all virtual address aliases are guaranteed to be used then the CPU will eventually evict the adversary's entry from the TLB.

The adversary can prevent its entry from being evicted from the TLB by not using all the virtual address aliases. However, in this case, the adversary will have to fake the value of the PC and the data pointer for the unused virtual address aliases. Since each iteration of the checksum code selects the virtual address aliases with which to load the PC and the data pointer in a pseudo-random manner, the adversary will have to check which aliases are used to load the PC and the data pointer in each iteration of the checksum code. This will increase the adversary's checksum computation time.

The TLB-based memory copy attack can also be prevented by implementing the verification function as an SMM module. Since the CPU uses physical addresses in SMM and all virtual memory support is disabled, the memory copy attack that uses the TLBs is not possible anymore.

Why use Pioneer instead of trusted network boot? In trusted network boot, the BIOS on a host fetches the boot image from a trusted server and executes the boot image. In order to provide the guarantee of verifiable code execution, trusted network boot has to assume that: 1) the host has indeed rebooted; 2) the correct boot image has indeed reached the host; and 3) the BIOS will correctly load and transfer control to the boot image. To guarantee that the BIOS cannot be modified by the adversary, the BIOS will have to stored on an immutable storage medium like Read-Only Memory (ROM). This makes it impossible to update the BIOS without physically replacing the ROM, should any vulnerability be discovered in the BIOS code.

Pioneer does not require any code to reside in immutable storage media, thereby making it easy to update. Also, Pioneer provides the property of verifiable code execution without having to reboot the untrusted platform, without having to transfer code over the network and without relying on any unverified software on the untrusted platform to transfer control to the executable.

MMX and SSE instructions. x86 processors provide support for Single Instruction Multiple Data (SIMD) instructions in the form of MMX and SSE technologies [13]. These instructions can simultaneously perform the same operation on multiple data items. This is faster than operating on the data items one at a time. However, the adversary cannot use the MMX or SSE instructions to speed up its checksum code, since we design the checksum code to be non-parallelizable.

Pioneer and TCG. A promising approach for reducing exposure to network RTT and for achieving a trusted channel to the untrusted platform is to leverage a Trusted Platform Module (TPM). The TPM could issue the challenge and time the execution of the checksum code and return the signed result and computation time to the dispatcher. However, this would require that the TPM be an active device, whereas the current generation of TPMs are passive.

Directly computing checksum over the executable. Why do we need a hash function? Why can the checksum code not simply compute the checksum over the executable? While this simpler approach may work in most cases, an adversary could exploit redundancy in the memory image of the executable to perform data-dependent optimizations. A simple example is a executable image that contains a large area initialized to zeros, which allows the adversary to suppress memory reads to that region and also to suppress updating the checksum with the memory value read (in case of `add` or `xor` operations).

`skinit` and `senter`. AMD's Pacifica technology has an instruction called `skinit`, which can verifiably transfer control to an executable after measuring it [4]. Intel's LaGrande Technology (LT) has a similar instruction, `senter` [12]. Both `senter` and `skinit` also set up an execution environment in which the executable that is invoked is guaranteed to execute untampered. These instructions are used to start-up a Virtual Machine Monitor (VMM) or a Secure Kernel (SK). Both instructions rely on the TCG load-time attestation property to guarantee that the SK or the VMM is uncompromised at start-up. However, due to the vulnerability of the SHA-1 hash function, the TCG load-time attestation property is compromised as we describe in Section 1. Hence, there is no guarantee that the SK or the VMM that is started is not malicious.

Implementing Pioneer on other architectures. We use the x86 architecture as our implementation platform example for the following reasons: 1) since x86 is the most widely deployed architecture today, our implementation of Pioneer on x86 can immediately be used on many legacy systems; and 2) due to requirements of backward compatibility, the x86 is a complex architecture, with a non-orthogonal ISA. Therefore, implementing Pioneer on the x86 architecture is more challenging than implementing it on RISC architectures with more orthogonal instruction sets, such as the MIPS, and the Alpha.

Verifying the timing overhead. Pioneer relies on the execution time of the checksum code. Therefore, the dispatcher has to know ahead of time what the correct checksum computation time should be for the untrusted platform. The checksum computation time depends on the CPU of the untrusted platform. There are two ways by which the dispatcher can find out the correct checksum computation time: 1) if the dispatcher has access to a trusted platform having the same CPU as the untrusted platform, or a CPU simulator for the untrusted platform, it can run experiments to get the correct execution time; or 2) we can publish the correct execution time for different CPUs on a trusted web-site.

6 APPLICATIONS

In this section, we first discuss the types of applications that can leverage Pioneer to achieve security, given the assumptions we make. Then, we describe the kernel rootkit detector, the sample application we have built using Pioneer.

6.1 Potential Security Applications

Pioneer can be applied to build security applications that run over networks controlled by a single administrative entity. On such networks, the network administrator could configure the network switches so that an untrusted host can only communicate with the dispatcher during the execution of Pioneer. This provides the property of message-origin-authentication while eliminating proxy attacks. Examples of networks that can be configured in this manner are corporate networks and cluster computing environments. On these networks the network administrator often needs to perform security-critical administrative tasks on untrusted hosts, such as installing security patches or detecting malware like viruses and rootkits. For such applications, the administrator has to obtain the guarantee that the tasks are executed correctly, even in the presence of malicious code on the untrusted host. This guarantee can be obtained through Pioneer.

As an example of how Pioneer could be used, we briefly discuss secure code updates. To verifiably install a code update, we can invoke the program that installs the code update using Pioneer. Pioneer can also be used to measure software on an untrusted host after a update to check if the code update has been successfully installed.

6.2 Kernel Rootkit Detection

In this section, we describe how we build a kernel rootkit detector using Pioneer. Our kernel rootkit detector allows a trusted verifier to detect kernel rootkits that may be installed on an external untrusted host without relying on signatures of specific rootkits or on low-level file system scans. Sailer et al. propose to use the load-time attestation guarantees provided by a TPM to detect rootkits when the kernel boots [20]. However, their technique cannot detect rootkits that do not make changes to the disk image of the kernel but only infect the in-memory image. Such rootkits do not survive reboots. Our rootkit detector is capable of detecting both kinds of rootkits. The only rootkit detection technique we are aware of that achieves similar properties to ours is Copilot [19]. However, unlike our rootkit detector, Copilot requires additional hardware in the form of an add-in PCI card to achieve its guarantees. Hence, it cannot be used on systems that do not have this PCI card installed. Also, our rootkit detector runs on the CPU of the untrusted host, making it immune to the dummy kernel attack that we describe in Section 7 in the context of Copilot.

Rootkits primer. Rootkits are software installed by an intruder on a host that allow the intruder to gain privileged access to that host, while remaining undetected [19, 29]. Rootkits can be classified into two categories: those that modify the OS kernel, and those that do not. Of the two, the second category of rootkits can be easily detected. These rootkits typically modify system binaries (e.g., ls, ps, and netstat) to hide the intruder's files, processes, network connections, etc. These rootkits can be detected by a kernel that checks the integrity of the system binaries against known good copies, e.g., by computing checksums. There are also tools like Tripwire that can be used to check the integrity of binaries [1]. These tools are invoked from read-only or write-protected media so that the tools do not get compromised.

As kernel rootkits subvert the kernel, we can no longer trust the kernel to detect such rootkits. Therefore, Copilot uses special trusted hardware (a PCI add-on card) to detect kernel rootkits. All rootkit detectors other than Copilot, including AskStrider [26], Carbonite [14] and St. Michael [9], rely on the integrity of one or more parts of the kernel. A sophisticated attacker can circumvent detection by compromising the integrity of the rootkit detector. Recently Wang et al. proposed a method to detect stealth software that try to hide files [27]. Their approach does not rely on the integrity of the kernel; however, it only applies when the stealth software makes modifications to the file system.

Implementation. We implement our rootkit detector on the x86_64 version of the Linux kernel that is part of the Fedora Core 3 Linux distribution. The x86_64 version of the Linux kernel reserves the range of virtual address space above `0xffff800000000000`. The kernel text segment starts at address `0xffffffff80100000`. The kernel text segment contains immutable binary code which remains static throughout its lifetime. Loadable Kernel Modules (LKM) occupy virtual addresses from `0xffffffff88000000` to `0xfffffffffff00000`.

We build our kernel rootkit detector using a Kernel Measurement Agent (KMA). The KMA hashes the kernel image and sends the hash values to the verifier. The verifier uses Pioneer to obtain the guarantee of verifiable code execution of the KMA. Hence, the verifier knows that the hash values it receives from the untrusted host were computed correctly.

The KMA runs on the CPU at the kernel privilege level, i.e., CPL0; hence, it has access to all the kernel resources (e.g., page tables, interrupt descriptor tables, jump tables, etc.), and the processor state, and can execute privileged instructions. The KMA obtains the virtual address ranges of the kernel over which to compute the hashes by reading the *System.map* file. The following symbols are of interest to the KMA: 1) `_text` and `_etext`, which indicate the start and the end of the kernel code segment; 2) `sys_call_table` which is the kernel system call table; and 3) `module_list` which is a pointer to the linked list of all loadable kernel modules (LKM) currently linked into the kernel. When the Kernel Measurement Agent (KMA) is invoked, it performs the following steps:

1. The KMA hashes the kernel code segment between *_text* and *_etext*.
2. The KMA reads kernel version information to check which LKMs have been loaded and hashes all the LKM code.
3. The KMA checks that the function pointers in the system call table only refer to the kernel code segment or to the LKM code. The KMA also verifies that the return address on the stack points back to the kernel/LKM code segment. The return address is the point in the kernel to which control returns after the KMA exits.
4. The KMA returns the following to the verifier: 1) the hash of the kernel code segment; 2) the kernel version information and a list indicating which kernel modules have been loaded; 3) the hash of all the LKM code; 4) a success/failure indicator stating whether the function pointer check has succeeded.
5. The KMA flushes processor caches, restores the register values, and finally returns to the kernel. The register values and the return address were saved on the stack when the kernel called invoked the Pioneer verification function.

We now explain how the verifier verifies the hash values returned by the untrusted platform. First, because the kernel text is immutable, it suffices for the verifier to compare the hash value of the kernel code segment to the known good hash value for the corresponding kernel version. However, the different hosts may have different LKMs installed, and so the hash value of the LKM code can vary. Therefore, the verifier needs to recompute the hash of the LKM text on the fly according to the list of installed modules reported by the KMA. The hash value reported by the KMA is then compared with the one computed by the verifier.

Experimental results. We implemented our rootkit detector on the Fedora Core 2 Linux distribution, using SHA-1 as the hash function. The rootkit detector ran every 5 seconds and successfully detected adore-ng-0.53, the only publically-known rootkit for the 2.6 version of the Linux kernel.

	Standalone (s)	Rootkit Detect. (s)	% Overhead
PostMark	52	52.99	1.9
Bunzip2	21.396	21.713	1.5
copy large file	373	385	3.2

Table 1: Overhead of the Pioneer-based rootkit detector

We monitor the performance overhead of running our rootkit detector in the background. We use three representative tasks for measurements: PostMark, bunzip2, and copying a large file. The first task, PostMark [5], is a file system benchmark that carries out transactions on small files. As a result, PostMark is a combination of I/O intensive and computationally intensive tasks. We used bunzip2 to to uncompress the Firefox source code, which is a computationally intensive task. Finally, we modeled an I/O intensive task by copying the entire `/usr/src/linux` directory, which totaled to 1.33 GB, from one harddrive to another. As the table above shows, all three tasks perform reasonably well in the presence of our rootkit detector.

Discussion. As with Copilot, one limitation of our approach is that we do not verify the integrity of data segments or CPU register values. Therefore, the following types of attacks are still possible: 1) attacks that do not modify code segments but rely merely on the injection of malicious data; 2) if the kernel code contains jump/branch instructions whose target address is not read in from the verified jump tables, the jump/branch instructions may jump to some unverified address that contains malicious code. For instance, if the jump address is read from an unverified data segment, we cannot guarantee that the jump will only reach addresses that have been verified. Also, if jump/branch target addresses are stored temporarily in the general purpose registers, it is possible to jump to an unverified code segment, after the KMA returns to the kernel since the KMA restores the CPU register values. In conclusion, Pioneer limits a kernel rootkit to be placed solely in mutable data segments; it requires any pointer to the rootkit to reside in a mutable data segment as well. These properties are similar to what Copilot achieves.

Our rootkit detection scheme does not provide backward security. A malicious kernel can uninstall itself when it receives a Pioneer challenge, and our Pioneer-based rootkit detector cannot detect bad past events. Backward security can be achieved if we combine our approach with schemes that backtrack intrusions through analyzing system event logs [17].

7 Related Work

In this section, we survey related work that addresses the verifiable code execution problem. We also describe the different methods of code attestation proposed in the literature and discuss how the software-based code attestation provided by Pioneer is different from other code attestation techniques.

7.1 Verifiable Code Execution

Two techniques, Cerium [8] and BIND [23], have been proposed. These use hardware extensions to the execution platform to provide a remote host with the guarantee of verifiable code execution. Cerium relies on a physically tamper-resistant CPU with an embedded public-private key pair and a μ-kernel that runs from the CPU cache. BIND requires that the execution platform has a TPM chip and CPU architectural enhancements similar to those found in Intel's LaGrande Technology (LT) [12] or AMD's Secure Execution Mode (SEM) [3] and Pacifica technology [4]. Unlike Pioneer, neither Cerium nor BIND can be used on legacy computing platforms. As far as we are aware, Pioneer is the only technique that attempts to provide the verifiable code execution property solely through software techniques.

7.2 Code Attestation

Code attestation can be broadly classified into hardware-based and software-based approaches. While the proposed hardware-based attestation techniques work on general purpose computing systems, to the best of our knowledge, there exists no software-based attestation technique for general purpose computing platforms.

Hardware-based code attestation. Sailer et al. describe a load-time attestation technique that relies on the TPM chip standardized by the Trusted Computing Group [20]. Their technique allows a remote verifier to verify what software was loaded into the memory of a platform. However, a malicious peripheral could overwrite code that was just loaded into memory with a DMA-write, thereby breaking the load-time attestation guarantee. Also, as we discussed in Section 1, the load-time attestation property provided by the TCG standard is no longer secure since the collision resistance property of SHA-1 has been compromised. Terra uses a Trusted Virtual Machine Monitor (TVMM) to partition a tamper-resistant hardware platform in multiple virtual machines (VM) that are isolated from each other [11]. CPU-based virtualization and protection are used to isolate the TVMM from the VMs and the VMs from each other. Although the authors only discuss load-time attestation using a TPM, Terra is capable of performing run-time attestation on the software stack of any of the VMs by asking the TVMM to take integrity measurements at any time. All the properties provided by Terra are based on the assumption that the TVMM is uncompromised when it is started and that it cannot be compromised subsequently. Terra uses the load-time attestation property provided by TCG to guarantee that the TVMM is uncompromised at start-up. Since this property of TCG is compromised, none of the properties of Terra hold. Even if TCG were capable of providing the load-time attestation property, the TVMM could be compromised at run-time if there are vulnerabilities in its code. In Copilot, Petroni et al. use an add-in card connected to the PCI bus to perform periodic integrity measurements of the in-memory Linux kernel image [19]. These measurements are sent to the trusted verifier through a dedicated side channel. The verifier uses the measurements to detect unauthorized modifications to the kernel memory image. The Copilot PCI card cannot access CPU-based state such as the pointer to the page table and pointers to interrupt and exception handlers. Without access to such CPU state, it is impossible for the PCI card to determine exactly what resides in the memory region that the card measures. The adversary can exploit this lack of knowledge to hide malicious code from the PCI card. For instance, the PCI card assumes that the Linux kernel code begins at virtual address 0xc0000000, since it does not have access to the CPU register that holds the pointer to the page tables. While this assumption is generally true on 32-bit systems based on the Intel x86 processor, the adversary can place a correct kernel image starting at address 0xc0000000 while in fact running a malicious kernel from another memory location. The authors of Copilot were aware of this attack [6]. It is not possible to prevent this attack without access to the CPU state. The kernel rootkit detector we build using Pioneer is able to provide properties equivalent to Copilot without the need for additional hardware. Further, because our rootkit detector has access to the CPU state, it can determine exactly which memory locations contain the kernel code and static data. This ensures that our rootkit detector measures the running kernel and not a correct copy masquerading as a running kernel. Also, if the host running Copilot has an IOMMU, the adversary can re-map the addresses to perform a data substitution attack. When the PCI card tries to read a location in the kernel, the IOMMU automatically redirects the read to a location where the adversary has stored the correct copy.

Software-based attestation. Genuinity is a technique proposed by Kennell and Jamieson that explores the problem of detecting the difference between a simulator-based computer system and an actual computer system [16]. Genuinity relies on the premise that simulator-based program execution is bound to be slower because a simulator has to simulate the CPU architectural state in software, in addition to simulating the program execution. A special checksum function computes a checksum over memory, while incorporating different elements of the architectural state into the checksum. By the above premise, the checksum function should run slower in a simulator than on an actual CPU. While this statement is probably true when the simulator runs on an architecturally different CPU than the one it is simulating, an adversary having an architecturally similar CPU can compute the Genuinity checksum within the alloted time while maintaining all the necessary architectural state in software. As an example, in their implementation on the x86, Kennell and Jamieson propose to use special registers, called Model Specific Registers (MSR), that hold various pieces of the architectural state like the cache and TLB miss count. The MSRs can only be read and written using the special `rdmsr` and `wrmsr` instructions. We found that these instructions have a long latency ($\approx$ 300 cycles). An adversary that has an x86 CPU could simulate the MSRs in software and still compute the Genuinity checksum within the alloted time, even if the CPU has a lower clock speed than what the adversary claims. Also, Shankar et al. show weaknesses in the Genuinity approach [22]. SWATT is a technique proposed by Seshadri et al. that performs attestation on embedded devices with simple CPU architectures using a software verification function [21]. Similar to Pioneer, the verification function is constructed so that any attempt to tamper with it will increase its running time. However, SWATT cannot be used in systems with complex CPUs. Also, since SWATT checks the entire memory, its running time becomes prohibitive on systems with large memories.

8 Conclusions and Future Work

We present Pioneer, which is a first step towards addressing the problem of verifiable code execution on untrusted legacy computing platforms. The current version of Pioneer leaves open research problems. We need to: 1) deriving a formal proof of the optimality of the checksum code implementation; 2) proving that an adversary cannot use mathematical methods to generate a shorter checksum function that generates the same checksum output when fed with the same input; 3) deriving a checksum function that is largely CPU architecture independent, so that it can be easily ported to different CPU architectures; and 4) increasing the time overhead for different attacks, so that it is harder for an adversary to forge the correct checksum within the expected time. There are also low-level attacks that need to be addressed: 1) the adversary could overclock the processor, making it run faster; 2) malicious peripherals, a malicious CPU in a multi-processor system or a DMA-based write could overwrite the executable code image in memory after it is checked but before it is invoked; and 3) dynamic processor clocking techniques could lead to false positives. We plan to address all these issues in our future work.

This paper shows an implementation of Pioneer on an Intel Pentium IV Xeon processor based on the Netburst Microarchitecture. The architectural complexity of Netburst Microarchitecture and the complexity of the x86_64 instruction set architecture make it challenging to design a checksum code that executes slower when the adversary tampers with it in any manner. We design a checksum code that exhausts the issue bandwidth of the Netburst microarchitecture, so that any additional instructions the adversary inserts will require extra cycles to execute.

Pioneer can be used as a new basic building block to build security applications. We have demonstrated one such application, the kernel rootkit detector, and we propose other potential applications. We hope these examples motivate other researchers to embrace Pioneer, extend it, and apply it towards building secure systems.

9 Acknowledgments

We gratefully acknowledge support and feedback of, and fruitful discussions with William Arbaugh, Mike Burrows, George Cox, David Durham, David Grawrock, Jon Howell, John Richardson, Dave Riss, Carlos Rozas, Stefan Savage, Dawn Song, Jesse Walker, Yi-Min Wang, and our shepherd Emin Gün Sirer. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

References

[1] Tripwire. `http://sourceforge.net/projects/tripwire/`.

[2] VMware. `http://www.vmware.com/`.

[3] AMD platform for trustworthy computing. In *WinHEC*, September 2003.

[4] Secure virtual machine architecture reference manual. AMD Corp., May 2005.

[5] Network Appliance. Postmark: A new file system benchmark. Available at http://www.netapp.com/techlibrary/3022.html, 2004.

[6] W. Arbaugh. Personal communication, May 2005.

[7] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. Miller, P. Roussel, R. Singhal, B. Toll, and K.S. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(01), February 2004.

[8] B. Chen and R. Morris. Certifying program execution with secure procesors. In *Proceedings of HotOS IX*, 2003.

[9] A. Chuvakin. Ups and downs of unix/linux host-based security solutions. *;login: The Magazine of USENIX and SAGE*, 28(2), April 2003.

[10] Free Software Foundation. superopt - finds the shortest instruction sequence for a given function. `http://www.gnu.org/directory/devel/compilers/superopt.html`.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *In Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[12] Intel Corp. *LaGrande Technology Architectural Overview*, September 2003.

[13] Intel Corporation. *IA32 Intel Architecture Software Developer's Manual Vol.1*.

[14] K. J. Jones. Loadable Kernel Modules. *;login: The Magazine of USENIX and SAGE*, 26(7), November 2001.

[15] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 304–314, 2002.

[16] R. Kennell and L. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of USENIX Security Symposium*, August 2003.

[17] S. King and P. Chen. Backtracking intrusions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 223–236, 2003.

[18] A. Klimov and A. Shamir. A new class of invertible mappings. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 470–483, 2003.

[19] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of USENIX Security Symposium*, pages 179–194, 2004.

[20] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of USENIX Security Symposium*, pages 223–238, 2004.

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

[22] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of USENIX Security Symposium*, pages 89–101, August 2004.

[23] E. Shi, A. Perrig, and L. van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 154–168, 2005.

[24] Trusted Computing Group (TCG). `https://www.trustedcomputinggroup.org/`, 2003.

[25] Xiaoyun Wang, Yiqun Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *Proceedings of Crypto*, August 2005.

[26] Y. Wang, R. Roussev, C. Verbowski, A. Johnson, and D. Ladd. AskStrider: What has changed on my machine lately? Technical Report MSR-TR-2004-03, Microsoft Research, 2004.

[27] Y. Wang, B. Vo, R. Roussev, C. Verbowski, and A. Johnson. Strider GhostBuster: Why it's a bad idea for stealth software to hide files. Technical Report MSR-TR-2004-71, Microsoft Research, 2004.

[28] G. Wurster, P. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2005.

[29] D. Zovi. Kernel rootkits. `http://www.cs.unm.edu/~ghandi/lkr.pdf`.