Verifying the System State for the Absence of Malware on Commodity Platforms

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Yanlin Li

B.S., Electrical and Information Engineering, Tianjin University, China
M.S., Electrical and Information Engineering, Tianjin University, China
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

August, 2015

Thesis Committee:

Prof. Adrian Perrig, Chair (Carnegie Mellon University)

Prof. Virgil Gligor (Carnegie Mellon University)

Prof. Greg Ganger (Carnegie Mellon University)

Dr. Jesse Walker (Intel)

Dr. Jonathan McCune (Google)

This research was supported in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389, MURI W 911 NF 0710287, W911NF-09-1-0273 from the Army Research Office, grant NGIT2009100109 from the Northrop Grumman Information Technology Inc. Cyber Security Consortium, grant N66001-13-2-4040 from the Defense Advanced Research Projects Agency (DARPA), and grant from the General Motors (GM) Corporation and by a gift from Lockheed Martin Corporation.

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, DARPA, GM, Northrop Grumman, CMU, LMC, or the U.S. Government or any of its agencies.

Dedicated to my beloved parents and wife

Abstract

Several techniques exist to verify the integrity of the software image to guarantee the absence of malware on commodity computers or embedded platforms based on a hardware- or software-based root of trust. However, as modern embedded platforms have become increasingly complex, existing software-based attestation techniques for embedded platforms no longer cover the new hardware features. In addition, malware can infect peripherals' firmware in a commodity computer. Such malware, once inside a peripheral, may also compromise other peripherals' firmware or the host operating system. Unfortunately, none of the existing techniques provides a mechanism for verifying the integrity of peripherals' firmware to guarantee the absence of malware.

In the first two parts of this thesis, we investigate the feasibility of addressing the following two challenges: (1) establishing a software-only root of trust on an embedded platform to verify the system state of the embedded platform, and (2) verifying the integrity of peripherals' firmware on commodity computers. For the first challenge, we identify three new classes of attacks against existing software-based attestation mechanisms and propose countermeasures to detect these attacks. For the second challenge, we propose a software-based scheme enabling a piece of trusted code running on the main CPU, bootstrapped through a hardware- or software-based root of trust, to verify the integrity of peripherals' firmware.

The software stack on commodity computers contains an increasingly large number of vulnerabilities. Verifying the integrity of the entire software image on commodity computers in a hostile world is impractical for protecting security-sensitive operations. To protect security-sensitive operations, e.g., paying bills,

shopping online, accessing medical records, establishing an isolated execution environment on commodity computers for security-sensitive operations with integrity measurement is a desirable functionality. The software-based mechanism for peripheral firmware integrity verification can be integrated with the isolated execution environment to guarantee the absence of malware in peripherals, providing an isolated malware-free operation environment with trusted peripherals for security-sensitive operations.

However, one-way protected malware-free operation environment is insufficient in some practical scenarios, e.g., Cloudlets, in which two-way protection is required. In the third part of this thesis, we propose MiniBox, the first two-way sandbox for x86 native code that not only protects a benign OS from a misbehaving program, but also protects a running program from a malicious OS. To achieve two-way protection, MiniBox verifies the system state including the integrity of peripherals' firmware to prevent malware from spreading to either side.

Acknowledgements

Foremost, I would like to express my deepest appreciation and thanks to my academic advisor, Professor Adrian Perrig, for his invaluable guidance, inspiration, and encouragement. Adrian has been a tremendous mentor for me. His advice and patience trained me to grow as a research scientist and his enthusiasm in research always motivates me to improve and aim high. I also would like to express my sincere gratitude to Dr. Virgil Gligor and Dr. Jonathan McCune, who also advised me during my Ph.D. study, for their invaluable guidance and encouragement. My sincere thanks also go to my other committee members Dr. Greg Ganger and Dr. Jesse Walker for their abundant encouragement and insightful comments on this thesis.

I am also very grateful to many collaborators: Brandon Baker, Dr. Rekha Bachwani, Dr. David Brumely, Dr. Bradley Chen, Dr. Haibo Chen, Dr. Yueqiang Cheng, Dr. Anupam Datta, Will Drewry, Dr. Jie Liu, Dr. Petros Maniatis, Dr. James Newsome, Dr. Bodhi Priyantha, Dr. Bryan Parno, Dr. Dina Papagiannaki, Dr. Ning Qu, Dr. Anmol Sheth, Dr. Amit Vasudevan, Dr. Maverick Woo, Ted Wobber, Dr. Yinglian Xie, Dr. Fang Yu, Dr. Xin Zhang, and Dr. Zongwei Zhou. I cannot be thankful enough to them for their invaluable feedback and fruitful collaborations.

I also thank colleagues at Carnegie Mellon University, who made my Ph.D. journey so colorful: Dr. Lujo Bauer, Youzhi Bao, Dr. Sang Kil Cha, Chen Chen, Michael Farb, Geoff Hasker, Dr. Hsu-Chun Hsiao, Han Jun, Dr. Limin Jia, Dr. Tiffany Hyun-jin Kim, Dr. Jialiu Lin, Dr. Yue-Hsun Lin, Steve Matsumoto, Emmanuel Owusu, Dr. Edward Schwartz, Michael Stroucken, Dr. Vyas Sekar, Miao Yu, and Dr. Jun Zhao.

Finally, I must express my sincere thanks to my family members, whose support and encouragement in both my life and research inspired me whenever I was in doubt. In particular, I must thank my wife, Lingjuan, without whose love, support, and encouragement over the past years in my Ph.D. journey, the dissertation would not have been completed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As modern commodity computers evolve, their software stack becomes progressively larger. Today, the Trusted Computing Base (TCB) of commodity computers includes the operating system, the device drivers, applications and peripherals' firmware. Unfortunately, such software is mainly designed for features, not security. As a result, the software stack on commodity computers contains an increasingly larger number of vulnerabilities. Consequently, once a commodity computer is connected to a hostile environment (e.g., to the Internet or with an untrusted USB drive), an adversary can exploit the vulnerabilities on a commodity computer. In addition, today's supply chains for commodity computers and embedded devices (e.g., desktops, wireless routers, smartphones) are globalized and diversified, making it easier for adversaries to insert malware into commodity platforms during the manufacturing or shipping process. Consequently, a commodity computer or embedded platform might be infected with malware even before it is delivered to the user. For instance, in 2015, Kaspersky Lab (a Russian security software maker) reported [59] that it had discovered spyware on hard drives' firmware on personal computers from about 30 countries. The infected hard drives came from major

manufacturers including Seagate, Western Digital (WD), IBM, and Toshiba.

To guarantee the absence of malware on commodity computers or embedded platforms, a number of techniques [32, 77, 88, 89, 95, 121] have been proposed to verify the integrity of the software image on a commodity computer or embedded platform based on a hardware- or software-based root of trust. However, a number of limitations exist in existing techniques.

1.1 Existing Techniques and Limitations

1.1.1 Software-Based Attestation

Software-based attestation techniques for embedded platforms (Section 2.1 in Chapter 2) establish a software-based root of trust on an embedded platform, and then measure the integrity of the software image on the platform to guarantee the absence of malware [32, 88, 89, 95, 121]. However, modern embedded platforms have become increasingly complex with the inclusion of more hardware features, such as multiple heterogeneous processors, multiple-level caches, and complex system configurations. The software-based attestation techniques proposed years ago for much simpler embedded platforms do not cover the new hardware features available on modern embedded platforms. Consequently, an adversary might leverage the new hardware features to break existing software-based attestation mechanisms.

In the first part of this thesis (Chapter 3), we investigate the feasibility of establishing a software-only root of trust on a modern embedded platform to guarantee the absence of malware on the platform. In particular, we examine new hardware features of embedded platforms that are not covered by existing software-based attestation techniques, present new classes of attacks against existing software-based

attestation techniques, and propose countermeasures to detect these attacks.

1.1.2 Trusted Boot

On commodity computers, trusted boot technique [77] accumulates a list of the integrity measurements of the software executed on a commodity computer based on a hardware-based root of trust, thereby enabling a verifier to verify the integrity of the software. However, an often-overlooked software attack target is the firmware that executes in peripheral devices. Attackers can exploit vulnerabilities in peripherals' firmware or their firmware update tools [22, 29] to insert malware into peripherals' firmware. Such malware, once inside a peripheral, may also compromise other peripherals' firmware or the host operating system.

Unfortunately, none of the previous techniques provide a mechanism to verify the integrity of peripherals' firmware to guarantee the absence of malware in peripherals. Verifying the integrity of peripherals' firmware remains a significant challenge because (1) the limited memory and computational resources on peripherals make it difficult to deploy complex security primitives on the peripherals themselves and (2) hardware-based protection is impractical because it would add cost and complexity to devices already under severe economic constraints.

In the second part of this thesis (Chapter 4), we investigate the challenges in verifying the integrity of peripherals' firmware on commodity computers and present a novel approach to enable a securely protected program (e.g., bootstrapped based on a hardware-based root of trust) running on the main CPU to efficiently verify the integrity of peripherals' firmware to guarantee the absence of malware on peripherals.

1.1.3 One-Way Protection and Cloudlets

One-Way Protection To securely perform security-sensitive operations on a commodity computer in a hostile environment, a current trend is to establish an isolated operation environment for security-sensitive operations with integrity measurement to guarantee the absence of malware in the isolated operation environment [4, 31, 37, 66, 111, 126, 127]. Existing techniques for establishing an isolated operation environment focus on one-way protection that protects the security-sensitive portion of a program from a malicious OS (usually with two-way memory isolation) while the non-sensitive portion of the program is not isolated from the OS. Such one-way protection is insufficient for some practical scenarios, such as Cloudlets [86,87], in which two-way protection is highly desirable. In the two-way protection, a security-sensitive program is protected from a malicious OS while the OS is protected from a misbehaving program. However, in existing techniques, the non-isolated portion of the program may contain malware that can compromise the system.

Cloudlets As mobile computing and cloud computing converge, Satyanarayanan et al. envision *Cloudlets* [86, 87], a middle tier of a mobile device-Cloudlet-cloud hierarchy, enabling resource-poor mobile devices to offload compute-intensive or data-intensive programs to nearby public computers (e.g., a public computer in a cafe or airport) with a small communication latency. Meanwhile, users can also benefit from the rich I/O peripherals available on Cloudlet public computers for complex operations that are inconvenient on mobile devices with limited I/O peripherals (e.g., small display with poor resolution).

Two-Way Protection Obviously, a two-way protection mechanism is highly desirable in Cloudlets. In the two-way protection, the offloaded program and corre-

sponding security-sensitive data are protected from malicious code on the Cloudlet public computer (including malicious code in the OS and other programs offloaded by other users) while the OS on the Cloudlet computer is protected from malicious offloaded programs. Unfortunately, it is still an open challenge to provide such a two-way protection mechanism on commodity computers.

In the third part of the thesis (Chapter 5), we investigate the design options for providing such two-way protection by combining an isolation module (e.g., the TrustVisor [66, 111] or Intel Software Guard Extensions (Intel SGX) [4, 31, 37]) with a one-way sandbox (i.e., Google Native Client Sandbox [122, 123]). We then present MiniBox, the first two-way sandbox for x86 native code. To achieve two-way protection with trusted peripherals, MiniBox can verifies the integrity of peripherals' firmware to prevent malware from spreading to either side.

1.2 Thesis Statement

In this dissertation, we examine the feasibility of verifying the system state to guarantee the absence of malware on commodity platforms and make the following *thesis statement*:

It is possible to guarantee the absence of malware in the entire system or in an isolated operation environment (with or without two-way protection) by verifying the system state (including the peripherals' state) of the entire system or the isolated operation environment based on a hardware- or software-based root of trust on commodity platforms.

1.3 Outline and Overview

Now, we present the outline of this dissertation and an overview of each chapter.

1. In Chapter 2, we describe the background knowledge including existing software-based attestation techniques and known attacks against the software-based attestation, the hardware architecture of a modern motherboard, the features of malware on peripherals on commodity computers, existing hardware-based trusted computing techniques, and the Google Native Client sandbox [122, 123].

2. In Chapter 3, we present three new classes of attacks against existing software-based attestation schemes and propose corresponding countermeasures to establish a software-only root of trust on embedded platforms. In particular, we leverage a dynamically modified page table to force attackers to perform complex operations during an attack, thereby inducing detectable overhead. Based on the software-only root of trust, we can verify the integrity of the system image and reset all platform configurations to guarantee the absence of malware on embedded platforms.

3. In Chapter 4, we present VIPER, a novel software-based approach to enable a trusted verifier program running on the main CPU to efficiently verify the integrity of peripherals' firmware to guarantee of the absence of malware on peripherals. In this chapter, we also show how to integrate the mechanism for peripheral firmware integrity verification with existing trusted computing techniques to establish an isolated malware-free operation environment with trusted peripherals on commodity computers.

4. In Chapter 5, we present MiniBox, a two-way sandbox for x86 native code, which can provide efficient two-way protection for x86 native code with a small TCB. The two-way sandbox provides a mutually isolated execution environment for x86 native code with an efficient service runtime. To protect

the host OS, the two-way sandbox also constrains the system call interface exposed to the native code from the OS. Integrated with VIPER, the two-way sandbox can offer a two-way protected malware-free operation environment, thereby enabling users to perform security-sensitive operations on a public computer (e.g., a Cloudlet computer).

5. In Chapter 6, we review the related work in software-based attestation, peripheral malware detection, schemes to protect security-sensitive applications, and sandbox for the host OS protection.

6. In Chapter 7, we conclude this dissertation.

1.4 Summary of Contributions

In the journey to examine the feasibility of verifying the system state to guarantee the absence of malware, this dissertation makes the following high-level contributions.

1. An investigation into the new challenges and corresponding countermeasures to perform software-based attestation to guarantee the absence of malware on embedded platforms.

2. A software-based approach to verify the integrity of peripherals' firmware on commodity computers.

3. A system architecture for providing a two-way protected operation environment with a small TCB on commodity computers.

Chapter 2

Background

In this chapter, we present background knowledge. Existing software-based attestation techniques and the known attacks against existing techniques are presented in Section 2.1. The hardware architecture of a modern motherboard and the features of malware on peripherals are described in Section 2.2. The existing trusted computing techniques to achieve secure code execution and on-demand isolated I/O are described in Section 2.3. In Section 2.4, we present Google Native Client sandbox [122, 123], based on which we design a two-way sandbox.

2.1 Software-Based Attestation Techniques

2.1.1 SWATT: SoftWare-based ATTestation

SoftWare-based ATTestation for embedded devices (SWATT) is based on a challenge-response protocol between a trusted verifier and an untrusted embedded device, and a predicted computation time constraint. First, the verifier sends a random nonce to the embedded device. Using this nonce as a seed, a verification function in the embedded device computes a checksum over the entire memory contents and re-

turns the checksum result to the verifier. The verifier has a copy of the expected memory contents of the embedded device, so it can verify the checksum result. Also, the verifier knows the exact hardware configuration of this untrusted embedded device, enabling the verifier to exactly predict the checksum computation time. Because the checksum function is well optimized, the presence of any malicious code in memory will either invalidate the checksum result or introduce a detectable time delay. Therefore, only the checksum result received within the expected time range is valid. During checksum computation, the checksum function reads memory in a pseudo-random traversal, thus preventing an attacker from precomputing the checksum result. SWATT requires that the embedded device can only communicate with the verifier during attestation. This prevents a malicious device from communicating with a faster machine to compute the checksum.

2.1.2 ICE: Indisputable Code Execution

Indisputable Code Execution (ICE) sets up a dynamic software-based root of trust on an untrusted device through a challenge-response protocol between a trusted verifier and the untrusted embedded device, and a predicted computation time constraint. The dynamic software-based root of trust also sets up an untampered execution environment, which in turn is used to demonstrate verifiable code execution to the verifier. As in SWATT, the verifier first sends a random nonce to the untrusted device. Upon receiving the random nonce, the verification function in the untrusted device sets up an untampered execution environment. The verification function includes code to set up an ICE environment by disabling interrupts, a checksum function that computes a checksum over the contents of the verification function, a communication function (send function) that returns computation results to the verifier, and a hash function that computes a hash of the executable that

will be invoked in the untampered environment. After checksum computation, the send function sends the checksum result to the verifier. As in SWATT, the verifier can verify the checksum result and predict the checksum computation duration. If the verifier receives the correct checksum within the expected time, the verifier obtains assurance that the untampered execution environment (dynamic root of trust) has been set up in the untrusted device. The send function invokes the hash function to compute a hash of the executable in the embedded device and sends the hash result to the verifier. Then the verification function invokes the executable on the untrusted device. Simultaneously, the verifier obtains the guarantee of the integrity of the executable through verifying the hash of the executable.

2.1.3 Attacks Against Existing Techniques

In this section, we describe known attacks against the existing software-based attestation schemes.

Memory Copy and Memory Substitution Attacks There are two types of memory copy attacks. In a first type of memory copy attack, attackers run a modified checksum function in the correct memory location, and save a correct copy of the original checksum function in spare memory space. During the checksum computation, the malicious checksum function computes the checksum over the correct copy. In a second type of memory copy attack, attackers load the original checksum function to the correct memory location, but run a malicious checksum function in another memory location that computes the checksum over the original copy. In a memory substitution attack, attackers run a modified checksum function in the correct location and save a correct copy of the original checksum function in spare memory. During the checksum computation, the modified checksum function checks every memory address to read and redirects the memory read

to the correct copy when the modified memory contents are read. In the checksum computation, the Program Counter (PC) value or the Data Pointer (DP) value (the memory address to read) are incorporated into the checksum. In a memory copy or memory substitution attack, the malicious checksum function has to forge the correct PC or DP value to compute the expected checksum, thereby causing a computation overhead.

Proxy Attack In a proxy attack, the untrusted device under attestation (prover device) asks a remote faster computer (a proxy that can compute the expected checksum results faster than the prover device) to compute the expected checksum. The proxy attack can be detected if the user monitors all the communication channels of the prover device. For example, the user can use a Radio Frequency analyzer (e.g., RF-Analyzer HF35C) to detect any wireless communications of the prover device, thus detecting wireless proxy attacks.

Split-TLB Attack In a split-TLB attack, attackers carefully configure the Instruction Translation Lookaside Buffer (I-TLB) and the Data Translation Lookaside Buffer (D-TLB) such that the entries for the checksum function memory pages point to different physical addresses in the I-TLB and the D-TLB separately. In this way, attackers can run a malicious checksum function, but compute the checksum over the correct copy of the checksum function without additional operations. However, attackers must guarantee that the carefully configured entries in the D-TLB and the I-TLB will be preserved during the checksum computation.

Return Oriented Programming Attack Return oriented programming (ROP) [16, 40, 97] performs computation on a system by executing several pieces of code that are terminated by a return instruction. These pieces of code are executed through

well-controlled stack content. If there is sufficient existing binary code in the system, an adversary can execute arbitrary computations through a ROP attack without injecting any code, except for overwriting the stack with well-designed content. Castelluccia et al. [17] present that an adversary can use a ROP attack to protect malicious code from being detected by software-based attestation schemes. Briefly, the adversary code first saves a copy of the adversary code on data memory before attestation. Then the adversary code modifies the contents of data memory by embedding ROPs on the stack. Through these ROPs, the attacker erases all the malicious code in program memory and restores the original code before checksum computation. Then, during checksum computation, the contents of program memory are exactly as expected. After attestation, the attacker restores malicious code in program memory through an additional ROP. The ROP attack generates little computation overhead. For example, in the attack described by Castelluccia et al. [17], the computation overhead caused by a ROP attack is undetectable, only 0.3% of the expected checksum computation time. To prevent this attack, the checksum function can incorporate the stack contents into the checksum to measure the integrity of the stack contents.

Memory Compression Attack One important enabler of a memory copy or memory substitution attack is that the malicious code can remember or predict the constant values of empty memory during attestation. Therefore, Seshadri et al. [96] propose to fill the empty space of program memory with pseudo-random values and leave no available free space for attackers to make a memory copy or memory substitution attack. However, an attacker can still create free space through compressing the existing code on program memory. Some compression algorithms, such as the Canonical Huffman Encoding [39], can decompress the compressed stream from an arbitrary position. Thus, the malicious code can de-

compress the compressed steam on-the-fly during attestation and obtain the correct checksum result though the checksum code reads memory in a pseudo random traversal. However, the decompression procedure causes a detectable computation overhead because of the complexity of the decompression algorithm.

Two-part checksum computation attack Another attack against software-based attestation mechanisms is two-part checksum computation attack [90]. In software-based attestation schemes, a checksum function executes the checksum blocks for a large number of iterations (assuming the checksum function executes the checksum blocks for N iterations). In a two-part checksum computation attack, an adversary runs a malicious checksum function (with additional operations to protect malicious contents from being detected) for only K iterations, and then runs the original checksum function for the left $N-K$ iterations. Because the checksum function reads the memory contents in a random pattern (for integrity measurement), it is possible that the malicious contents would not be measured during the left $N-K$ iterations. The overhead caused by the malicious operations in the K iterations might be not be detectable.

To prevent this attack, we can increase the number of checksum iterations performed on the prover device. Based on the result of the Coupon Collector's Problem [42], we denote the minimal number of checksum iterations to measure every memory location in high probability as M. The verifier machine can request the checksum function to execute the checksum blocks for $c \times M$ iterations (assuming the checksum function measures the memory contents one time in each checksum iteration). c is a number greater than 1. To prevent two-part computation attacks, we set a detection threshold that could detect malicious operations in $(c-1) \times M$ iterations.

2.2 Modern System Architecture and Buses

2.2.1 Modern System Architecture

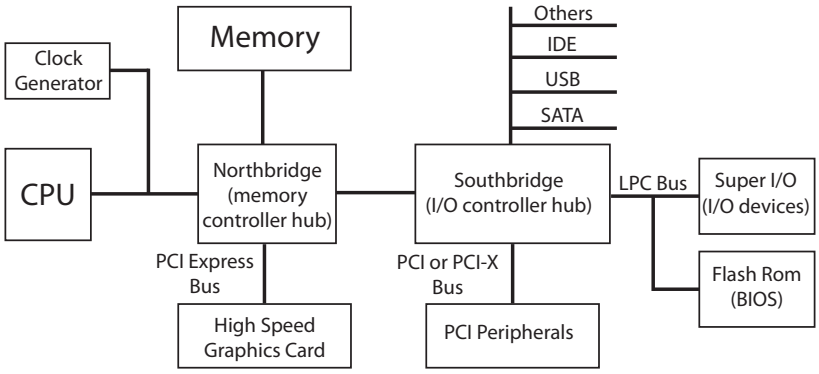

Figure 2.1: Hardware Architecture of a Modern Motherboard.

Figure 2.1 shows a diagram of a modern motherboard. Two logical chipsets (north- and southbridge) connect the host CPU(s) with memory, PCI-family buses, and numerous other buses and peripherals. The northbridge (memory controller hub) typically deals with communication among the CPU, main memory, any PCI Express (PCIe) peripherals, and the southbridge (I/O controller hub). The southbridge primarily handles communication among the northbridge, IDE, SATA, USB, LPC, PCI or PCI-X buses / peripherals, and so on.

Memory-mapped I/O (MMIO) [47] maps part of the memory inside peripherals (MMIO memory) to the main memory address space of the host CPU, and enables the host CPU to access the MMIO memory on peripherals through ordinary memory read or write instructions. A separate I/O address space also exists and can be used to interface with some peripherals, in which case the host CPU accesses the peripherals through special I/O instructions (e.g., `outb`).

Direct Memory Access (DMA) enables peripherals to transfer data between

main memory and the device's local memory without involving the host CPU. The memory addresses in the main memory that a peripheral can access through DMA can be controlled in newer systems with hardware support for virtualization by using an input/output memory management unit (IOMMU) [6, 7, 48].

2.2.2 PCI, PCI-X and PCI Express

The original PCI bus is a parallel bus shared by all PCI peripherals. Any peripheral on the PCI bus can initiate a transaction by requesting permission to become the *bus master* from the PCI bus arbiter. If its request is granted, the initiator starts the transaction by sending the target address, followed by one or more data phases. All the PCI peripherals receive the transmissions, but only the PCI peripheral with the target address processes the transaction. PCI-X is a faster version of PCI, which was designed for higher-speed peripherals, such as gigabit Ethernet. A PCI-X peripheral shares the same bus with other PCI peripherals. Unlike the PCI or PCI-X buses, PCI Express (PCIe) is a serial communication bus. PCIe peripherals communicates with each other via a logically named interconnect, which creates point-to-point links between any two PCIe peripherals on the motherboard. On a modern motherboard, the clock speed of the northbridge and southbridge can exceed 1 GHz. The capacity of various versions of the vanilla PCI bus are from 1066 Mbps (32-bit at 33.3 MHz) to 4266 Mbps (64-bit at 66.6 MHz). The capacity of a PCI-X bus is 4266 Mbps (64-bit at 66.6 MHz) or 8512 Mbps (64-bit at 133 MHz). PCIe supports 2 Gbps (v1), 4 Gbps (v2), and 8 Gbps (v3) on each lane, with up to 32 lanes in each PCIe slot.

2.2.3 Peer-to-Peer Peripheral Communication

Based on the PCI and PCIe specifications [68, 69], two PCI / PCIe peripherals can engage in peer-to-peer communication. However, under typical workloads on a commodity PC, one endpoint is almost always the host CPU or main memory. Nevertheless, on a modern motherboard, DMA potentially enables a peripheral device to read or write other peripherals' MMIO memory. For instance, the GPU often has a large amount of memory (1 GB or more) mapped into the main memory address space using MMIO. A NIC can write or read the GPU's MMIO memory using DMA [85, 110]. In today's systems, the IOMMU is in the northbridge, and it is responsible for configuring the main memory addresses that peripherals can access through DMA. Any DMA access to main memory must go through the IOMMU. However, two PCI peripherals can often avoid the IOMMU, especially if both peripherals connect via the southbridge [85].

Access Control Services The PCI Express specification, especially as of revision 2.0, includes support for *Access Control Services* (ACS), intended to restrict the ability of devices to engage in unintended peer-to-peer exchanges [78, 79, 85]. While this is a promising mechanism that may help restrict the damage that can be inflicted by malicious peripheral devices, we emphasize that not all peripheral devices are PCI Express, and the restricting mechanism provides no any guarantee of the integrity of the firmware inside the peripheral.

2.2.4 Malware on Peripherals

On a modern computer motherboard, all the firmware-enabled peripheral devices (e.g. PCI or PCIe-based NIC, GPU, BIOS, USB peripherals) are vulnerable to attack. Once malware infects computer peripherals, it has the following features:

1. Malware on a peripheral can eavesdrop on sensitive data handled by the peripheral (e.g., passwords).

2. Malware on a peripheral may modify executable programs or scan sensitive data in main memory via DMA if the IOMMU is not perfectly configured or not present on a computer system.

3. Malware on one peripheral may spread malicious code to other peripherals through DMA.

4. Malicious peripherals can collude using peer-to-peer bus communication without involving the host CPU.

5. Malware on peripherals cannot be removed by firmware update tools if the firmware update procedure assumes that the existing firmware is benign.

2.3 Hardware-Based Trusted Computing Techniques

2.3.1 Dynamic Root of Trust for Measurement

AMD's Secure Virtual Machine (SVM) [5, 7] and Intel's Trusted eXecution Technology (TXT) [45, 46, 48] provide a mechanism called Dynamic Root of Trust for Measurement (DRTM), that can *late-launch* a piece of code with hardware-based protection and integrity measurement. When launching a piece of code, the DRTM mechanism reinitializes all CPUs to a known state, sets up DMA protection to protect the launched code, measures a cryptographic hash of the protected code, extends the integrity measurement into a Platform Configuration Register (PCR) in the system's Trusted Platform Module (TPM) for integrity verification, and then starts to execute the protected code. The DRTM mechanism was originally designed for dynamically launching a Virtual Machine Monitor (VMM) with

hardware-based protection and integrity measurement at any time when the OS is running. The protected code can run in the root privilege having full control over the system. Instead of launching a VMM, McCune et al. demonstrated how to run a Piece of Application Logic (PAL) using the DRTM mechanism to achieve secure code execution on an untrusted commodity computer in Flicker [67].

2.3.2 TrustVisor

TrustVisor [66] is a minimized hypervisor that isolates a Piece of Application Logic (PAL) from the rest of the system and offers efficient trustworthy computing abstractions (via a μTPM API) to the isolated PAL with a small TCB. Figure 2.2 shows the architecture of TrustVisor.

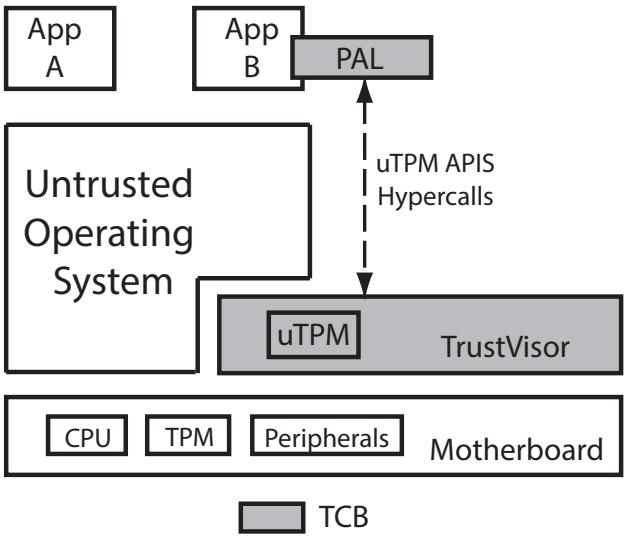

Figure 2.2: TrustVisor Architecture

Memory Protection TrustVisor isolates the memory pages containing itself and any registered PALs from the guest OS and DMA-capable devices by configuring

nested page tables and the IO Memory Management Unit (IOMMU). TrustVisor exposes hypercall interfaces for applications in the guest OS to register and unregister a PAL. When a PAL is *registered*, information including the memory pages of the PAL is passed to TrustVisor. TrustVisor configures nested page tables to isolate the memory pages of the PAL from the guest OS. Any access from the guest OS to the PAL or from the PAL to the guest OS causes a nested page fault that will be intercepted by the hypervisor. When a PAL is unregistered, TrustVisor zeroes the data memory in the PAL, and removes the memory protections on the PAL's address space.

Integrity Measurement TrustVisor employs a two-level integrity measurement mechanism for measuring the integrity of the hypervisor and registered PAL. TrustVisor is booted using the DRTM mechanism (Section 2.3.1) available on commodity x86 processors. The chipset computes an integrity measurement (cryptographic hash) of the hypervisor and extends the resulting hash into a Platform Configuration Register (PCR) in the Trusted Platform Module (TPM). TrustVisor computes an integrity measurement for each registered PAL, and extends that measurement result into the PCR of the PAL's μTPM instance. The TPM Quote from the hardware TPM and the μTPM Quote from the PAL's μTPM instance comprise the complete chain of trust for remote attestation.

2.3.3 On-Demand Isolated I/O

Zhou et al. [127] expanded TrustVisor using a wimpy kernel to support on-demand isolated I/O for an isolated PAL with a small TCB, thereby enabling an isolated PAL to securely access I/O peripherals. Figure 2.3 shows the system architecture providing on-demand isolated I/O.

In this architecture, the wimpy kernel runs in an isolated execution environment

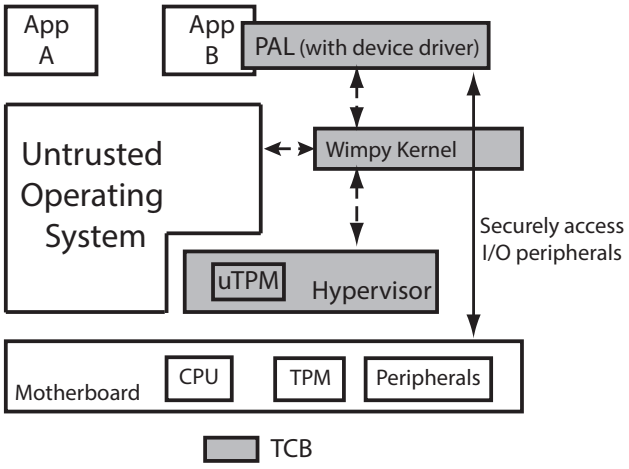

Figure 2.3: On-Demand Isolated I/O Architecture

established by a hypervisor. To provide isolated I/O with a small TCB, the wimpy kernel outsources the complex peripheral initialization processes to the OS, but verifies the peripheral information with a small TCB, and exports device drivers to the isolated PAL. The wimpy kernel (running in the root privilege) also configures the system registers to protect the I/O-port, I/O-memory, and device configurations from a malicious OS. However, the proposed I/O protection mechanism does not defend against malware in peripherals; consequently, malware in peripherals might obtain users' security-sensitive data (e.g., credit card information, bank account password).

2.4 Google Native Client

Google Native Client (NaCl) [122, 123] is a sandbox for x86 native code (called Native Module) using Software Fault Isolation (SFI) [65, 112].

2.4.1 Validator

To guarantee the absence of privileged x86 instructions that can break out of the SFI sandbox in a Native Module, a validator in NaCl reliably disassembles the Native Module and validates the disassembled instructions as being safe to execute.

2.4.2 Runtime Framework

NaCl provides a simple service runtime including a context switch function and a system call dispatcher to support the execution of a Native Module. On 32-bit x86, the service runtime and the Native Module are isolated using the CPU's segmentation mechanism [48]. NaCl simulates system calls for a Native Module using a *Trampoline Table and Springboard*. There is a Trampoline Table in each Native Module, and a 32-byte entry in the Trampoline Table for each supported system call. For each system call, the Google NaCl toolchain ensures that control transits to one of the entries in the Trampoline Table, instead of to a traditional system call. The Trampoline Table entries switch the active data and code segments, and jump to the context switch function in NaCl. The context switch function saves the thread context of the Native Module and transfers control to the system call dispatcher in NaCl. The system call dispatcher exposes only a subset of the OS system call interface to the Native Module, sanitizes the system call parameters, conducts access control to constrain the file access of the Native Module, and finally calls the corresponding handler in the OS. After the handler execution completes, the context switch function restores the execution context of the Native Module and calls the *Springboard*, which performs the inverse of the control transitions in Trampoline Table entries.

Chapter 3

Mead: Establishing a Malware-Free System State on Embedded Platforms

An adversary who can insert malware into a system poses a persistent threat. Malware can survive across repeated boot operations and can be remotely activated at the adversary's discretion. Attempts to detect persistent malware in a system usually require off-line forensic analysis and hence do not offer timely recourse after a successful attack. In contrast, on-line detection of adversary presence in a system can be fast (e.g., a matter of seconds or minutes), but typically requires some form of hardware- or software-based attestation by a verifier to test the system's state. When strong guarantees are sought in attestation despite persistent adversary presence, designers usually rely on secrets protected in tamper-resistant hardware and standard cryptography; viz., the private keys of a Trusted Platform Module (TPM) [108].

However, hardware-protected secrets can still be successfully attacked by ex-

ploiting compelled/stolen/forged certificates for hardware private keys [55, 83], side channels [119], and padding oracles [13]. Equally important, managing hardware-based attestation (e.g., TPM-based) poses significant usability challenges; e.g., the Cukoo attack [76].

Software-based attestation aims to avoid management of secret keys and their protection in hardware. Software-based attestation uses verifiers that challenge adversary-controlled systems with the execution of checksum functions whose output is verified and execution time is measured [10, 58, 91, 93, 94, 96]. Hence, a checksum function must have *accurately measurable* execution time bounds. Inaccurate verifier measurements would allow an adversary to exploit the gap between the verifier's expected measurements and the adversary's lower actual execution time. To avoid numerous false alarms on realistic system configurations, a verifier's time measurement must be dilated to account for a checksum's execution-time jitter. This includes clock variation due to static skew and dynamic (e.g., peak-to-peak) jitter, each of which can easily extend a processor's clock period by 3–8% [106]. Unfortunately, the attacking detection threshold of the measured time in previous schemes is limited (e.g., less than 1.7% over the normal condition in [58]). In addition, to act as a root of trust, software-based attestation must be uninterruptably linked to other functions. Otherwise, an adversary could pre-plan unpleasant surprises for a verifier. Modern embedded platforms have became increasingly complex with new hardware features. The new hardware features on modern embedded platforms might enable attackers to break the link.

Contributions In this chapter we address the problem of establishing a malware-free system state on untrusted embedded platforms using software-only approaches. In particular, we make the following contributions.

1. We present new architecture features of embedded system platforms that pose heretofore unexpected challenges to all prior software-based attestation protocols.

2. We define three new classes of attacks against software-based attestation protocols that are enabled by both new architecture features and scalability considerations on commodity computing platforms.

3. We present the implementation of practical protocols based on a new checksum design to counter these attacks, and explain their applicability to complex multiprocessor systems comprising heterogeneous processors; e.g., processors for DSP and ARM platforms.

4. We evaluate our countermeasures on a popular embedded system platform (i.e., the Gumstix FireStorm COM) running a Linux operating system (i.e., Pocky 3.5 and Linux kernel 3.5) and present the evaluation results.

3.1 Assumptions & Attacker Model

Assumptions We assume attackers cannot physically change the hardware configuration of the prover device (the untrusted embedded platform under attestation), such as adding additional memory, replacing the device's CPU with a faster CPU or over-clocking the device's CPU frequency. Preventing or detecting physical attacks is out of scope. We assume that a proxy attack (recall Section 2.1.3) can be detected by the user and attackers cannot control or access the communication channel between the prover device and the verifier machine.

We assume that the verifier program running on a verifier machine is free of vulnerabilities and that the verifier machine is free of malware. Malware on the prover device thus cannot compromise the verifier program or the verifier machine.

We assume that the verifier program knows the exact configuration of the prover device (e.g., the memory size and the kernel image version) and has a golden image (the correct device image without malware) of the prover device.

We assume that the checksum function we deploy to the prover device is optimal for performance: attackers cannot optimize the checksum function to run it faster or find an alternative algorithm to compute the result faster. However, we assume that an adversary can compress the checksum function size and run a compressed checksum function with smaller memory space (e.g., by reducing duplicated instruction blocks). Finally, attackers cannot break standard cryptographic primitives [72].

Attacker Model Attackers may deploy arbitrary software images or files to the prover device and attempt to hide the malicious contents from being detected. For example, attackers may run a modified checksum function on the device and attempt to forge the correct checksum result within the expected time (to protect malicious content from being detected) during the attestation procedure. Attackers may also attempt to configure the platform to generate interrupts or other events to break the untampered execution environment (software-based root of trust) without being detected.

3.2 Mead Overview

Mead establishes an untampered execution environment on a commodity embedded platform, verifies the integrity of the system image, and resets system configurations to achieve a malware-free state on the embedded platform. Figure 3.1 shows the Mead system architecture and verification procedure.

In Mead, a verifier program runs in a trusted verifier machine and performs

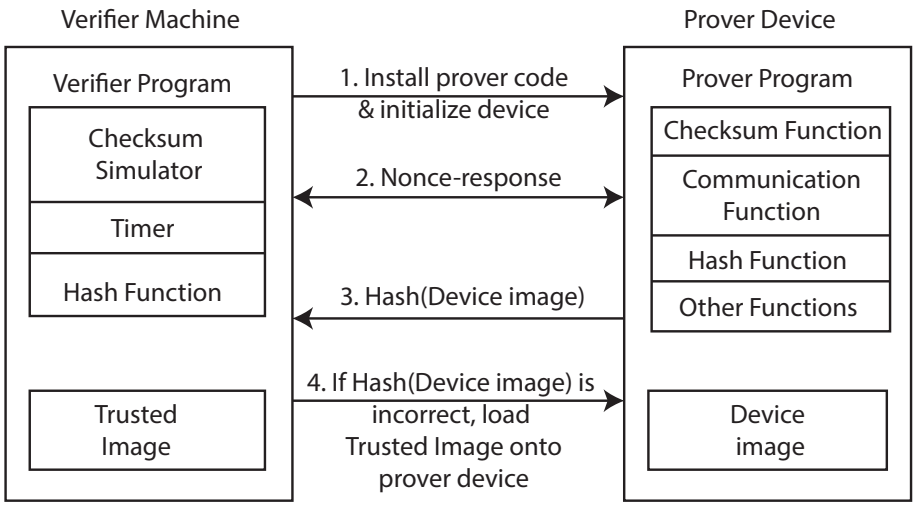

Figure 3.1: Mead system architecture and verification procedure.

attestation on a prover device through a nonce-response protocol. The verifier program comprises a checksum simulator, a timer, and a cryptographic hash function. The checksum simulator generates nonces for the attestation, constructs a copy of the device memory contents, and computes the expected response (i.e., checksum result) by simulating the checksum computation on the prover device. The timer measures the elapsed time of the nonce-response reception. A trusted image (correct image without malware) is in the verifier machine and the hash function computes a hash of the trusted image for integrity verification.

On the prover device, a prover program is installed and includes a checksum function, a communication function, a cryptographic hash function, and other functions. The checksum function disables interrupts on the prover device, resets system configurations in a known state (critical system configurations will be incorporated into the checksum), computes a checksum over the prover program and other critical contents (e.g., page table, stack, exception handler table, communication buffer, and critical system configurations), and establishes an untampered

execution environment for the attestation.

Because the checksum function is carefully designed, any additional operations will invalidate the checksum result or cause a detectable computation overhead. The verifier program verifies the response (checksum result) and the elapsed time of the nonce-response pair. If the checksum result is correct and the measured time is within a detection threshold, the verifier program obtains the guarantee that an untampered execution environment has been established on the prover device, and subsequent results sent by the prover device are to be trusted.

After sending the checksum result to the verifier program, the prover program sets system configurations and any memory locations that are not measured by the checksum function with known values (to erase malicious contents on system configuration space and memory locations that are not measured by the checksum function), and then calls a hash function to compute an integrity measurement (i.e., cryptographic hash) of the entire device image on the secondary storage (e.g., NAND Flash) and sends the measurement to the verifier program. The verifier program verifies the integrity of the device image, and if the device image has been changed, the verifier program loads the trusted image to the device to establish a malware-free state on the prover device.

In Mead, the checksum function can also fill all the spare memory space with pseudo-random values and then computes a checksum over the entire memory contents (instead of only over the prover program and other critical contents). In this way, the verifier program can prevent attackers from using the spare memory space on the prover device to perform malicious operations (Section 2.1.3). However, modern embedded platforms might have a large memory (e.g., 1GB). Verifying the integrity of the entire memory space by the checksum function might significantly increase the attestation time.

3.3 New Attacks & Challenges

In this section, we describe three new classes of attacks against existing software-based attestation protocols and new challenges that we need to address on modern embedded platforms.

3.3.1 Future-Posted Event Attacks

Some modern embedded-system platforms allow the configuration of *future-posted* events. These events can be set during system configuration (e.g., during Step 1 of Figure 3.1) and trigger at a future time when the system runs (e.g., after Step 2 of Figure 3.1). An example of such an event is the *future-posted Watch-Dog Timer (WDT) reset*. Other examples include the future-posted DMA transfers.

On some embedded platforms, attackers can configure the WDT to reset the device after a specific timer period. For example, on TI DM3730-based platforms, the CPU [107] supports the future-posted WDT reset, and the specific time period to reset the device can be configured as between about 62.5 microseconds and 74 hours and 56 minutes. As a result, attackers can perform *future-posted WDT reset attacks*.

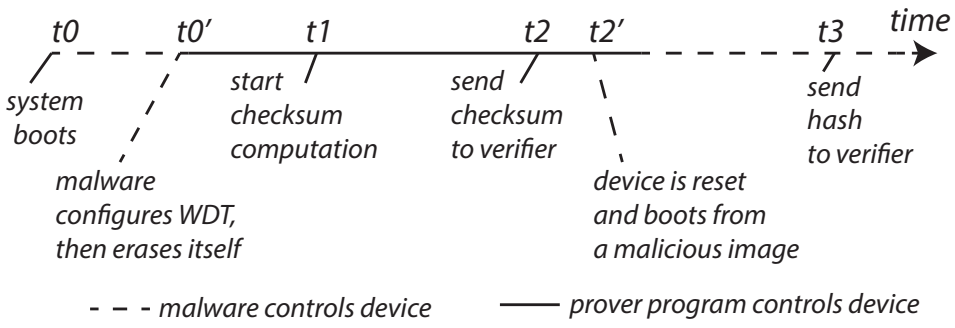

Figure 3.2: Timeline of a WDT reset attack.

Figure 3.2 shows the timeline of this attack. Suppose that malware controls the platform during the installation of the prover code (i.e., Step 1 of Figure 3.1) and configures the WDT to reset the device after the *correct checksum result* is sent to the verifier program; i.e., after Step 2 of Figure 3.1. Then the malware erases itself from memory and invokes the prover program. During the attestation, the prover program calls the checksum function to compute a checksum over the memory contents based on the nonce from the verifier program, and then sends the correct checksum result to the verifier program. After the checksum result is sent, the WDT reset event is triggered and the platform boots from an adversary-modified device image. After reboot, the malware of the device image controls the platform and sends a forged hash result (i.e., integrity measurement of the device image) to the verifier program.

3.3.2 I-cache Inconsistency Attacks

Modern embedded processors have multiple-level caches. However, to save energy, some embedded processors may not have hardware support for cache coherence between Instruction-cache (I-cache) and Data-cache (D-cache), and software has to maintain cache coherence. For example, the ARM Cortex-A8 processor, which is widely deployed on embedded platforms, does not have hardware support for cache coherence between I-cache and D-cache. Software has to use cache maintenance instructions to ensure cache coherence. Therefore, the contents of the I-cache may differ from those of the D-cache, and attackers can leverage this feature to hide malicious instructions (e.g., malicious instructions in the communication function or hash function) in the I-cache without being detected. We call this attack the *I-cache inconsistency attack*.

This attack is similar in spirit to the Split-TLB attacks (Section 2.1.3), where

the I-TLB and D-TLB contain inconsistent mappings for the checksum function pages. Experience with those attacks suggests that the *I-cache inconsistency attack* is equally practical, particularly since its setup is simpler.

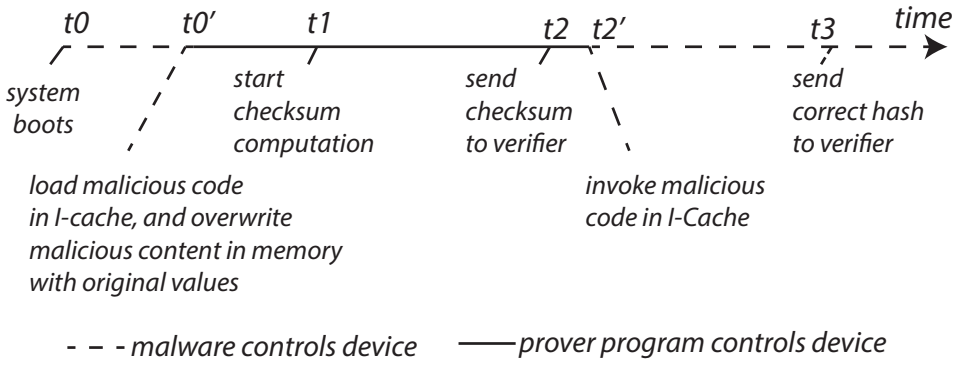

Figure 3.3: Timeline of an *I-cache inconsistency attack*.

Timeline. The timeline of an I-cache inconsistency-based attack is shown in Figure 3.3. Here, malware first loads malicious instructions into the I-cache, then overwrites the malicious content in memory with the original values to guarantee that only legitimate contents are in the memory during checksum computation. The malicious code needs to comprise only a few instructions of the hash or communication function. The checksum function is computed over the legitimate memory contents and the correct checksum result is sent to the verifier program. After the checksum result is sent to the verifier program, the malicious instructions in the I-cache are invoked, and then the adversary controls the system.

3.3.3 Measured Time-Variance Based Attacks

In software-based attestation protocols, the measured time of one nonce-response round is utilized to detect malicious operations on the prover device; e.g., malicious

operations to forge correct checksum results. Typically the attack-detection threshold is set based on the overhead caused by possible attacks. If a measured time exceeds the threshold, it is highly likely that malicious operations are performed on the prover device. However, the measured time may exhibit significant variance caused by CPU clock variance, Translation Look-aside Buffer (TLB) misses, and possibly cache misses. Hence, to avoid false-positive malware detection, the attack-detection threshold must be extended over the maximal value of the measured time in normal (no-attack) conditions.

Recent research [106] shows that the modern CPU clock variance can be up to 3-8% and it increases with program execution times. Furthermore, because the traditional checksum functions read the memory contents in a pseudo-random pattern, the resulting TLB misses could increase the measured execution time significantly. To account for these types of execution-time jitter, the maximal value of the measured time in normal (no-attack) conditions must be extended significantly; e.g., by nearly 3% of the average execution time.

Previous software-based attestation schemes did not have to account for *high* execution-time variance in setting the maximal execution time threshold. Their designs made cache behavior fairly predictable; e.g., the checksums fit into the cache and random access patterns resulted in predictably high overheard. TLBs need not be used since measurements were taken in physical RAM, and clock jitter was small because checksum execution times were short for relatively small memory configurations. In contrast, some of the new embedded-system processors force virtual memory (and hence the TLB) use whenever caches are used, and large memory configurations (i.e., GB size) cause checksum functions to execute for minutes instead of tens of milliseconds. Consequently, attackers can now exploit the high execution-time jitter on embedded systems to launch successful time-variance based attacks with *non-negligible probability*.

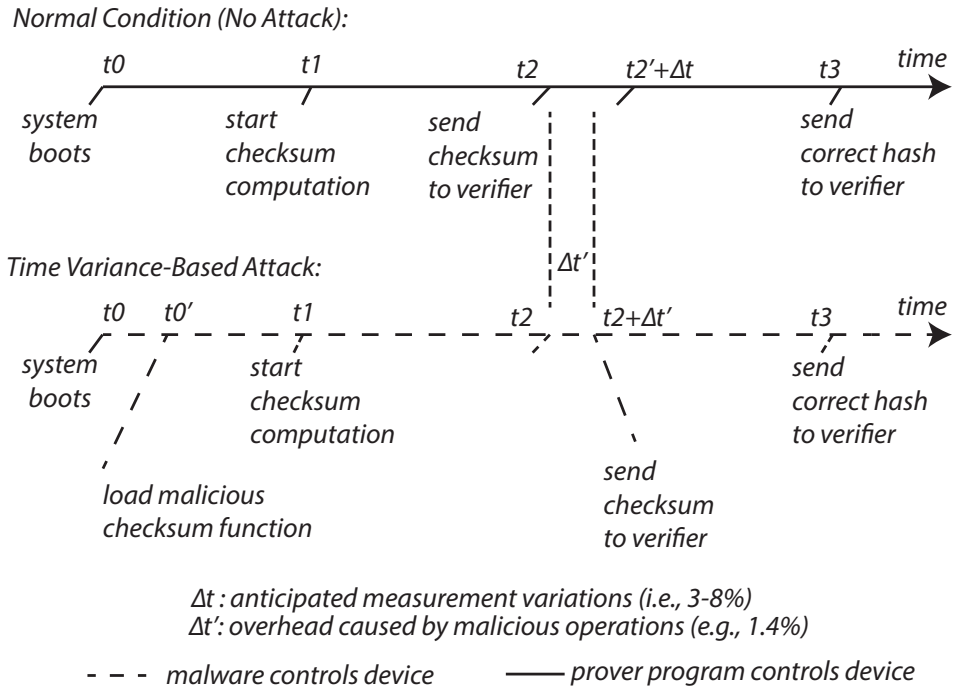

Figure 3.4: Timeline of normal condition and time-variance based attack.

Timeline. Figure 3.4 shows the timeline of a *time-variance based attack*. Here, malware that controls the platform loads a modified checksum function that computes the expected checksum result. To protect the modified contents (i.e., malicious code) from being detected, the modified checksum function performs additional operations to forge the expected checksum, and these operations cause an overhead $\Delta t'$. However, as shown in the figure, under normal (no-attack) conditions, the anticipated measured time variation is Δt (i.e., the timing detection threshold), is larger than $\Delta t'$. Consequently, the verifier receives the correct checksum result within the timing detection threshold, and hence the verifier cannot detect this attack; i.e., a false-negative detection result.

3.3.4 New Challenge: Heterogeneous Processor Architecture

Modern commodity-embedded platforms may have multiple processors that are heterogeneous processors. Figure 3.5 shows an example of a modern commodity-

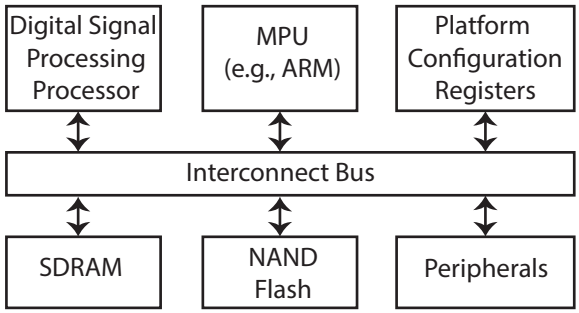

Figure 3.5: An example of a commodity embedded platform with heterogeneous processors.

embedded platform with multiple processors. In this example, the platform has a Main Processing Unit (MPU) processor (e.g., ARM processor) and a Digital Signal Processing (DSP) processor (for video and audio processing). The MPU and the DSP share the same interconnect bus to access the SDRAM and NAND Flash. To establish a software-based root of trust on such a platform, we may need to run checksum functions on both processors. However, it could be extremely challenging to design and implement the checksum functions for special-purpose processors (e.g., DSP). For example, the DSP processor for video or audio processing may have complex architecture (e.g., complex pipelines for parallel computation), and it may be extremely challenging to design and implement a checksum function for the DSP processor. If the checksum function on one processor is broken, the malicious code on one processor will in turn subvert the execution of other processors.

3.4 Countermeasures

We expand software-based attestation model to cover new hardware features of commodity embedded platforms. In this section, we describe the mechanisms to address the new attacks and challenges described in Section 3.3.

3.4.1 Verifying Critical Configurations

To establish an untampered execution environment on embedded platforms, it is important to verify or reset critical platform configurations to guarantee that attackers cannot leverage malicious configurations to tamper with the prover program without being detected Pioneer [94] proposes calling on a piece of *Epilog* code to reset or verify all critical configurations *before* sending the checksum result to the verifier program but *after* the checksum computation, in order to guarantee that the platform will be in a known legitimate state after the checksum result is sent to the verifier program. On embedded platform with multiple heterogeneous processors, the checksum function might need to call the *Epilog* code several times in a random pattern (details are described in Section 3.4.5). In addition, modern embedded platforms have rich and complex configurations; consequently, the *Epilog* code might need to verify or reset a number of configuration registers with a high overhead to read or write those configuration registers. We analyze the requirement for the overhead caused by the *Epilog* code.

Requirements To reconfigure the critical configurations in the memory-mapped register space, the prover program needs to either configure the memory-mapped register space as non-cacheable or perform additional operations to clean the corresponding cache blocks (pushing the corresponding cache blocks to the memory-mapped register space) in the *Epilog* code using cache maintenance instructions.

However, a read or write operation on a non-cacheable memory-mapped register space is much slower than a cacheable read or write operation. Cache maintenance operations are also expensive. Attackers could configure the memory-mapped register space as cacheable (by changing the page table attributes) or skip the cache maintenance instructions to reduce the overhead of the *Epilog* code. In an attack in which attackers need to perform additional operations, attackers can get time compensation by configuring the memory-mapped register space as cacheable or skip the cache maintenance instructions, thereby reducing the overall overhead of the attack.

Under normal conditions, a round-trip communication time between the verifier program and the prover program is T^{comm} while the time to verify or reconfigure critical configurations is T^{Epilog}_{normal}, and other checksum operation time is T^{cksum}. The measured time variance is ΔT. The time of one nonce-response pair is

$$T^{veri}_{normal} = T^{comm} + T^{cksum} + T^{Epilog}_{normal} \pm \Delta T \tag{3.1}$$

In an attack in which attackers configure the memory-mapped register space as cacheable or skip the cache maintenance instructions, the time to verify or reconfigure device configurations is T^{Epilog}_{attack} while the overhead caused by instructions to forge the correct checksum is T^{ops}_{attack}. Thus, the time of one nonce-response pair in this attack is

$$T^{veri}_{attack} = T^{comm} + T^{cksum} + T^{Epilog}_{attack} + T^{ops}_{attack} \pm \Delta T \tag{3.2}$$

The minimal overhead of this attack is

$$T^{overhead}_{attack} = T^{ops}_{attack} - (T^{Epilog}_{normal} - T^{Epilog}_{attack}) - 2\Delta T \tag{3.3}$$

We set the detection threshold as T_{max}. Then the detection overhead is

$$T^{overhead}_{detection} = T_{max} - T^{veri}_{normal} \tag{3.4}$$

To detect this attack, we must have

$$T_{detection}^{overhead} < T_{attack}^{ops} + T_{attack}^{Epilog} - T_{normal}^{Epilog} - 2\Delta T \tag{3.5}$$

3.4.2 Prevention of I-cache Inconsistency-Based Attacks

To prevent an I-cache inconsistency-based attack, the checksum function must guarantee that attackers cannot keep malicious instructions in the I-cache without being detected during the checksum computation.

One way to prevent attackers from hiding malicious instructions in I-cache is to run a checksum function, whose size is several times that of the I-cache (e.g., two times or four times the size of the I-cache). During the checksum computation, the checksum function causes a large number of cache block replacements in every I-cache block, and the cache blocks that contain malicious instructions are replaced by the checksum function blocks. However, attackers might be able to compress the checksum function size (e.g., by reducing duplicated instruction blocks) to be smaller than the I-cache size, and then run a compressed checksum function in the I-cache to avoid cache block replacements (and protect malicious instructions in the I-cache from being replaced). Although the compressed checksum function might need to run additional instructions (e.g., additional jump instructions), the overhead caused by additional instructions might not be detectable.

To prevent attackers from running a compressed checksum function in the I-cache without being detected, we propose a novel mechanism, in which the checksum function dynamically updates the Virtual Address-to-Physical Address (VA-to-PA) mappings in the page table to randomize the virtual addresses of the memory pages to verify. In this section, we first describe the dynamic page table mechanism and then describe possible attacks that attackers can perform.

Dynamic Page Table

In software-based attestation schemes, the checksum function computes checksum over memory contents that include all memory pages of the prover program (including the checksum function pages) and other critical memory contents (e.g., stack, communication buffer, exception handler table, and page table) to establish an untampered execution environment. During the checksum computation, the memory contents to verify are mapped in contiguous virtual memory and the checksum function reads the memory contents using virtual addresses.

In the mechanism we propose, the checksum function dynamically updates the VA-to-PA mappings of the virtual memory of all memory pages to verify (e.g., randomly pick two mappings of the memory pages to verify in the page table and exchange the values of the two mappings). To avoid a data memory page being mapped to the virtual address that the checksum function is running, we run the checksum function in a separate virtual memory. Thus, the memory pages of the checksum function are mapped to two virtual memory spaces. The checksum function also dynamically updates the VA-to-PA mappings of the virtual memory for the checksum execution. To access the page table in constant virtual addresses, the page table memory pages are also mapped to a separate virtual memory space. Figure 3.6 shows an example of the virtual memory mappings. In this example, the memory pages of all the memory pages to verify (including the checksum function and the page table) are mapped in the Virtual Address (VA) space D; the memory pages of the checksum function are also mapped in a separate virtual memory (the VA space A) for checksum execution; the *Epilog* code memory pages are mapped in the VA space B; and the page table memory pages are mapped in the VA space C (to guarantee that the checksum function can update the page table contents in constant virtual memory addresses). During the checksum computation, the checksum

function runs in the VA space A and computes checksum over all the VA spaces (i.e., A, B, C, and D).

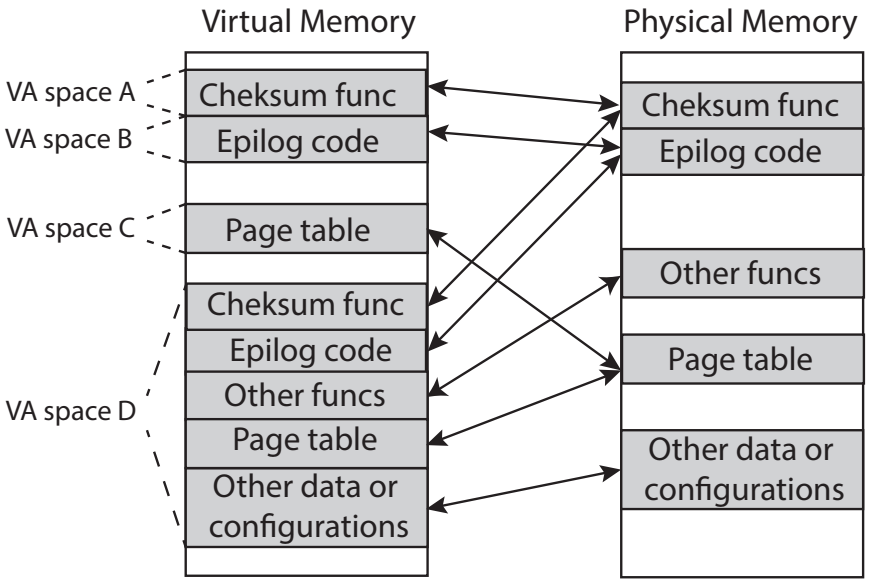

Figure 3.6: Virtual memory and physical memory mappings.

Novelty The novelty of our approach is that, when the checksum function dynamically updates the VA-to-PA mappings of the VA spaces A and D, the checksum function does not explicitly invalidate the VA-to-PA mappings in the TLB. As a result, previous VA-to-PA mappings cached in the D-TLB or the I-TLB might still be utilized by the CPU for data access or code execution. Only when a TLB miss happens, is a VA-to-PA mapping loaded from the page table to the TLB. Figure 3.7 shows an example of the TLB buffer contents and the page table contents. In this example, the TLB buffer has different mappings from the page table: only when a TLB miss happens, is the VA-to-PA mapping in the page table loaded into the TLB and utilized by the CPU.

Consequently, the virtual addresses of the memory pages in the VA spaces A

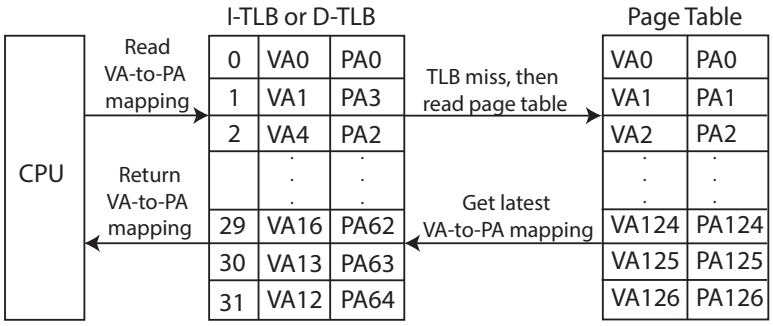

Figure 3.7: Page Table and TLB. TLB contains different mappings from the page table.

and D are dynamically changing during the checksum computation. Any additional memory operations or code executions that cause TLB misses might change the TLB replacement, and then invalidate the checksum result. In this way, we force attackers to perform sophisticated operations to guarantee that the malicious operations will not invalidate the checksum result, causing detectable overhead.

Requirements This dynamic page table mechanism requires that the TLB replacement policy be deterministic (e.g., the round-robin replacement policy or the least recently used replacement policy). Otherwise, the checksum function execution is not deterministic and the verifier program can not predict the correct checksum results. In addition, the checksum function execution and data access (for integrity measurement) must cover a number of memory pages (greater than the number of TLB entries) to guarantee that TLB cannot contain all mappings and TLB misses (in both I-TLB and D-TLB)will happen during checksum execution.

Furthermore, if the cache is Virtually Indexed and Virtual Tagged (VIVT), cache block contents are indexed and tagged by the corresponding virtual addresses. Only when a cache miss happens, does the CPU load the expected memory

content into the cache using the corresponding physical address. As a result, when the CPU reads a virtual address, the content that has been *previously* mapped to that virtual address can still exist in the cache, and it will be utilized for the checksum computation. In this condition, to make the checksum execution deterministic, the dynamic page table mechanism requires that the cache replacement policy be deterministic. When the cache Virtually Indexed and Physically Tagged (VIPT) or Physically Indexed and Physically Tagged (PIPT), the cache content is identified by its corresponding physical address. The cache replacement policy will not affect the checksum execution.

Security Analysis

When attackers compress the checksum function size, they might need to change the virtual addresses of some checksum blocks. Because the VA-to-PA mappings of virtual memory for the checksum execution are dynamically changing, the legitimate virtual address of each checksum block is dynamically changing during the checksum computation. In every checksum block, the Program Counter (PC) can be included in the checksum during checksum computation. As a result, the compressed function running in the I-cache has to compute the expected PC value. However, the compressed checksum function cannot simply read the page table to get the VA-to-PA mapping (to compute the expected PC value) because the I-TLB might contain a different mapping. Therefore, the compressed checksum function might need to simulate the I-TLB replacement to calculate the expected PC value, which will significantly increase the overhead of this attack.

Random Jump Furthermore, to increase the overhead of this attack, we can force attackers to simulate the I-TLB replacement in every checksum block by performing a random jump at the end of each checksum block. In the random jump

operation, the checksum block (that has been executed) randomly jumps to another checksum block, whose legitimate virtual address might be on another memory page (that might cause an I-TLB miss). For every code execution that might go to another memory page, the compressed checksum function has to simulate the I-TLB replacement.

When simulating I-TLB replacements, the compressed checksum function might need to perform additional memory read or write operations to save simulated TLB entry values in the memory and operate on those values (read, write, and update these values). However, the additional memory operations might change the D-TLB replacement and then invalidate the checksum result. Attackers can analyze the data pages used by the checksum function, find a memory page whose mapping is always in the D-TLB, and then save the simulated I-TLB entries on that data page. However, in this attack, attackers change the memory content to verify and need to perform additional operations to forge the correct checksum result (a memory copy attack or memory substitution attack in Section 2.1.3). To prevent this attack, we can carefully configure the data pages accessed by the checksum function and guarantee that no data page mappings are always in the D-TLB.

There are several other ways that could enable the compressed checksum function to simulate the I-TLB replacements without invalidating the checksum result. Now we describe and analyze the attacks.

Attack (I) The compressed checksum function can simulate both I-TLB and D-TLB replacements. Although the D-TLB replacement sequence is changed, the compressed checksum function can still calculate the legitimate address to read or write and guarantee that the checksum result is correct. However, simulating both I-TLB and D-TLB replacements will cause a high overhead that is not expected by attackers.

In this attack, the checksum computation time by a compressed checksum function is T_{comp}^{cksum}; the overhead to simulate I-TLB replacements is T_{ITLB}^{simu}; and the overhead to simulate D-TLB replacements is T_{DTLB}^{simu}. Then the measured time of one nonce-response pair is

$$T_{comp}^{veri} = T^{comm} + T_{comp}^{cksum} + T_{ITLB}^{simu} + T_{DTLB}^{simu} \pm \Delta T \tag{3.6}$$

The minimal overhead of this attack is

$$T_{comp}^{overhead} = T_{comp}^{cksum} + T_{ITLB}^{simu} + T_{DTLB}^{simu} - T^{cksum} - 2\Delta T \tag{3.7}$$

To detect this attack, we must have

$$T_{detection}^{overhead} < T_{comp}^{cksum} + T_{ITLB}^{simu} + T_{DTLB}^{simu} - T^{cksum} - 2\Delta T \tag{3.8}$$

In this attack, the compressed checksum function is forced to simulate the D-TLB replacement for every memory operation that might cause a D-TLB miss. Thus, to increase the overhead of this attack, we can carefully increase the number of memory read operations (that might cause D-TLB misses) in every checksum block, thereby forcing the malicious checksum function to perform more additional operations.

Attack (II) On embedded platforms, TLB entries might be preserved when the MMU is disabled (e.g., on an ARM Cortex A8 processor, all TLB entries are preserved when the MMU is disabled). After the MMU is disabled, attackers can perform arbitrary additional memory access without affecting the TLB replacements. Thus, the compressed checksum function can disable the MMU before starting to simulate the I-TLB replacement and enable the MMU after the simulation. However, disabling and enabling the MMU could be expensive (e.g., hundreds of CPU cycles). Furthermore, we find that the data cache might be disabled when the MMU

is disabled on some processors. When the data cache is disabled, data access would slow down and simulating the I-TLB replacements could consume a high execution overhead.

In this attack, the overhead to simulate I-TLB replacements when the MMU is disabled is $T_{simu\_ITLB}^{no\_mmu}$. Then the measured time of one nonce-response pair is

$$T_{comp}^{veri} = T^{comm} + T_{comp}^{cksum} + T_{simu\_ITLB}^{no\_mmu} \pm \Delta T \tag{3.9}$$

The minimal overhead of this attack is

$$T_{comp}^{overhead} = T_{comp}^{cksum} + T_{simu\_ITLB}^{no\_mmu} - T^{cksum} - 2\Delta T \tag{3.10}$$

To detect this attack, we must have

$$T_{detection}^{overhead} < T_{comp}^{cksum} + T_{simu\_ITLB}^{no\_mmu} - T^{cksum} - 2\Delta T \tag{3.11}$$

Attack (III) Another way to avoid affecting the D-TLB replacement is to save the simulated I-TLB entries values in spare control registers that are not in the memory-mapped register space. For example, ARM system control coprocessor registers are not memory-mapped. Some ARM system control registers are spare and can be used to save values. However, accessing control registers could be expensive and might take hundreds of CPU cycles.

In this attack, the overhead to simulate I-TLB replacements using the spare control registers is $T_{simu\_ITLB}^{regs}$. Then the measured time of one nonce-response pair is

$$T_{comp}^{veri} = T^{comm} + T_{comp}^{cksum} + T_{simu\_ITLB}^{regs} \pm \Delta T \tag{3.12}$$

The minimal overhead of this attack is

$$T_{comp}^{overhead} = T_{comp}^{cksum} + T_{simu\_ITLB}^{regs} - T^{cksum} - 2\Delta T \tag{3.13}$$

To detect this attack, we must have

$$T_{detection}^{overhead} < T_{comp}^{cksum} + T_{simu\_ITLB}^{regs} - T^{cksum} - 2\Delta T \tag{3.14}$$

3.4.3 Overcome Measured Time Variance

The measured time variance can cause false positive detection results if the overhead of the possible attacks is lower than the measured time variance. To reduce the false positive detection results, we can carefully design the checksum function to guarantee that the overhead caused by the possible attacks is higher than the measured time variance.

When we discuss possible attacks, we focus on the memory copy attacks and the memory substitution attacks (Section 2.1.1). We assume that other countermeasures already overcome the attacks, in which attackers do not need to perform additional operations during the checksum computation (e.g., the split-TLB attacks [90], the I-cache inconsistency-based attacks, or the future-posted event attacks). Please note that when the dynamic page table-based approach is applied, the I-TLB and D-TLB entries are frequently updated from a shared page table; consequently, attackers cannot preserve malicious mappings in either the I-TLB or the D-TLB to perform a split-TLB attack.

Increase Overhead of Memory Copy Attacks

In a memory copy attack, Program Counter (PC) and Data Pointer (DP) values are incorporated into the checksum, and attackers are forced to perform additional operations to forge the PC or DP values to compute the expected checksum, resulting in an execution overhead. However, attackers might only need to add several instructions in every checksum block to forge the PC or DP values, meaning the overhead is limited.

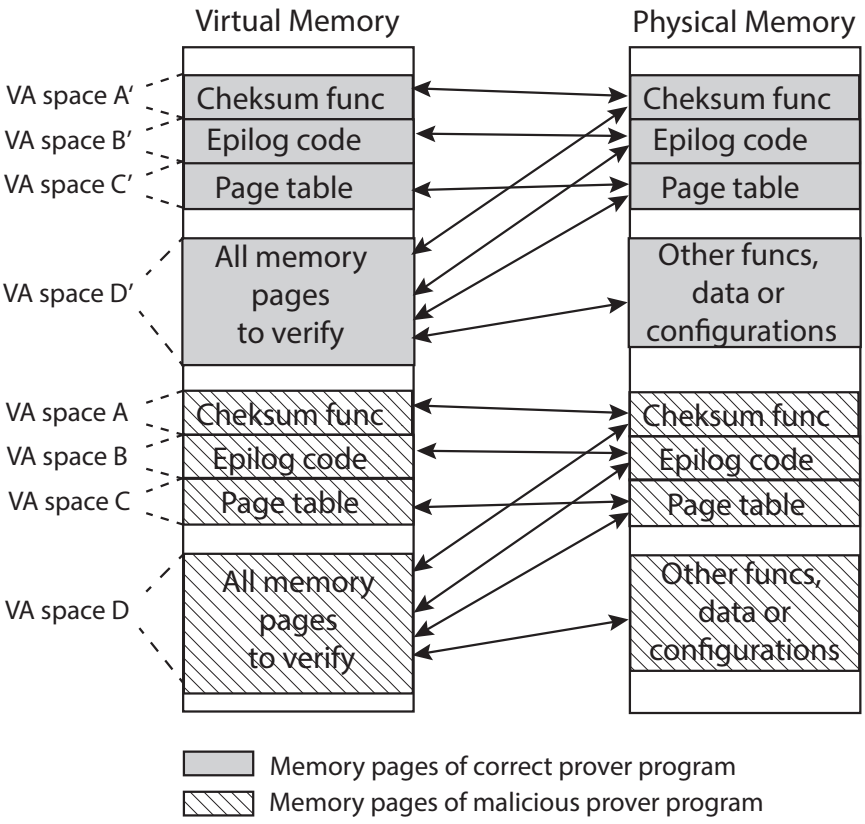

Figure 3.8: Memory mappings of Attack IV, Attack VI, or Attack VII.

One way to increase the overhead of memory copy attacks is to apply dynamically modified instructions in the checksum function (discussed in Pioneer-NG [90]), forcing attackers to perform additional memory operations to update the correct copy of the checksum function. However, the additional memory operations can be efficient if the dynamically modified instructions in the correct copy of the checksum function are in the D-cache; consequently, the overhead caused by the additional operations might still be limited and within the measured time variance.

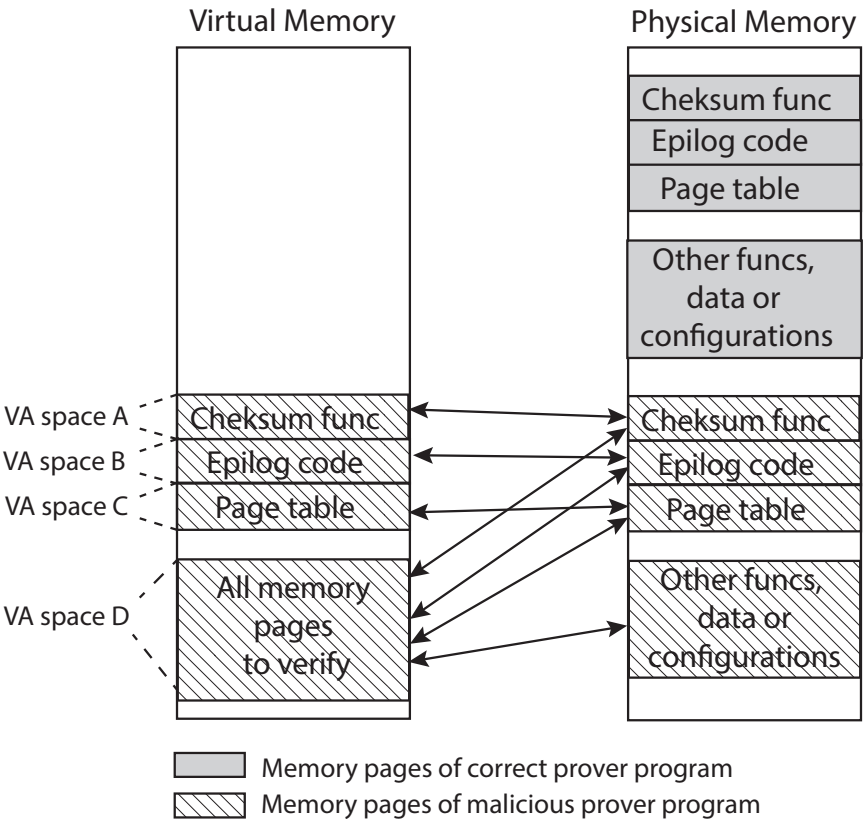

Figure 3.9: Memory mappings of Attack V or Attack VIII.

Dynamic Page Table Fortunately, the dynamic page table mechanism described in Section 3.4.2 not only prevents the I-cache inconsistency-based attacks, but also significantly increases the overhead of the memory copy attacks.

In the first type of memory copy attacks (Section 2.1.3), the VA spaces *A* and *D* (Figure 3.6) contain a malicious checksum function. Without the dynamic page table mechanism, when the checksum function memory pages are read, the malicious checksum function can simply add a constant offset to the DP to redirect the memory read. However, when the dynamic page table mechanism is applied, the virtual addresses of memory pages in the VA spaces *A* and *D* are dynamically

changing and, consequently, the malicious checksum function is forced to perform additional operations to calculate the virtual address of the expected content before redirecting the memory read.

In the second type of a memory copy attack (Section 2.1.3), a malicious checksum function runs in another VA space and computes the checksum over the VA spaces A, B, C, and D (the original copy of the memory contents to verify). In this attack, the malicious checksum function needs to perform similar operations to compute the expected checksum, except that the malicious checksum function needs to forge the PC value instead of the DP value.

Using a first type of memory copy attack as an example, we analyze the overhead of a memory copy attack when the dynamic page table mechanism is applied.

Attack (IV) In the first way, the malicious checksum function uses constant VA-to-PA mappings for the VA spaces A and D; simulates the D-TLB replacement to calculate the memory address to read or write; adds a constant offset to the memory address to read to redirect the memory read; updates the page table in the VA space C' and D' to guarantee that the expected page content is in the VA space C' and D'; forges the DP and PC values; and simulates the I-TLB replacement to compute the expected checksum result.

In this attack, the checksum computation time by the malicious checksum function is $T_{Att\_IV}^{cksum}$; the overhead to forge the DP value is T_{DP}^{forge}; the overhead to forge the DP value is T_{PC}^{forge}. Then the measured time of one nonce-response pair is

$$T_{Att\_IV}^{veri} = T^{comm} + T_{Att\_IV}^{cksum} + T_{ITLB}^{simu} + T_{DTLB}^{simu} + T_{DP}^{forge} + T_{PC}^{forge} \pm \Delta T \tag{3.15}$$

The minimal overhead of this attack is

$$T_{Att\_IV}^{overhead} = T_{Att\_IV}^{cksum} + T_{ITLB}^{simu} + T_{DTLB}^{simu} + T_{DP}^{forge} + T_{PC}^{forge} - T^{cksum} - 2\Delta T \tag{3.16}$$

To detect this attack, we must have

$$T_{detection}^{overhead} < T_{Att\_IV}^{cksum} + T_{ITLB}^{simu} + T_{DTLB}^{simu} + T_{DP}^{forge} + T_{PC}^{forge} - T^{cksum} - 2\Delta T \qquad (3.17)$$

The main overhead of this attack is caused by the operations to simulate the I-TLB and D-TLB replacements. Thus, we can increase the number of memory read operations (that might cause D-TLB misses) and the number of random jumps to increase the overhead of this attack.

Attack (V) In the second way, the malicious checksum function dynamically updates the VA-to-PA mappings of the VA spaces A and D; performs necessary memory operations over the VA spaces A, B, C, and D (to avoid changing the D-TLB replacement); gets the PA of the correct memory content in certain ways; performs an additional memory read to get the correct memory content for every memory read operation; incorporates the correct value into the checksum to compute the expected checksum.

In this attack attackers do not need to modify the page table to create VA spaces for the correct copy of the memory content to verify because the malicious checksum function reads the correct copy using the corresponding PA.

There are two possible ways for the malicious checksum function to get the PA of the correct memory content.

1. The malicious checksum function might be able to get the PA of a VA via the system control coprocessor (without changing the D-TLB replacement). For example, on an ARM Cortex A8 processor, the system control coprocessor can provide VA-to-PA translations for software. The malicious checksum function can call the system coprocessor to get the PA of the VA to read, and then add a constant offset to the PA to get the PA that saves the expected memory content. However, accessing the system control coprocessor can

take hundreds of CPU cycles, and frequently accessing the system control coprocessor for every memory read might cause a high execution overhead. In this attack, the malicious checksum function needs to call the system control coprocessor to perform the VA-to-PA translation for every memory read over the VA spaces A and D. The overhead of each VA-to-PA translation by the system control processor is T_{VA2PA}^{cp15} while the numbers of the memory reads over the VA spaces A and D are N_A and N_D separately. The overhead to get the PAes of the correct memory content in this attack is

$$T_{VA2PA}^{overhead} = (N_A + N_D) \times T_{VA2PA}^{cp15} \tag{3.18}$$

2. Attackers can save the PA of the memory page that contains the correct memory content in a constant offset of the corresponding memory page in the VA spaces A and D. In this way, for every memory read over the VA spaces A and D, the malicious checksum function can easily get the PA of the memory page that contains the correct memory content by a memory read and then calculate the PA of the correct memory content to read. In this attack, the malicious checksum function must read the memory page of every expected memory addresses to read in the VA spaces A, B, C, and D to avoid changing the D-TLB replacement (but without incorporating the read result to the checksum). Thus, the additional memory read to get the PA of the memory page to read will not cause an overhead to this attack.

After obtaining the PA of the correct memory content, the malicious checksum function performs an additional memory read to get the expected memory value. To avoid changing the D-TLB replacement, the malicious checksum function needs to disable the MMU before the additional memory read, read the memory content using the corresponding PA, and enable the MMU after the read. When the MMU

is disabled, the D-cache might be disabled, the CPU accesses the memory using the PA and a memory read operation might take hundreds of CPU cycles, thereby causing a high overhead.

To guarantee that the additional code execution (for malicious operations) does not change the I-TLB replacement, attackers have to guarantee that the malicious code to perform malicious operations in every checksum block is in the same memory page with the corresponding checksum block. Therefore, attackers are forced to compress the checksum function size to get spare memory space for malicious code in every memory page of the checksum function. During the checksum computation, the page table content is dynamically updated, and the malicious checksum function has to guarantee that the correct page table content is incorporated into the checksum.

To reduce the number of additional memory operations, the malicious checksum function can configure the CPU to utilize the page table in the VA space C; dynamically update the page table in the VA space C as expected; read the VA space C to get the expected page table content when the VA space C is measured; check the memory address to read and read the VA space D to get the page table content when the page table memory pages in the VA space D are measured. The malicious checksum function can identify the page table memory pages by the PA of the memory address to read (save in a constant offset of the page table memory pages). When the memory location that saves the PA of the page table memory page is read, the checksum function function can either forge the expected value or get the expected value from the correct copy by an additional memory read.

T_{mmu} denotes the overhead to disable and enable the MMU one time; $T_{read}^{nocache}$ denotes the overhead of a non-cached memory read; N_B and N_C denote the numbers of memory reads over the VA spaces B and C separately; and N_D^{PT} denotes the number of memory read over the page table memory pages in the VA space D.

Then the overhead caused by additional memory reads is

$$T_{read}^{overhead} = (N_A + N_B + N_D - N_D^{PT}) \times (T_{mmu} + T_{read}^{nocache}) \tag{3.19}$$

Attack (VI) Using another way to perform the first type of memory copy attacks, attackers run the malicious checksum in the VA space A, dynamically update the VA-to-PA mappings for the VA spaces A, dynamically update the VA-to-PA mappings of the correct copies of the VA spaces A and D (i.e., the VA spaces A' and D'), and perform all necessary memory read operations over the copy of the original prover program (the VA spaces A', B',C', and D'). This approach is equal to that attackers adding a constant offset to the memory address to read, but without changing the D-TLB replacement sequence.

In this attack, to avoid invalidating the I-TLB replacement, attackers can choose to compress the checksum function size to get spare memory space for the malicious code. Another way to guarantee the correct checksum execution for the malicious checksum function is to simulate the I-TLB replacement (to compute the correct PC value incorporated into the checksum and the target address to jump to in a random jump operation) without dynamically updating the mappings of the VA space A.

In this attack, the malicious checksum function also needs to guarantee that the correct page table content is incorporated into the checksum. The malicious checksum function can incorporate the correct page table content in two ways:

1. The malicious checksum function configures the CPU to utilize the page table in the VA space C, dynamically updates the page table in the VA space C, dynamically updates the page table in the VA space C' (to guarantee that the VA spaces C' and D' contain the expected page table contents), and incorporates the page table content in the VA spaces C' and D' into the checksum.

In this attack, to avoid changing the D-TLB replacement sequence, the malicious checksum function disables the MMU to preserve the D-TLB entries when updating the page table in the VA space C and enables the MMU after the update. During the checksum computation, the page table content is frequently updated; consequently, the malicious checksum function is forced to frequently disable/enable the MMU and perform memory write operations when the D-cache is disabled, causing a high overhead. When the checksum function updates the mappings for the VA space A', it also needs to update the mappings of the VA space A. To randomly update two entries of the page table, the checksum function needs to perform two memory read operations and two memory write operations. In the checksum computation, the checksum function updates the mappings of the VA space A' for $N_{A'}^{PT}$ times; updates the mappings of the VA space A for N_{A}^{PT} times ($N_{A'}^{PT}$ is equal to N_{A}^{PT}); and updates the mappings of the VA space D' for $N_{D'}^{PT}$ times. We denote the overhead of a non-cacheable memory write is $T_{write}^{nocache}$ Then the overhead of updating the page table in the VA space C in this attack is

$$T_{PT}^{overhead} = N_{A}^{PT} \times (T_{mmu} + 4 \times T_{read}^{nocache} + 4 \times T_{write}^{nocache}) + N_{D'}^{PT} \times (T_{mmu} + 2 \times T_{read}^{nocache} + 2 \times T_{write}^{nocache}) \quad (3.20)$$

2. Alternatively, the malicious checksum function configures the CPU to utilize the page table in the VA space C'; dynamically updates the page table in the VA space C'; computes the checksum over the content in the VA spaces A', B', C', and D'. In this attack, the malicious checksum function adds the VA-to-PA mappings for the VA spaces A', B', C', and D' in the page table being measured (in the VA space C'); but the memory space that saves the additional mappings should contain pseudo-random values (invalid mapping

values). Thus, the malicious checksum function is forced to perform additional operations to guarantee that the expected values (instead of values of the additional mappings) are incorporated into the checksum. Because the additional mapping values only exist in the page table and can be easily distinguished from other mapping values or other memory contents, the malicious checksum function can check the returned value of every memory read when the VA spaces C' and D' are measured and performs an additional read over the VA space C to get the expected value if the returned value is an additional mapping. When the VA space C' is measured, the malicious checksum function could also check every memory address to read and redirect the memory read to the VA space C if the address containing an additional mapping is measured. But in this way, to avoid changing the D-TLB replacement, the malicious checksum function still needs to perform a read or write on the expected memory page to read in the VA space C'. The malicious checksum function might only need several instructions to check the returned value or the memory address to read. The overhead to check the memory address to read is similar to the overhead to check the returned value.

We denote the numbers of memory reads over the VA spaces C' and D' as $N_{C'}$ and $N_{D'}$ separately; the overhead to check one returned value or one memory address as T_{check}; the probabilities to read additional mappings (modified contents) in the VA spaces C' and D' are $P_{C'}$ and $P_{D'}$. The overhead to incorporate the expected values into this attack is

$$T_{mappings}^{overhead} = T_{check} \times (N_{C'} + N_{D'}) + (T_{mmu} + T_{read}^{nocache}) \times (P_{C'} \times N_{C'} + P_{D'} \times N_{D'}) \tag{3.21}$$

The memory range containing the additional mappings may be only a small

portion of the memory pages to verify in the VA spaces C' and D' ($P_{C'}$ and $P_{D'}$ are small). Although the additional memory read over the VA space C might be expensive, the main overhead of this attack might be caused by the additional operations to check the returned value or the memory address to read. However, T_{check} is small (only several additional instructions) and consequently, the overhead of this attack might be limited.

Dynamically Modified Instructions To increase the overhead of the second type Attack (VI), we can apply a large number of dynamically modified instructions in the checksum function. In this way, we can force the malicious checksum function running in the VA space A to perform additional operations to compute the expected checksum.

The malicious checksum function can compute the expected checksum result in three ways.

1. In the first way, the malicious checksum function dynamically updates the checksum instructions in the VA spaces A' and D', and performs additional memory write operations on the VA space A to update the checksum function instructions in the VA spaces A and D. To avoid changing the D-TLB replacement, the malicious checksum function has to disable the MMU (to preserve the D-TLB entries) when updating the instructions in the VA space A and enable the MMU after the update.

2. In the second way, the malicious checksum function dynamically updates the checksum instructions in the VA spaces A' and D', performing read-decode-execute operations to compute the expected checksum. In a read-decode-execute operation, the malicious checksum function running in the VA space A reads the dynamically modified instructions in the VA space A',

decodes them, and then forges the execution based on the decoding result. In this attack, attackers have to perform additional memory read operations to read the dynamically modified instructions, which might change the D-TLB replacement and invalidate the checksum result. Thus, the malicious checksum function has to disable MMU when reading the instructions and enable MMU after the read operations.

3. In the third, to reduce the number of additional memory operations (to reduce overhead), the malicious checksum function can save the random information (bits that are dynamically updated) of the dynamically modified instructions in the spare memory space or the spare system control registers. When the dynamically modified instructions need to be executed, the malicious checksum function reads the random information from the space memory space or the spare system control registers and then forges the execution. However, performing additional memory operations or frequently accessing the system control registers might cause a high execution overhead. In addition, we can increase the number of random bits updated to the dynamically modified instructions to force the malicious checksum function to perform more operations on the space memory space or the spare system control registers, thereby increasing the overhead of this attack.

Using dynamically modified checksum instructions also increases the overheads of previously described attacks (I), (II), (III), (IV), (V), and first type (VI), because in those attacks the malicious checksum function has to perform similar additional operations to compute the expected checksum result. In the Attacks (I and IV), the malicious checksum function does not need to disable the MMU when performing the additional read or write operations. In the Attack (V), the malicious checksum function can avoid the additional memory operations (to re-

duce the overhead) by checking every memory address to read when the VA spaces A and D are measured and read the VA spaces A and D when the dynamically modified instructions are measured.

Increase Overhead of Memory Substitution Attacks

The dynamic page table mechanism also significantly increases the overhead of the memory substitution attacks (Section 2.1.1). When the dynamic page table mechanism is applied, attackers can perform a memory substitution attack in two ways: Attack (VII) and Attack (VIII).

Attack (VII) The malicious checksum function uses constant mappings for the VA spaces A and D, simulates the I-TLB replacement for the correct checksum execution, simulates the D-TLB replacement to compute the memory address to read, checks every memory address to read, and redirects the memory address to read to the correct copy (by adding a constant offset to the memory address to read) when the modified content is being measured. The overhead of this attack is mainly caused by the operations to simulate the I-TLB and D-TLB replacements, which is similar to the overhead of the Attack (IV).

Attack (VIII) In this attack, the malicious checksum function dynamically updates the mappings of the VA spaces A and D, checks every memory address to read to identify if the modified content is being measured, and performs an additional memory read on the correct copy to get the expected value if the modified content is being measured. To avoid changing the D-TLB replacement, the malicious checksum function is forced to disable the MMU to preserve the D-TLB entries before the memory read, read the memory content using the PA, and enable the MMU after the memory read.

Attackers can reduce the number of the additional memory read operations by constraining the malicious code to be a small portion of all memory pages to verify. For example, attackers can only modify the *Epilog* code and a small part of the checksum function to perform attacks. In this condition, the malicious checksum function performs additional memory read operations only when the modified checksum instructions or the modified *Epilog* code instructions are being measured. As we discussed in Section 3.4.2, when the dynamic page table mechanism is applied, any additional code execution in extra memory pages might change the I-TLB replacement and invalidate the checksum result. Therefore, in this attack attackers are forced to keep the malicious instructions of every checksum block in the same memory page with the corresponding checksum block. Attackers need to compress the checksum function to get the spare memory space for malicious instructions in every memory page of the checksum function. Consequently, a large portion of the VA space A might contain the modified content and the malicious checksum function has to perform additional memory read operations in a high probability when the checksum function memory pages (e.g., in the VA space A) is being measured. Thus, we can increase the number of memory reads (for integrity measurement) over the VA space A in every checksum block to force the malicious checksum function to perform more additional memory read operations, thereby increasing the overhead of this attack.

During the checksum computation, the virtual addresses of the checksum function memory pages in the VA space D are dynamically changing. Thus, the malicious checksum function is forced to perform additional operations to identify if the memory page to read (in the VA space D) is a checksum function memory page before performing the additional memory read operations. As we discussed in the Attack (V) the malicious checksum function can save a special tag in a constant offset of every memory page of the checksum function (further guaranteeing that

the tag does not exist in other memory pages). The malicious checksum function can easily read the tag to identify if it is reading a memory page of the checksum function. The special tag can be the PA of the corresponding memory page that contains the correct copy of the original checksum function. For other memory pages that contain the modified content (e.g., the *Epilog* code memory page), the malicious checksum function can perform similar operations to identify the memory page to read, then check if the modified content is being read.

We denote the probabilities to read the modified content in the VA spaces A and D as P_A and P_D separately; denote the overheads to check one memory address to read in the VA space A or the VA space D (to identify if the modified content is being measured) as T^A_{check} and T^D_{check} separately. The overhead to get the expected memory content in this attack is

$$T^{overhead}_{sub} = T^A_{check} \times N_A + T^D_{check} \times N_D + (T_{mmu} + T^{nocache}_{read}) \times (P_A \times N_A + P_D \times N_D) \tag{3.22}$$

The modified content might be only a small portion of the VA space D (P_D is small). When the VA space D is measured, the overhead is mainly caused by the operations to check the memory address to read. In this attack, T^D_{check} is limited because the malicious checksum function might only need several instructions to check the memory address. However, P_A might be large as analyzed above and hence we can increase the overhead of this attack by increasing the value of N_A.

3.4.4 Measure The Entire Physical Memory

To establish a software-only root of trust, the checksum function can also fill the free memory space with pseudo-random values and then compute a checksum over the entire physical memory (the VA space D covers the entire physical memory). This section analyzes the attacks and limitations when the checksum function mea-

sures the entire physical memory.

Attack IX When the entire physical memory is measured during the checksum computation, attackers cannot find the free memory space to save a correct copy of the original prover program or to run a malicious checksum function. However, on some embedded platforms with secondary storage (e.g., NAND Flash), attackers can save a correct copy of the original checksum function in the second storage and then perform a memory substitution attack. On embedded platforms, accessing the secondary storage could be much more expensive than accessing the SDRAM or SRAM memory. For example, to get a 32-bit word from a NAND Flash (for the checksum computation), the CPU might need to (1) send a *read-data* command to the NAND Flash, (2) receive a block of data (e.g., 512 bytes) that includes the expected word from the NAND Flash, (3) read the corresponding Error-Correction Code (ECC) from the NAND Flash, and finally (4) perform error correction over the block of data to get the expected word. In this attack, the malicious checksum function might only need to access the secondary storage in a low probability because the modified content (e.g., the malicious checksum function) is only a small portion of the entire physical memory. For example, on a 512MB SDRAM, a 256KB checksum function is only 0.05% of the entire SDRAM memory. Consequently, the main overhead of this attack is caused by the operations to check the memory address to read.

The dynamic page table mechanism can significantly increase the overhead of this attack. In a memory substitution attack, attackers run a modified checksum function (the memory contents in the VA space A are modified), check every memory address to read, and access the secondary storage to get the expected value when the modified contents are measured. When the dynamic page table mechanism is applied, the malicious checksum function is forced to perform similar

operations as Attack VII or Attack VIII, except that the malicious checksum function accesses the secondary storage (instead of the free memory space) to get the expected value. In every checksum block, the VA space *A* might be measured several times (to increase the overhead of Attack VII or Attack VIII). To reduce the number of operations to access the secondary storage, attackers can save a correct copy of the original checksum function in the memory and save the original values of the memory that saves the correct checksum function in the secondary storage. During the checksum computation, the malicious checksum function checks every memory address to read when the VA space *A* is measured, reads the correct copy in the memory when the modified content in the VA space *A* is measured, checks every memory address to read when the VA space *D* is measured, gets the expected value from the secondary storage when the memory space saving the correct checksum function is measured, and gets the expected value from the correct copy of the checksum function when the modified parts of the checksum function in the VA space *D* is measured. Consequently, the overhead of this attack should be close to the overhead in Attack VII or Attack VIII, depending on the approaches (i.e., the approach in Attack VII or Attack VIII) attackers choose to perform.

Limitations An approach that measures the entire physical memory during the checksum computation is not scalable to the memory size. On the platform with a large memory (e.g., 512MB SDRAM), the checksum function needs to perform a large number of memory read operations in a random pattern to measure every memory location in high probability (based on the result of the Coupon Collector's Problem). As a result, the entire checksum execution time will be significantly increased, compared with the approach that only measures part of the memory. Also, the measured time variance caused by CPU jitter will be increased (Section 3.3.3). When the checksum function measures the entire physical memory (e.g., SDRAM)

instead of a part of the memory, the D-cache miss rate will increase. Consequently, the average execution time of every checksum block is increased (because the average memory read time is increased).

3.4.5 Attestation on Heterogeneous Processor Architectures

On embedded platforms, a large number of system control registers are available to the MPU, enabling the MPU to configure or control the hardware components, such as interconnect bus, peripheral interfaces, and the clock of other processors. We find that on embedded platforms it is possible that the MPU could configure system registers to disable special purpose processors or prevent other processors from accessing the main memory. Leveraging the hardware features, we might be able to establish a software-only root of trust on embedded platforms with multiprocessors without running a checksum function on every processor.

To establish a software-only root of trust, we can run a checksum function on the MPU (the fastest processor). The checksum function configures the platform to disable other processors or prevent other processors from accessing the main memory (where the checksum function is running) and the memory-mapped register space (where the critical configurations are), includes the configuration values in the checksum (i.e., in the *Epilog* code), and establishes an untampered execution environment using the software-based attestation approach. To prevent malicious code in other processors from overwriting the *Epilog* code, the checksum function calls the *Epilog* code in a random pattern and includes the critical configurations in the checksum.

However, malicious code on other processors might predict the time at which the *Epilog* code will be invoked and overwrite the *Epilog* code (i.e., to bypass it) after the checksum computation, but before the *Epilog* code is called. To bypass the

Epilog code, attackers can overwrite register-write instructions (that reset platform configurations) with *NOP* instructions and overwrite register-read instructions (that read configuration values) with instructions that construct correct configuration values without any reads.

To prevent this attack, the checksum function can call the *Epilog* code several times during the checksum computation in a random pattern and consequently, malicious operations to bypass the *Epilog* code either invalidate the checksum result or cause a detectable overhead.

In addition, because the checksum function cannot be computed in parallel, the attackers cannot run the checksum function on multiple processors in parallel to speed up the checksum computation.

This approach requires that the MPU be the fastest processor on the platform. When other processors are faster than the MPU the processors that run at a faster frequency (than the MPU) might have a different TLB architecture or TLB replacement, and consequently, have to simulate the TLB replacements to compute the expected checksum result for the MPU, thereby causing a detectable overhead.

3.5 Implementation

We implement a proof of concept prototype of Mead and evaluate our system using a Gumstix FireStorm COM[1] as the prover device and an HP laptop as the verifier machine. The device and the laptop are directly connected via Ethernet cables and a Linksys wireless-G broadband router. The router is set using the default configurations, allowing both the verifier machine and prover device to dynamically get IPs via DHCP protocol. We run a Linux operating system (based on Yocto

[1] https://store.gumstix.com/

Poky Dylan 9.0.0[2]) on the Gumstix FireStorm COM and the Linux kernel version is 3.5.7. We implement the prover program as a loadable kernel module. In the prover kernel module, we implement the Ethernet communication function and the function to read and write NAND Flash partition contents. We implement the checksum function using ARM assembly in the prover kernel module.

The HP laptop has an Intel Quad-Core i5 CPU running at 2534 MHz, 4GB of RAM, and runs 32-bit Ubuntu 12.04 LTS as the guest OS. We implement a verifier program that runs on the HP laptop. The verifier program communicates with the prover kernel module on the Gumstix FireStorm COM through ICMP packets. A timer in the verifier program measures the time of one nonce-response pair using the *RDTSC* instructions.

3.5.1 Gumstix FireStorm COM

The Gumstix FireStorm COM is a TI DM3730-based platform, and it has 512M SDRAM, 64KB SRAM, and 512MB NAND Flash. The FireStorm COM is also equipped with a wireless module for WiFi and Bluetooth communications. We expand the FireStorm COM with a Gumstix Tobi expansion board[3] that expands the platform with external interfaces (e.g., 10/100 Ethernet and USB ports).

The TI DM3730 [107] Central Processing Unit has an ARM Cortex-A8 processor (MPU) with 1-GHz maximum frequency, and a High Performance Image Video Audio (IVA) subsystem. In the IVA subsystem, there is a TMS320C64x+ Digital Signal Processing (DPS) core (800-MHz) and a ARM9 core (200-MHz). The MPU and the IVA subsystem share the memory system. DM3730 supports firewalls to restrict accesses of SDRAM, SRAM and NAND Flash from other subsystems (e.g., IVA subsystem or the DMA controller). By leveraging the access

[2]https://www.yoctoproject.org/

[3]https://store.gumstix.com/tobi.html

control enforced by the firewalls, the MPU can prevent illicit accesses from other subsystems and only allows the MPU itself to access SDRAM, SRAM and Flash contents.

ARM Cortex-A8 The MPU subsystem has an ARM Cortex-A8 core [8, 9]. The MPU core has 32 entries of I-TLB and D-TLB respectively. TLBs use round-robin replacement policy, which is deterministic. Besides TLBs, MPU has two levels of caches. The first-level (L1) has 32KB I-cache (4-Way Set Associative) and 32KB D-cache (4-Way Set Associative), while the second level (L2) is a 256KB unified cache (8-Way Set Associative) for both instructions and data. Page table content in L1 D-cache is not available for MMU to read on ARM Cortex-A8. When TLB miss happens, MMU reads the L2 cache to get the page table contents. On Cortex A8, there is no hardware-support for cache coherence in L1 level caches. Caches use random replacement policy. The data cache (L1 data cache and unified L2 cache) could be configured as write-through, which allows any modification on the L1 data cache will be reflected on the L2 cache or SDRAM. The L1 I-cache is Virtually Indexed and Physically Tagged (VIPT) while the L1 D-cache and the L2 unified cache is Physically Indexed and Physically Tagged (PIPT). ARM Cortex A8 has a NEON coprocessor for Single-Instruction-Multiple-Data (SIMD) and floating point operations. The NEON coprocessor has 32 64-bit general purpose registers and share the same instruction Fetch unit with the ARM processor.

Device Registers On DM3730, there are more than 5000 readable and writable device control registers in memory-mapped register space, and those control registers could be used by an adversary to hold malicious code/data. To achieve a malware-free system state, we reset all device registers (with known values) immediately after the checksum computation. There are several control registers that

could subvert the untampered execution environment established on the MPU. In particular, the WDT could be configured to reset the embedded device once the pre-configured timer threshold is triggered. Moreover, the DMA could be configured to issue a DMA request to transfer malicious contents to the prover kernel module. To handle such control registers, the checksum function periodically calls the *Epilog* code to incorporate the WDT and firewall configurations in the checksum.

3.5.2 Checksum Function Implementation

We implement the checksum function as 512 checksum blocks and each checksum block has 128 ARM instructions that take 512B memory space. The checksum function is deployed on 4KB-aligned virtual memory and the 512 checksum blocks take 64 4KB memory pages (256KB memory). On the Gumstix FireStorm COM, accessing the SRAM is faster than accessing the SDRAM (our measurements shows that on the Gumstix FireStorm COM, a non-cached memory read over SRAM consumes 95 CPU cycles while a non-cached memory read over SDRAM consumes 135 CPU cycles when the CPU runs at 1GHz). To obtain the best performance, we deploy part of the checksum function on the 64KB SRAM[4]. In particular, 128 checksum blocks are deployed on the 64KB SRAM while 384 checksum blocks are deployed on the SDRAM.

Pseudo-Random Number Generator In the checksum function, we use a 32-bit T-function to build a Pseudo-Random Number Generator (PRNG) and construct the memory address to read using the PRNG outputs. Because T-function is a Pseudo-Random Permutation Function (PRPF), after every 2^{16} checksum itera-

[4]We also evaluated other configurations (e.g., deploying the frequently-accessed page table in the SRAM) and evaluation results show that our current implementation have the best performance compared with other configurations.

tions, the checksum function updates the seed to the T-function to build a PRNG from the T-function [35]. In particular, the checksum function saves a large number of Pseudo-Random Numbers (PRNs) in the memory before the checksum computation (the memory space saving the PRNs is measured by the checksum function during checksum computation) and then generates the seed to the T-function based on the T-function outputs, checksum states, and saved PRNs during checksum computation. In this way, the verifier program does not need to send new nonces (as the seed) to the prover program during checksum computation. Note that an adversary could not precompute the seed based on the saved PRNs because the T-function outputs and checksum states are incorporated with the saved PRNs to generate the new seed. In addition, the initial seed to the T-function is sent from the verifier and the prover program generates the PRNs using an Advanced Encryption Standard (AES)-based PRNG based on the random values sent from the verifier before the checksum computation (another way is that the verifier program sends all required PRNs to the prover device before the checksum computation).

Strongly-Ordered Checksum Function In each checksum block, the checksum function updates one 32-bit checksum state, out of a total 30 checksum states, using strongly-ordered *ADD*, *XOR*, and *SHIFT* operations. Each checksum block takes as input: other checksum states, the memory address being read (Data Pointer), memory contents, current processor status (i.e., the Current Processor Status Register (CPSR) value), Program Counter (PC), the pseudo-random numbers generated by a T-function, and a counter. We carefully order the checksum instructions and guarantee that they cannot be executed in parallel. In the checksum function, all 30 available ARM General Purpose Registers (GPRs) are used (2 GPRs in monitor mode are not available to access on ARM Cortex A8): 25 GPRs are used to save checksum states; r0 stores the pseudo-random value from T-function; r1, r2, and

r3 are used as temporary variables; r4 stores the counter value. Figure 3.10 shows the ARM assembly instructions that measure the VA space D in one checksum block. To prevent attackers from using NEON coprocessor to perform malicious operations, we save pseudo-random values in all 32 64-bit NEON GPRs. In each checksum block, all NEON GPR values are updated (based on the checksum states) and incorporated into the checksum.

$r12$ C	checksum state to update Carry flag
Assembly Instruction	Explanation
umull r2, r1, r0, r0	$tmp = PRN \times PRN$, T-function computation
orr r1, r2, #0x5	$tmp = tmp \mid 5$, T-function computation
add r0, r0, r1	$PRN = PRN + tmp$, T-function computation
lsr r1, r0, #12	$tmp = PRN >> 12$
and r2, r1, #0xFFFFFFFC	$addr\_VA\_D = tmp \ \& \ mask$
ldr r1, [r2]	$tmp = mem[addr\_VA\_D]$
eor r12, r12, r1	$r12 = r12 \oplus tmp$
adcs r12, r12, r11	$r12 = r12 + r11 + C, \ update \ C$
eor r12, r12, r0	$r12 = r12 \oplus PRN$
adcs r12, r12, r15	$r12 = r12 + PC + C, \ update \ C$
eor r12, r12, r13	$r12 = r12 \oplus r13$
adcs r12, r12, r2	$r12 = r12 + addr\_VA\_D + C, \ update \ C$

Figure 3.10: Assembly instructions that measures the VA space D in one checksum block. The checksum state is saved in $r12$ in this checksum block.

Memory Read Over VA Spaces As analyzed in Section 3.4, we could increase the overhead of attacks by increasing the number of memory read operations in the checksum function. Also, the overhead of Attack VIII is mainly caused by mali-

cious operations when the VA space *A* is measured. Thus, the checksum function performs multiple memory reads over the VA spaces in every checksum block with more memory read operations over the VA space *A* than other VA spaces. In particular, the checksum function performs five memory reads over the VA spaces *A*, *B*, *C*, or *D* for integrity measurement in every checksum block (total 2560 memory reads in the 512 checksum blocks). In the 2560 memory reads, the checksum function performs 2423 memory reads over the VA space *A* (256KB), one memory read over the VA space *B* (8KB), eight memory reads over the VA space *C* (64KB), and 128 memory reads over the VA space *D* (1MB) (for 128 checksum blocks, the checksum function performs one memory read over the VA space *D* and four memory reads over other VA spaces in every checksum block). In our implementation, the program program and other critical memory contents (e.g., page table, stack, communication buffer) take about 912KB memory. The spare memory in the 1MB VA space *D* are filled with PRNs before checksum computation.

Dynamic Page Table The memory contents to verify (including the checksum function, the communication function, hash function, stack, communication buffer, page table, exception handler table, and so on) are mapped in a 1MB virtual memory (256 4KB pages) for integrity measurement. The memory pages containing checksum function is also mapped to another 256KB virtual memory (64 4KB pages) for checksum execution. The checksum function dynamically updates the mappings of the 1MB virtual memory and the mappings of the 256 KB virtual memory (for checksum execution) separately. Between the five memory reads, the checksum function dynamically updates page table contents in each checksum block. In 128 checksum blocks (out of the 512 checksum blocks), the checksum function dynamically updates the VA-to-PA mappings of the 256KB virtual memory for checksum execution while in other 384 checksum blocks, the checksum

function dynamically updates the VA-to-PA mappings of the 1MB virtual memory for integrity measurement. The L1 D-cache is configured as write-through, so the updated mappings are available in L2 cache immediately. When TLB miss happens, the MMU reads the updated page table contents in L2 cache directly.

Dynamically Modified Instructions In additional, each checksum block contains 13 ARM instructions that are dynamically modified during checksum execution. After every 2^{11} checksum iterations (executing one checksum block is one checksum iteration), the checksum function modifies 13 ARM instructions in every checksum block, and invalidate all I-cache blocks. The 13 ARM instructions in every checksum block perform arithmetic computation with *SHIFT* operations. The checksum function incorporates 7 random bits to each instruction to update the number of bits to shift (5 random bits) and the way to shift (2 random bits) in the instruction. In our implementation, the L1 D-cache is configured as write-through, so the dynamically modified instructions are available in L2 cache after modification. After I-cache blocks are invalidated, the dynamically modified instructions in L2 cache are executed by the CPU.

Untampered Execution Environment The checksum function disables maskable interrupts by configuring the CPSR value and incorporating the CPSR value in the checksum in every checksum block. To prevent nonmaskable interrupts (e.g., undefined instruction exception or data access abort), all Save Processor Status Registers (SPSR) are used to save checksum states. When an exception or abort happens, the MPU (ARM Cortex A8) automatically overwrites the SPSR value of the exception handling mode with the CPSR value of the operating mode in which the exception happens, consequently invalidating the checksum state saved in the SPSR. Every checksum block randomly jumps to another checksum block after

checksum computation. In this way, we can force attackers to simulate I-TLB replacement in every checksum block in an I-cache inconsistency-based attack. the checksum function calls the *Epilog* code to incorporate critical platform configurations in the checksum or reconfigure critical control registers values in a random pattern (in one out of the 128 checksum blocks that perform memory reads over the VA space *D*, the checksum function checks the value of the memory address to read and calls the *Epilog* code when a specific memory range in the VA space *D* is measured). The implemented *Epilog* code first verifies the firewall configuration (to guarantee that the IVA subsystem and the DMA cannot access the SDRAM, SRAM, or NAND Flash), then verifies other critical configurations (e.g., WDT configuration, exception handler table base address, and page table base address).

Attestation on DM3730 During attestation processor, we only run the checksum function on the Cortex A8 processor (1-GHz). The DSP processor (800-MHz) or the ARM9 processor (200-MHz) in the IVA subsystem cannot tamper with the execution environment established on the Cortex A8 by writing malicious instructions to the main memory (SDRAM and SRAM) because the checksum function configures the firewall to prevent the IVA subsystem from accessing the main memory where the prover program is running and verifies the firewall configuration during the checksum computation. The DSP processor (800-MHz) or the ARM9 processor (200-MHz) in the IVA subsystem cannot compute the checksum for the Cortex A8 processor (1-GHz) in a shorter time than the Cortex A8 processor, because (1) the DSP and the ARM9 processors run in slower CPU frequencies than the Cortex A8 processor and they cannot compute the expected checksum in parallel; (3) the IVA subsystem has only a unified TLB for the DSP and the ARM9 processors, so a checksum function running on the DSP or the ARM9 processor has to simulate the I-TLB and D-TLB replacements (on Cortex A8) to compute the expected check-

sum causing high overhead; (4) a checksum function running on the DSP or the ARM9 processor has to change the firewall configuration (to access the memory contents to verify in main memory) and perform additional operations to forge the expected checksum result causing overhead. After a software-only root of trust has been established on the Cortex A8 processor, the prover program can reset the IVA subsystem to set the IVA subsystem in a known state (cleaning all malicious contents in the IVA subsystem).

3.6 Evaluation

We evaluated Mead and measured the overhead caused by possible attacks on a Gumstix FireStorm COM.

3.6.1 Attacks and Malicious Operations

Table 3.1 summarizes the attacks against Mead and corresponding malicious operations described in this chapter. In these attacks, Attack (VI-1) performs more operations than Attack (VI-2), causing a higher overhead than Attack (VI-2). Thus, we did not measure the overhead of Attack (VI-1) as Attack (VI-2) is a better choice for adversaries. In addition, Attack (V) performs an additional non-cached memory read operation for nearly every memory read operation in the checksum function, causing a much higher overhead than Attack (VIII), which performs additional non-cached memory read operations only when modified contents (a small portion of the prover program) are measured. Thus, we did not measure the overhead of Attack (V).

As shown in Table 3.1, the main malicious operations in other attacks (except Attacks (V) and (VI-1)) include simulating the I-TLB replacement by saving variables in spare memory space (when MMU is disabled) or by saving variables in

spare system control coprocessor registers, simulating both I-TLB and D-TLB replacements when MMU is enabled, handling dynamically modified instructions by performing additional memory write operations when MMU is disabled or by performing *read-decode-execute* operations, and operations in Attack (VIII) (checking every memory address to read and performing additional memory read only when modified contents are measured). We evaluated these attacks by measuring the overhead caused by these malicious operations.

3.6.2 Simulating I-TLB Replacement

In Attack (II) or (III), a malicious checksum function simulates the I-TLB replacement. In our evaluation, we focused on the overhead caused by the operations that simulate the I-TLB replacement and assumed that attackers already compressed the checksum function size to be smaller than the I-cache size and a compressed malicious checksum function runs in the I-cache (with spare I-cache blocks for malicious instructions). The compressed checksum function runs in a virtual memory with constant VAes. The I-TLB entries for the compressed checksum function are preserved. At the end of each checksum block, the malicious checksum function simulates the I-TLB replacement using the round-robin replacement policy and calculates the target address to jump.

Saving Variables in Spare Memory We implemented the malicious code that simulates the I-TLB replacements by saving variables in the spare memory for the checksum function. Our implementation of simulating the I-TLB replacement contains 63 ARM instructions. The implemented malicious code saves 32 simulated I-TLB entries (each entry contains a 32-bit PA) and 384-bit status bits for 64 VA-to-PA mappings of the checksum function pages in the spare memory. The status bits indicate if a VA-to-PA mapping is cached in I-TLB (1 bit for each mapping)

Table 3.1: Attacks against Mead and Malicious Operations.

Attacks	**Main Malicious Operations**
Attack (I)	Simulate I-TLB and D-TLB replacements (MMU is enabled) (Equation 3.6); Perform *read-decode-execute* to handle dynamically modified instructions
Attack (II)	Simulate I-TLB replacement (MMU is disabled) (Equation 3.9); Perform *read-decode-execute* to handle dynamically modified instructions
Attack (III)	Simulate I-TLB replacement using spare coprocessor registers to save variables (Equation 3.12); Perform *read-decode-execute* to handle dynamically modified instructions
Attack (IV)	Simulate I-TLB and D-TLB replacements (MMU is enabled) (Equation 3.15)
Attack (V-1)	Call coprocessor to perform VA-to-PA translation (Equation 3.18); Perform additional memory read (when MMU is disabled) when VA spaces A, B, D are measured (Equation 3.19)
Attack (V-2)	Perform additional memory read (when MMU is disabled) when VA spaces A, B, D are measured (Equation 3.19)
Attack (VI-1)	Perform additional memory write to update page table (MMU is disabled) (Equation 3.20); Handle dynamically modified instructions by additional write operations when MMU is disabled or by *read-decode-execute*
Attack (VI-2)	Handle dynamically modified instructions by additional write operations when MMU is disabled or by *read-decode-execute*
Attack (VII)	Simulate I-TLB and D-TLB replacements when MMU is enabled
Attack (VIII)	Check memory address to read and perform additional memory read (MMU is disabled) when modified contents are measured (Equation 3.22)

and also include the entry index number in I-TLB (5 bits for each mapping) if the mapping is cached in I-TLB. In total, the malicious code saves 1408-bit values $(1024 + 384)$ values in the spare memory.

For each memory read, the malicious code checks the status bits of the memory page to read. If the mapping is already in I-TLB, the malicious code reads the mapping value in the simulated I-TLB to get the corresponding PA. However, if the mapping is not in I-TLB, the malicious code reads the mapping in the page table to get the PA, updates the simulated I-TLB entries using the round-robin replacement policy, and then updates the status bits. To avoid changing the D-TLB replacement, the malicious checksum function first disables the MMU before simulating the I-TLB replacement, and then enables the MMU after the simulation. The measurement results are described in Section 3.6.6.

Saving Variables in Spare Coprocessor Registers As previously described, a malicious checksum function might be able to save the variable values for simulating the I-TLB replacement in spare coprocessor registers to avoid expensive non-cached memory operations. On the ARM Cortex-A8 processor, we found 14 spare system control coprocessor (cp15) registers that could be used to save variable values for the simulation. The spare cp15 registers are mainly used to save fault status information or context ID, whose values will not affect the processor status. In these 14 32-bit cp15 registers, 77 bits are reserved (cannot be used to save values), and only 371 bits can be used to save variable values. As a result, we could only save part of the variable bits (e.g., part of status bits) in the spare cp15 registers. If accessing a spare cp15 register is faster than a non-cached memory read or write, we could speed up the simulation.

We first measured the time for a non-cached read or write operation, the time for reading or writing a spare cp15 register, and the time for enabling and enabling

MMU. Our measurement results show that on the Cortex-A8 processor (1GHz) on a Gumstix FireStorm COM, a non-cached read operation (over SDRAM) consumes about 135 CPU cycles; meanwhile, a non-cached write operation consumes the same amount of CPU cycles. However, reading a spare cp15 register consumes about 30 CPU cycles on average[5], and writing a spare cp15 register consumes about 38 CPU cycles on average[6]. Thus, using the cp15 registers to save part of variable values will reduce the overhead. Our measurement results also show that disabling and then enabling MMU on the Cortex-A8 processor (1GHz) consumes about 113.6 CPU cycles.

In our simulation algorithm, simulated I-TLB entries are accessed less frequently than status bits. When the mapping is in I-TLB, the malicious code performs one read on the status bits and one read on the simulated I-TLB entries, disables and enables MMU one time; when the mapping is not in I-TLB, the malicious code performs one read and two writes on the status bits, one read and one write on the simulated I-TLB entries, and one read on the page table, and disables and enables MMU one time. Thus, it is reasonable to use the spare cp15 registers to save the status bits to reduce the overhead.

We denote the time of one non-cached read or write operations as T_{mem}, the time of reading a spare cp15 register as T^{r}_{cp15}, the time of writing a spare cp15 register as T^{w}_{cp15}, the time of disabling and then enabling MMU one time as T_{mmu}, and the average time to simulate one I-TLB replacement as T^{simu}_{ITLB}. The checksum function executes the checksum blocks in 64 memory pages by random jumps (using the same probability). The I-TLB has 32 entries, so the I-TLB miss rate should be close to 50%. Ignoring other additional operations, when the status bits are saved

[5]Reading the 14 cp15 registers consumes about 420 CPU cycles in total

[6]Writing the 14 cp15 registers consumes about 532 CPU cycles in total

in spare cp15 registers, the average time to simulate one I-TLB replacement is

$$T_{ITLB}^{simu} = 0.5 \times (T_{cp15}^{r} + T_{mem} + T_{mmu}) + 0.5 \times (T_{mem} \times 3 + T_{cp15}^{w} \times 2 + T_{cp15}^{r} + T_{mmu}) \tag{3.23}$$

$$T_{ITLB}^{simu} = T_{cp15}^{r} + T_{cp15}^{w} + 2 \times T_{mem} + T_{mmu} \tag{3.24}$$

When all variables are saved in spare memory space, the average time is

$$T_{ITLB}^{simu} = 4 \times T_{mem} + T_{mmu} \tag{3.25}$$

Although we did not implement the malicious code that simulates the I-TLB replacement by saving status bits in cp15 coprocessors, based on the above equations, we can estimate that—when using cp15 registers to save status bits—the overhead to simulate I-TLB replacement can be reduced by about 30.9%[7].

3.6.3 Simulating I-TLB and D-TLB Replacements

In Attacks (I), (IV), and (VII), the malicious checksum function simulates both I-TLB and D-TLB replacements. We implemented the malicious code to simulate both I-TLB and D-TLB replacements in the checksum function. In every checksum block, the malicious code simulates the D-TLB replacement for every memory read operation to compute the legitimate memory address to read. Our implementation of simulating D-TLB replacement contains 82 ARM instructions. In addition, at the end of every checksum block, the malicious code simulates the I-TLB replacement to compute the legitimate target address to jump. Our implementation of simulating I-TLB replacement contains 55 ARM instructions (this implementation contains less instructions than the implementation in Section 3.6.2 because it does not disable or enable MMU). Note that, when the D-TLB replacement is

[7] $1 - \frac{30+38+2\times135+113.6}{4\times135+113.6} = 0.309$

simulated, the malicious checksum function does not need to disable MMU when performing additional memory operations. The measurement results are described in Section 3.6.6.

3.6.4 Handling Dynamically Modified Instructions

In Attacks (I), (II), (III), and (VI), a malicious checksum function needs to handle the dynamically modified instructions to compute the correct checksum results. There are two ways to compute the correct checksum (as detailed in Section 3.4.3): (1) performing additional write operations to update the malicious checksum function and (2) performing *read-decode-execution* operations. When performing additional write operations, the malicious checksum function disables MMU to preserve D-TLB entries. In the *read-decode-execute* operation, a malicious checksum function can save the random bits in spare memory or in spare cp15 registers.

We implemented the malicious code that performs additional write operations to dynamically update the instructions in the malicious checksum function (the malicious checksum function is saved on the correct memory location). In our implementation, to update each instruction, the malicious code disables the MMU, performs one memory write operation (write the updated instruction to the malicious checksum function), and then enables the MMU. Thus, to update one checksum block, the malicious code performs 13 additional memory write operations and also disables and enables MMU thirteen times. The measurement results are described in Section 3.6.6.

In *read-decode-execute* operations, the malicious code has to save 91 random bits (13×7 bits) in spare memory space or in spare cp15 registers for every checksum block. For all 512 checksum blocks, the malicious code needs to save $46,592$

random bits. Thus, only a small portion of the random bits (0.8%[8]) can be saved in the 14 spare cp15 registers that can save 371-bit values. When saving the random bits in spare memory, the malicious code disables MMU, performs 3 memory write operations (saves random bits in memory), and then enables MMU. During the checksum execution, to read the random bits, the malicious checksum function disables MMU, performs 3 memory read operations (to get the random bits), enables MMU, and then decodes and executes the instructions correctly. The current checksum function has 512 checksum blocks and the checksum instructions are dynamically updated after every 2048 checksum iterations (one checksum iteration is the execution of one checksum block). Thus, each checksum block is executed four times on average before the checksum instructions are dynamically updated again. Therefore, before the instructions are updated again, the malicious checksum function needs to perform 3 additional memory write operations and 12 additional memory read operations for every checksum block on average (15 additional memory operations total). The malicious checksum function also needs to disable and enable MMU five times for every checksum block on average. Note that 64KB checksum function are deployed on the SRAM while 192KB checksum function are deployed on the SDRAM. A non-cached memory write over the SRAM is faster than a non-cached memory write over the SDRAM. However, in Attacks I, II, and III, the malicious checksum function cannot save random bits on the SRAM because the data memory (data cache, SRAM, and SDRAM) should not contain any modified contents in these attacks (Section 3.3.2). In Attack VI-2, the malicious checksum function can save all random bits in the SRAM, reducing the overhead of *read-decode-execute* operations (in this attack, attackers have to move several checksum blocks from the SRAM to SDRAM to get spare memory space

[8] $0.008 = \frac{371}{46592}$

on the SRAM for saving the random bits).

Our measurements show that disabling and then enabling MMU on an ARM Cortex-A8 processor (1 GHz) consumes about 113.6 CPU cycles while a non-cached read or write operation over the SDRAM consumes about 135 CPU cycles. A non-cached read or write operation over the SRAM consumes about 95 CPU cycles. Ignoring the operations to decode the random bits and then execute the instructions correctly, when random bits are saved in the SDRAM (Attacks I, II and III) the *read-decode-execute* operation could reduce the overhead by 16.4%[9], compared with performing additional writes to update the malicious checksum function. When random bits are saved in the SRAM (Attacks VI-2) the *read-decode-execute* operation could reduce the overhead by 35.7%[10], compared with performing additional writes to update the malicious checksum function. Note that we did not implement the malicious code to perform *read-decode-execute* operations. In addition, the spare cp15 registers can be used to save random bits to speed up the *read-decode-execute* operations. However, as the spare cp15 registers can only save 0.8% of the random bits, we ignored the spare cp15 registers in the computation.

3.6.5 Memory Substitution Attacks

We implemented the malicious operations in Attack (VIII) and measured the overhead of this attack. To avoid changing the I-TLB replacements, we kept the malicious code for each checksum block in the same memory page with the corresponding checksum block. To get sufficient spare memory space for malicious operations, we reduced the size of three checksum blocks (out of the eight checksum blocks) in every checksum function memory page by reducing the duplicated

[9] $1-\frac{15\times135+5\times113.6}{(13\times135+13\times113.6)\times0.75+(13\times95+13\times113.6)\times0.25}=0.164$

[10] $1-\frac{15\times95+5\times113.6}{(13\times135+13\times113.6)\times0.75+(13\times95+13\times113.6)\times0.25}=0.357$

T-function instructions. The implemented malicious code checks every memory address to read and reads the correct copy (an additional memory read) if malicious contents are measured. To avoid changing the D-TLB replacement, the malicious code disables MMU before performing the additional memory read and enables MMU after the memory read.

In every checksum block, the memory read instructions are replaced with jump instructions that jump to the malicious code. In every checksum function memory page, when the memory space that contains the three compressed checksum blocks is read, the malicious code performs an additional memory read to get the correct values; when the other five checksum blocks are read, the malicious code reads the memory, checks the returned value, and replaces it with the value of the memory read instruction if the returned value is a jump instruction. In this attack, the malicious checksum function saves the PAes of the memory pages containing the correct copy in a constant offset of the corresponding checksum function memory page (Section 3.4.3). Because a non-cached read over the SRAM is faster than a non-cached read over the SDRAM, the malicious checksum function might be able to reduce the overhead by saving a correct copy of the modified contents on the SRAM (running the malicious checksum function on the SDRAM). Therefore, we implemented this attack in two ways and measured the overheads of both implementations. In the first approach (Attack VIII Implementation A), we ran the malicious checksum function on the original memory location (64KB of the checksum function are on the SRAM) and saved the correct values of modified contents on the SDRAM. In the second approach (Attack VIII Implementation B), we ran the malicious checksum function on the SDRAM, but saved 64KB (out of 96KB) of the correct values of modified contents on the SRAM (other 32KB were saved on the SDRAM). The measurement results are described in Section 3.6.6.

3.6.6 Evaluation Results

Counter Value Based on the result of the Coupon Collector's Problem [42], the minimal counter value (the number of checksum iterations to run) to measure all memory locations in the VA spaces A, B, C, and D is $0xc80000$. We set the counter value as $0x2000000$ (approximately three times of $0xc80000$) in all measurements against two-part checksum computation attacks (see Section 2.1.3).

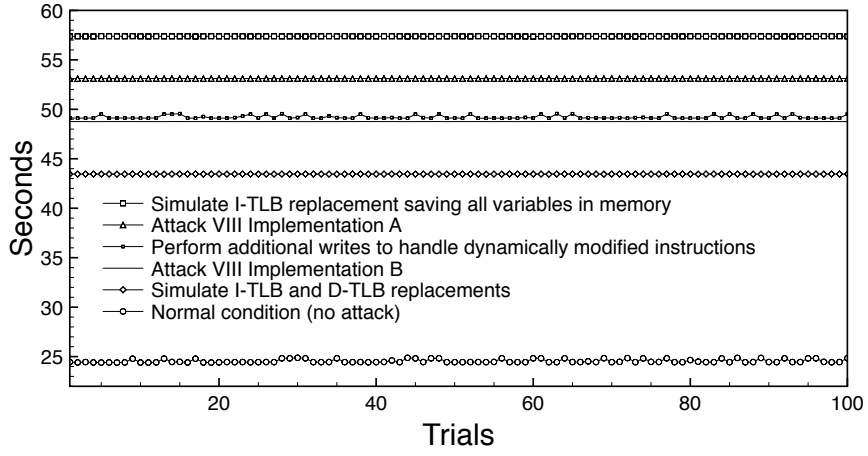

Figure 3.11: Measured time of a single nonce-response pair in seconds (iteration counter is $0x2000000$). The malicious checksum function performs malicious operations in all $0x2000000$ iterations.

Normal Conditions On normal conditions without attacks, the average latency of a single nonce-response pair measured by the verifier program over 100 trials is 24.55 seconds (standard deviation is 0.18 seconds). We also measured the time of a single nonce-response pair when the *Epilog* code is skipped (simulating the attacks in which attackers configure the memory-mapped register space as cacheable described in Section 3.4.1). The measurements show that the overhead caused by the *Epilog* code is negligible. In addition, in Attacks (I), (II), and (III), because

the malicious checksum function runs inside the I-cache without I-cache misses or I-TLB misses, the I-cache misses and I-TLB misses are avoided in these attacks. To estimate the time reduced by avoiding I-cache misses and I-TLB misses, we ran a modified checksum function that contains only 64 checksum blocks (32KB). The modified checksum function runs in the 32KB I-cache and uses only 8 (out of 32) I-TLB entries. The average latency of a single nonce-response pair over 100 trials is 21.33 seconds (standard deviation is 0.33 seconds) (reducing the checksum execution time by 13%).

Measurement Results The measurement results (Figure 3.11) show that, in Attack VIII Implementation A (Section 3.6.5), the average latency of a nonce-response pair over 100 trials is 53.11 seconds (standard deviation is 0.003 seconds and the caused overhead is 116% of the average latency on normal conditions). Meanwhile, in Attack VIII Implementation B, the average latency of a nonce-response pair over 100 trials is 48.76 seconds (standard deviation is 0.003 seconds and the caused overhead is 98.6% of the average latency on normal conditions). Thus, Attack VIII Implementation B demonstrated better performance than Attack VIII Implementation A. The Attack VIII measurement results showed a smaller standard deviation than the measurement results for normal conditions, because Attack VIII performed a large number of non-cached memory read operations, causing less cache misses than normal conditions. When the malicious checksum function performs additional memory writes to handle the dynamically modified instructions, the average latency of a nonce-response pair over 100 trials is 49.21 seconds (standard deviation is 0.17 seconds). The caused overhead is 100.5% of the average latency on normal conditions. Based on the analysis in Section 3.6.4, the overhead caused by the *read-decode-execute* operations would be about 84.0%[11]

[11] $100.5\% \times (1 - 16.4\%) = 84.0\%$

(when random bits are saved in the SDRAM) and about 64.6%[12] (when random bits are saved in the SRAM). When the malicious checksum function simulates the I-TLB replacement by saving the variables in the spare memory (on the SDRAM) when MMU is disabled, the average latency of a nonce-response pair is 57.38 seconds (standard deviation is 0.007 seconds). The caused overhead is 133.7% of the average latency on normal conditions. Based on the analysis in Section 3.6.2, the overhead caused by simulating the I-TLB replacement when using spare cp15 registers to save status bits would be about 92.4%[13]. The average latency of a nonce-response pair when the malicious checksum function simulates both I-TLB and D-TLB replacements (when MMU is enabled) is 43.45 seconds (standard deviation is 0.004 seconds and the caused overhead is 77.0% of the average latency on normal conditions), which is less than the latency when simulating only the I-TLB replacement when the MMU is disabled. The reason for this is that, when the MMU is disabled, the non-cached memory operations in the I-TLB simulation consume a large number of CPU cycles, significantly slowing down the simulation operations. The measurement results when the checksum function simulates TLB replacements showed a smaller standard deviation than the measurement results for normal conditions. The reason for this is that, when simulating TLB replacements the checksum execution causes less TLB misses than the checksum execution on normal conditions.

Attack Overhead Based on the measurement results, we analyze the overhead of the attacks summarized in Table 3.1 (except Attacks V and VI-1). Note that we did not evaluate Attacks V or VI-1 because they cause higher overhead than other attacks (see Section 3.6.1). Figure 3.12 shows the overheads caused by these

[12] $100.5\% \times (1-35.7\%) = 64.6\%$
[13] $133.7\% \times (1-30.9\%) = 92.4\%$

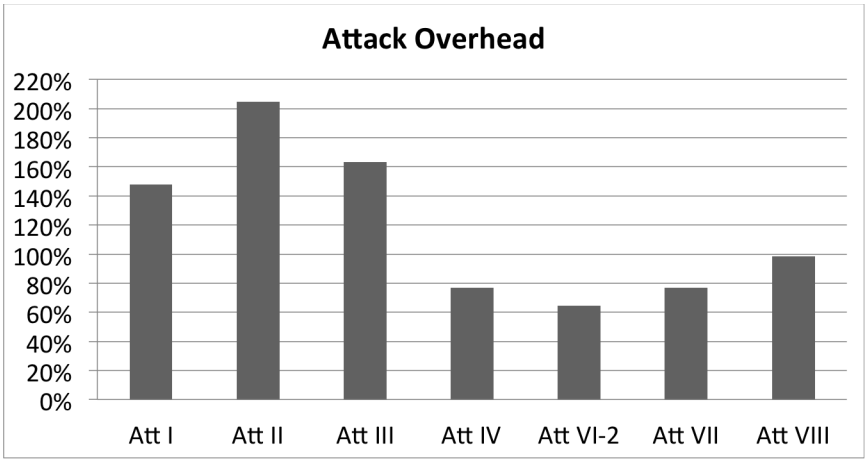

Figure 3.12: Attack Overhead.

attacks.

In Attacks I, II, and III, although the malicious checksum function can obtain time compensation by avoiding I-cache and I-TLB misses (reducing the execution time of the original checksum function by 13%), it needs to not only simulate the TLB replacements, but also perform *read-decode-execute* operations to handle the dynamically modified instructions (saving random bits on the SDRAM). We compute the overheads of the Attacks I, II and III by combining the overhead caused by simulating TLB replacement and the overhead caused by *read-decode-execute* operations; we then subtract the compensation time reduced by avoiding I-cache and I-TLB misses. Note that in Attacks II and III, the malicious checksum function cannot save the variables for simulating the I-TLB replacements on the SRAM, because the SRAM should not contain any malicious contents in these attacks. The results shows that Attacks I, II, and III cause high overheads (148.0%, 205.0%, and 163.0% of the average latency on normal conditions, respectively). In Attack VI-2, we assume that the malicious checksum function saves the random bits (for dynamically modified instructions) on the SRAM and performs *read-decode-*

execute operations to handle the dynamically modified instructions. The overhead in Attack VI-2 could be 64.6% of the average latency on normal conditions. Attacks IV and VII performs additional operations to simulate I-TLB and D-TLB replacements (the overhead is 77.0% of the average latency on normal conditions).

In a two-part checksum computation attack, the malicious checksum function could perform malicious operations in only 0x1380000 ($0x2000000 - 0xc80000$) iterations. Thus, if an adversary also performs a two-part checksum computation attack with these attacks, the attack overhead would be reduced by 39.1%[14]. For Attack VI-2 with two-part checksum computation operations, the overhead caused by malicious operations would be about 39.3%[15] of the average latency on normal conditions. To detect this attack, we could set a detection threshold (e.g., 15.0% of the average latency on normal conditions) that is much lower than 39.3%, but still higher than the possible measured time variance (e.g., 3% to 8%).

Note that the overheads shown in Figure 3.12 are caused by the main malicious operations in these attacks. Real attacks require additional operations. For example, to successfully perform a *read-decode-execute* operation, a malicious checksum function needs to perform the decoding and execution operations, which are ignored in our measurements. In addition, to successfully perform Attack VI-2, a malicious checksum function also needs to perform the operations presented in Equation 3.21. We ignored these operations in our measurements. Furthermore, we can increase the overhead of these attacks by increasing the number of dynamically modified instructions in every checksum block (forcing attackers to perform more non-cached memory operations) or increasing the number of memory read operations or the number of random jumps (forcing attackers to simulate TLB replacement more frequently). Although our current implementation for simulating

[14] $\frac{0xc80000}{0x2000000} = 39.1\%$

[15] $64.6\% \times (1 - 39.1\%) = 39.3\%$

TLB replacements is not optimal, it would be extremely challenging for attackers to significantly reduce the attack overhead to be lower than the detection threshold. For example, our current implementation for simulating both I-TLB and D-TLB replacements includes 82 ARM instructions. To reduce the overhead to be lower than 15.0%, an adversary might have to simulate the TLB replacements by less than 27 ARM instructions, which could be extremely challenging. The measurement results indicate that the dynamic page table mechanism significantly increases the overhead of possible attacks.

3.7 Discussion

Measured Time Variance The measured time variance might be caused by the CPU clock variance, the cache misses, or the TLB misses. The dynamic page table mechanism dynamically updates the TLB entries and might cause a high TLB miss rate, consequently increasing the measured time variance. In addition, the cache misses might increase the measured time variance. To overcome the variance caused by the TLB misses and the cache misses, the verifier program can run the integrity measurement with the same nonce and the same checksum iteration number on a known malware-free device that has the same configuration a the prover device. With the same nonce and the same checksum iteration number, the checksum executions on identical devices are the same (the same TLB misses and cache misses). In this way the verifier program can use the measured time of the nonce-response reception on the known malware-free device as the baseline for configuring the timing-detection threshold for the prover device, thereby overcoming the variances caused by the TLB misses and cache misses. However, we cannot avoid the variance caused by the CPU clock, which is due to the hardware limitation.

Optimal Attack Implementation When the dynamic page table mechanism is applied, the malicious checksum function might simulate the I-TLB or D-TLB replacements (or both) to compute the expected checksum. However, it is difficult to evaluate the minimal overhead caused by the TLB replacement because it is extremely challenging to have an optimal implementation of simulating the TLB replacement. We cannot guarantee that attackers cannot have a faster implementation that simulates the TLB replacement. However, we might be able to analyze the simulation procedure, summarize the minimal operations (e.g., disable and enable the MMU, read the mapping values from the simulated TLB, and update the TLB), and then evaluate the overhead of the minimal operations. In future work, we will analyze the attacks, estimate the minimal operations attackers are forced to perform, and evaluate the overhead of the minimal operations.

Malware in Peripherals A modern commodity embedded platform might also have peripherals with firmware running in them. Malware in peripherals may have access to the main memory and be able to inject malicious code in the main memory to break the execution of the prover program without being detected. On DM3730, the firewall feature can prevent peripherals from accessing the main memory (e.g., SDRAM or SRAM). Thus, the checksum function running on the MPU can configure the firewall to prevent peripherals from accessing the main memory (assuming that the MPU is faster than peripherals), then establish an untampered execution environments in main memory (a software-based root of trust). After a software-based root of trust been established on the MPU, the verifier program can verify the integrity of peripherals' firmware using the mechanisms described in Chapter 4.

Optimal Checksum Implementation Existing software-based attestation schemes require that the checksum implementation be optimal. If an adversary could optimize the checksum implementation (e.g., by reducing one instruction), the adversary might obtain time compensation for malicious operations and then run a malicious checksum function within the attacking detection threshold without being detected. One observation is that in the checksum function, memory read instructions (for integrity measurement) consume more CPU cycles than arithmetic instructions (on the ARM Cortex-A8, one memory read over SDRAM with cache miss consumes more than one hundred CPU cycles while an arithmetic instruction consumes only one CPU cycle). However, it is extremely challenging for the adversary to reduce the number of memory read operations (for integrity measurement) in the checksum function. Optimizing the checksum function by reducing several arithmetic instructions would not significantly reduce the checksum execution time (the time compensation is limited). In Mead, the dynamic page table mechanism significantly increase the overhead of malicious operations. If an adversary could optimize the checksum function by reducing only several arithmetic instructions, the time compensation would not be enough to protect malicious operations for being detected.

3.8 Summary

Nowadays, embedded platforms are used pervasively and consequently are becoming targets of software attacks [57]. Approaches to establish malware-free state on embedded platforms are necessary. However, hardware-based approaches would increase the cost of embedded platforms. In Mead, we propose novel software-based approaches to establish a software-only root of trust for integrity verification on modern commodity embedded platforms. The proposed approaches pre-

vent software attacks and we anticipate that the proposed approaches will make software-based attestation practical on current commodity embedded platforms.

Chapter 4

VIPER: Verifying the Integrity of Peripherals' Firmware

Malware on peripherals' firmware is becoming a popular trend for next-generation malware. In 2008, Triulzi demonstrated how to exploit a vulnerability in a Broadcom Tigon Network Interface Card (NIC), and inject malware into the NIC to eavesdrop on all traffic [109]. Triulzi also showed that the malware on the NIC can deploy malicious code into the GPU, causing the GPU to store and analyze the data sent through the NIC [110]. In 2009, Chen exploited a vulnerability in the Apple keyboard firmware update tool, which enables attackers to inject malicious code into the firmware of an Apple Aluminum Keyboard during firmware update [22]. In 2010, a buffer overflow vulnerability in a Broadcom NIC firmware was published [29], through which a remote attacker can compromise the NIC firmware by sending malicious packets to this NIC, then execute arbitrary code on the NIC.

On modern commodity computers, a peripheral might have a dedicated microcontroller and dedicated internal non-volatile memory (e.g., NAND Flash) or

volatile memory (e.g., SDRAM), making the peripheral essentially a separate system that cannot be fully accessed or controlled by the main CPU. Since the main CPU cannot access peripherals' internal memory, it is considerably challenging for any antivirus programs running on the main CPU to detect malware in peripherals. In addition, malware in a peripheral can be persistent (stored in non-volatile memory) and survive after repeated reboots. Even worse, malware in a peripheral might be able to perform arbitrary read or write over the entire memory space (including the main memory containing the operating system) through the Direct Memory Access (DMA). Consequently, the malware can compromise users' privacy and safety, such as eavesdropping a user's bank account password or credit card number, or embedding a kernel-level rootkit into a clean re-installed operating system. Stewin [104] demonstrates the capabilities of a peripheral-based malware that analyzes the contents in the main memory to obtain security-sensitive information (e.g., disk encryption password).

At first glance, software-based attestation [94, 96] may provide an approach for verifying the integrity of firmware. Device vendors could embed an attestation function in their firmware. Driver code executing on the main CPU could query the attestation function to verify firmware integrity. The advantages of software-based attestation are that no costly hardware changes are needed, and that the OS can validate firmware integrity (e.g., as a standard part of device driver initialization). Unfortunately, previously proposed approaches for software-based attestation have several shortcomings that preclude applicability in this context. The most serious shortcoming is a *proxy attack*, in which a queried device contacts a faster device (the proxy) to compute the correct answer to the time-sensitive checksum computation, which enables malware on the device to go undetected.

Figure 4.1 demonstrates a proxy attack. Peripherals, such as a NIC, can communicate with a remote proxy server to compute the expected answer. Also, faster

Figure 4.1: A Proxy Attack.

peripherals can work as a proxy server to compute correct answers for slower peripherals in the face of previous software-based attestation mechanisms.

Thanks to several new approaches, we improve software-based attestation for devices and bring these approaches into the realm of practicality. In fact, we leverage intricacies of the system buses to create a software-based attestation function that prevents proxy attacks and dramatically increases the time overhead that malicious code exhibits. More specifically, we propose to verify the peripheral firmware integrity on a modern computer system, and propose a software-only primitive, Verifying Integrity of PERipherals (VIPER). In the spirit of software-based attestation, VIPER is based on a timed challenge-response protocol between the host CPU and each peripheral. Our attestation protocols can detect all known software-based attacks on peripherals.

This chapter makes the following contributions:

1. We propose a software-only primitive, VIPER, to verify the integrity of peripheral devices' firmware.

2. We propose novel attestation protocols that prevent all known software-only attacks. Specifically, our attestation protocols can prevent a proxy attack that would have been successful against previous software-based attestation mechanisms.

3. We evaluate VIPER on a Netgear GA620 network adapter in an off-the-shelf computer and on an Apple Aluminum keyboard in an Apple Macbook

laptop, and also implement an Ethernet-based proxy attack. Our evaluation shows that VIPER can efficiently verify the integrity of peripherals' firmware.

4.1 Problem Definition

Problem Definition In today's computer systems, all peripheral devices with firmware, such as network adapters, USB and disk controllers, and even the BIOS, are at risk from computer malware. Verifying the integrity of these components' firmware, and guaranteeing the absence of malware, is the main problem we address in this chapter.

Assumptions Our focus is in protecting peripherals from network-based threats. Attacks where an attacker physically accesses the target device to change its hardware configuration (e.g., over-clocking peripherals' CPUs or increasing their memory) are out of scope. We assume that the verification program on the host CPU is correct, and that the operating system on the host CPU is secure and trustworthy during verification. While this is a strong assumption [84], recent work in OS-level security and trusted computing may in fact provide a reasonable platform from which to attempt peripheral device verification [11, 67, 113]. We also require that the verifier program on the host CPU has been configured with sufficient information about peripheral devices installed in a computer system, i.e., the verifier knows what is *supposed* to be there.

Attacker Model The attacker may compromise firmware executing inside peripheral devices. The attacker may also control remote machines that may assist a compromised device in responding to challenges. This machine may have con-

siderable computation and memory resources, though the attacker is still unable to break standard cryptographic primitives [72]. However, we assume practical communication constraints, such as the bandwidth and latency characteristics of PCI [68, 69] and Gigabit Ethernet.

4.2 System Design

We describe the VIPER system architecture, attestation protocols, and checksum function.

4.2.1 VIPER Overview

VIPER is a software-only solution to verify the integrity of peripherals' firmware using a timed challenge-response protocol between the host CPU and peripherals.

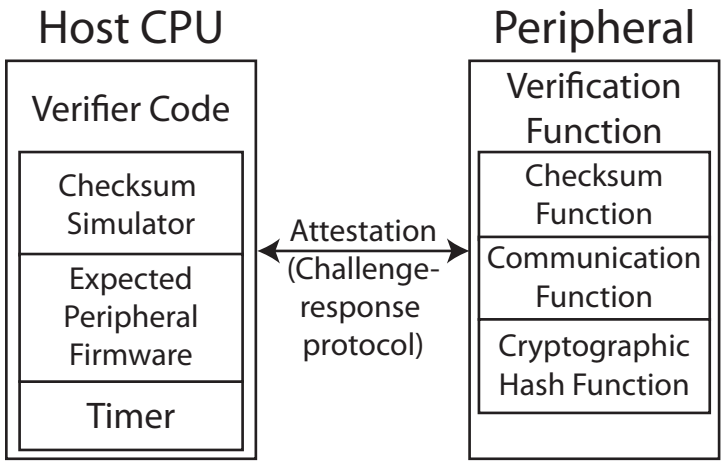

Figure 4.2: VIPER System Architecture.

System Architecture In VIPER (Figure 4.2), a verifier program executes on the host CPU and performs the verification procedure over all peripherals one-by-one

on a computer system. The verifier program has correct copies of all peripheral firmware (e.g., bundled with device drivers) in the computer. A checksum simulator in the verifier program generates challenges (cryptographic nonces) and the corresponding expected responses by simulating the verification procedure over the correct copies of peripherals' firmware. A timer is used to measure the time of the verification procedure from inside the verifier program ("Verifier Code" in Figure 4.2). On each peripheral device, a verification function comprised of three main parts engages in the VIPER verification protocol to set up an untampered execution environment and compute a special checksum function over the contents of the verification function's components (the checksum function itself, a communication function, and a cryptographic hash function). The checksum function is carefully designed to offer optimal performance. Any malicious code or operations during verification either invalidate the checksum result, or cause a detectable delay in the verification function's response. When the checksum computation finishes, the checksum function invokes the hash function over the entire memory contents of the peripheral. By verifying the checksum result and the computation time, the verifier program obtains the guarantee that an untampered execution environment has been set up inside the peripheral device, and that the subsequent computation of the complete hash of the peripheral's firmware is trustworthy.

Full System Verification In VIPER, the host CPU verifies the firmware integrity of all peripherals one-by-one. However, a faster peripheral on the motherboard can work as a proxy helper for a slower peripheral. Consider a resource-impoverished device such as a keyboard. Such devices are likely equipped with 8-bit microcontrollers running at a few tens of MHz. The computational latency imposed by running a checksum algorithm on such devices may actually be large enough to cover up the communication latency induced by forwarding nonces and responses

to a faster malicious device elsewhere in the system or even on an external system.

The solution for verifying a device with a particular level of computational capability is that all devices with greater capabilities must be verified first. For example, to verify a slow 8-bit microcontroller, all high-speed peripheral devices (e.g., NIC, SATA controller, GPU, USB 3.0) must first be verified. After the attestation of a faster peripheral, the verification function on the faster peripheral continues running until all peripherals have been verified. In this way, VIPER can prevent the faster peripheral from being compromised during the time interval between initial verification of the faster peripheral and completion of the verification of all peripherals. Thus, the verifier program on the host CPU can conclude that the devices capable of masquerading as the weak device are all benign, and will not interfere with the verification process.

Verification Procedure We now detail the verification procedure for a single peripheral.

1. The verifier program calls the checksum simulator to generate nonces, and expected checksum and hash results by simulating the verification procedure.

2. The verifier program sends an attestation request to the peripheral. The checksum function on the peripheral resets the peripheral into a known-good state.

3. The verifier program starts a timer, and begins to perform the attestation by sending the nonces generated by the checksum simulator to the target peripheral over the system's bus (Section 4.2.2).

4. After receiving the nonces, the verification function executing inside the peripheral sets up an untampered execution environment, performs the checksum computation, and sends the result back to the verifier program on the

host CPU (note that the nonces are used to initialize the initial checksum states and other registers. The verification function cannot start the checksum computation before receiving the nonces). The verification function then calls the hash function to compute a hash over the full memory contents of the target peripheral.

5. The verifier program confirms that the checksum results are correct and timely.

6. The verification function on the peripheral sends the hash result to the verifier program.

7. The verifier program validates the hash result.

4.2.2 Attestation Protocol

Though an on-board proxy attack can be prevented or detected as described in Section 4.2.1, it is a challenge to detect a remote proxy attack. A network-enabled peripheral device can communicate with a remote proxy helper through its network interface. Also, the network-enabled peripheral can work as a communication medium in a hybrid proxy attack, e.g., when a USB peripheral is being verified, a NIC may help the USB peripheral to contact a remote proxy helper, even if the NIC's CPU is slower than the USB peripheral's. In this section, we propose novel attestation protocols that detect such remote proxy attacks.

Latency-based Attestation Protocol

In a proxy attack, the peripheral to be verified always incurs some latency to communicate with a proxy helper. If the checksum computation time is well-controlled and smaller than the minimal communication latency between the peripheral and a

proxy helper, the additional latency caused by the proxy attack is detectable, even if the proxy helper has infinitely fast computation resources. In this section, we detail a latency-based attestation protocol based on these observations. Also, we describe a technique to increase the communication overhead between a peripheral and a proxy helper, and a technique to accelerate the attestation procedure by synchronizing the host CPU and peripheral. Figure 4.3 shows the time line of one challenge-response pair in a latency-based attestation protocol, including both the normal computation, and the proxy attack.

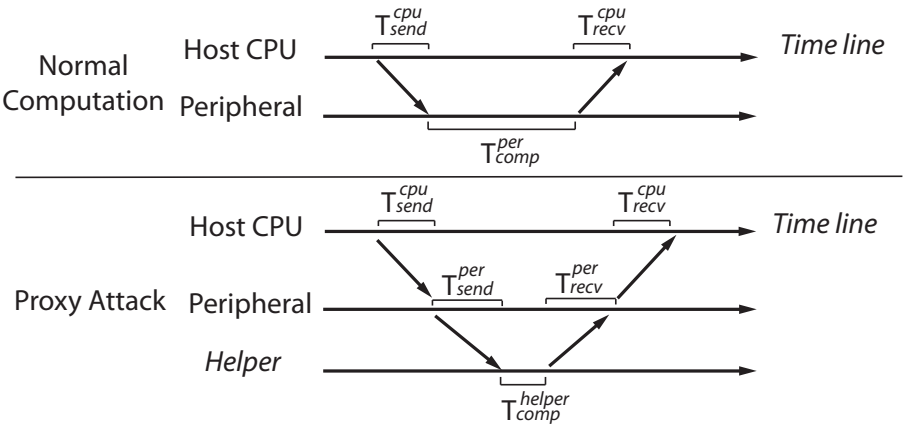

Figure 4.3: One challenge-response pair for latency-based attestation under both normal computation and a proxy attack.

Under normal conditions, the host CPU sends a challenge to the peripheral requiring time T^{cpu}_{send}, and the checksum computation consumes time T^{per}_{comp}. After checksum computation, communication consumes time T^{cpu}_{recv} to send the checksum back to the host CPU. Thus, the time of one challenge-response pair is:

$$T^{normal}_{comp} = T^{cpu}_{send} + T^{per}_{comp} + T^{cpu}_{recv} \tag{4.1}$$

In a proxy attack, the peripheral forwards the challenge sent by the host CPU to a proxy helper, which consumes time T^{per}_{send}. The remote helper consumes T^{helper}_{comp}

to compute the correct checksum, and then takes T_{recv}^{per} to send the result back to the peripheral. Thus, in the proxy attack, the time of each challenge-response pair is:

$$T_{comp}^{proxy} = T_{send}^{cpu} + T_{send}^{per} + T_{comp}^{helper} + T_{recv}^{per} + T_{recv}^{cpu} \tag{4.2}$$

We assume that T_{comp}^{helper} is zero because we conservatively assume that the remote helper has massive computational and memory resources available. Then, the overhead caused by a proxy attack is:

$$T_{overhead}^{proxy} = T_{send}^{per} + T_{recv}^{per} - T_{comp}^{per} \tag{4.3}$$

To detect a proxy attack, $T_{overhead}^{proxy}$ must be positive and detectable. Therefore, T_{comp}^{per} must be well-controlled to guarantee that T_{comp}^{per} is smaller than the minimal detectable proxy overhead. Note that modern network interfaces can have extremely low latency for short connections (e.g., consider a gigabit Ethernet crossover cable). Thus, the practical bound for minimal proxy overhead is a function of the application scenario. Internet-based attacks are unlikely to be less than one millisecond away, but an "evil maid" attack can easily manage sub-millisecond latencies.

If T_{comp}^{per} must be small, it is unlikely that the entire memory region containing the verification function can be checked during a single challenge-response pair. Thus, VIPER employs multiple challenge-response iterations to guarantee that the entire verification function memory region is verified. Between two consecutive challenge-response pairs, the communication function waits for the next challenge. Note that the communication function is also part of the verification function (Figure 4.2) and is verified by the checksum function. However, during the time interval between two consecutive challenge-response pairs, an adversary can accurately guess the expected behavior of the checksum function. To remove this potential attack surface, we work to minimize any idle waiting time by overlapping checksum computation and challenge-response exchange.

Increasing Proxy Communication Overhead The checksum function maintains state as an array of bit vectors. During one iteration of the checksum computation, several checksum vectors may be updated. However, to make the communication between the verifier program and the peripheral efficient, only one randomly-selected checksum vector is returned to the host CPU during each challenge-response pair. To increase the communication overhead between peripherals and the proxy helper, we design the protocol such that the host CPU sends a new challenge to the peripheral before the checksum vector is returned, and the checksum vector to be returned is chosen based on this newly-received challenge sent by the verifier program. This is illustrated in Figure 4.4, where $cksum[i]$ denotes a single checksum vector randomly selected from the full checksum state as it exists after an iteration computed with $nonce[i]$ as an input. The random selection is chosen based on the value of $nonce[i+1]$. T^{n}_{diff}, the time interval between receiving the new challenge and sending the correct checksum result, is so small that a proxy helper is forced to return the entire set of correct checksum vectors or at least all the checksum vectors that have been updated during the checksum computation back to the peripheral before missing the time deadline. (A proxy that randomly guesses which vector to return will quickly be detected as additional checksum iterations drive the probability of repeated successful guessing to a negligible level.) This technique then increases the value of T^{per}_{recv}. The additional communication overhead of sending the entire set of vectors makes it overwhelmingly likely that the attacker will miss the deadline.

Continuous Checksum Computation It is desirable to eliminate any idle waiting on the peripheral between checksum iterations, both for efficiency and to reduce the time during which an attacker knows that the checksum's internal state remains constant. The previous paragraph describes how a portion of the checksum

state as influenced by $nonce[i]$ is not returned to the host CPU until $nonce[i+1]$ is received. If the peripheral device supports concurrent computation and data exchange (as is commonly the case with memory-mapped IO), then the reception of new nonces can be closely synchronized with the runtime of checksum iterations, thereby enabling continuous checksum computation. This is evident in Figure 4.4, when viewing a checksum iteration as the elapsed time at the host CPU between transmission of $nonce[i]$ and reception of $chsum[i]$. Excluding the very first and last checksum iterations, T^{cpu}_{send} for iteration $i+1$ and T^{cpu}_{recv} for iteration i do not impose any additional latency on the total checksum computation time.

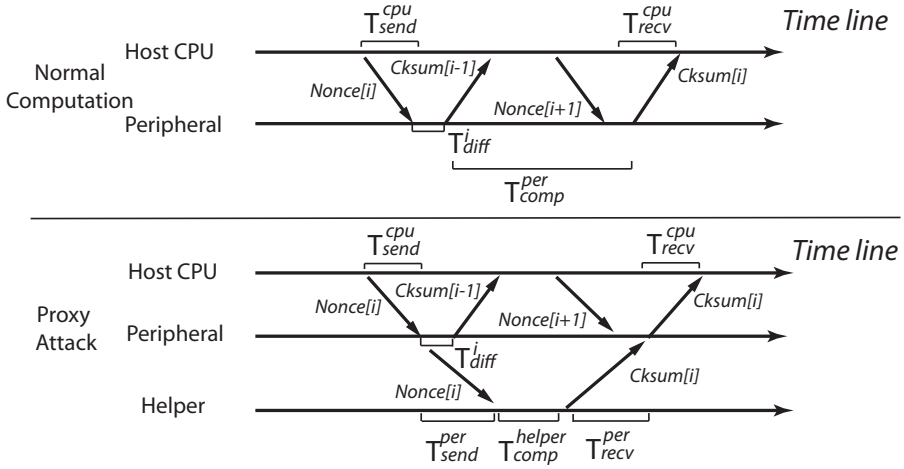

Figure 4.4: The latency-based attestation procedure after speed-up under benign and (hypothetically successful) attack conditions.

The time for a single challenge-response pair is:

$$T^{normal}_{comp} = T^{cpu}_{send} + T^{n}_{diff} + T^{per}_{comp} + T^{cpu}_{recv} \tag{4.4}$$

In this equation, T^{n}_{diff} is the time interval between receiving the n^{th} nonce and sending the $(n-1)^{th}$ checksum result on the peripheral.

In a proxy attack, the malicious code on the peripheral sends the challenge to a proxy helper as soon as it receives the challenge. The time of a challenge-response pair in the proxy attack is:

$$T_{comp}^{proxy} = T_{send}^{cpu} + T_{send}^{per} + T_{comp}^{helper} + T_{recv}^{per} + T_{recv}^{cpu} \tag{4.5}$$

We still assume that T_{comp}^{helper} is zero. Thus, the time overhead caused by a proxy attack is:

$$T_{overhead}^{proxy} = T_{send}^{per} + T_{recv}^{per} - T_{diff}^{n} - T_{comp}^{per} \tag{4.6}$$

Through this equation, we can see that T_{diff}^{n} decreases the value of $T_{overhead}^{proxy}$. Thus, it is desirable that the CPU and peripheral are well-synchronized so that $T_{overhead}^{proxy}$ remains positive and detectable.

Other Potential Attestation Solutions

Other features of communication channels, such as communication latency variance, packet loss, and throughput, may also be viable tools to detect a proxy attack when verifying the integrity of peripherals' firmware. Compared with Ethernet communication, the communication channel between the host CPU and the peripheral is very efficient and stable, with low communication latency variance and near-zero packet loss rate. In a communication latency variance-based attestation protocol, if the communication variance on the proxy communication channel is larger than the well controlled checksum computation time, the proxy helper cannot always send the expected checksum result back to the peripheral in time. A packet loss-based attestation protocol is similar. Network devices suffer from different levels of packet loss with Ethernet. However, on the motherboard, the communication between the host CPU and peripherals over system buses has near-zero loss rate. Therefore, a packet loss-based attestation protocol may also represent a

practical solution to verify the integrity of peripherals' firmware in a computer system. A throughput-based attestation protocol requires that the throughput between the host CPU and the peripheral is larger than the throughput between the peripheral and the proxy helper. The host CPU can send a large amount of random data to the peripheral and require that the random data is incorporated into the checksum computation. To attempt an attack, all of the random data must be sent from the peripheral to the proxy helper. The protocol can be constructed such that the checksum computation on the NIC will complete before the necessary challenge and response can be exchanged with the proxy. We leave the detailed investigation of these mechanisms as future work.

4.2.3 Design of the Checksum Function

In this section, we describe the design of our checksum function in detail. Similar to other software-based attestation schemes [60,94,96], our checksum function sets up an untampered execution environment, computes a fingerprint over the contents of the verification function (i.e., the checksum function itself, and communication and hash functions). Through the checksum result and elapsed computation time, the checksum function provides a guarantee to the verifier program that the verification function has not been modified and the following hash computation was carried out in the untampered execution environment, and is therefore trustworthy. As discussed in the above sections, the checksum function needs to be carefully designed to achieve the necessary timing properties.

There are many different system architectures and instruction sets on peripheral devices. It is difficult to design a single generic checksum function for all cases. However, we first discuss the general principles that apply to the design of the checksum function for any peripheral:

1. All available registers are used during checksum computation. For any additional operations (malicious operations), an attacker has to utilize memory operations (read and write) to save the register values first. This causes large computational overhead since memory operations are much slower than register operations.

2. Each iteration of the checksum function should fit into the Instruction Cache if there is an Instruction Cache, and cause as few cache misses as possible. Any additional operations inserted by malicious code should cause more cache misses.

3. To prevent an attacker from predicting the memory addresses to read, the checksum function reads from memory addresses in a pseudo-random pattern.

4. To prevent an attacker from computing the expected checksum result over a correct copy of the verification function located in some other memory address, the data pointer (DP) value used to address memory should be included in the checksum computation, i.e., the checksum computation is position-*dependent*.

5. To further prevent malicious code from performing the computation at other memory addresses, the program counter (PC) value is also included in the checksum computation if the PC value can be efficiently read by the checksum function.

6. The checksum function is simple enough that it is feasible to determine that the implementation is optimal but non-parallelizable.

We design our checksum function using a sequence of strongly-ordered ADD and XOR operations, since they are naturally non-parallelizable. Strongly-ordered

means that the sequence of checksum operations cannot be changed without causing the checksum result to be different with high probability. Each checksum state update incorporates the value of the program counter (PC), data pointer (DP), contents of the memory referenced by the DP, the most recent nonce sent by the verifier, and the existing checksum states. The carry bit should be included during addition operations if a carry bit is supported on the target peripheral, to avoid losing entropy due to repeated invocation of the checksum. We use intermediate checksum results to select the memory addresses to read in a pseudo-random fashion. This helps to optimize the implementation of the checksum function, since we do not need additional instructions to generate pseudo-random numbers. Malware, in an attempt to remain undetected, must forge the correct PC or DP values during checksum computation. However, forging the PC or DP value will require additional register and memory operations, and cause cache misses and extra memory operations, which will result in detectable computational overhead. We describe our specific implementation of the checksum function on a Netgear GA620 NIC [44] in Section 4.3.3.

4.3 Implementation on Netgear GA620 NIC

We implement and evaluate VIPER on an x86-class computer system. Because of the limited availability of source code for peripherals' firmware, we focus on a PCI-X Netgear GA620 Gigabit Ethernet Adapter (NIC) that uses open source firmware [43]. We installed this card in a Sun Fire V20z 1U rack-mount server that includes a single-core AMD Opteron processor running at 1.795 GHz, 2 GB of RAM, and two PCI-X expansion slots. In this section, we describe the hardware architecture of the Netgear GA620 NIC, and present the detailed implementation of our latency-based attestation scheme.

4.3.1 Netgear GA620 Network Adapter

The Netgear GA620 is a Gigabit Ethernet adapter with a 64-bit PCI-X interface. The theoretical maximum throughput between the host CPU and the GA620 NIC is 8.5 Gbps on a 133.3 MHz, 64-bit PCI-X bus. The maximal bandwidth of the Ethernet link of the Netgear GA620 is 1 Gbps. Figure 4.5 illustrates the architecture of a Netgear GA620.

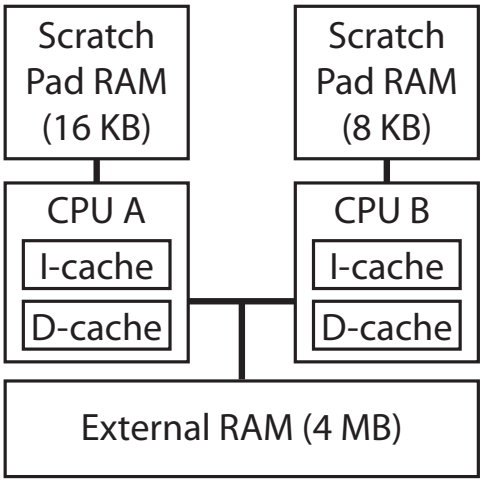

Figure 4.5: Netgear GA620 System Architecture.

The GA620 features two MIPS microcontrollers "A" and "B" running at 200 MHz. The two microcontrollers work simultaneously, and the firmware assigns work to both microcontrollers. In firmware version 12.4.3 [43], Microcontroller A works as the main controller in charge of packet transmission, and microcontroller B assists by preparing DMA descriptors. On each microcontroller, there is a 64-byte Instruction Cache (which fits 16 instructions), and an 8-byte Data Cache. On microcontroller A there are 16 KB of scratch pad memory, though microcontroller B has only 8 KB of scratch pad memory. The scratch pad memory of each microcontroller is located in the same memory address range, and one microcon-

troller physically cannot address the other's scratch pad memory. The host CPU and DMA transactions are also unable to address either scratch pad memory region. During NIC initialization, the firmware moves some time-critical functions into the scratch pad memory. A 4 MB SRAM is shared by both microcontrollers.

Instruction Set Architecture The microcontrollers implement a 32-bit MIPS instruction set architecture. There are 32 registers, where r0 is always zero, and r1 – r31 are used for common operations. All arithmetic operations, logical operations, memory operations, and jump operations are supported while the rotation-shift, multiply, and divide operations, which are available in general MIPS microcontrollers, are removed. In arithmetic operations the carry bit value is not included, which makes the design of an attestation function significantly more challenging because the lack of the carry bit results in entropy loss. The lack of a multiplier or rotation shift also complicates implementation of a size-optimized cryptographic hash function. However, firmware can read the program counter value indirectly using jump instructions (e.g., *JAL* or *JALR*).

Memory Layout Figure 4.6 illustrates the memory layout of the external SRAM on a Netgear GA620 NIC.

The first 16 KB of the external SRAM is mapped into the memory addresses of the host CPU via a memory-mapped IO (MMIO) interface. Both the host CPU and NIC firmware can read or write this MMIO memory. Following the shared MMIO memory, the NIC firmware is in the space from 0x04000 to 0x16000. After the firmware space follows the space for each microcontroller's stack, RX/TX DMA descriptors and RX/TX buffers. On microcontroller A, the 16 KB internal scratch pad memory is addressable from 0x00c00000 to 0x00c04000. On microcontroller B, the 8 KB internal scratch pad memory occupies 0x00c00000 to 0x00c02000.

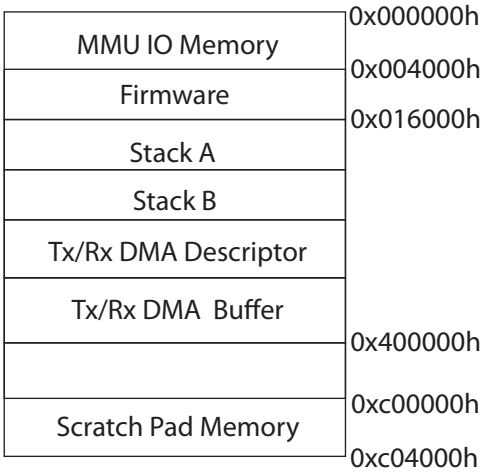

Figure 4.6: Netgear GA620 Memory Layout.

NIC – Host CPU Communication The Netgear GA620 NIC and host CPU communicate via a *Mailbox* abstraction, which is a bank of 32 8-byte communication registers that are mapped into the host CPU's MMIO address space. Microcontroller A uses the lower 16 mailbox registers and microcontroller B uses the higher 16 mailbox registers. The host CPU can read or write to the mailbox registers directly using ordinary memory operations (e.g., `mov`). When the host CPU writes a mailbox register, an event is generated on the GA620 NIC and the NIC firmware can detect the event by checking the event register. However, the GA620 NIC cannot cause interrupts to the host CPU by writing values into the mailbox registers, because the interrupt mechanism on the host CPU is too slow to support Gigabit-speed communication. For large amounts of data, such as network packets, the NIC transmits the data between local memory (TX/RX buffer) and main memory through DMA.

4.3.2 Verification for Microcontrollers A and B

We conduct the attestation protocol on both microcontrollers A and B to verify the entire memory contents of the Netgear GA620 NIC. To prevent the verification functions running on one microcontroller from being modified by malicious code running on the other microcontroller, we execute the verification functions within the scratch-pad memory of each microcontroller. On each microcontroller, we implement: a checksum function *VCF* (*Viper Checksum Function*) (Section 4.2.3) that computes a checksum over the entire verification function memory contents; a communication function that initializes the checksum states, reads nonces from the host CPU, and randomly returns a 32-bit checksum state vector to the host CPU for each nonce-checksum pair. *VCF* and the communication function are implemented using 656 MIPS instructions, and are deployed into the scratch pad memory of each microcontroller. A SHA-1 hash function is deployed into the scratch pad memory of Microcontroller A and its binary code consumes 2 KB. In detail, the attestation procedure performs the following operations:

1. The verifier program on the host CPU sends an attestation request to both microcontrollers A and B on the NIC. The checksum functions on both microcontrollers A and B set the NIC to a known state.

2. The verifier program conducts the latency-based attestation protocol for microcontroller B first.

3. During the attestation, *VCF*, which is in microcontroller B's scratch-pad memory, sets up an untampered execution environment and computes a checksum over the entire 8 KB scratch pad memory on Microcontroller B. Because microcontroller A cannot access microcontroller B's scratch pad memory, any malicious code on microcontroller A cannot tamper with the execution

environment on microcontroller B.

4. Because *VCF* verifies the entire scratch pad memory on Microcontroller B, it is not necessary to call the SHA-1 hash function to compute a hash over the scratch pad memory on Microcontroller B. After the attestation procedure for microcontroller B, the communication function on microcontroller B continues to run but waits for an *EXIT* command from the host CPU. The host CPU will not send an *EXIT* until the attestation for both microcontrollers B and A are complete. Note that while it waits, the program counter of microcontroller B remains within the scratch-pad memory that has just been verified, so its behavior is known.

5. The verifier program on the host CPU verifies the checksum results and computation time during the attestation for microcontroller B. If the attestation for microcontroller B is successful, the verifier program conducts the attestation for microcontroller A, also using the latency-based attestation protocol.

6. During the attestation for microcontroller A, *VCF* on microcontroller A first sets up an untampered execution environment, and computes checksums over *VCF* itself, the communication function, and the SHA-1 hash function.

7. The *VCF* calls the SHA-1 hash function to compute a cryptographic hash over the memory contents of the entire scratch-pad memory on microcontroller A, and of the external SRAM. It then sends the hash result to the host CPU. Because the verification function on microcontroller B is running during the attestation of microcontroller A, the attestation procedure of microcontroller A cannot be tampered with by B.

8. The verifier program on the host CPU confirms the attestation results (checksum results, timing results, and hash result) of the attestation for microcon-

troller A.

9. The verifier program informs both microcontrollers A and B to exit their attestation functions.

4.3.3 Checksum Function Implementation

We implement the checksum function *VCF* as 32 checksum computation blocks. Each block has 16 MIPS CPU instructions and fits precisely into the 64-byte instruction cache, since all instructions are 32 bits long. As each block executes, one 32-bit checksum vector, out of a total of 26 checksum state vectors, are updated using alternating *ADD* and *XOR* operations. Each block takes as input: a subset of the contents of scratch pad memory in the NIC, other checksum states, nonces from the verifier program on the host CPU, the memory addresses being read (data pointer), and the program counter. All 31 available general purpose registers ($r1$ to $r31$) are used by the checksum computation: 26 registers ($r5$ to $r30$) are used to save checksum states; $r1$ and $r31$ are used as temporary variables to save memory addresses and the values of recently read memory; $r2$ stores the nonce provided by the host CPU; $r3$ stores the end address of each checksum block; $r4$ stores the starting address of each checksum block (both $r3$ and $r4$ essentially reflect program counter values). Following is the pseudo-code to update one checksum state in each block.

```
/* Pseudo Code to update one checksum state:
   C is the checksum vector,
   i is the index of a checksum vector register,
   tmp is a temporary variable,
   addr is the memory address to read,
   memory_base can be the beginning address of
```

```
   a checksum block or the end address of
   a checksum block. */
/* in odd blocks */
tmp = mem[addr] xor C[(i-2) mod 26] + addr
/* construct another memory address */
addr = memory_base xor ( tmp & mask )
/* update one checksum state */
C[i] = mem[addr] xor C[i] + PC xor nonce + tmp
/* create dependency on C[i] for next iteration */
nonce = nonce + C[i]

/* in even blocks */
tmp = mem[addr] + C[(i-2) mod 26] + addr
/* construct another memory address */
addr = memory_base xor ( tmp & mask )
/* update one checksum state */
C[i] = mem[addr] + C[i] xor PC + nonce xor tmp
/* create dependency on C[i] for next iteration */
nonce = nonce xor C[i]
```

Figure 4.7 shows the assembly code of one checksum computation block. In this checksum block, one checksum state ($r7$) is updated based on the contents of two memory addresses, the values in $r3$, and another checksum state ($r5$). At the end of this checksum block, the value of $r3$ is updated by the instruction *JALR*, which reads the program counter (PC) value into $r3$ as part of a jump to the memory address saved in $r4$. Note that MIPS executes the instruction following a jump instruction even if the jump is taken; it executes prior to the instruction residing at

Assembly Instruction	Explanation
xor r31, r4, r1	$addr = memory\_base \oplus offset$
lw r1, 0(r31)	*memory read*
xor r1, r5, r1	$tmp1 = r5 \oplus mem[addr]$
add r31, r31, r1	$tmp2 = addr + tmp1$
andi r1, r31, 0x1ffc	$offset = tmp2$ & $mask$
xor r1, r3, r1	$addr = memory\_base \oplus offset$
lw r1, 0(r1)	*memory read*
xor r1, r7, r1	$tmp3 = r7 \oplus mem[addr]$
add r1, r3, r1	$tmp3 = PC + tmp3$
xor r1, r2, r1	$tmp3 = nonce \oplus tmp3$
add r7, r31, r1	$r7 = tmp2 + tmp3$
add r2, r7, r2	$nonce = r7 + nonce$
andi r1, r7, 0x7c0	$tmp4 = r7$ & $mask1$
xor r4, r4, r1	$r4 = r4 \oplus tmp4$
jalr r3, r4	$r3 = PC + 8$; *jump to* $r4$
andi r1, r2, 0x1ffc	$offset = nonce$ & $mask$

Figure 4.7: Assembly Instructions for One Checksum Block.

the jump target. The value of $r4$ is updated using five bits (bits 6 to bits 10) of $r7$. Because the target address ($r4$) of the *JALR* instruction is updated randomly, the PC jumps to the beginning address of one of the 32 checksum blocks at the end of each checksum block in a pseudo-random fashion. *In this way, we can prevent an attacker from predicting the target address of the jump instruction.* Out of the 32 checksum blocks, 4 checksum blocks are chosen as 'exit' blocks, which deterministically jump to the communication code following checksum computation. The communication code returns one 32-bit checksum state vector to the host CPU and reads the nonce most recently sent from the host CPU (i.e., the verifier program).

One checksum state is read in each block, and one state is updated (written) in each block. Cumulatively across all 32 checksum blocks, all 26 checksum states are updated. Since an attacker cannot predict which block will be used for computation until the current block has completed, the attacker cannot use any of the registers that store checksum states for malicious operations, unless the attacker first uses memory operations to save the values stored in those registers.

4.3.4 Latency-Based Attestation

As described in Section 4.2.2, to prevent a proxy attack, the checksum computation time must be well controlled, and small enough that the overhead of a proxy attack ($T^{proxy}_{overhead}$ from Eqn. 4.3) is detectable. In this section, we calculate the theoretical minimal time of a proxy attack over a 1 Gbps Ethernet link. Then, we describe the mailbox communication overhead between the host CPU and the GA620 NIC, the checksum computation time for one challenge-response pair, and an optimization to speed up the attestation procedure via synchronization.

Theoretical Fastest Time for a Proxy Attack

In an Ethernet-based proxy attack, to explore the best case for the attacker, we assume that both the peripheral and the proxy helper need no time to prepare the network packets. However, the packets used in a proxy attack must go through the hardware Ethernet MAC (physical serial communications port) of the NIC. Therefore, theoretically the fastest time of an Ethernet-based proxy attack is the time that it takes the packets to go through the Ethernet MAC of both the sender's and receiver's NICs, and the time that the data actually spends on the wire. We assume that the peripheral and the proxy helper utilize 72 byte raw Ethernet frames to exchange data during a proxy attack, as 72 bytes is the minimal allowable Ethernet

frame size [41]. Assuming that both the peripheral and the proxy helper use 1 Gbps Ethernet MAC, the time consumed by packet transmission is 1152 nanoseconds for a round trip.[1] This is useful to set a lower bound for the shortest possible proxy attack (i.e., the fastest attack that could ever be performed with this hardware configuration), and sets $T^{per}_{send} + T^{per}_{recv} = 1152\ ns$ from Eqns. 4.3 and 4.6. Under these conditions, to detect a proxy attack, T^{per}_{comp} (Eqn. 4.3) and (if using the synchronized version) T^{n}_{diff} (Eqn. 4.6) must be sufficiently small that $T^{proxy}_{overhead}$ remains positive and detectable.

Communication Overhead

During attestation, the host CPU measures the time for each challenge-response pair between the host CPU and the peripheral device. To detect the time overhead caused by malicious operations, the communication between the host CPU and peripheral should be efficient and stable. Figure 4.8 shows the mailbox communication architecture between the host CPU and the microcontrollers on the GA620 NIC.

Determining Communication Delay We now describe our approach to empirically determine the CPU-NIC communication overheads (T^{cpu}_{send} and T^{cpu}_{recv} from Section 4.2.2). Essentially, we exchange the smallest possible amount of data between the CPU and the NIC, with the NIC performing the absolute minimum amount of computation to return a result. This is as close as we can practically come to setting $T^{per}_{comp} = 0$. First, the host CPU writes a 32-bit value into address A (a mailbox register address), which generates an event on the GA620 NIC. As soon as it detects the mailbox event, the firmware on the GA620 NIC updates the 32-bit value

[1] $2 \cdot \frac{72\ bytes \cdot 8\ bits/byte}{1{,}000{,}000{,}000\ bits/second} = 1152\ ns$. 72 bytes is the minimal usable Ethernet frame size with payload [41].

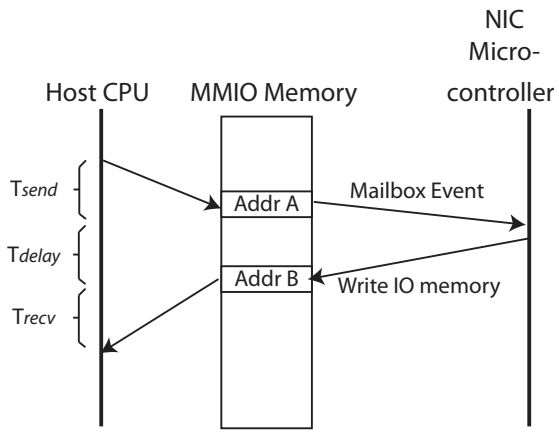

Figure 4.8: Host CPU to NIC Communication via GA620's *Mailbox*.

in address B. After a time delay, the host CPU repeatedly reads address B until it obtains the updated value from address B.

Since memory operations can take hundreds of CPU cycles, the communication between the host CPU and NIC is the most efficient when the host CPU can predict the precise time to read the updated value, and obtain the updated value from address B in a single read operation. Therefore, we design an experiment to predict the time delay between a mailbox write and mailbox read on the host CPU, so that the host CPU can communicate efficiently and reliably. In our experiment, the host CPU stalls for a fixed delay interval between the MMIO write and MMIO read. For each delay period, we repeat the MMIO write and MMIO read 200 times, and record the frequency that the host CPU obtains the updated value from address B in a single MMIO read. We then increase the delay, and repeat the same experiment until the delay is large enough that the host CPU can always read the updated value in a single MMIO read.

We implement the measurement code on both the host CPU and the GA620 NIC in assembly for efficiency. On the host CPU, we disable all interrupts on the

CPU core where the measurement code is executing. We choose the instruction *RDTSC* to read the current CPU counter as a timer, taking care to incorporate a serializing instruction (i.e., *CPUID*) to prevent instruction reordering from impacting the accuracy of our measurements. We implement the delay by spinning in a tight loop that consumes exactly two clock cycles per loop iteration.

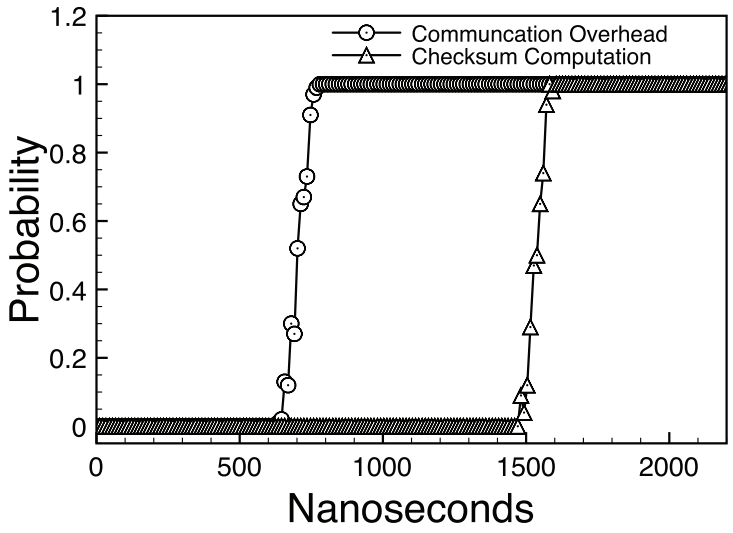

Figure 4.9: Impact of delay on probability that host CPU reads expected value from address B in a single MMIO read.

Figure 4.9 shows our experimental results. In this figure, the X-axis is the delay in nanoseconds, N. The Y-axis is the probability that the host CPU reads the updated value from address B in a single MMIO read when the host CPU stalls for N nanoseconds between writing the mailbox at address A and reading the value from address B. The experimental results show that when the delay is larger than 790 ns, the probability is 1.

Demonstrating Communication Reliability We then fix the delay at 790 ns (determined from the previous results), and repeat the measurement another 200 times to confirm that communication is reliable. In this experiment, the host CPU measures the time between writing the mailbox event and obtaining the updated value from address B after a delay of 790 ns. Figure 4.10 shows our measurement results. The X-axis is individual trials and the Y-axis is the timing result in nanoseconds computed from CPU cycles. The average result of the 200 trials is $T^{cpu}_{send} + T^{cpu}_{recv} =$ 1375 ns. The standard deviation is 4 nanoseconds.

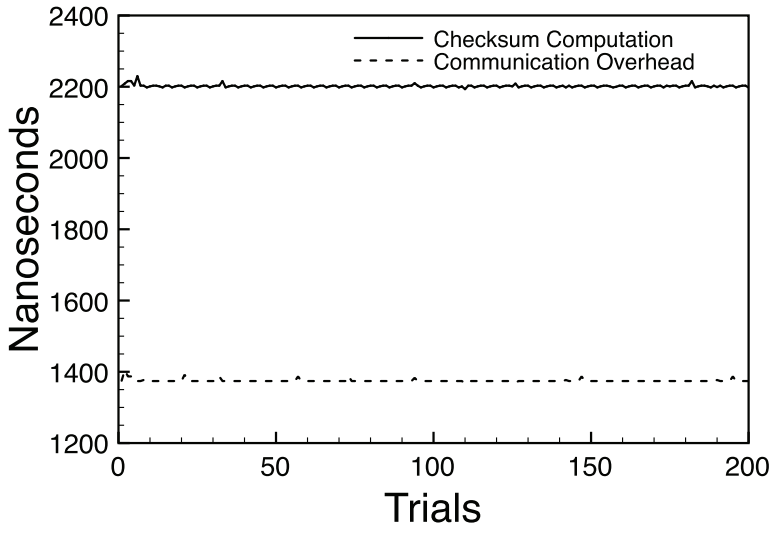

Figure 4.10: Communication overhead and checksum computation time (time of a challenge-response pair) measured by the host CPU.

Checksum Computation Time

We conduct two experiments to measure the time for checksum computation on the GA620 NIC. These experiments are similar to the experiments used to measure the communication overhead between the host CPU and NIC in Section 4.3.4.

The only difference is that in the communication overhead measurement, the NIC writes a 4-byte value to MMIO memory immediately upon receiving a mailbox event, while in these experiments the NIC executes three checksum blocks before writing to MMIO memory. *Because we have implemented a checksum simulator, the checksum simulator always selects nonces where the NIC returns a checksum state after executing precisely three checksum blocks.*

As with the communication overhead measurement, we first perform experiments to predict the necessary delay on the host CPU to guarantee that the host CPU can obtain the expected checksum result in a single MMIO read operation. Figure 4.9 shows the probability that the host CPU gets the expected checksum result using a single MMIO read operation while varying the delay. For each delay period, the experiments are repeated 200 times. The experimental results show that after the delay reaches 1616 ns, the host CPU starts to read the expected checksum result for all 200 experiments, i.e., it becomes sufficiently reliable.

We conduct a second experiment to measure the entire time between the host CPU writing the mailbox event to MMIO memory, and reading the checksum result after a delay of 1616 ns (one challenge-response pair). In each trial, the checksum function on the NIC computes three checksum blocks. Figure 4.10 shows the time of 200 challenge-response pairs measured by the host CPU. The average value of a single challenge-response pair (T_{comp}^{normal}) is 2202 nanoseconds, with a standard deviation 4 nanoseconds. Based on these results, we can calculate that the time required for computing three checksums blocks (T_{comp}^{per}) on the NIC is about 827 nanoseconds, (i.e., $T_{comp}^{per} = T_{comp}^{normal} - (T_{send}^{cpu} + T_{recv}^{cpu}) = 2202 \text{ ns} - 1375 \text{ ns} = 827$ ns). Thus, the overhead caused by the theoretical fastest proxy attack over 1 Gbps Ethernet is about 325 nanoseconds ($T_{overhead}^{proxy} = (T_{send}^{per} + T_{recv}^{per}) - T_{comp}^{per} = 1152 \text{ ns} - 827 \text{ ns} = 325$ ns).

Host CPU – NIC Synchronization

A design goal of VIPER is to maximize the utilization of the system buses and the CPUs in the NIC, and to minimize the overall attestation runtime. Recall (Section 4.2.2) that we can parallelize bus communication and NIC computation; the host CPU sends the *next* nonce before the NIC writes the *current* checksum result into MMIO memory. These tight timing constraints require that the host CPU and NIC be synchronized, to guarantee (1) that the host CPU is able to send the nonce to the NIC before the NIC starts to return current checksum states, and (2) that the host CPU reads the checksum result from MMIO memory only after the NIC has updated the result. We describe the design and implementation of our synchronization mechanisms and the experiments that demonstrate their effectiveness.

To remain synchronized, the time interval between two consecutive MMIO reads by the host CPU should match the time required to compute checksum blocks on the NIC CPU. Therefore, given the initiation time of a read of $cksum[i]$, the time when the host CPU should start to read the value of $cksum[i+1]$ can be predicted. In our implementation, the verifier code again uses *RDTSC* to read the CPU time stamp counter before starting to read $cksum[i]$, and then predicts the future time stamp counter value when the host CPU should start to read the following checksum state. The host CPU busy-waits in a tight loop that consumes exactly 2 CPU cycles per iteration until the necessary time arrives, although we convert iterations to nanoseconds to streamline presentation here. Figure 4.11 shows the verification procedure with synchronization between the host CPU and NIC. $nonce1$ is the first nonce that the host CPU sends to the NIC, while $cksum1$ is the first checksum result that the NIC returns to the host CPU.

To implement synchronization between the host CPU and NIC, the checksum computation time must be long enough that the host can perform one MMIO read

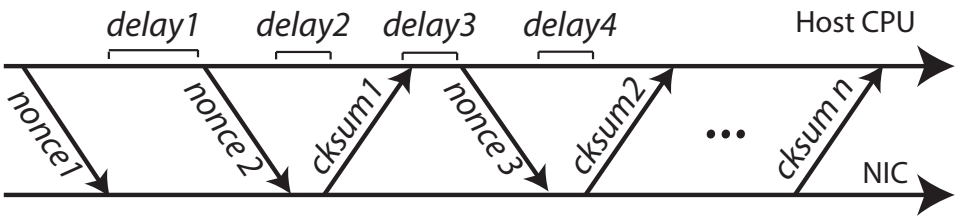

Figure 4.11: Verification procedure with synchronization between host CPU and NIC.

operation (read a checksum result) and one MMIO write operation (write a nonce to the NIC) inside the time interval where two consecutive checksum results are returned by the NIC. When the NIC computes three checksum blocks for each nonce-checksum pair, the checksum computation time is not long enough to keep synchronization between the host CPU and NIC. Therefore, we increase the number of checksum blocks to compute on the NIC for each nonce-checksum pair. Our synchronization experiments show that the host CPU and NIC can remain synchronized for over 300 nonce-checksum response pairs when the NIC computes six checksum blocks for each nonce-response pair.[2]

Figure 4.12 illustrates the first few iterations of this procedure, yielding $delay1 =$ 780 ns, $delay2 = 670$ ns, and $delay3 = 390$ ns. Although 300 nonce-response iterations are not sufficient to verify the entire memory contents of the verification function (this is a simple application of the coupon collector's problem), the same procedure can be repeated multiple times to verify the entire memory with overwhelming probability.

The average time for a single challenge-response pair is 3106 ns (T_{comp}^{normal} from Eqn. 4.4), with a standard deviation of 19 ns. The entire time for performing 300

[2]Note that the time for computing six checksum blocks on the NIC is longer than the time of the theoretical fastest proxy attack described in Section 4.3.4. However, it is much shorter than the time of the real proxy attack we have implemented. We discuss this discrepancy further in Section 4.8.

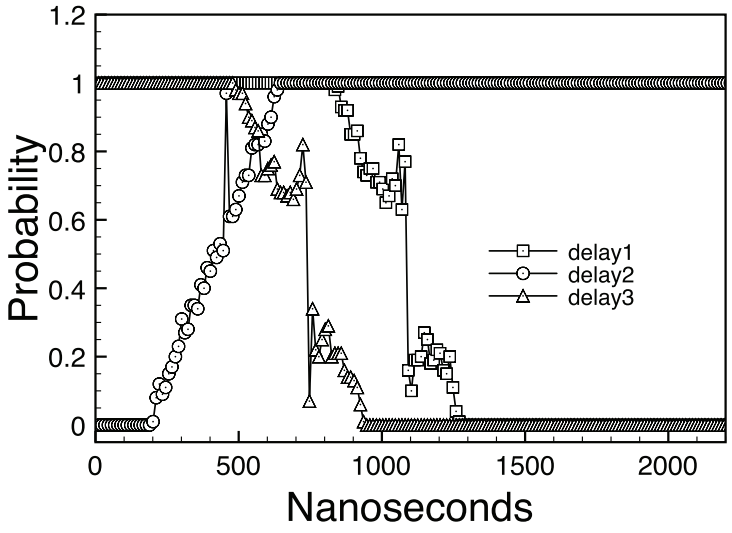

Figure 4.12: Impact of *delay*1, *delay*2, and *delay*3 in Figure 4.11.

challenge-response pairs is 535 microseconds. The entire verification procedure (excluding the time for SHA-1 to process the firmware in SRAM on the NIC) consumes about 2 milliseconds.

4.4 Evaluation on Netgear GA620 NIC

We implement a real Ethernet-based proxy attack, the *forging DP attack*, and the *forging PC attack* on the Netgear GA620 NIC to evaluate VIPER's ability to detect the attacks.

4.4.1 Ethernet-based proxy attack

We implement a real Ethernet-based proxy attack (Figure 4.13). Computers A and B connect directly (without a switch) through a crossover cable. The NICs in both computers are 1 Gbps Netgear GA620s. On computer A, the host CPU verifies the

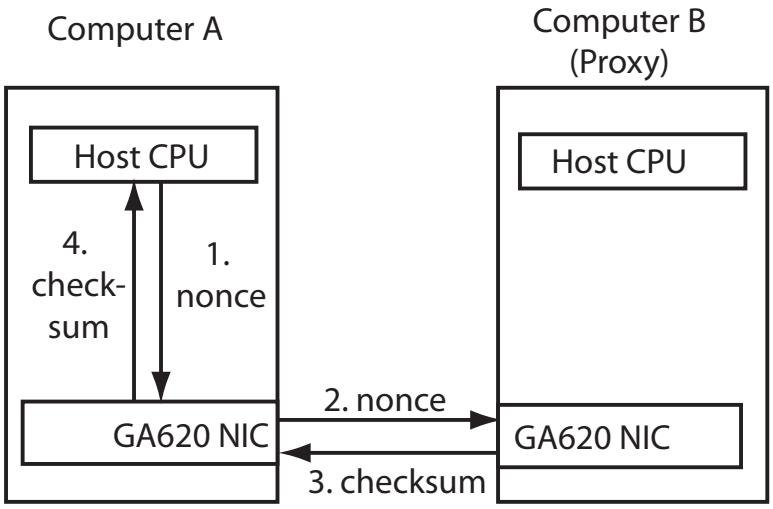

Figure 4.13: Proxy Attack Implementation.

firmware integrity of the GA620 NIC using the latency-based attestation protocol. Once the NIC on computer A receives a challenge from the host CPU, it sends the challenge to computer B (the proxy) over the crossover cable. It then waits for the reply from computer B. In the real implementation, we assume that the proxy is very fast, and needs no time to compute the excepted checksum result. Therefore, on computer B, as soon the NIC receives the packet that contains the challenge for attestation, it sends the response, which includes the expected checksum, to computer A. The NIC on computer B (the proxy) generates the response immediately within its firmware, without bus activity and without involving the host CPU. Then, on computer A, the NIC receives the checksum from computer B, and returns the checksum to the host CPU.

Note that there is no timer on the NIC. Thus, the host CPU measures the time of the proxy attack indirectly, since it measures the time of the proxy procedure plus the communication overhead between the host CPU and the NIC. The average latency of a single challenge-response pair measured by the host CPU over

200 trials during the proxy procedure is 43.72 ± 0.38 microseconds. The proxy attack we implement consumes much more time than the checksum computation time for each challenge-response pair in our implementation on the GA620 NIC. Our implementation shows that computer B cannot send the expected checksum back to the NIC on computer A on time because the latency to communicate with the proxy is longer than the expected checksum computation time. Note that our simple C code implementation in the NIC firmware is slower than the theoretical fastest gigabit Ethernet proxy attack, although this limitation is only a weakness for an attacker with a direct physical connection (no intermediate network hops) to the target system's Ethernet port. Further optimization of our attack is interesting future work.

4.4.2 Forging Data Pointer (DP) attack

In a *forging DP attack*, the attacker maintains a shadow copy of the correct verification function in unused memory, and then executes malicious code out of the original address range of the checksum function, computing the expected checksum over the shadow copy. In this attack, the malicious code needs to add or subtract a constant offset to the DP value to redirect the memory addresses to read (one instruction). Because the DP value is also included in the checksum computation, the malicious code also needs to forge its value before the computation (another instruction). To keep the PC value correct, the malicious code cannot inject additional instructions in the checksum computation block. Thus, the malicious code has to jump out of the main checksum computation block (one jump instruction) to change the DP value to compute the expected checksum. After the computation, malicious code has to jump back (another jump instruction) to the main checksum computation block to obtain the expected PC value. In our current checksum de-

sign, two memory addresses (DP) are checked in each checksum block and the DP value is included in the checksum computation once in each checksum block, so a *forging DP attack* needs five additional instructions (two jump instructions and three arithmetic or logical instructions) to compute the expected checksum result in each checksum block. The two jump instructions also cause two cache misses in each checksum block.

4.4.3 Forging PC attack

In a *forging PC attack*, malicious code is deployed in some other memory address and computes the expected checksum over the original copy of the verification function. In this attack, the malicious code does not need to forge the DP value since the checksum is computed over the original copy of the verification function. However, the malicious code does need to forge the correct PC. In *VCF*, $r4$ stores the beginning address of the current checksum computation block while $r3$ stores the end address of the previous checksum computation block. $r4$ and $r3$ are updated at the end of each checksum computation block. In the forging PC attack, the memory address of the malicious checksum computation block has a constant offset from the address of the original checksum computation block. To jump to the malicious code block, the malicious code adds a constant offset to $r4$ (one instruction) before the jump instruction in each checksum block. To forge the correct PC values (both $r3$ and $r4$) before the checksum computation, the malicious code also needs to subtract a constant value from $r3$ and $r4$ (two instructions). As with the forging DP attack, to guarantee that the value of $r3$ and $r4$ have a constant offset from the correct value, the malicious code has to jump out of the malicious checksum block (one jump instruction) to modify the PC value, and then jump back (another jump instruction) before the jump instruction at end of each check-

sum block. Therefore, the forging PC attack needs five additional instructions (two jump instructions, three arithmetic or logical instructions) and causes two additional cache misses for each checksum computation block to compute the expected checksum result.

4.4.4 Evaluation Results

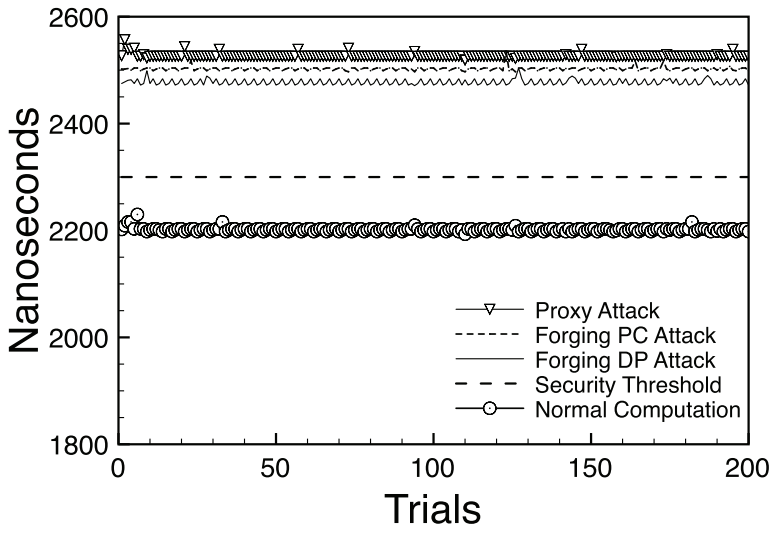

Figure 4.14: Attacker Performance.

Figure 4.14 shows the verification time (checksum computation time plus communication overhead) of normal computation (the host CPU and NIC are not synchronized; three checksum blocks are computed for each nonce-checksum pair), a *forging PC attack*, a *forging DP attack*, the theoretical best proxy attack, and a time threshold to detect attacks. The theoretical proxy attack line represents the communication overhead between the host CPU and the NIC plus the time of the theoretically fastest proxy attack (1152 nanoseconds for a round-trip) between the NIC and a proxy helper over 1 Gbps Ethernet. Our results show that a *forging PC*

attack or a *forging DP attack* cause over 280 nanoseconds of computation overhead, while the theoretical fastest proxy attack causes over 325 nanoseconds of overhead compared with the normal computation. These overheads are readily detected by the host CPU executing in a tight loop with interrupts disabled. Thus, VIPER successfully detects all of the attacks.

4.5 Implementation on Apple Aluminum Keyboard

We also evaluate VIPER with a wired Apple Aluminum keyboard. We assume the Apple Aluminum keyboard is the last (slowest) peripheral to verify and malicious code inside the keyboard cannot perform a local or remote proxy attack. Thus, we utilize a simple nonce-response protocol to verify the integrity of the firmware on the Apple Aluminum keyboard.

In this Section, we first describe the hardware feature of the Apple Aluminum keyboard, then detail the verification function design and implementation on the Apple Aluminum keyboard.

4.5.1 The Apple Aluminum Keyboard

The Apple Aluminum Keyboard connects to a computer via a USB interface. Inside the Apple Aluminum keyboard, a Cypress CY7C63923 microcontroller controls the keyboard matrix. During a firmware update, the firmware on the CY7C63923 microcontroller is updated. The Cypress CY7C63923 microcontroller belongs to the Cypress enCoReTM II family and is primarily designed for low-speed USB peripheral controllers, such as mice, keyboards, joysticks, game pads, barcode scanners, and remote controllers. The Cypress CY7C63923 is a Harvard Architecture, 8-bit programmable microcontroller with 256 bytes of RAM and 8 KB of Flash. Five registers on this microcontroller control the operations of its

CPU. These five registers are the Flag register (F), Program Counter (PC), Accumulator Register (A), Stack Pointer (SP), and Index Register (X). PC is 16-bits in length, while all the other registers are 8-bits long. A and I are used during arithmetic or logical operations on this microcontroller.

4.5.2 Verification Function Design

Due to the constrained computation and memory resources in the simple microcontrollers that are deployed on low-speed peripherals, the verification functions that are used in previous proposals cannot be deployed directly on the Apple Aluminum Keyboard. In this section, we detail the verification by describing the pseudo-random number generator, our design for filling data memory with pseudo-random values, and our checksum function.

Pseudo-Random Number Generator (PRNG). In the verification function, the PRNG is used for two purposes:

1. output PRNs to fill the data memory;
2. output PRNs to construct memory address to read in a pseudo-random fashion.

In previous proposals [96], T-functions [56] or RC4 [118] are used to output PRNs. However, on low speed peripherals, it is challenging to implement the same PRNGs efficiently due to constrained computation or memory resources. T-functions need a multiplication unit to generate PRNs efficiently. However, a hardware multiplication unit is not available in many low-speed microcontrollers that are used in peripherals. Software-based multiplication is too slow to be a viable option. For instance, on a CY7C63923 microcontroller [25], a software-based multiplication requires thousands of cycles to complete a 16-bit multiplication. An RC4-based

PRNG outputs pseudo-random numbers through simple arithmetic and logical operations. However, RC4 requires at least 256 bytes of RAM, which consumes all memory resource on some microcontrollers (such as the CY7C63923). Laszlo et al. [36] propose several efficient PRNGs that are primarily designed for low speed embedded devices. The PRNGs proposed by Laszlo et al. only require simple addition, XOR, or shift operations and few memory resources to output PRNs efficiently. From the PRNGs that Laszlo et al. propose, we select a 2-stage PRNG in our design. Other PRNGs that have the same features are also potential choices. The PRNG we select outputs PRNs as follows:

$$x[i+1] = x[i-1] + (x[i] \oplus rot(x[i-1], 1)) \tag{4.7}$$

$\oplus$ is the logical XOR operation and *rot* is the left rotation shift operation. x is the output of this PRNG, a 32-bit long stage. The value of one stage is updated based on the values of the previous two stages in each iteration.

Filling Data Memory With Pseudo-Random Values. The verification function fills data memory in a pseudo-random fashion. Such a design is required to prevent an attacker from reserving one small block at the end of data memory to store malicious data, and then generating the PRNs that are expected to be in that small block of data memory on-the-fly when they are needed by the checksum routine. In our design, the verification function determines the data memory addresses to be filled based on the outputs of the PRNG. Each address is then filled using the XOR of two bytes of PRNG output. This prevents the attacker from generating the PRNs that are expected in data memory based on the values of existing PRNs in other locations in data memory, since only XOR results are stored in data memory. To make sure that all data memory is filled, the verifier can obtain the number of loop iterations upon which all data memory has been filled. This value is determined by

simulating the filling procedure before sending the attestation request.

Checksum Function Design. The checksum function computes a fingerprint over the entire contents of both program memory and data memory. As in SWATT or ICE, the checksum is computed through a strongly ordered sequence of addition, XOR, and rotation shift operations. If the sequence of the operations is altered or some operations are removed, the checksum result will be different with a high probability. Also, the checksum function reads memory in a pseudo-random traversal. If the memory size is N bytes, each memory location is accessed at least once after $O(NlnN)$ memory read with a high probability [96]. The input to the checksum function is a 16-byte pseudo-random value, which is used to seed the PRNG (i.e., to provide stages $x[0]$ and $x[1]$) and to initialize an 8-byte checksum vector. The output of the checksum function is also an 8-byte checksum vector. Each byte of the checksum vector is called a checksum state. For each iteration of the checksum function, the value of one checksum state is updated based on the current memory contents, the pseudo-random value, and the values of other checksum states. In order to preserve the entropy of the carry bit, we add each carry bit to the checksum state. Following is the pseudo code of one iteration of the checksum function:

```
/* C is the checksum vector, i is its current index. */
/* PRN is the pseudo-random number */
/* addr is the memory address */
   addr = PRN & MASK     /* Construct memory address */
/* update one checksum state */
   C[i] = C[i] + ( Mem[addr] xor C[(i-2) mod 8] )
   tmp = C[i] mod 256
   C[i] = rotation_left_shift(tmp, 1) + ( C[i] >> 8 )
```

```
C[i] = C[i] + i
i = (i + 1) mod 8           /* update the index i */
```

To optimize the computation time of the checksum, we unroll the checksum loop eight times and each time one checksum state is updated by either the contents of program memory or the contents of data memory, which can be adjusted based on the memory size proportion of each. For example, on a peripheral that has 8 KB of programmable Flash and 256-bytes of RAM, seven checksum states can be updated based on the contents of Flash memory while one checksum state can be updated based on the contents of RAM. To make the computation of each checksum state different, we also add the index value to the checksum state.

4.5.3 Verification Function Implementation

Following a keyboard firmware update, the Flash memory from 0xe00 to 0x1300 (1280 bytes) is available free space, where we implement our verification function. Figure 4.15 shows the final memory layout of keyboard Flash memory.

The verification function is located at addresses 0x0e00 – 0x1268 in the Flash memory. The Flash memory from 0x1268 to 0x1300 is filled with pseudo-random values. In the verification function, a 'Send Function' is the communication module that handles the attestation request from the verifier and returns checksum results through the USB channel to the verifier following checksum computation. Before the attestation, the contents of RAM is unpredictable to the verifier. Therefore, an 'Initial Function' sets the contents of data memory to a known state by filling the data memory with pseudo-random values (we fill data memory in a linear sequence instead of in a pseudo-random fashion as designed). The data memory from 0x18 to 0xff is filled with with pseudo-random values, while the data memory

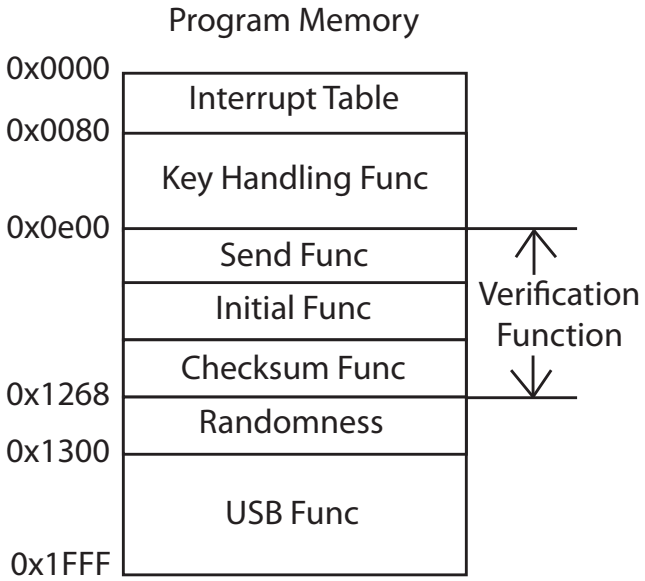

Figure 4.15: Memory Layout of Program Memory

from 0x00 to 0x17 is used to store variables for the verification function. Also, the 'Initial Function' disables all interrupts on the CY7C63923 microcontroller, which prevents the contents of data memory from being modified by an interrupt call during checksum computation. A 'Checksum Function' is implemented, which computes a checksum over the entire contents of both program memory (Flash) and data memory (RAM). After attestation, we reset the Apple Aluminum Keyboard. A two-stage pseudo-random number generator (PRNG) is implemented in both the 'Initial Function' and 'Checksum Function'. The 8-byte nonce sent by the verifier is used to seed the PRNG in the 'Initial Function'. After filling RAM, the PRNG in 'Initial Function' outputs a 16-byte random number to serve as input to the 'Checksum Function', which is used to seed the PRNG in 'Checksum Function' and to initialize the 8-byte checksum vector. All of these functions are implemented in assembly. The two-stage PRNG is implemented using 23 assem-

bly instructions. It outputs 4 bytes of pseudo-random values every 157 CPU cycles on the Apple Aluminum Keyboard. We unroll the checksum iteration eight times. Each time one checksum state is updated. The first seven checksum states are updated based on the content of Flash memory while the last checksum state is updated based on the content of RAM. Including the two-stage PRNG, 'Checksum Function' only requires 19.5 instructions and 133.5 CPU cycles on the average to update one checksum state on the Apple Aluminum Keyboard. Following is the assembly we implement to update one checksum state:

```
; [0x00] to [0x07] saves outputs of PRNG
; [0x08] to [0x0f] saves temp variables, such as counter
; [0x10] to [0x17] saves checksum states
; ROMX is the instruction to read flash memory
; CPU loads memory address from register A
; and register X when ROMX is executed
; the result of ROMX is saved in register A automatically by CPU
; Update checksum[0]
MOV X, [0x00]    ; read pseudo-random values
MOV A, [0X01]    ; to register X and A
AND A,0X1F       ; construct memory address
ROMX             ; read Flash memory, result is saved in A
XOR A, [0x16]    ; Mem[addr] xor checksum[6]
ADD [0x10], A    ; add previous checksum value
RLC [0x10]       ; left rotation shift 1 bit, add carry bit
ADC [0x10],0x00  ; add carry bit (add index too)
```

4.6 Evaluation on Apple Aluminum Keyboard

In this section, we detail the evaluation results on the Apple Aluminum keyboard.

4.6.1 Verification Time.

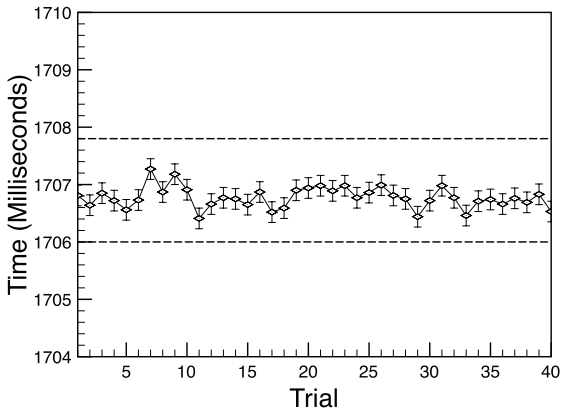

Figure 4.16: Verification Time

Figure 4.16 shows the verification time for 40 trials. In each trial, the verifier measures the entire verification time between sending a nonce to the Apple Aluminum Keyboard and receiving the checksum result from the keyboard. The average verification time of the 40 trials is 1706.77 ms while the standard deviation is only 0.18 ms.

4.6.2 USB Communication Overhead.

In this experiment, the verifier first sends an attestation request to the Apple Aluminum Keyboard. Upon receiving a request from the verifier, the verification function on the Apple Aluminum Keyboard returns an 8-byte value to the verifier immediately without computing the checksum. To obtain accurate experimental re-

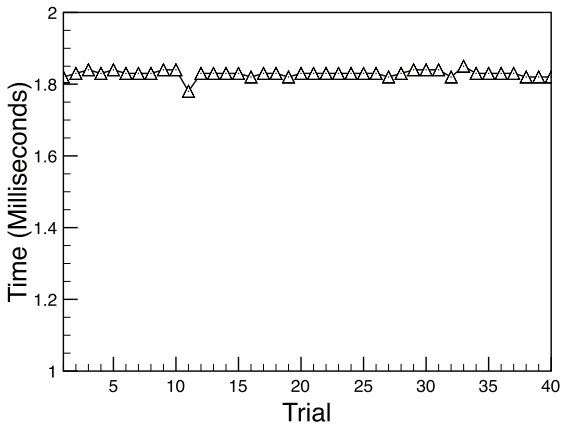

Figure 4.17: USB Communication Overhead

sults, the verifier measures the entire time of 1000 runs of the communication in each trial. Figure 4.17 shows the average communication time of the 1000 runs in each trial. The average value of the USB communication overhead for all the experiments is 1.83 ms and the standard deviation is only 0.01 ms.

4.6.3 Analysis

The experimental results show that the verification procedure is very stable. As shown in Figure 4.16, the verification time for all 40 trials varies from about 1706 ms to about 1708 ms. An attacker cannot hide malicious code from an attestation unless the malicious code computes the correct checksum result with a computation overhead less than 3 ms, which is only about 0.2 percent of the verification time. This kind of attack is extremely challenging for the attacker since there is not any free space left in program or data memory. Also, the experimental results show that the communication overhead does not affect the detection of the computation overhead caused by malicious code, since the communication is also very efficient and stable.

4.7 Integration: A Malware-Free Operation Environment

We can integrate VIPER with on-demand I/O isolation (Section 2.3) to establish a malware-free operation environment with trusted I/O (isolated I/O with trusted I/O peripherals). Figure 4.18 shows the system architecture of such an isolated malware-free operation environment on commodity computers.

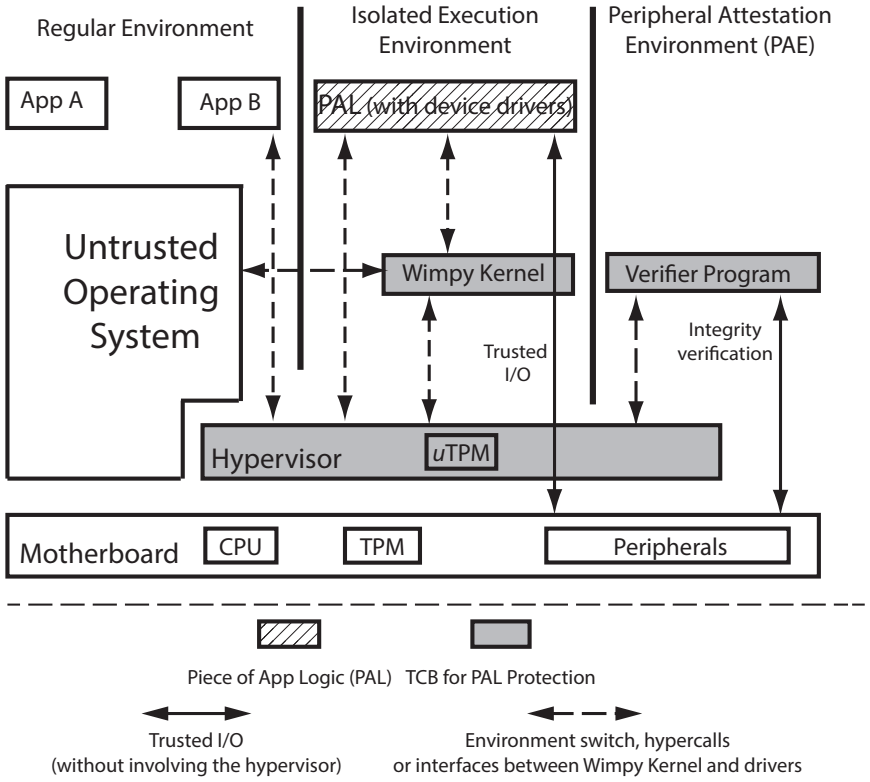

Figure 4.18: System Architecture of a Malware-Free Operation Environment with Trusted I/O.

Architecture As shown in Figure 4.18, in the isolated execution environment, a wimpy kernel [127] (Section 2.3) provides on-demand I/O isolation to a PAL.

To guarantee the absence of malware in peripherals, a peripheral attestation environment (PAE) is established and isolated from the regular environment and the isolated execution environment. In the PAE, a verifier program is responsible for verifying the integrity of peripherals' firmware using the VIPER attestation mechanism.

An alternative design is to include the verifier program in the isolated execution environment. However, such a design will increase the complexity of the code running inside the isolated execution environment. Establishing a separate isolated execution environment (PAE) for the verifier program is an on-demand *add-on* approach. Only when the PAL needs to access I/O peripherals, the PAE is included in the TCB.

Assumptions Please note that we assume that the chips on the I/O channel (i.e., the Northbridge, the Southbridge, the USB host controller, the USB hub) do not have firmware and focus on verifying the firmware integrity of the end-point peripherals (e.g., a keyboard connected to a USB hub, a NIC connected to the PCI). We also assume that the verifier program has the information of all peripherals on the commodity computer (e.g., the peripherals' memory address, the USB hierarchy, a copy of the original firmware that should run inside the peripherals).

Verifying the Integrity of Peripherals' Firmware Before enabling the PAL to access the I/O peripherals, the wimpy kernel first invokes the verifier program in the PAE (by an environment switch or a hypercall to the hypervisor that will invoke the verifier program) to verify the integrity of peripherals' firmware.

The verifier program first verifies the peripheral information to guarantee that all peripherals' configurations are not modified. The verifier program utilizes similar approaches with wimpy kernel (*outsource-and-verify*) to verify the peripheral

information. For instance, the verifier program scans through all MMIO memory mappings to prevent MMIO mapping attacks [127]. However, as the verifier program already has the expected peripheral information, the verification process is simpler. For example, because the prover program already has the information of the USB hierarchy and connected USB devices on the computer, the prover program can quickly verify the USB hierarchy and detect any hidden USB devices without the complex operations in the original wimpy kernel [127].

After verifying the peripheral information, the verifier program calls the hypervisor to (1) configure IOMMU, Nested Page Table (NPT) or Extended Page Table (EPT), the I/O port-access-interception bitmap, and the PCI eXpress (PCIe) Access Control Services (ACS) (Section 2.2) to isolate the I/O peripherals from the malicious OS and other peripherals, and (2) configure the Programmable Interrupt Controller (PIC) to temporarily disable interrupts from all peripherals.

If the I/O peripherals are not on the PCIe bus or the PCIe ACS is not available on the computer, the I/O peripherals might be able to engage in peer-to-peer communication with other peripherals by avoiding the IOMMU (see Section 2.2). Consequently, if an I/O peripheral under attestation is in such a peer-to-peer bus network (e.g., in the south-bridge), a faster peripheral in the same peer-to-peer bus network could help the I/O peripheral under attestation compute the expected checksum (a local proxy attack). In addition, after the I/O peripheral has been verified, malware in other peripherals in the same peer-to-peer network might compromise the verified I/O peripheral by exploiting vulnerabilities in the firmware of the I/O peripheral. Therefore, to prevent proxy attacks and run-time attacks from other peripherals, all following peripherals have to be isolated from the malicious OS and other peripherals on the computer and be associated with the PAE the isolated execution environment: (1) the I/O peripherals requested by the PAL; (2) and all other peripherals (that cannot be isolated by configuring the PCIe ACS) in the

same peer-to-peer bus network with the I/O peripherals requested by the PAL on the motherboard. The verifier program then verifies the firmware integrity of all associated peripherals to guarantee the absence of malware on these peripherals.

After verifying the integrity of the associated peripherals' firmware, the verifier program invokes the wimpy kernel (via an environment switch or a hypercall to the hypervisor that will invoke the wimpy kernel) and informs the wimpy kernel that the firmware integrity of I/O peripherals (and other peripherals in the same peer-to-peer bus network) has been verified. Note that during and after integrity verification, all verified peripherals are isolated from the malicious OS and other peripherals. In addition, after integrity verification, the wimpy kernel does not need to verify the peripheral information (e.g., the USB hierarchy, the MMIO address) again as the verifier program already verifies these values. To establish I/O isolation, the wimpy kernel also enables interrupts from the I/O peripherals, and establishes interrupt isolations. The wimpy kernel then enables the PAL to access the I/O peripherals with the guarantee of I/O isolation and the absence of malware in peripherals' firmware. In this system, a user needs to verify the integrity of the isolated PAL and the entire TCB for PAL protection (including the hypervisor, the wimpy kernel, and the verifier program) using an external verifier device [127].

Limitations To prevent proxy attacks, the wimpy kernel might isolate not only the I/O peripherals but also other peripherals in the same peer-to-peer bus network with the I/O peripherals. However, a malicious OS might need to access the isolated peripherals for non-sensitive operations. For instance, when a PAL requests access to the keyboard, wimpy kernel establishes isolated I/O to access the keyboard for the PAL. Because the keyboard and the NIC are in the same peer-to-peer bus network (PCIe ACS is not available), the wimpy kernel also isolates the NIC from the malicious OS and other peripherals; however, when the system switches

back to the regular environment, the OS might need to access the NIC for network communication.

To address the problem, the wimpy kernel can temporarily disable the I/O isolation to enable the OS to access the isolated peripherals when the system switches back to the regular environment (the wimpy kernel might also need to reset the isolated I/O peripherals to clean any temporary security-sensitive data in I/O peripherals). When the I/O isolation is disabled, a malicious OS could insert malware into the I/O peripherals (or other peripherals that need to be isolated). Therefore, after the system switches back to the isolated execution environment, all peripherals are no longer trusted for the wimpy kernel and the PAL; consequently, the wimpy kernel needs to invoke the verifier program (to guarantee the absence of malware in peripherals' firmware) and establishes I/O isolation for the PAL again. Frequently performing peripheral firmware integrity verification might cause a high overhead for the execution of the PAL. This problem is caused by the peer-to-peer communication feature available on modern motherboards. If the peer-to-peer communication could be prevented, the wimpy kernel only needs to isolate the I/O peripherals without needing to invoke the verifier program to verify the firmware integrity frequently.

4.8 Discussion

We now discuss open problems, limitations, and known issues with VIPER.

It remains an open problem to prove that a program of any appreciable complexity is time-optimal. This has been a significant hurdle for all software-based attestation proposals to date. Two requirements have proven especially challenging: (1) The Checksum algorithm design does not have any flaws that allow an attacker to obtain the expected result with less than the expected data available.

(2) The code design of the checksum algorithm must require precisely the smallest number of cycles to complete.

A primary goal of the present work has been to suggest that additional sources of asymmetry may be viable primitives for software-based attestation, and that these other sources of asymmetry are easier to quantify. We used the asymmetry of the latencies from CPU-to-peripheral, as compared to the latencies from peripheral-to-proxy.

Interestingly, we also face the challenge of being unsure as to whether our attack implementations are optimal. For example, the theoretically fastest RTT for an Ethernet frame is significantly shorter than the RTT that we observed with our implementation of a proxy attack. We believe our VIPER prototype to be secure against proxy attacks facing the empirically measured proxy attack time, but an attacker who can communicate at gigabit Ethernet's theoretical speeds may have an advantage. In practice, this limitation is minor, since we primarily consider attacks arriving via multiple network hops on the Internet, which even today entails several orders of magnitude higher latency.

Full System Verification An overview of full-system verification with VIPER is presented in Section 4.2. While theoretically straightforward, learning the expected configuration of all programmable elements of a computer system is a considerable practical challenge. Today's vendor ecosystem does not propagate such information, and we were unable to obtain enough information about a complete system to attempt such verification. We suggest such endeavors as fertile ground for future research.

Quiescing the System to Enable Verification We ran our experiments (Section 4.4) on hardware that is several generations removed from the latest systems.

Our motivation for doing so was in reducing the amount of system activity for which we could not account. Multicore processors, system management interrupts (SMIs), and platform management tools such as OPMA, IPMI, or Intel AMT are all capable of generating system activity that may be difficult or impossible to quantify from a vantage point on a single platform CPU. We view this as one instance of the challenges faced in attempting to identify expected or baseline system behavior with high-assurance.

It is also worth mentioning that modern platforms include significant support for power management. While the logic that governs these operations is itself in scope for verification, one way to achieve necessary levels of system quiescence may be to power down peripherals that cause (possibly benign) interference. Failure to respond to power-down requests is itself an indictment of a particular peripheral.

Hardware Variability Nightingale et al. study 1,000,000 consumer PCs and find that a full 1% run outside 0.5% of their rated clock speed, even when intentional overclocking is taken into consideration [71]. This level of variability may complicate the process of establishing baseline, or expected, behavior for VIPER on a particular platform. Additional investigation is warranted.

Why Not Hide? Attackers may be incentivized to infect peripherals with malware that deletes itself when interrogated for verification. In principle the system must have had a vulnerability somewhere, and the attacker may be able to reinfect the system post-verification. However, it is not easy to correlate infected firmware in one device with a vulnerability in that device's expected firmware, e.g., the vulnerability may have been in the OS and the driver that updates device firmware may have been compromised. The malware-free operation environment with on-

demand I/O offers the mechanism to solve Time Of Check To Time Of Use (TOCTTOU) problem.

Network Infrastructure as the Verifier In an enterprise network it may be reasonable to let network infrastructure such as gateway systems act as verifiers. While feasible for verifying NIC firmware, this approach does not trivially allow verification of all other peripherals in a full system.

Denial of Service One malicious device can easily create excessive bus traffic such that verification of another device would fail. This can be interpreted as potentially being a form of inter-device "blackmail", but ultimately one has detected that something is amiss in the system. Localizing the source of the attack is a secondary problem.

4.9 Summary

Attackers have elevated malware to a new frontier: executing invisibly on devices within a computer system. Such malware can exploit DMA to compromise the OS or misuse PCI buses to compromise other devices. We address the research challenge of how to reliably detect such malware. This work shows how we extend previous software-based attestation mechanisms to defend against proxy attacks, where the untrusted system obtains help for computing the time-critical checksum from a remote party. By harnessing the inherent properties of PCI buses, we have developed a new approach for software-based attestation that can prevent the proxy attack and simultaneously achieve lower verification time overhead. We anticipate that our proposed techniques will make software-based attestation practical on current platforms and provide uncircumventable advantages to defenders

without relying on specialized hardware.

Chapter 5

MiniBox: A Two-Way Sandbox for x86 Native Code

Mobile devices are starting to increasingly change the world. Although the hardware performance of mobile devices has been significantly improved over the past decade, mobile devices are still *resource-poor* relative to server-level hardware platforms. In addition, energy consumption is alway critical for mobile devices. Consumers would never complain a mobile device's battery lasts too long. However, compute-intensive or data intensive tasks on mobile devices (e.g., speech recognition or face recognition) can dramatically decrease the battery life of mobile devices. Thus, compute-intensive and data-intensive operations are often offloaded from mobile devices to the remote cloud data center (e.g., via the wireless network). For example, Google Glass might offload its face recognition tasks to a remote cloud server. Apple Siri voice recognition is another example of offloading the compute-intensive task to a remote cloud server. However, humans are sensitive to delays. The Round Trip Time (RTT) between a mobile device and a remote commercial cloud service (e.g., Amazon EC2) over the Wide Area Network (WAN) is

unsatisfactory for delay-sensitive tasks (e.g., real-time speech translation).

Cloudlets Satyanarayanan et al. propose *Cloudlets* [86, 87], a middle tier of a mobile device-Cloudlet-cloud hierarchy, in which mobile devices can offload its tasks to nearby public computers (Cloudlets) to avoid the high and unstable communication latency with a remote cloud server. For example, when a user with a Google glass is having a rest in a cafe, the user's Google glass may automatically connect with the public computer in the cafe via a local wireless network and offload the execution of compute-intensive voice recognition task to the public computer. The communication latency in the local wireless network is much lower than the WAN latency and hence the user benefits from the Cloudlet service.

Furthermore, mobile devices, especially wearable devices usually provide limited user-device interaction interfaces (e.g., a small display or a small keypad for user input). In contrast, public Cloudlet computers can provide rich user-interaction I/O peripherals (e.g., keyboard or high resolution monitor). Thus, a user might also prefer to offload operations from a mobile device to a Cloudlet computer, perform complex operations (e.g., editing document) that are not convenient on small wearable devices.

Security Issues However, there are practical issues for deploying such a middle tier. When such Cloudlet is widely deployed, how can we protect the Cloudlet public computers from being compromised by a large number of *untrusted* offloaded programs from users' mobile devices? In the meantime, users may offload security-sensitive data (e.g., face images or voice data) to Cloudlet computers. If a Cloudlet public computer is compromised, the adversary can breach users' privacy and secrecy. How can we protect the users' security-sensitive data on the Cloudlet computers? When a user edits an document she offloads to the Cloudlet computer,

how can we protect the user's document from being leaked?

Satyanarayanan et al. propose a Virtual Machine (VM) based system architecture to provide a clean and isolated environment (an initialized Virtual Machine in a clean state) for each offloaded program in Cloudlets. In this way, the offloaded program (including the user's security-sensitive data) is isolated from other offload programs through VMs. However, such a VM-based architecture has a large TCB. An adversary can exploit vulnerabilities in the VM to control the Cloudlet public computer, then access the security-sensitive data from other users.

Two-Way Sandbox We think the security model of Cloudlets, and argue that a two-way sandbox is desirable in Cloudlet public computers. The two-way sandbox not only protects a benign OS from a misbehaving program offloaded by a user (*OS protection*) but also protects one offloaded program from a malicious OS (*application protection*). Researchers have explored several approaches for either protecting the OS from an *untrusted* application [27, 51, 54, 122] or protecting security-sensitive applications (or security-sensitive PALs) from a malicious OS [12, 20, 21, 23, 24, 26, 33, 38, 52, 66, 67, 100–102, 105, 120]. Unfortunately, none of these schemes provides two-way protection, and many challenges remain to design a two-way sandbox.

TrustVisor [66] and Intel Software Guard Extensions (Intel SGX) [4, 31, 37] are examples of systems that provide efficient memory space isolation mechanisms to protect a Security-Sensitive Portion of an Application (security-sensitive PAL) from a malicious OS (Figure 5.1-B). On TrustVisor or Intel SGX, memory access from the OS to the security-sensitive PAL or from the security-sensitive PAL to the OS is disabled by an isolation module, which is a hypervisor (on TrustVisor) or CPU hardware extensions (on Intel SGX). However, the Non-Sensitive Portion of an Application (non-sensitive PAL) is not isolated from the OS, and the non-

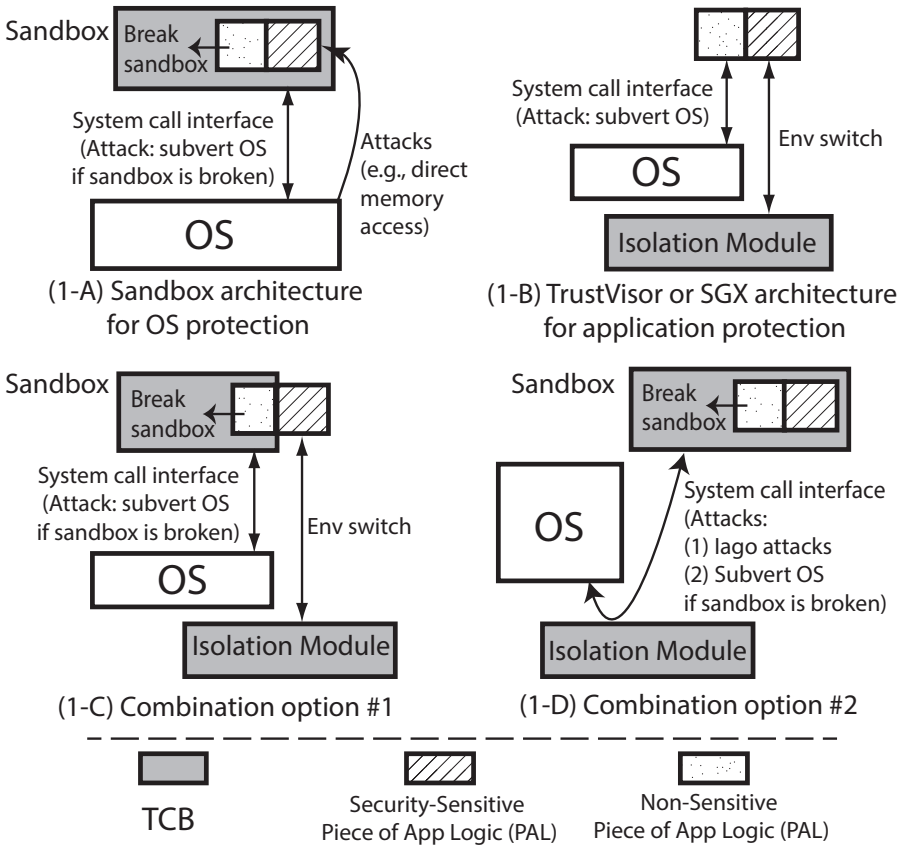

Figure 5.1: Sandbox Architecture, TrustVisor or Intel SGX Architecture, and Combination Options.

sensitive PAL may contain malware that can compromise the OS.

Google Native Client (NaCl) [122] and Microsoft Drawbridge [27, 81] are examples of application-layer one-way sandboxes for native code. We found that combining an application-layer sandbox and an efficient memory space isolation mechanism is promising for the two-way sandbox design. However, it it not straightforward. Figure 5.1-C and 5.1-D show two combination options. In option #1, the security-sensitive PAL runs in an isolated memory space while a sandbox confines

the non-sensitive PAL. However, in this design application developers need to split the application into security-sensitive and non-sensitive PALs, requiring substantial porting effort. In option #2, the sandbox is included inside the isolated memory space to avoid porting. The isolation module forwards system calls (from the sandbox) to the OS. However, there are several issues with this option. First, because the sandbox is complex and exposes a large interface to the application, a malicious application may exploit vulnerabilities in the sandbox and in turn subvert the OS. Second, a malicious OS may be able to compromise the application through Iago attacks [19]. In Iago attacks, a malicious OS can subvert a protected process by returning a carefully chosen sequence of return values to system calls. For instance, if a malicious OS returns a memory address that is in the application's stack memory for an *mmap* system call, sensitive data (e.g., a return address) in the stack may subsequently be overwritten by the mapped data. Finally, because the OS is isolated from the sandbox and the application, it is challenging to support the application execution in an isolated memory space. Thus, both options have obvious shortcomings and we shall not choose them for the two-way sandbox design.

MiniBox In this chapter, we present MiniBox, a two-way sandbox for x86 native applications. Leveraging a hypervisor-based memory isolation mechanism (proposed by TrustVisor) and a mature one-way sandbox (NaCl), MiniBox offers efficient two-way protection. MiniBox splits the NaCl sandbox into OS protection modules (software modules performing OS protection) and service runtime (software modules supporting application execution), runs the service runtime and the application in an isolated memory space (Section 5.2.1), and exposes a minimized and secure communication interface between the OS protection modules and the application (Section 5.2.2). MiniBox also splits the system call interface available to the isolated application as sensitive calls (the calls that may cause Iago attacks)

and non-sensitive calls (the calls that cannot cause Iago attacks), and protects the application against Iago attacks by handling sensitive calls inside the service runtime in the isolated memory space (Section 5.2). MiniBox also provides secure file I/O for the application (Section 5.2.6). Integrated with the on-demand isolated I/O mechanism (recall Section 2.3) and VIPER (recall Chapter 4), MiniBox not only offers trusted I/O to the isolated application, but also prevents malware from spreading to either side through peripherals by verifying the integrity of peripherals' firmware in both sides (Section 5.5). Using a special toolchain, application developers can concentrate on application development with small porting effort (Section 5.4). We implement a MiniBox prototype based on the Google Native Client (NaCl) [122] open source project and the TrustVisor hypervisor [66, 111] (Section 5.3), and port several applications to MiniBox. Evaluation results show that MiniBox is practical and provides an efficient execution environment for isolated applications (Section 5.4).

Contributions

1. We design, implement, and evaluate MiniBox, the first attempt toward a practical two-way sandbox for x86 native applications.

2. MiniBox demonstrates it is possible to provide a minimized and secure communication interface between OS protection modules and the application to protect against each other.

3. MiniBox demonstrates it is possible to protect against Iago attacks, and provide an efficient execution environment with secure file I/O for the application.

5.1 Assumptions and Attacker Model

Assumptions We assume that the attacker cannot conduct physical attacks against the hardware units (e.g., CPU and TPM). We assume that the attacker cannot break standard cryptographic primitives and that the TCB of MiniBox is free of vulnerabilities. For application protection, we also assume that the application does not have any memory safety bugs (e.g., buffer overflows) or insecure designs. One example of the insecure designs is that an application seeds a pseudo-random number generator by the return value of a system call handled by the untrusted OS. It is the developer's responsibility to take measures to eliminate memory safety bugs or insecure designs. For OS protection, we assume that the system call interface that the OS protection modules expose to the application (a subset of the OS system call call interface) is free of vulnerabilities, and that the OS does not have concurrency vulnerabilities [114] in system call wrappers.

Attacker Model For Application Protection We assume that the attacker can execute arbitrary code on the OS. For example, the attacker may compromise and control the OS, and then attempt to tamper with the protected application by accessing the application memory contents or handling the system calls of the application in malicious ways (Iago attacks). The attacker may attempt to inject malicious code into the application binary or into the service runtime binary before the application runs in an isolated memory space *without being detected*. The attacker may subvert DMA-capable devices on the platform in an attempt to modify memory contents through DMA. The attacker may also attempt to access security-sensitive files of the application. However, we do not prevent denial of service attacks. Finally we do not prevent side-channel attacks [125].

Attacker Model For OS Protection The untrusted application may attempt to subvert the hypervisor or break out of the hypervisor-based memory isolation. The application may also attempt to read or modify sensitive files that do not belong to the application on the system. The application may attempt to subvert the OS by making arbitrary system calls with carefully-chosen parameters.

5.2 System Design

5.2.1 MiniBox Architecture

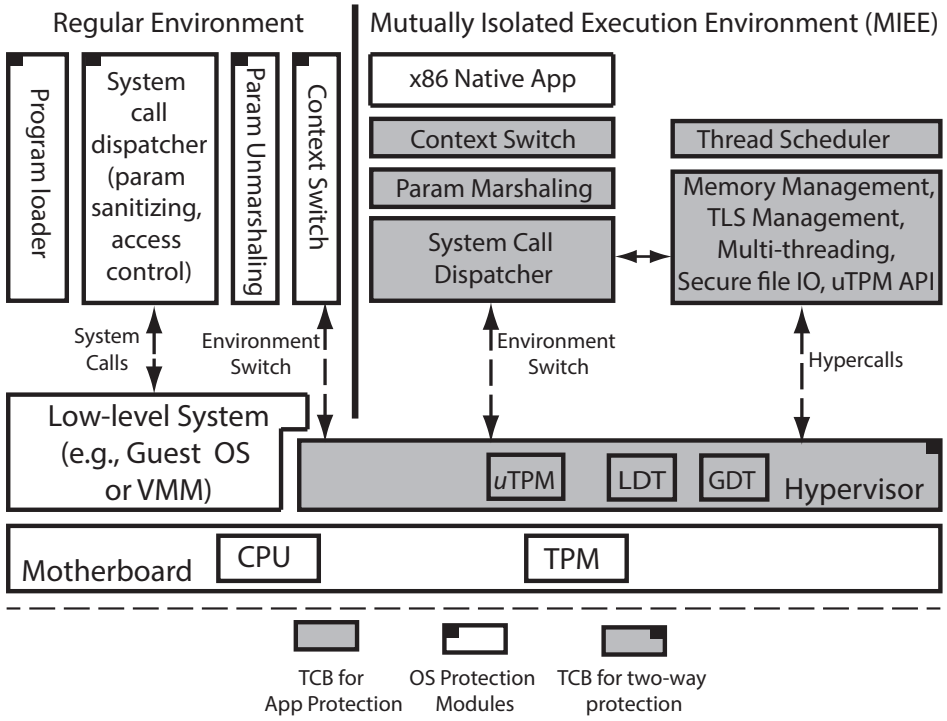

Figure 5.2: MiniBox System Architecture.

Figure 5.2 shows the MiniBox architecture. As shown in this figure, a hypervisor underpins the system. The hypervisor sets up the two-way memory space

isolation between the Mutually Isolated Execution Environment (MIEE) and the regular environment, and creates a μTPM instance for the MIEE.

On MiniBox, the hypervisor and a service runtime in the MIEE comprise the runtime TCB for application protection. In the MIEE, beyond the x86 native application, a service runtime is included, containing: a context switch module that stores and switches thread contexts between the application and the service runtime; a system call dispatcher that distinguishes between non-sensitive and sensitive calls, calls handlers in the MIEE for sensitive calls, or invokes the parameter marshaling module for non-sensitive calls; a parameter marshaling module that prepares parameter information for non-sensitive calls (for the hypervisor); system call handlers for handling sensitive calls; and a thread scheduler that schedules the execution of multiple threads comprising an application; device drivers to access I/O peripherals (recall Section 2.3). In sensitive call handlers, the service runtime supports dynamic memory management, thread local storage management, multi-threading management, secure file I/O, and μTPM API. Integrated with the isolated I/O mechanism [127] (recall Section 2.3) and VIPER (Chapter 4), MiniBox can provide trusted I/O (isolated I/O with trusted peripherals) to the isolated application. We present how to integrate MiniBox with isolated I/O and VIPER to provide such trusted I/O in Section 2.3.

On MiniBox, the OS protection modules include a user-level program loader, a context switch module, a parameter unmarshaling module, a system calls dispatcher in the regular environment, and a kernel module containing a prover code and a wimpy kernel. In the regular environment, the user-level program loader sets up the MIEE and loads the application into the MIEE; the context switch module stores and restores the thread context of the regular environment during environment switches between the regular environment and MIEE; the parameter unmarshaling module unmarshals system call parameters; and the system call dispatcher

confines the system call interface exposed to the application (allowing only a subset of the OS system calls), sanitizes the system call parameters, conducts access control to constrain the file access of the application, and forwards the non-sensitive system calls to corresponding handlers in the regular environment.

Finally, MiniBox adopts TrustVisor's integrity measurement (recall Section 2.3) to enable a remote verifier to verify the integrity of the hypervisor, the service runtime, and the isolated application. In this way, MiniBox prevents adversaries from injecting malicious code into the hypervisor, the service runtime or the application before the memory isolation is established without being detected. *This is also the reason that the program loader is not in the TCB for application protection.*

5.2.2 Communication Interfaces

The MiniBox hypervisor exposes a small interface to the rest of the system. MiniBox minimizes and secures the communication interface between OS protection modules and the application to protect against each other.

Hypervisor Interface Other than passing system call information between the MIEE and the regular environment, the hypervisor exposes a small interface (i.e., only several hypercalls) to the rest of the system. Thus, assuming the small hypercall interface is free of vulnerabilities, malicious code in the rest of the system cannot compromise the hypervisor or break out of the hypervisor-based memory isolation.

Minimizing Communication Interface On MiniBox, the communication interface between OS protection modules and the application consists of only the program loader and the system call interface. Because privileged instructions cannot break out of the hypervisor-based memory isolation, the NaCl validator (that val-

idates that the application binary does not contain privileged instructions) is not included in MiniBox, *which significantly reduces the interface exposed to the application*. Without the validator, privileged instructions in the application can break out of the segmentation-based isolation and compromise the service runtime in the MIEE. However, a malicious service runtime in the MIEE cannot break out of the hypervisor-based memory isolation.

Secure Communication On MiniBox, the hypervisor is the only communication channel between the regular environment and the MIEE. Each non-sensitive system call causes environment switches between the MIEE and the regular environment. For each environment switch from the MIEE out to the regular environment, the parameter marshaling module in the MIEE updates the parameter information of the system call that will be used by the hypervisor for copying parameters between the two environments. However, the parameter marshaling module in the MIEE cannot specify where the parameters will be stored in the regular environment. The hypervisor copies the system call parameters to a parameter buffer in the regular environment, and constrains the total data size of system call parameters (to prevent buffer overflow attacks). In this way, malicious code in the MIEE cannot overwrite critical data (e.g., stack contents) in the regular environment. To prevent a misbehaving application from sending arbitrary system call parameters to the regular environment, the system call dispatcher in the regular environment checks the system call parameters before sending them to the OS. For example, the system call dispatcher checks the value of every pointer parameter and guarantees that it is safe to access the memory space the parameter points to. If a check fails, the system call dispatcher returns an error code without calling the corresponding system call handler.

After the system call is handled, the system call dispatcher copies return values

to the parameter buffer in the regular environment and triggers the environment switch back to the MIEE. When MiniBox switches from the regular environment *back to* the MIEE, the hypervisor uses the same parameter information specified by the MIEE to copy parameters from the parameter buffer in regular environment to the MIEE. This prevents malware in the regular environment from attempting to compromise MIEEs by manipulating parameter information.

5.2.3 Dynamic Memory Management

MiniBox supports three system calls (`sysbrk`, `mmap`, and `munmap`) to provide dynamic memory management for the application running inside the MIEE. To prevent the OS from returning arbitrary memory addresses for the `sysbrk` or `mmap` system calls (Iago attacks) or removing arbitrary data memory pages from the MIEE, memory management system calls are handled inside the MIEE.

Design One naive design is pre-allocating and registering a large amount of data memory in the MIEE as data memory for the application. This design has low execution time overhead, but it wastes memory and is inflexible. Another design is allowing the hypervisor to allocate memory pages as the application's data memory. However, the MiniBox hypervisor does not support swapping of memory pages to disk, and cannot be sure that pages marked as unused by the guest OS are actually present in memory. To resolve this issue, we design the system call handlers that request more data memory (i.e., `sysbrk` and `mmap`) in two modules: one in each of the isolated and regular environments. When the application requests more data memory but the requested data memory is not in the MIEE, the system call handler in the MIEE calls the module in the regular environment that allocates the memory page(s) and writes zero to them to ensure that the new memory page(s) are loaded into physical memory, and then returns to the handler inside the MIEE. The system

call handler inside the MIEE then makes a hypercall to the hypervisor to add the new memory page(s) to the MIEE. The `munmap` handler inside the MIEE makes a hypercall to unregister memory from the MIEE.

Security Protection To prevent Iago attacks caused by `mmap` or `sysbrk`, the hypervisor checks that the newly registered pages are not already registered to the MIEE (so that the malicious OS cannot overwrite stack contents of the application in the MIEE). To prevent leakage of sensitive data in either direction, the MiniBox hypervisor *zeroes* memory pages during registration and unregistration. To prevent a misbehaving or malicious application from adding privileged data pages (e.g., kernel pages) into MIEE, the hypervisor checks that the newly registered pages are user-level memory pages that are in ring 3, and correspond to the same OS process that originally registered the MIEE. Presently MiniBox does not allow additional memory to be mapped as executable, as this represents a significant increase in attack surface. Thus, the hypervisor checks that the requested memory pages are data pages that are not executable. In *data* memory page unregistration, the hypervisor checks that the unregistered memory pages are data pages that are already registered to the MIEE.

5.2.4 Thread Local Storage Management

Background On 32-bit Linux, the native code on vanilla NaCl stores the memory address of its Thread Local Storage (TLS) as the base address of a segment descriptor in the Local Descriptor Table (LDT) [48]. During program initialization or when a new thread is created, `tls_init` system call initializes the TLS base address and updates the appropriate LDT entry. During execution, the `tls_get` system call is frequently called to get the TLS base address.

Design Because the TLS and LDT represent critical configuration data, MiniBox handles the `tls_init` and `tls_get` entirely within the MIEE. The MiniBox hypervisor creates an LDT instance for each MIEE and supports a hypercall interface to the MIEE to handle `tls_init` system call. MiniBox supports caching the latest TLS address inside the MIEE, so that the `tls_get` handler can quickly return the latest TLS base address to the application without calling the hypervisor.

5.2.5 Multi-threading

Background NaCl applies a 1:1 thread model (i.e., creating a kernel thread for each Native Module user-level thread) and uses the OS to handle thread-related system calls (e.g., thread synchronization system calls) and schedule the execution of Native Module threads.

Design If MiniBox applies the same multi-threading mechanism, the OS controls the thread context of the application threads. A malicious OS could break the Control Flow Integrity (CFI) [1–3] of the isolated application by changing the thread context. Also, when the OS handles all thread synchronization system calls, a malicious OS could break the application CFI by arbitrarily changing application thread states. To protect the application thread context from being accessed by the OS, MiniBox can store the thread context in the MIEE and never leak it out of the MIEE. Also, the service runtime in the MIEE can verify the thread synchronization results by duplicating all supported thread synchronization system call handlers. In this design, all thread context and the application CFI are protected from a malicious OS. However, the complexity of this design is comparable to implementing the multi-threading operations within the MIEE. Also, if thread-related system calls are handled by the OS, the environment switches caused by thread-related system calls will increase the overhead of application execution in the MIEE. Thus,

to reduce execution overhead and avoid duplicated operations, MiniBox supports multi-threaded application execution via a user-level multi-threading mechanism entirely within the MIEE. System calls to create, exit and synchronize threads are handled in the MIEE.

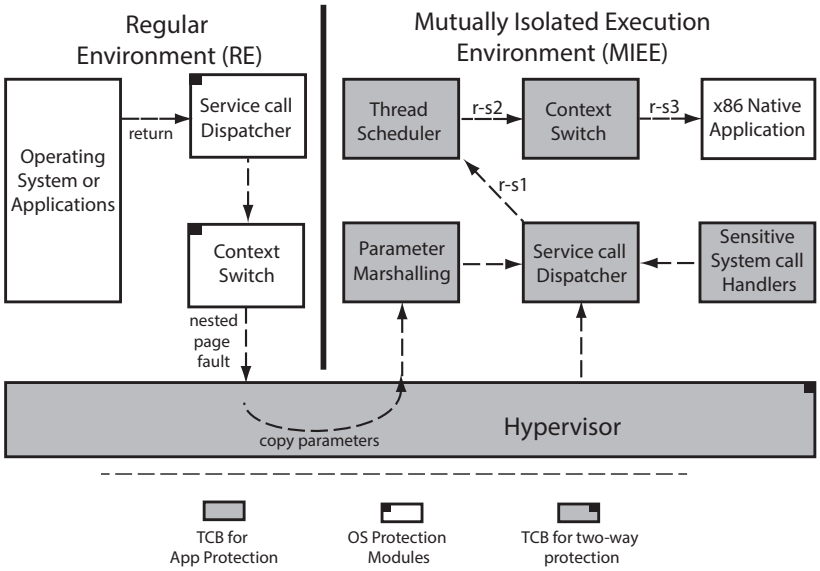

Figure 5.3: System Call Return Flow.

Thread Scheduler MiniBox provides a thread scheduler to schedule the thread execution of the application in the MIEE. The thread scheduler is invoked each time there is a call from an entry of the *Trampoline Table* (recall Section 2.4). After the call is handled, control returns to the *thread scheduler* inside the MIEE before the context switch module is invoked (r-s1 in Figure 5.3). Before scheduling the execution of application threads, the thread scheduler first obtains the thread ID of the thread that made the system call, and saves the thread context in the corresponding thread context data structure. The scheduler checks the state of each thread, and schedules the execution of runnable threads using a round-robin

algorithm. The thread scheduler finally calls the context switch module (r-s2 in Figure 5.3), which resumes the execution of the scheduled thread by restoring the thread context of the scheduled thread and jumping to the Springboard within the application (r-s3 in Figure 5.3).

Note that, before calling the thread scheduler, the hypervisor and the parameter marshaling module in the MIEE have already unmarshaled the system call parameters, and copied the returned parameters to the application's stack or data memory.

5.2.6 Secure File I/O

On MiniBox, the application running in the MIEE still needs to access the file system in the regular environment, so the file system calls are forwarded to the OS. However, to protect the file contents and metadata of an isolated application, MiniBox supports secure file I/O for applications running in the MIEE through five system calls: `secure_write`, `secure_read`, `secure_open`, `secure_close`, `create_siokey`. The five system calls are handled in the MIEE.

Confidentiality and Integrity `secure_write` encrypts the data written by the application (with a symmetric secret key) and sends the encrypted data to the general file I/O, while `secure_read` decrypts the data and returns the decrypted data to the application in the MIEE. In `secure_write` and `secure_read`, the data is written or read by a chain of blocks of a constant size. To protect the integrity of file contents and file metadata, including file name and path, a hash tree is constructed and computed over the blocks of file contents and file metadata in the MIEE (this approach has been demonstrated in the Trusted Database System [63], VPFS [116] and jVPFS [117]). A HMAC of the master hash is computed in the MIEE and stored at the end of the file (as file contents). When a file created by secure file I/O

is opened, `secure_open` reads the HMAC and verifies the integrity of the file contents and metadata by reconstructing the hash tree. `secure_open` stores the hash tree in the MIEE. When a data block is read, `secure_read` verifies the integrity of the data block based on the stored hash tree. When file contents are modified, `secure_write` updates the hash tree stored in the MIEE. When a file is closed, `secure_close` recomputes the master hash and the HMAC, and stores the updated HMAC at end of the file. This allows the integrity of file contents and file metadata to be verified. The attacker cannot remove, add, or replace data blocks in the file because any changes will invalidate the HMAC. The attacker cannot replace the file with other files that are created by the same application running in the MIEE either because file metadata is also verified.

Rollback Prevention (Freshness) MiniBox adds a counter in each HMAC computation to guarantee freshness of files stored through the secure file I/O. The counter is sealed by the μTPM. Because the μTPM cannot provide freshness for sealed contents, the integrity of the counter is measured every time the same application runs in the MIEE (the measurement result is extended into μPCR for remote attestation). This allows a verifier to verify the freshness during remote attestation.

Key Management Before using secure file I/O, the application running in the MIEE must call `create_siokey` to create the secret keys used in secure file I/O (i.e., a symmetric encryption key and a HMAC key). The application specifies the file name and file path for storing the keys when calling `create_siokey`. `Create_siokey` first checks if the file already exists. If not, `create_siokey` creates new secret keys, seals the secret keys with the current μPCR values. Then it stores the sealed secret keys in the file, and returns the key ID to application. If the file already exists (i.e., keys are already created), `create_siokey` reads the sealed

keys from the untrusted file system, unseals the keys and returns the key ID to the application.

Access Control and Migration Because the secret keys are sealed with the current μPCR (i.e., the integrity measurement of the application), the sealed keys can only be unsealed by the μTPM when the same application runs in the MIEE. Thus, any data encrypted through secure File I/O can only be decrypted and verified when the same application runs in the MIEE. To share the sensitive files with other applications running in the MIEE (e.g., an updated version of the application), the application can seal the secret keys with the integrity measurement result of other applications, and share the sealed keys to other applications. Then, other applications running in the MIEE can unseal the secret keys (using `create_siokey`) and access the secret files.

Cache Buffer On MiniBox, environment switches between the MIEE and the regular environment cause high overhead in file I/O (Section 5.4). To reduce the number of environment switches, MiniBox creates a cache buffer in the MIEE for each opened file descriptor. Both general file I/O and secure file I/O benefit from the cache buffer because the number of environment switches is reduced.

5.2.7 MIEE Preemption and Scheduling

As described in Section 5.2.5, MiniBox does not preempt an application thread running in the MIEE. However, if an application thread is in an endless loop, the thread will not freeze the entire system because the MIEE is preemptive on MiniBox. When the system switches into a MIEE, the hypervisor starts a timer for the MIEE and preempts the code execution in the MIEE when the timer expires. After preempting the MIEE, the hypervisor stores the MIEE context and transfers control

to the regular environment by simulating a special system call (i.e., *MIEE_sleep*). The *MIEE_sleep* handler sleeps for a while and then calls the hypervisor to resume the code execution in the MIEE. In this way, the hypervisor transfers the control to the OS, which can schedule the execution of other processes. When multiple MIEEs are registered (one MIEE in each process), the OS can implicitly schedule the execution of multiple MIEEs by scheduling process execution. However, the question is how much CPU time should be assigned to each MIEE by the hypervisor. One design is that the hypervisor exposes a hypercall interface to the regular environment and the MIEE to enable the OS and the isolated application in the MIEE to configure the MIEE process priority. The hypervisor assigns CPU time to each MIEE based on the MIEE process priority.

5.2.8 Exceptions, Interrupts, and Debugging

Exceptions and Interrupts While the code in a MIEE is running, the processor cannot access exception and interrupt handlers in the OS. Thus, the hypervisor is configured to intercept exceptions (e.g., *segmentation fault*, *invalid opcode*) and Non-Maskable Interrupts (NMIs) when system runs in a MIEE. Maskable interrupts are disabled when system runs in a MIEE. When NMIs happen, the hypervisor handles NMIs and resumes the code execution in the MIEE. When an exception happens, the hypervisor first checks whether the exception is because the application in the MIEE needs more stack pages. If so, the hypervisor calls a module in the regular environment to allocate more data pages as stack pages, adds the stack pages into the MIEE, and resumes the code execution in the MIEE. If not, the hypervisor terminates the code execution in the MIEE by simulating an *Exit* system call. The *Exit* call is forwarded to the program loader, which unregisters the MIEE from the hypervisor via hypercall.

Debugging Though the MiniBox execution environment is compatible with NaCl's, the NaCl debugging tool for application development cannot be directly used on MiniBox because on MiniBox the OS cannot access the memory contents in the MIEE. However, MiniBox can be configured in a debugging mode, in which the hypervisor functionalities are disabled, and an application layer module passes parameters between the two environments. In debugging model, memory management and TLS management calls are handled by the OS. In this way, the memory isolation is disabled and application developers can use the NaCl debugging tool for MiniBox application development. An alternative way is including the NaCl debugging tool in the MIEE and supporting an interface to access the debugging tool from the regular environment. In this way, the developers can debug the application when the memory isolation is enabled.

5.2.9 Program Loader

In MiniBox, a user-level program loader prepares the service runtime for the application, loads the application binary to page-aligned memory, registers the whole thing as a MIEE through hypercalls, and finally launches the execution of the application.

Initialization After loading the application into page-aligned memory, the program loader initializes the relevant LDT for segments of the application (code, data and stack), initializes the system call parameter information for environment switch, and populates the initial thread context of the application. The program loader allocates a 32 MB stack section for each application at the high end of the application's address space. The application accepts *arguments* upon its initial invocation like a typical process. The program loader copies the arguments into the application's stack memory.

MIEE Registration Before launching the execution of the application, the program loader registers the MIEE comprising the service runtime, and the application's code, data and stack sections. During registration, the hypervisor sets up memory protection for the MIEE, instantiates a fresh μTPM instance, instantiates a GDT and LDT for the MIEE, measures the memory contents of the MIEE, and extends the measurement results into a PCR in the μTPM instance for remote attestation. Before registration, the program loader has full access permissions to the application and the service runtime. Thus, it could potentially maliciously modify the contents of the application or service runtime. *However, any such modifications or injected malicious contents will cause the μTPM's PCR to take on a different value than expected. As a result, the MIEE will be unable to generate correct μTPM Quotes in remote attestation or unseal the secret data that are sealed with the expected μTPM PCR value.*

Launching Application After registration, the program loader launches application execution by triggering the environment switch into the MIEE. Inside the MIEE, the context switch module initializes the application thread context, switches segment selector registers, and starts application execution.

MIEE Unregistration After the application completes its execution it invokes an *Exit* system call that is forwarded to the program loader. After receiving this system call, the program loader unregisters the MIEE from the hypervisor via hypercall. The MIEE data memory is zeroed and the memory protections on the MIEE's memory space are removed by the hypervisor.

5.3 Implementation

We implement a MiniBox prototype running on recent x86 multi-core systems from Intel or AMD, with 32-bit Ubuntu 10.04 LTS as the guest OS. This section describes the MiniBox implementation in details.

5.3.1 Hypervisor

The implementation of the MiniBox hypervisor is based on the public implementation of TrustVisor hypervisor (version 0.1.2) [66, 111] with support for multi-core and both AMD and Intel processors. We changed the parameter marshaling implementation [61] and added a hypercall interface for handling sensitive system calls. We added code to create new Global Descriptor Table (GDT) [48] entries and instantiate an LDT for every MIEE, and added code to handle GDT- and LDT-related operations. The original implementation of TrustVisor hypervisor has 14414 source lines of code (SLoC), computed using the sloccount tool[1]. Our implementation adds an additional 691 SLoC. Table 5.1 shows the code size of each module that we added or modified in TrustVisor hypervisor. Note that the parameter marshaling module and the extended hypercall interface are independent of the CPU manufacturer, The GDT- and LDT-related module support both AMD and Intel processors. Thus, as with TrustVisor hypevisor, the MiniBox hypervisor also works on both AMD and Intel processors.

5.3.2 Program Loader and Service Runtime

We implement the user-level program loader, the service runtime in the MIEE, the context module and the system call dispatcher in the regular environment based on the Google Native Client (NaCl) open source project (SVN revision 7110). We

[1] http://www.dwheeler.com/sloccount/

Table 5.1: Source Lines of Code (SLOC) Added to TrustVisor Hypervisor.

MiniBox hypervisor module	**SLOC**
Parameter Marshaling	201
Extended Hypercall Interface and Handlers	230
GDT and LDT-related	260
Total	691

have focused our work on the 32-bit x86 architecture, though there are no fundamental barriers to expanding to 64-bit. In the NaCl source code, we implement code to conduct MIEE registration and unregistration in 299 SLoC. We implement the service runtime in the MIEE within the NaCl source code, adding 3550 SLoC. The secure file I/O module has a large code base (1065 SLoC) because it contains cryptographic primitives for AES and HMAC. The implemented service runtime can be configured in debugging mode for application development (recall Section 5.2.8). The parameter marshaling module in the MIEE has a large codebase because it needs to be capable of preparing parameter information for the 42 different system calls that are handled in the regular environment (23 SLOC for each system call on average). The sensitive system call handlers we implemented are mainly for handling thread synchronization (e.g., mutexes, semaphores, and condition variables), which results in a larger codebase than other modules. We use a custom linker script when building the NaCl ELF loader to link the service runtime framework in page-aligned memory pages. Table 5.2 summarizes the SLOC added to Google's NaCl source code.

Table 5.2: Source Lines of Code (SLOC) of Modules Added to Google Native Client.

Module	**SLOC**
MIEE registration and unregistration	299
Context Switch in regular environment.	29
Total in regular environment.	**328**
System Call Dispatcher in MIEE	379
Parameter Marshaling in MIEE	970
Thread Synchronization in MIEE	711
Secure File I/O in MIEE	1065
Other Sensitive Call Handlers in MIEE	373
Context Switch in MIEE.	52
Total in MIEE	**3550**

5.3.3 System Calls

MiniBox adopts NaCl system call interface to expose a closed set of system call interface to the isolated application. MiniBox does not support dynamic code for the application, so NaCl dynamic code system calls are removed on MiniBox. However, MiniBox extends the NaCl system call interface with μTPM API, network I/O system calls, and secure file I/O calls, supporting a total of 75 system calls for the application.

Table 5.3 shows the system calls supported by MiniBox. The implementation entails adding header files and statically linked libraries into the NaCl Newlib toolchain, and modifying the NaCl source code to (1) add extended system call entries to the application's Trampoline Table and add corresponding parameter marshaling functions and (2) add corresponding hypercalls to the MiniBox ser-

Table 5.3: System calls supported by MiniBox. Starred calls (∗) are handled inside the MIEE or hypervisor; the remaining calls are forwarded to the regular environment and handled by the OS.

Operations	**System Calls**
μTPM∗	μTPM_PCR_Read, μTPM_PCR_Extend, μTPM_Random, μTPM_Seal, μTPM_unSeal, μTPM_PCR_Quote
Memory∗	sysbrk, mmap, munmap
TLS∗	tls_init, tls_get
Thread∗	thread_create, thread_exit, thread_nice, sched_yield
Mutex∗	mutex_create, mutex_lock, mutex_trylock, mutex_unlock
Condition∗	cond_create, cond_wait, cond_signal, cond_broad
Semaphore∗	sem_create, sem_wait, sem_post, sem_getvalue
Secure File∗	secure_read, secure_write, secure_open, secure_close, create_siokey, read_siokey
File	dup, dup2, open, close, read, write, lseek, ioctl, stat, fstat
Time	time_of_day, clock, nanosleep
Inter-Module Communication (IMC) [122]	imc_bound, imc_accept, imc_connect,imc_send, imc_recv, imc_objcreate,imc_socket
Socket	accept, bind, connect, send, recv, listen, getpeername, getsockname, getsockopt, recvmsg, recvfrom, sendmsg, sendto setsockopt, shutdown, socket, socketpair
Others	nameservice, getdents, exit, getpid, sysconf

vice runtime. MiniBox supports 17 socket system calls. The network I/O system calls are forwarded to the regular environment, because they are treated as part of the untrusted communication channel. Secure communication (e.g., SSL) can be implemented in the application layer to protect the data in network I/O.

In the MIEE, the supported thread synchronization system calls include semaphores, mutexes, and condition variables, which have the same functionality as the corresponding POSIX APIs. The thread synchronization implementation passes the internal thread synchronization test suite included in the NaCl source code. The secure file I/O calls encrypt/decrypt the data using AES with a 128-bit key in CBC mode and computes HMAC-SHA-1 using a 160-bit key.

The μTPM API is exposed to applications through system calls. The implementation entails adding μTPM header files and a statically linked μTPM library into the NaCl Newlib toolchain, and modifying the NaCl source code to (1) add μTPM API entries to the application's Trampoline Table, and (2) add corresponding hypercalls to the MiniBox service runtime framework.

5.4 Evaluation

In this section, we present the evaluations including system call overhead, file I/O overhead, network I/O, and application performance in the MIEE on MiniBox. Experiments were conducted on a Dell PowerEdge T105 server with a Quad-Core AMD Opteron Processor running at 2.3 GHz with 4 GB memory. The operating system is Ubuntu 10.04 with 32-bit kernel Linux 2.6.32.27. To obtain accurate timing results, the hypervisor does not preempt the MIEE.

5.4.1 Performance Impact

MiniBox hypervisor extends the TrustVisor with hypercall interface and modified parameter marshaling [61], neither of which affects the guest OS performance. Thus, MiniBox hypervisor imposes similar guest overhead to the TrustVisor [111]. Yee et al. [122,123] presented that the NaCl toolchain can cause significant increase in code size (2% to 57% on SPEC2000 benchmarks), but non-significant impact

on performance (on average less than 5% on SPEC2000 benchmarks).

5.4.2 Porting Effort

MiniBox uses the NaCl toolchain with extended API for application development and imposes similar porting efforts to the NaCl. Yee et al. [122,123] presented that porting an internal implemented H.264 decoders (11K lines of C code) to NaCl requires adding about twenty lines of C code, and porting Bullet[2] to NaCl took only a few hours. Compared to NaCl, MiniBox requires additional porting effort for application protection. For instance, application developers must understand the MiniBox protection mechanisms and avoid insecure application designs (recall Section 5.1). Application developers must understand the trustworthy computing abstractions exposed to every MIEE, and correctly use them.

5.4.3 MiniBox Microbenchmarks

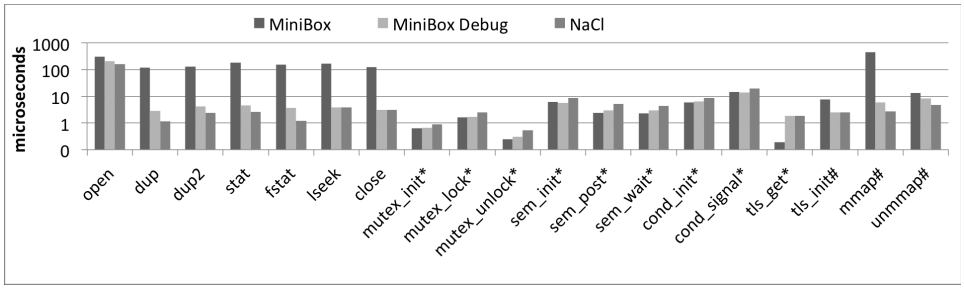

Figure 5.4: System Call Benchmarks in *us*. Average of 100 runs and standard deviation is less than 5%. Calls with $*$ are sensitive calls handled inside the MIEE without environment switches. Calls with # are sensitive calls that involve hypercall or environment switches.

[2] http://www.bulletphysics.com

System Call Overhead In the MIEE, non-sensitive system calls are handled in the OS with environment switches while sensitive system calls are handled either in the application layer inside the MIEE or by the hypervisor. The system call overhead in the MIEE was measured, and compared with the corresponding system calls on vanilla NaCl, and MiniBox in debugging model (recall Section 5.2.8). The evaluation results (Figure 5.4) show that the non-sensitive system calls (e.g., file operation calls) that involve environment switches on MiniBox are slower than on vanilla NaCl. However, the corresponding system calls on MiniBox in debugging mode have similar performance to those on vanilla NaCl. Thus the overhead of these system calls on MiniBox is mainly caused by environment switches. The sensitive system calls that are handled within the MIEE without any environment switch (e.g., thread synchronization calls) have similar performance to those on vanilla NaCl. The sensitive system calls that involve hypercall and environment switches (e.g., memory management system calls) on MiniBox are slower than on vanilla NaCl.

File I/O We evaluate the file I/O overhead on MiniBox and compare it to the file I/O on vanilla NaCl and MiniBox in debugging mode. We measure reads & writes of 32B for both general file I/O and secure file I/O. The measurement results (Figure 5.5) show that when the data is cached in the MIEE (cache-hit), the cache buffer significantly reduces the file I/O overhead for both general file I/O and secure file I/O.

Network I/O We evaluate the network I/O throughput on MiniBox and compare it to the network I/O throughput on MiniBox in debugging mode and vanilla NaCl. The server runs in the MIEE using MiniBox on the Dell T105 while the client runs on plain Linux on a Dell Optiplex 755 desktop with two Intel Core2 Duo

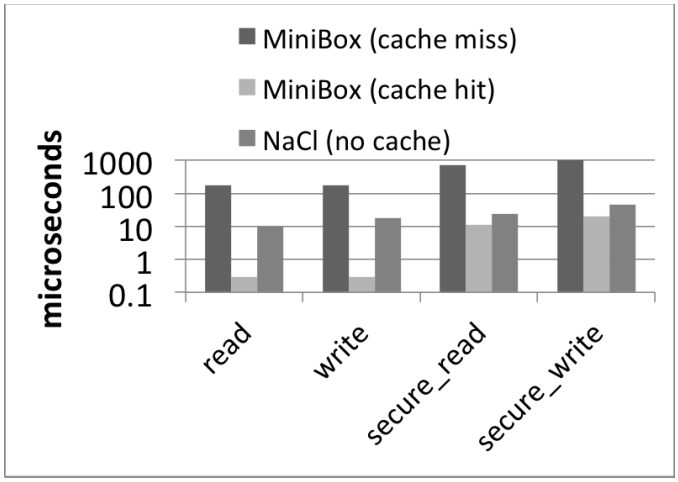

Figure 5.5: File I/O Benchmarks in *us*. Average of 100 runs and standard deviation is less than 2%.

processors running at 2.0 GHz with 2 GB memory. The operating system on the Dell Optiplex machine is Ubuntu 8.04.4 LTS with a 32-bit Linux kernel 2.6.24.30. Both the server and the client connect to a Netgear Gigabit Ethernet Switch using a Gigabit Ethernet Adapter. During each connection, the client sends 16 KB data to the server and we measure the network I/O throughput. The results (Figure 5.6) show that network I/O on MiniBox is about 10% slower than on vanilla NaCl. *Thus, although the environment switches impose a small overhead on MiniBox, the network throughput remains high.*

5.4.4 Application Benchmarks

CPU-bound application (AES key search and BitCoin) We measure the performance of CPU-bound applications on MiniBox and compare it to the performance of equivalent applications on vanilla NaCl and MiniBox in debugging mode. We first evaluate *AES key search*, which encrypts a 128-Byte plain-text using a 128-

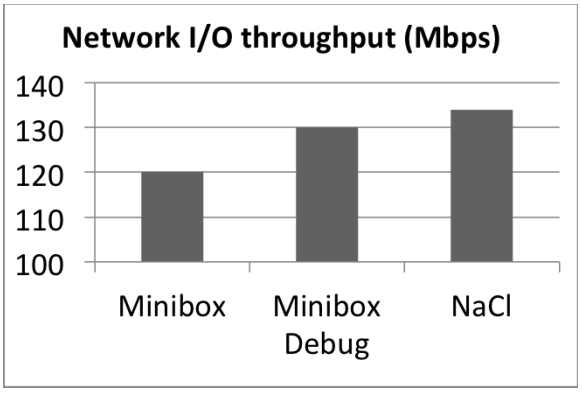

Figure 5.6: Network I/O Benchmarks in *Mbps*. Average of 100 runs and standard deviation is less than 2%.

bit key in CBC mode 200,000 times, simulating a AES key search operation. We port CBitCoin [70]), an open source BitCoin implementation to run on MiniBox. We measure the time to construct a BitCoin block, requiring 200,000 SHA-256 computations. The results show that *MiniBox does not add any noticeable overhead (less than 1% [61]) for CPU-bound applications over NaCl.*

I/O-bound application (Zlib) We evaluate the performance of I/O-intensive applications on MiniBox by testing Zlib [62], an open source library used for data compression. Zlib is already ported to run on NaCl as part of the naclports project, and does not require additional porting efforts to run on MiniBox. We measure the time elapsed to read 1 MB of file data from the file system over the general file I/O, and then compress the read data. The file data always misses the cache buffer, so every *read* operation involves an environment switch. The evaluation results (Figure 5.7) show that because of environment switches, the zlib application on MiniBox is slower than on vanilla NaCl. The slowdown is mainly caused by the environment switches since MiniBox in debugging mode has the same perfor-

mance as vanilla NaCl. We repeat the measurement on MiniBox while storing the file data in the cache buffer in the MIEE. The zlib application read file data with cache-hit without environment switches. The measurement result shows that the overhead is significantly reduced. *Thus, while file I/O in MiniBox can be expensive in the worst case, we expect that the cache buffer will significantly improve the application performance in practice.*

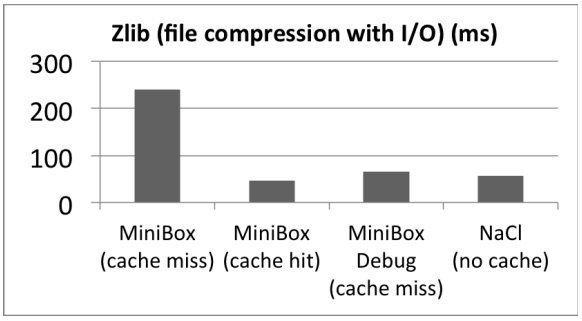

Figure 5.7: zlib File Compression with File I/O Benchmarks in *ms*. Average of 10 runs and standard deviation is less than 2%.

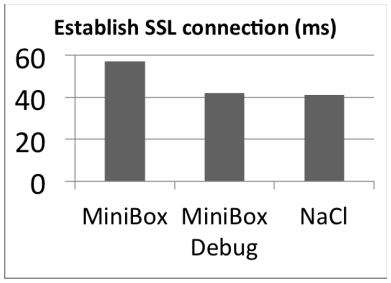

Figure 5.8: SSL Connection Benchmarks in *ms*. Average of 10 runs and standard deviation is less than 3%.

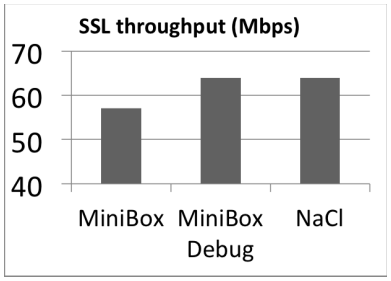

Figure 5.9: SSL Throughput Benchmarks in *Mbps*. Average of 10 runs and standard deviation is less than 1%.

SSL Server We port the entirety of OpenSSL [74] (version 1.0.0.e) to run on MiniBox. We also run the SSL server on NaCl by adding socket system call interface on the NaCl. In this experiment, the Dell Optiplex machine serves as the SSL client, and the Dell T105 acts as the SSL server. The SSL client runs on plain Linux while the SSL server runs inside the MIEE on MiniBox. We recorded both the time required to create an SSL connection and the overall SSL throughput. The SSL client sends 16KB of data to the SSL server during each connection. As in previous experiments, both machines connect to a Netgear Gigabit Ethernet Switch via a Gigabit Ethernet Adapter. The results show that MiniBox impose about a 15% overhead to SSL connections (Figure 5.8) and that SSL throughput on MiniBox has about a 10% slowdown (Figure 5.9). The overhead is mainly caused by environment switches, since MiniBox in debugging mode has the same performance as NaCl.

5.5 Integration: MiniBox with Trusted I/O

We can expand MiniBox with on-demand isolated I/O (Section 2.3) and VIPER (Chapter 4) to provide trusted I/O to the isolated application in the MIEE. Fig-

ure 5.10 shows the MiniBox architecture with trusted I/O.

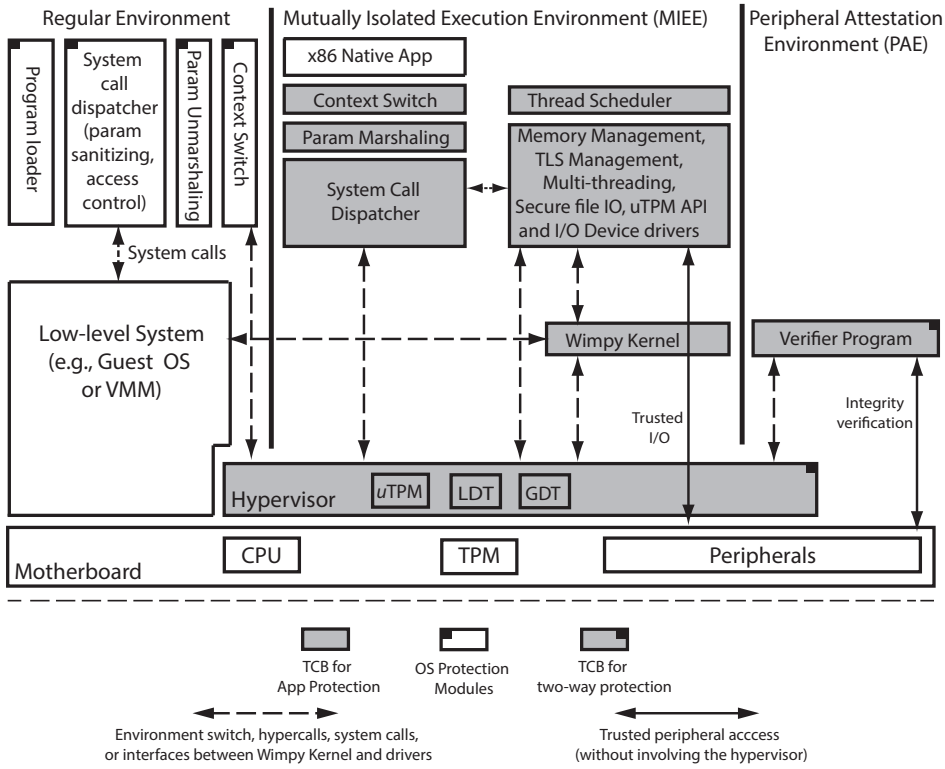

Figure 5.10: MiniBox System Architecture with Trusted I/O.

Architecture As shown Figure 5.10, a wimpy kernel is included in the MIEE to establish an on-demand isolated I/O for the MIEE; I/O device drivers are included in the service runtime in the MIEE for I/O access. To guarantee the absence of malware in peripherals, a peripheral attestation environment (PAE) is registered and isolated from the regular environment and the MIEE. In the PAE, a verifier program verifies the integrity of peripherals' firmware to guarantee the absence of malware on peripherals (trusted peripherals). We assume that the verifier program has the detailed information of the peripherals on the commodity computer and that

the chips in the I/O channel (e.g., Northbridge, Southbridge) do not have firmware. Before enabling the isolated application in the MIEE to access I/O peripherals, the wimpy kernel in the MIEE invokes the verifier program in the PAE (via a hypercall to the hypervisor or an environment switch) to verify the integrity of peripherals' firmware.

Two-Way I/O Isolation The hypervisor configures IOMMU, Nested Page Table (NPT) or Extended Page Table (EPT), the I/O port-access-interception bitmap, the PCIe Access Control Services (ACS) (Section 2.2), and Programmable Interrupt Controller (PIC) to establish two-way I/O isolation: malicious code in the regular environment or peripherals manipulated by the OS cannot access the isolated peripherals associated with the MIEE or PAE; similarly, a misbehaving isolated application, the wimpy kernel in the MIEE and peripherals manipulated by the wimpy kernel cannot access the peripherals manipulated by the OS in the regular environment. After the two-way I/O isolation is established, the wimpy kernel and the OS in the regular environment cannot bypass the hypervisor to arbitrarily change the system configurations, breaking the two-way isolation.

The original I/O isolation mechanism [127] also establishes two-way I/O isolation. However, the wimpy kernel is in the TCB for establishing two-way isolation. For example, in the original I/O isolation mechanism [127], the hypervisor enables the wimpy kernel to access the memory space of Programmable Interrupt Controller (PIC) to configure the interrupts while protecting the PIC memory space from the OS. In MiniBox, the wimpy kernel is not in the TCB for OS protection. Thus, the hypervisor performs all system configurations to establish two-way I/O isolation protection and protect the configurations from being modified by the wimpy kernel or the OS without being detected. For example, in MiniBox, the hypervisor protects the PIC memory space from both the wimpy kernel in MIEE

and the OS in the regular environment.

OS Protection MiniBox applies the mechanisms described in Section 4.7 to establish the trusted I/O for the isolated application. Now we describe how to protect the OS from malware in the peripherals that were manipulated by the MIEE. Although the wimpy kernel in the MIEE mediates the I/O access from the application to the associated I/O peripherals, it is still possible that a misbehaving application could insert malware into the firmware of the peripherals associated with the MIEE by exploiting firmware vulnerabilities. When the isolated peripherals are released to the regular environment, malware in the infected peripherals could compromise the OS or other peripherals. Therefore, to protect the OS and other peripherals from malware in peripherals associated with the MIEE, before releasing the isolated peripherals to the regular environment, the hypervisor configures the system to guarantee that only the prover program in the PAE can access the peripherals that were manipulated by the wimpy kernel and invokes the prover program in the PAE to verify the integrity of these peripherals' firmware (to guarantee the absence of malware on these peripherals). Only after obtaining the guarantee that all verified peripherals are free of malware, the hypervisor configures the system to release the verified peripherals to the regular environment (note that during and after the integrity measurement, code in the MIEE could not access these verified peripherals).

5.6 Limitations and Future Work

Application Interface MiniBox includes the entire application (the security-sensitive and non-sensitive PALs) in the MIEE and does not prevent adversaries from compromising the application through malicious inputs. The application can

measure the integrity of critical inputs (known inputs) and extend the results into the μTPM PCR for remote attestation. However, the isolated application may expose a large interface to unknown inputs. Schemes that focus on protecting a security-sensitive PAL [12, 33, 66, 67, 100, 105] can significantly reduce the attack surface by exposing a constrained interface between the security-sensitive PAL and the untrusted OS. On those schemes, the security-sensitive PAL remains secure when the application is compromised by the OS. Thus, for protecting the security-sensitive PAL, MiniBox may expose a larger attack surface to the untrusted OS than schemes that focus on protecting the security-sensitive PAL.

Thread Scheduling Application developers must consider that MiniBox does not make the scheduler work preemptively (recall Section 5.2.5), and so must always use supported system calls for thread synchronization (e.g., avoid situations where a thread performs busy waiting by watching a global variable in a loop instead of calling a blocking system call). In addition, the application-layer thread scheduler does not support multi-thread parallel computation to improve the performance of threaded applications on multi-core systems. One design is to allow the hypervisor to conduct thread scheduling and to manage the parallel computation on multiple cores, which will significantly increase the hypervisor complexity. As future work, we will investigate how to support parallel computation for a threaded application running inside the MIEE on multi-core systems. However, security-sensitive applications more concerned with a small TCB than performance may prefer not to include code for such complex operations in the hypervisor. To solve this issue, MiniBox can allow the application to configure the hypervisor functionality (e.g., disable the support for multi-thread parallel computation) at registration time, and can boot the hypervisor with the application-preferred configurations.

System Call Interface Exposing a large system call interface to the application increases the attack surface for OS protection; thus, MiniBox exposes a subset of the OS system call interface to the application to confine the application's operations. However, it will be interesting to investigate how to support the entire OS system call interface on MiniBox. If the entire OS system call interface is supported, statically linked legacy applications may be able to run on MiniBox. As future work, we will examine the OS system call interface, obtain a comprehensive list of sensitive calls, and investigate how to support the entire OS system call interface on MiniBox.

Improving Performance The hypervisor-based isolation mechanism causes overhead in environment switches. It is expected that the hardware-based isolation mechanism provided by Intel SGX will decrease the environment switch overhead. The *VMFUNC* instruction [48] released on the latest Intel 4th Generation Processor enables software in a guest Virtual Machine to switch nested page tables without a Virtual Machine exit. It is expected that the *VMFUNC* instruction will decrease the environment switch overhead. However, the *VMFUNC* instruction does not switch other critical system configurations (e.g., the GDT or IDT). As future work we will investigate how to perform secure environment switch using the *VMFUNC* instruction.

Supporting Multi-tenant Cloud Platform The MiniBox hypervisor prototype supports only a single guest OS. There is no fundamental barrier to port MiniBox with a virtual machine monitor like Xen [14] that supports multiple tenants, though doing so increases the TCB size. CloudVisor [124] demonstrates the approach to minimize the TCB on multi-tenant cloud platforms by leveraging nested virtualization technology. Nested virtualization can be added in MiniBox to sup-

port multi-tenant cloud platforms. On multi-tenant cloud platforms, the virtual machine (VM) may be constructed, destructed, saved, restored, or migrated. It is critical to protect the MIEE during VM management. The MiniBox hypervisor can encrypt or decrypt the memory contents of MIEEs in VM management, and verify the integrity of the MiniBox hypervisor on other machines to guarantee that MIEEs are only migrated to machines with a verified hypervisor. Also, the MiniBox hypervisor needs to encrypt or decrypt the μTPM instance together with a MIEE in VM management, to make the trustworthy computing abstractions provided to the MIEE transparent to the VM management.

Control Flow Integrity (CFI) Since the application that runs on MiniBox is isolated using nested page tables at the hypervisor level, and always runs in ring 3, MiniBox does not share NaCl's CFI requirement to be able to reliably disassemble and validate all instructions. Therefore, the CFI mechanisms implemented in NaCl are not necessary in MiniBox. The NaCl CFI mechanism depends upon its toolchain inserting many `nop` instructions into the compiled program, which decreases performance. The benefit of keeping the CFI mechanism, however, is that programs compiled by the same toolchain will be compatible with both NaCl and MiniBox. Also, the NaCl CFI mechanism does raise the bar for an adversary who wants to attack a specific application running in the MIEE.

5.7 Summary

MiniBox is a hypervisor-based sandbox that provides two-way protection between x86 native applications and the guest OS. MiniBox protects the guest OS through hypervisor-based memory isolation and OS protection modules. MiniBox significantly reduces the attack surface for both OS protection and application protection

by minimizing and securing the interface between OS protection modules and the application, and protects against Iago attacks on the application. The MiniBox design and protection mechanisms are promising for establishing two-way protection on commodity computer systems. In addition, MiniBox significantly decreases the porting effort compared to previous systems for isolating security-sensitive PALs, making MiniBox practical for wide adoption. Thus, we anticipate that MiniBox will be widely adopted on systems where two-way protection is desired.

Chapter 6

Related Work

6.1 Software-Based Attestation Techniques

We now chronologically review previous work on software-based attestation. Spinellis proposes "reflection" as an approach to verify the software running on a system [103]. Spinellis sketches an approach that fills the memory with random content, clears the system state and disables all interrupts, computes a hash function over the entire memory, and finally returns the system state and hash to a verifier. The verifier checks the execution time and returned information. Unfortunately, Spinellis only presents a high-level approach but no implementation details.

Kennell and Jamieson present Genuinity [53], an approach where a verifier executes a verification function on an untrusted system to validate the system configuration. Genuinity is based on the observation that simulating low-level hardware is slower than actual execution on that hardware, and intentionally creates randomized memory accesses that create many TLB misses. By validating the number of TLB misses the verifier can inspect whether the code was correctly executed. Mead is the first scheme to use the dynamic page tables to detectably increase at-

tackers overhead; i.e., virtual addresses (VAes) are dynamically changed during the checksum computation to force attackers incur verifier-detectable overhead in any checksum modification. In contrast, Genuinity uses constant page tables with random mappings from constant VAes to physical addresses (PAes), performs random reads to cause TLB misses, and includes the TLB miss counter in the checksum to detect attacks. Adversary modified memory contents have constant VAes, so Genuity cannot significantly increase the attackers overhead. Genuity is already broken by memory substitution/copy attacks [99].

Seshadri et al. introduced software-based attestation and developed SWATT, a system to verify the software of an embedded device [96]. SWATT relies on a checksum function that computes a checksum over the entire memory contents and is constructed to force an attacker to induce overhead to compute the correct checksum. Seshadri et al. proposed a variety of extensions: enable verification of a small amount of memory on sensor nodes through the ICE function [92], verification of code running on an Intel Pentium IV processor through the Pioneer function [94], and code running on an AMD Opteron K8 architecture through the Outpost function [90]. Castelluccia et al. point out weaknesses in the specific SWATT and ICE functions [18], triggering significant discussion [30, 80]. The basic approach of software-based attestation remains sound, but special care has to be paid to ensure security, as the current work demonstrates.

Concurrently, Gratzer and Naccache have presented a more theoretical treatment of software-based attestation, which relies on the assumption that the verifier can physically observe and reset the untrusted device and assuming that the reset and execution times can be accurately observed [34]. Park and Shin have proposed soft tamper-proofing, an approach that fills memory with random data and executes a hash function, however, without considering timing [75]. Shaneck et al. explore the use of encrypted and self-modifying code to verify software on sensor

nodes [98]. Their approach also relies on randomized traversal and timing. Jakobsson and Johansson have studied new approaches to software-based attestation on mobile devices [49, 50].

6.2 Peripheral Malware Detection

Duflot et al. [28] propose runtime firmware integrity verification of a network adapter by utilizing the debugging features available on a Broadcom network adapter. The debugging features enable the host CPU to single-step the microcontroller on the Broadcom NIC and inspect the memory contents on the NIC by accessing the NIC's MMIO registers. Unfortunately, similar debugging features are not available on all peripherals, and these features may themselves be susceptible to impersonation. A more general mechanism is needed to conduct the firmware integrity verification.

Lone Sang et al. discuss peer-to-peer attacks within computer systems by leveraging DMA-based communication [85]. They propose approaches to prevent unauthorized communication between devices within a computer system, but they do not propose any detection mechanisms for verifying the integrity of the firmware of devices.

Stewin [104] investigated the mechanisms to detect peripheral DMA malware that scans the entire main memory to obtain security-sensitive information (e.g., credit number). To detect such DMA malware, Stewin proposed to run a piece of trusted code on the main CPU that monitors the bus activities to detect abnormal DMA memory access from peripherals. However, malware on periperhals could protect itself from being detected by avoiding DMA-based memory access.

6.3 Protecting Applications

Systems aspiring to protect entire applications from a potentially compromised OS have been proposed (e.g., [15, 21, 23, 24, 26, 38, 64, 73, 105, 120]). Most of these schemes mainly focus on protecting application data from malicious code on an operating system and expose sensitive system calls to the untrusted OS, thus making the protected application vulnerable to Iago attacks. InkTag [38] secures applications running on an untrusted OS by verifying that the untrusted OS behaves correctly using a trustworthy hypervisor. It prevents `mmap`-based Iago attacks by verifying memory address invariants. However, in InkTag some other security-sensitive system calls (e.g., thread synchronization and TLS-related calls) are still performed by the untrusted OS without being verified. Proxos [105] splits system calls and forwards sensitive system calls to a trusted private OS to protect applications from an untrusted OS. However, Proxos needs application developers to specify the splitting rule. Baumann et al. [15] propose Haven that protects a legacy application in the isolated memory space provided by Intel SGX, and propose to include a library OS in the isolated memory space to prevent Iago attacks. The proposed protection mechanisms (for application protection) are similar to the mechanisms on MiniBox. However, Haven contains a larger TCB than MiniBox. Mai et al. [64] proposed mechanisms to prove that the OS implements the application security invariants (e.g., secure storage and memory isolation) correctly. The proposed verification approach is promising for application isolation.

Researchers have explored many systems for isolating sensitive code using virtualization, microkernels, and other low-level mechanisms [12, 33, 66, 67, 100, 105], or by running the code inside trusted hardware [20, 52, 101, 102]. The virtualization-based schemes contain a large TCB. Other schemes either do not enjoy compatibility with a large set of commodity systems or require significant porting effort.

TrustVisor [66] and Flicker [67] isolate a PAL from an untrusted OS with a small TCB. However, porting security-sensitive applications on TrustVisor or Flicker requires significant efforts. Nizza [100] also requires developers to perform similar operations to port sensitive applications to Nizza.

6.4 Sandbox for x86 Native Code

Google Native Client [122] confines untrusted native code using SFI [65, 112] and enables developers to port native code as web applications. Drawbridge [27, 81] isolates an application in a picoprocess and provides a library OS to the isolated application. However, Native Client and Drawbridge provide only one-way protection. TxBox [51] confines an untrusted application by executing the application in a system transaction and conducting security check. MBox [54] protects the host file system from an untrusted application by exposing a virtual file system on top of the host file system for the application. Capsicum [115] supports capability-sandbox for applications on UNIX-like OS (e.g., FreeBSD). It focuses on application compartmentalization and fine-grained access control. Systrace [82] improves the host OS security by confining the program privilege using a configurable system call policy. The protection mechanisms provided by MBox, Capsicum and Systrace can be applied on MiniBox as part of the OS protection modules.

Chapter 7

Conclusion

As malware-detection mechanisms have become increasingly advanced, so has malware that tries to hide itself from being detected. Sophisticated malware can hide itself in system configuration registers without changing any existing software or hide itself inside peripherals to prevent it from being detected, raising significant challenges for establishing malware-free system states or operation environments on commodity platforms. A hardware- or software-based root of trust on a commodity platform is the foundation for addressing the challenges. However, establishing a software-based root of trust on commodity platforms with complex hardware features is challenging and requires considerable efforts. This thesis examines those challenges and shows how to expand existing software-based attestation techniques to detect malware on peripherals or to establish a software-based root of trust on embedded platforms with complex architectures.

The techniques proposed in this thesis are generic and can be applied in other application scenarios. For example, the mechanisms to defend against proxy attacks can also be applied in other scenarios where proxy attacks are possible, for establishing a software-based root of trust. In addition, the two-way protection

mechanism in MiniBox can be applied in Platform-as-a-Service (PaaS) cloud computing platforms to provide efficient two-way protection for customers' programs as well as security-sensitive data. Furthermore, the isolated malware-free operation environment can be applied on a user's host computer to enable a remote server to authenticate not only a user (to prevent spam users), but also the integrity of the client-side program, significantly improving the end-to-end security.

We can imagine that defending against malware to protect security-sensitive programs or data on commodity platforms will become increasingly challenging. However, the techniques proposed in this thesis open up new directions for defenders, and we anticipate that trusted computing technology-based protection mechanisms will be widely adopted on commodity platforms in the future.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity Principles, Implementation, and Applications. *ACM Transaction on Information and System Security (TISSEC)*, 13:1 – 40, 2009. (Referenced on page 158.)

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. CFI: Principles, implementations, and applications. In *Proc. ACM Conference and Computer and Communications Security (CCS)*, 2005. (Referenced on page 158.)

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *Proc. Conference on Formal Engineering Methods*, 2005. (Referenced on page 158.)

[4] Uttau Abatu, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. ACM. (Referenced on pages 4, 5, and 147.)

[5] Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01, May 2005. (Referenced on page 17.)

[6] Advanced Micro Devices. AMD I/O virtualization technology (IOMMU) specification. Publication No. 34434, Revision: 1.26, February 2009. (Referenced on page 15.)

[7] Advanced Micro Devices. AMD64 architecture programmer's manual volume 2: System programming. Publication No. 24593, Revision: 3.17, June 2010. (Referenced on pages 15 and 17.)

[8] ARM. Cortex-A8 technical reference manual. Revision:r3p2, May 2010. (Referenced on page 64.)

[9] ARM. Arm architecture reference manual. ARMv7-A and ARMV7-R edition, July 2012. (Referenced on page 64.)

[10] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 1–12. ACM, 2013. (Referenced on page 23.)

[11] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of ACM conference on Computer and Communication Security*, 2010. (Referenced on page 93.)

[12] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In

Proc. ACM Conference on Computer and Communications Security, 2011. (Referenced on pages 147, 180, and 187.)

[13] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient padding oracle attacks on cryptographic hardware. In *CRYPTO*, pages 608–625, 2012. (Referenced on page 23.)

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. Symposium on Operating Systems Principles*, 2003. (Referenced on page 181.)

[15] Andrew Baumann, Marcus Peinado, Galen Hunt, Krystof Zmudzinski, Carlos V. Rozas, and Matthew Hoekstra. Secure execution of unmodified applications on an untrusted host. http://research.microsoft.com/apps/pubs/default.aspx?id=204758, 2013. (Referenced on page 187.)

[16] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return oriented programming to RISC. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008. (Referenced on page 11.)

[17] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2009. (Referenced on page 12.)

[18] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of ACM*

Conference on Computer and Communications Security (CCS), November 2009. (Referenced on page 185.)

[19] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2013. (Referenced on page 149.)

[20] B. Chen and R. Morris. Certifying program execution with secure processors. In *Proceedings of HotOS*, 2003. (Referenced on pages 147 and 187.)

[21] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P.C. Yew, and W. Mao. Tamper-resistant execution in an untrusted operating system using a VMM. Technical Report FDUPPITR-2007-0801, Fudan University, 2007. (Referenced on pages 147 and 187.)

[22] K. Chen. Reversing and exploiting an Apple firmware update. In *Black Hat*, 2009. (Referenced on pages 3 and 90.)

[23] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, 2008. (Referenced on pages 147 and 187.)

[24] Yueqiang Cheng, Xuhua Ding, and Robert Deng. AppShield: Protecting applications against untrusted operating system. In *Singaport Management University Technical Report, SMU-SIS-13-101*, 2013. (Referenced on pages 147 and 187.)

[25] CYPRESS. CYPRESS enCoRe II low-speed USB peripheral controller (CY7C639XX). (Referenced on page 128.)

[26] Prashant Dewan, David Durham, Hormuzd Khosravi, Men Long, and Gayathri Nagabhushan. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proc. Spring Simulation Multiconference*, 2008. (Referenced on pages 147 and 187.)

[27] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the USENIX conference on Operating systems design and implementation*, OSDI'08, pages 339–354, Berkeley, CA, USA, 2008. USENIX Association. (Referenced on pages 147, 148, and 188.)

[28] L. Duflot, Yves-Alexis Perez, and Benjamin Morin. Run-time firmware integrity verification: what if you can not trust your network card? In *CanSecWest*, 2011. (Referenced on page 186.)

[29] Loic Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card? CanSecWest, 2010. (Referenced on pages 3 and 90.)

[30] Aurélien Francillon, Claude Castelluccia, Daniele Perito, and Claudio Soriente. Comments on "refutation of on the difficulty of software-based attestation of embedded devices". http://planete.inrialpes.fr/~perito/papers/2010_CCS_attestation_comments_on_rebutal.pdf, October 2010. (Referenced on page 185.)

[31] McKeen Frank, Alexandrovich Ilya, Berenzon Alex, Rozas Carlos V, Shafi Hisham, Shanbhogue Vedvyas, and Savagaonkar Uday R. Innovative in-

structions and software model for isolated execution. In *Proceedings of International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM. (Referenced on pages 4, 5, and 147.)

[32] A. Frederik, S. Ahmad-Reza, S. Steffen, and W. Christian. A security framework for the analysis and design of software attestation. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 1–12, 2013. (Referenced on page 2.)

[33] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. ACM Symposium on Operating System Principles (SOSP)*, 2003. (Referenced on pages 147, 180, and 187.)

[34] Vanessa Gratzer and David Naccache. Alien vs. quine, the vanishing circuit and other tales from the industry's crypt. In *Proceedings of Eurocrypt*, May 2006. (Referenced on page 185.)

[35] C. Hall, D. Wagner, J. Kelsey, and B. Schneier. Building PRFs from PRPs. In *CRYPTO*, pages 370–389, 1998. (Referenced on page 66.)

[36] Laszlo Hars and Gyorgy Petruska. Pseudo-random recursions: Small and fast pseudo-random number generator for embedded applications. In *EURASIP Journal on Embedded Systems*, 2007. (Referenced on page 129.)

[37] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of International Workshop on Hardware and*

Architectural Support for Security and Privacy, HASP '13, pages 11:1–11:1, New York, NY, USA, 2013. ACM. (Referenced on pages 4, 5, and 147.)

[38] Owen Hofmann, Alan Dunn, Sangman Kim, Michael Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2013. (Referenced on pages 147 and 187.)

[39] D.A Huffman. A method for the construction of minimum redundancy codes. In *Proceedings of the IRE 40*, 1962. (Referenced on page 12.)

[40] R Hund, T Holz, and Freiling F. C. Return oriented rootkit: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, 2009. (Referenced on page 11.)

[41] IEEE Computer Society: 802.3 Working Group. IEEE standard 802.3x-1997, 1997. (Referenced on page 115.)

[42] Virtual Laboratories in Probability and Statistics. The coupon collector problem. http://www.math.uah.edu/stat/urn/Coupon.html. (Referenced on pages 13 and 81.)

[43] Alteon Networks Inc. Tigon Open Firmware. http://alteon.shareable.org. (Referenced on pages 105 and 106.)

[44] Alteon Networks Inc. Tigon/PCI Ethernet Controlller (revision 1.04). http://alteon.shareable.org, 1997. (Referenced on page 105.)

[45] Intel Corporation. Trusted execution technology – preliminary architecture specification and enabling considerations. Document number 31516803, November 2006. (Referenced on page 17.)

[46] Intel Corporation. Intel trusted execution technology – software development guide. Document number 315168-005, June 2008. (Referenced on page 17.)

[47] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture. Order Number: 253665-073US, January 2011. (Referenced on page 14.)

[48] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual volume 3b: System programming guide, part 2. Order Number: 325384-048US, September 2013. (Referenced on pages 15, 17, 21, 157, 166, and 181.)

[49] Markus Jakobsson and Karl-Anders Johansson. Assured detection of malware with applications to mobile platforms. DIMACS Technical Report 2010-03, http://dimacs.rutgers.edu/TechnicalReports/abstracts/2010/2010-03.html, 2010. (Referenced on page 186.)

[50] Markus Jakobsson and Karl-Anders Johansson. Assured detection of malware with applications to mobile platforms. In *Proceedings of the Workshop on Hot Topics in Security (HotSec)*, August 2010. (Referenced on page 186.)

[51] Suman Jana, Donald E. Porter, and Vitaly Shmatikov. Txbox: Building secure, efficient sandboxes with system transactions. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP '11, pages 329–344, Washington, DC, USA, 2011. IEEE Computer Society. (Referenced on pages 147 and 188.)

[52] S. Jiang, S. Smith, and K. Minami. Securing web servers against insider attack. In *Proc. Computer Security Applications Conference*, 2001. (Referenced on pages 147 and 187.)

[53] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. pages 295–308. (Referenced on page 184.)

[54] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *Proceedings of USENIX conference on USENIX annual technical conference*, USENIXATC'13, 2013. (Referenced on pages 147 and 188.)

[55] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable key infrastructure (aki): A proposal for a public-key validation infrastructure. In *Proceedings of International World Wide Web Conference (WWW 2013)*. WWW Consortium, 2013. (Referenced on page 23.)

[56] 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*, page 470 to 483, 2003. (Referenced on page 128.)

[57] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society. (Referenced on page 88.)

[58] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. New results for timing-based attestation. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 239–253. IEEE, 2012. (Referenced on page 23.)

[59] Kaspersky Lab. Equation Group: Questions and Answers. https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf, 2015. (Referenced on page 1.)

[60] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. SBAP: Software-Based Attestation for Peripherals. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, 2010. (Referenced on page 103.)

[61] Yanlin Li, Adrian Perrig, Jonathan M. McCune, James Newsome, Brandon Baker, and Will Drewry. MiniBox: A Two-Way Sandbox for x86 Native Code. Technical Report CMU-CyLab-14-001, Carnegie Mellon University, 2014. (Referenced on pages 166, 170, and 174.)

[62] Jean loup Gailly and Mark Adler. zlib open source library. http://www.zlib.net. (Referenced on page 174.)

[63] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association. (Referenced on page 160.)

[64] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. In *Proceedings of International Conference on Architectural Support for Pro-*

gramming Languages and Operating Systems, ASPLOS '13, pages 293–304, New York, NY, USA, 2013. ACM. (Referenced on page 187.)

[65] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *Proc. USENIX Security*, 2006. (Referenced on pages 20 and 188.)

[66] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010. (Referenced on pages 4, 5, 18, 147, 150, 166, 180, and 187.)

[67] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, April 2008. (Referenced on pages 18, 93, 147, 180, 187, and 188.)

[68] Mindshare Inc., Ravi Budruk, Don Anderson, and Tom Shanley. *PCI Express System Architecture*. Addison-Wesley Professional, September 2003. (Referenced on pages 16 and 94.)

[69] MindShare Inc., Tom Shanley, and Don Anderson. *PCI System Architecture (4th Edition)*. Addison-Wesley Professional, June 1999. (Referenced on pages 16 and 94.)

[70] Matthew Mitchell, Auston Sterling, and Andrew Miller. Cbitcoin open source project. http://code.google.com/p/naclports/. (Referenced on page 174.)

[71] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: an empirical analysis of hardware failures on a million

consumer PCs. In *Proceedings of the European Conference on Computer systems (EuroSys)*, 2011. (Referenced on page 142.)

[72] NIST. Recommendation for key management. Special Publication 800-57 Part 1, March 2007. (Referenced on pages 25 and 94.)

[73] Kaan Onarlioglu, Collin Mulliner, William Robertson, and Engin Kirda. Privexec: Private execution as an operating system service. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 206–220, Washington, DC, USA, 2013. IEEE Computer Society. (Referenced on page 187.)

[74] OpenSSL Project team. OpenSSL. http://www.openssl.org/, May 2005. (Referenced on page 176.)

[75] T. Park and K. G. Shin. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Transactions on Mobile Computing (TMC)*, 4(3), 2005. (Referenced on page 185.)

[76] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*, volume 10 of *SpringerBriefs in Computer Science*. Springer, 2011. (Referenced on page 23.)

[77] Bryan Jeffery Parno. *Trust Extension As a Mechanism for Secure Code Execution on Commodity Computers*. Association for Computing Machinery and Morgan, 2014. (Referenced on pages 2 and 3.)

[78] PCI-SIG. PCI Express access control services (ACS). PCI-SIG Engineering Change Notice, October 2006. (Referenced on page 16.)

[79] PCI-SIG. PCI express 2.0 frequently asked questions. http://www.pcisig.com/, March 2011. (Referenced on page 16.)

[80] Adrian Perrig and Leendert van Doorn. Refutation of "on the difficulty of software-based attestation of embedded devices". http://sparrow.ece.cmu.edu/group/pub/perrig-vandoorn-refutation.pdf, 2010. (Referenced on page 185.)

[81] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. *SIGPLAN Not.*, 46(3):291–304, March 2011. (Referenced on pages 148 and 188.)

[82] Niels Provos. Improving host security with system call policies. In *Proceedings of Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 18–18, Berkeley, CA, USA, 2003. USENIX Association. (Referenced on page 188.)

[83] Christopher Sagoian and Sid Stamm. Certified lies: Detecting and defeating government interception attacks against ssl. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 1–18, 2010. (Referenced on page 23.)

[84] Fernand Lone Sang, Èric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. In *Proceedings of IEEE Conference on Malicious and Unwanted Software (Malware)*, 2010. (Referenced on page 93.)

[85] Fernand Lone Sang, Vincent Nicomette, Yves Deswarte, and Loïc Duflot. Attaques DMA peer-to-peer et contremesures. In *Proceedings of the Symposium sur la Sécurité des Technologies de L'Information et des Communications (SSTIC)*, June 2011. (Referenced on pages 16 and 186.)

[86] Mahadev Satyanarayanan. Cloudlets: At the leading edge of cloud-mobile convergence. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, pages 1–2, New York, NY, USA, 2013. ACM. (Referenced on pages 4 and 146.)

[87] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, October 2009. (Referenced on pages 4 and 146.)

[88] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. SCUBA: Secure code update by attestation in sensor networks. In *ACM Workshop on Wireless Security (WiSe 2006)*, 2006. (Referenced on page 2.)

[89] A. Seshadri, M. Luk, A. Perrig, L. Van Doorn, and P. Khosla. SAKE: Software attestation for key establishment in sensor networks. In *International Conference on Distributed Computing in Sensor Systems*, 2008. (Referenced on page 2.)

[90] Arvind Seshadri. *A Software Primitive for Externally-verifiable Untampered Execution and its Applications to Securing Computing Systems*. PhD thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, 2009. (Referenced on pages 13, 44, 45, and 185.)

[91] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure Code Update By Attestation in sensor networks. In *Proceedings of ACM Workshop on Wireless Security*, pages 85–94. ACM, 2006. (Referenced on page 23.)

[92] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of ACM Workshop on Wireless Security (WiSe)*, September 2006. (Referenced on page 185.)

[93] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 335–350. ACM, 2007. (Referenced on page 23.)

[94] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–16, October 2005. (Referenced on pages 23, 34, 91, 103, and 185.)

[95] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: SoftWare-based ATTestation for embedded devices. (Referenced on page 2.)

[96] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: SoftWare-based ATTestation for embedded devices. (Referenced on pages 12, 23, 91, 103, 128, 130, and 185.)

[97] H Shacham. The geometry of innocent flesh on the bone: Return into libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007. (Referenced on page 11.)

[98] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *Proceedings of ESAS*, 2005. (Referenced on page 186.)

[99] Umesh Shankar, Monica Chew, and J.D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the USENIX Security Symposium*, 2004. (Referenced on page 185.)

[100] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications. In *EuroSys*, 2006. (Referenced on pages 147, 180, 187, and 188.)

[101] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, April 1999. Special Issue on Computer Network Security. (Referenced on pages 147 and 187.)

[102] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8), April 1999. (Referenced on pages 147 and 187.)

[103] Diomidis Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security*, 3(1):51–62, February 2000. (Referenced on page 184.)

[104] Patrick Stewin. *Detecting Peripheral-based Attacks on the Host Memory*. Springer, 2015. (Referenced on pages 91 and 186.)

[105] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *ACM SOSP*, 2006. (Referenced on pages 147, 180, and 187.)

[106] Simon Tam. Modern clock distribution systems. In *Clocking in Modern VLSI Systems*, Integrated Circuits and Systems, chapter 2, pages 6–95. Springer, 2009. (Referenced on pages 23 and 31.)

[107] Texas Instruments. AM/DM37X multimedia device technical reference manual. Version R, September 2012. (Referenced on pages 28 and 63.)

[108] The Trusted Computing Group. TPM Main specification version 1.2 (revision 116), 2011. (Referenced on page 22.)

[109] Arrigo Triulzi. Project Maux Mk.II, I Own the NIC, now I want a shell. In *The 8th annual PacSec conference*, 2008. (Referenced on page 90.)

[110] Arrigo Triulzi. The Jedi Packet takes over the Deathstar, taking NIC backdoor to the next level. In *The 12th annual CanSecWest conference*, 2010. (Referenced on pages 16 and 90.)

[111] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, May 2013. (Referenced on pages 4, 5, 150, 166, and 170.)

[112] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *ACM SOSP*, 1993. (Referenced on pages 20 and 188.)

[113] Jiang Wang, Angelos Stavrou, and Anup K. Ghosh. HyperCheck: A Hardware-Assisted Integrity Monitor. In *Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2010. (Referenced on page 93.)

[114] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of USENIX Workshop on Offensive Technologies*, WOOT '07, pages 2:1–2:8, Berkeley, CA, USA, 2007. USENIX Association. (Referenced on page 151.)

[115] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *Proceedings of USENIX Conference on Security*, USENIX Security'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association. (Referenced on page 188.)

[116] Carsten Weinhold and Hermann Härtig. Vpfs: building a virtual private file system with a small trusted computing base. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 81–93, New York, NY, USA, 2008. ACM. (Referenced on page 160.)

[117] Carsten Weinhold and Hermann Härtig. jvpfs: adding robustness to a secure stacked file system with untrusted local storage components. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 32–32, Berkeley, CA, USA, 2011. USENIX Association. (Referenced on page 160.)

[118] wikipedia. http://en.wikipedia.org/wiki/RC4. (Referenced on page 128.)

[119] Thomas Wollinger, Jorge Guajardo, and Christof Paar. Security on fpgas: State-of-the-art implementations and attacks. In *ACM Transactions on embedded computing systems (TECS)*, volume 3, 2004. (Referenced on page 23.)

[120] J. Yang and K. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proc. ACM Conference on Virtual Execution Environments (VEE)*, 2008. (Referenced on pages 147 and 187.)

[121] Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *Proceedings of IEEE International Symposium on Reliable Distributed Systems*, 2007. (Referenced on page 2.)

[122] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Orm, Shiki Okasaka, Neha Narula, Nicholas Fullagar, and Google Inc. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009. (Referenced on pages 5, 6, 8, 20, 147, 148, 150, 169, 170, 171, and 188.)

[123] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010. (Referenced on pages 5, 6, 8, 20, 170, and 171.)

[124] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2011. (Referenced on page 181.)

[125] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS

'12, pages 305–316, New York, NY, USA, 2012. ACM. (Referenced on page 151.)

[126] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2012. (Referenced on page 4.)

[127] Zongwei Zhou, Miao Yu, and Virgil D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated I/O. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 308–323. IEEE, 2014. (Referenced on pages 4, 19, 136, 138, 139, 153, and 178.)