VIPER: Verifying the Integrity of PERipherals' Firmware

Yanlin Li, Jonathan M. McCune, and Adrian Perrig
CyLab, Carnegie Mellon University
Pittsburgh, PA, USA
{yanlli,jonmccune,perrig}@cmu.edu

ABSTRACT

Recent research demonstrates that malware can infect peripherals' firmware in a typical x86 computer system, e.g., by exploiting vulnerabilities in the firmware itself or in the firmware update tools. Verifying the integrity of peripherals' firmware is thus an important challenge. We propose software-only attestation protocols to verify the integrity of peripherals' firmware, and show that they can detect all known software-based attacks. We implement our scheme using a Netgear GA620 network adapter in an x86 PC, and evaluate our system with known attacks.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*

General Terms

Security, Verification

Keywords

Integrity of Peripherals' Firmware, Proxy Attack, Software-based Attestation

1. INTRODUCTION

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 [5, 7]. The malware, once inside a peripheral, may also compromise other peripherals' firmware. 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 [35]. 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 [36]. 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 [5]. Such malicious code can be a key-logger or script that can compromise the host operating system. In 2010, a buffer overflow vulnerability in a Broadcom NIC firmware was published [7], through which a remote attacker can compromise the NIC firmware by sending malicious packets to this NIC, then execute arbitrary code on the NIC.

We expect that malware on peripherals' firmware will be a popular trend for next-generation malware. Unfortunately, it is still an open challenge to detect malware on peripherals or verify the integrity of peripherals' firmware because (1) the limited memory and computational resources on peripherals make it difficult to deploy complex security primitives on peripherals themselves; and (2) hardware-based protection is impractical because it would add cost and complexity to devices already under severe economic constraints.

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 paper.

At first glance, software-based attestation [30, 31] may provide an approach for verifying the integrity of firmware. Device vendors could embed an attestation function in their firmware, which driver code executing on the main CPU could query 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* (Figure 1), 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. Peripherals, such as a NIC, can communicate with a remote proxy server to compute the expected answer. Also, faster 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

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

Figure 1: A Proxy Attack.

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 paper makes the following contributions:

1. We systematically analyze the features of malware on computer peripherals.
2. We propose a software-only primitive, VIPER, to verify the integrity of peripheral devices' firmware.
3. 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.
4. We fully implement VIPER on a Netgear GA620 network adapter in an off-the-shelf computer, and also implement an Ethernet-based proxy attack. Our evaluation shows that VIPER can efficiently verify the integrity of peripherals' firmware.

The remainder of this paper is organized as follows: Section 2 provides background knowledge on the system bus and architecture of a modern computer system, and malware on peripherals. Section 3 describes our assumptions and attacker model. The VIPER architecture and attestation protocols are detailed in Section 4. Section 5 describes our implementation, and Section 6 gives evaluation results with the best known attacks. Open problems and limitations are treated in Section 7, related work in Section 8, and conclusions in Section 9.

2. BACKGROUND

We provide the necessary background on the system buses and architecture of a modern motherboard, and the features of malware on peripherals.

Modern System Buses and Architecture. Figure 2 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. On a modern motherboard, the clock speed of the northbridge and southbridge can exceed 1 GHz. The capacity of various versions of PCI buses 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) [21]. PCIe supports 2 Gbps (v1), 4 Gbps (v2), and 8 Gbps (v3) on each lane, with up to 32 lanes in each PCIe slot [20].

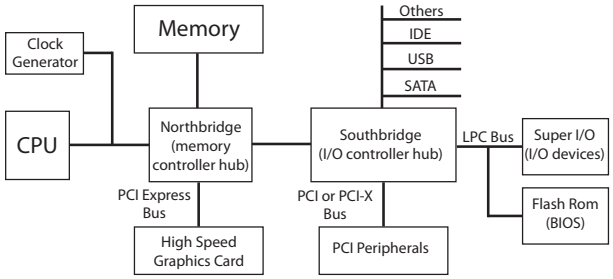

Figure 2: Hardware Architecture of a Modern Motherboard.

Memory-mapped I/O (MMIO) [13] 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) [1, 2, 14].

Peer-to-Peer Peripheral Communication. Based on the PCI and PCIe specifications [20, 21], 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 [27, 36]. 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 [27].

Malware on Peripherals. 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.

3. ASSUMPTIONS & ATTACKER MODEL

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 [26], recent work in OS-level security and trustworthy computing may in fact provide a reasonable platform from which to attempt peripheral device verification [3, 19, 37]. 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 considerable computation and memory resources, though the attacker is still unable to break standard cryptographic primitives [23]. However, we assume practical communication constraints, such as the bandwidth and latency characteristics of PCI [20, 21] and Gigabit Ethernet.

4. VIPER: VERIFYING THE INTEGRITY OF PERIPHERALS' FIRMWARE

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

4.1 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.

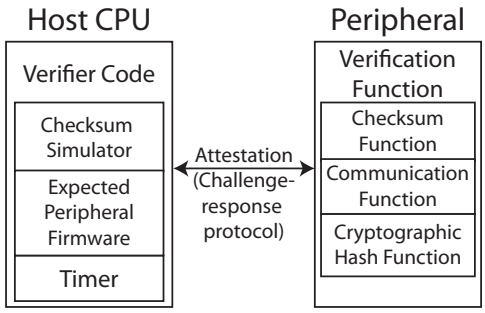

Figure 3: VIPER system architecture.

System Architecture. In VIPER (Figure 3), 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 3). 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).
4. 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. 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 Attestation Protocol

Though an on-board proxy attack can be prevented or detected as described in Section 4.1, 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.

4.2.1 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 shows the time line of one challenge-response pair in a latency-based attestation protocol, including both the normal computation, and the proxy attack.

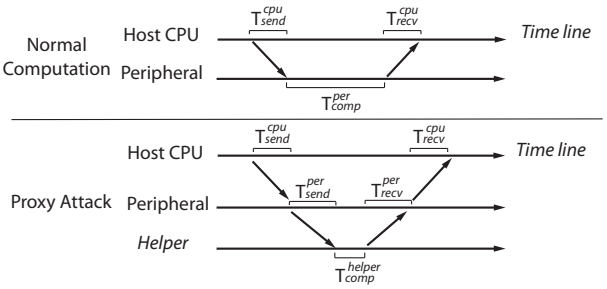

Figure 4: One challenge-response pair for latency-based attestation under both normal computation and a proxy attack.

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:

$$T^{normal}_{comp} = T^{cpu}_{send} + T^{per}_{comp} + T^{cpu}_{recv} \quad (1)$$

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^{per}_{recv} to send the result back to the peripheral. Thus, in the proxy attack, the time of each challenge-response pair is:

$$T^{proxy}_{comp} = T^{cpu}_{send} + T^{per}_{send} + T^{helper}_{comp} + T^{per}_{recv} + T^{cpu}_{recv} \quad (2)$$

We assume that T^{helper}_{comp} 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:

$$T^{proxy}_{overhead} = T^{per}_{send} + T^{per}_{recv} - T^{per}_{comp} \quad (3)$$

To detect a proxy attack, $T^{proxy}_{overhead}$ must be positive and detectable. Therefore, T^{per}_{comp} must be well-controlled to guarantee that T^{per}_{comp} 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^{per}_{comp} 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 3) 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 5, 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_{recv}^{per}. 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 5, 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_{send}^{cpu} for iteration $i+1$ and T_{recv}^{cpu} for iteration i do not impose any additional latency on the total checksum computation time.

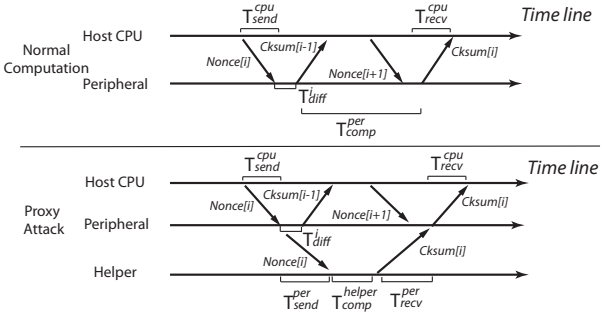

Figure 5: The latency-based attestation procedure after speed-up under benign and (hypothetically successful) attack conditions.

The time for a single challenge-response pair is:

$$T_{comp}^{normal} = T_{send}^{cpu} + T_{diff}^{n} + T_{comp}^{per} + T_{recv}^{cpu} \quad (4)$$

In this equation, T_{diff}^{n} 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:

$$T_{comp}^{proxy} = T_{send}^{cpu} + T_{send}^{per} + T_{comp}^{helper} + T_{recv}^{per} + T_{recv}^{cpu} \quad (5)$$

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

$$T_{overhead}^{proxy} = T_{send}^{per} + T_{recv}^{per} - T_{diff}^{n} - T_{comp}^{per} \quad (6)$$

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.

4.2.2 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.

4.3 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 [18, 30, 31], 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 ad-

ditional 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 [12] in Section 5.3.

5. IMPLEMENTATION

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 [11]. 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.

5.1 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 6 illustrates the architecture of a Netgear GA620. 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 [11], 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 microcontroller 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 7 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.

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

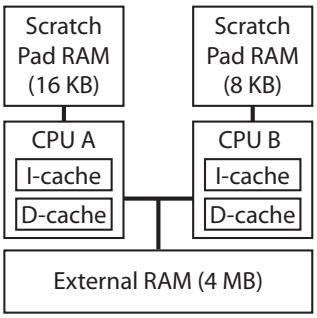

Figure 6: Netgear GA620 System Architecture.

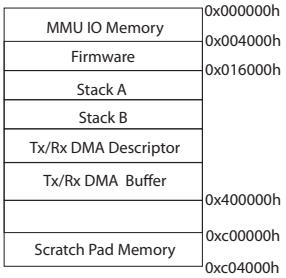

Figure 7: Netgear GA620 Memory Layout.

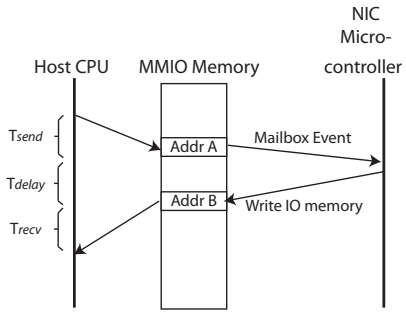

Figure 8: Host CPU to NIC communication via GA620's *mailbox*.

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.

5.2 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 4.3) 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 microcontroller A.
9. The verifier program informs both microcontrollers A and B to exit their attestation functions.

5.3 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. Figure 9 shows the pseudo-code to update one checksum state in each block. 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).

```
/* 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 9: Pseudo-code to update one checksum state vector.

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 10: Assembly instructions for one checksum block.

Figure 10 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 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.

5.4 Latency-Based Attestation

As described in Section 4.2.1, 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. 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.

5.4.1 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 [10]. 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. 3 and 6. Under these conditions, to detect a proxy attack, T^{per}_{comp} (Eqn. 3) and (if using the synchronized version) T^{n}_{diff} (Eqn. 6) must be sufficiently small that $T^{proxy}_{overhead}$ remains positive and detectable.

[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 [10].

5.4.2 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 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 4.2.1). 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 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 11 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 12 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.

5.4.3 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 5.4.2. 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 11 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 12 shows the time of 200 challenge-response pairs measured by the host CPU. The average value of a single challenge-response pair (T^{normal}_{comp}) 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^{per}_{comp}) on the NIC is about 827 nanoseconds, (i.e., $T^{per}_{comp} = T^{normal}_{comp} - (T^{cpu}_{send} + T^{cpu}_{recv}) =$ 2202 ns − 1375 ns = 827 ns). Thus, the overhead caused by the theoretical fastest proxy attack over 1 Gbps Ethernet is about 325 nanoseconds ($T^{proxy}_{overhead} = (T^{per}_{send} + T^{per}_{recv}) - T^{per}_{comp} =$ 1152 ns − 827 ns = 325 ns).

5.4.4 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 4.2.1) 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.

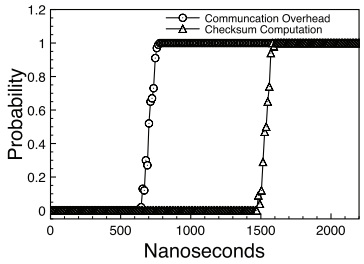

Figure 11: Impact of delay on probability that host CPU reads expected value from address B in a single MMIO read.

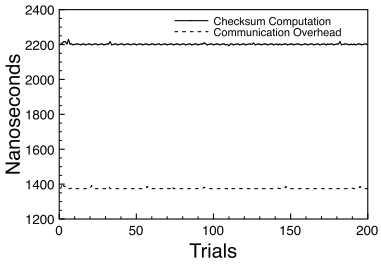

Figure 12: Communication overhead and checksum computation time (time of a challenge-response pair) measured by the host CPU.

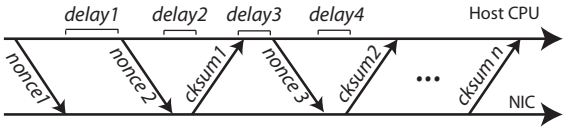

Figure 13: Verification procedure with synchronization between host CPU and NIC.

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 13 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 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 14 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), with a standard deviation of 19 ns. The entire time for performing 300 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.

[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 5.4.1. However, it is much shorter than the time of the real proxy attack we have implemented. We discuss this discrepancy further in Section 7.

6. EVALUATION

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.

Ethernet-based proxy attack. We implement a real Ethernet-based proxy attack (Figure 15). 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 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

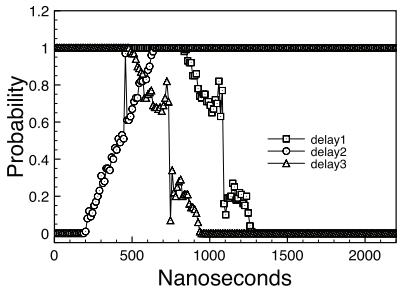

Figure 14: Impact of *delay*1, *delay*2, and *delay*3 in Figure 13.

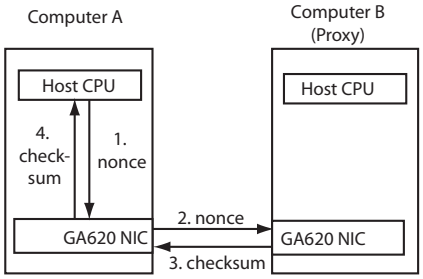

Figure 15: Proxy attack implementation.

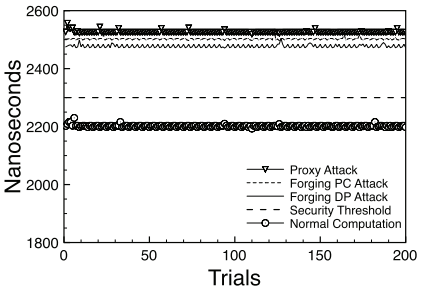

Figure 16: Attacker performance.

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.

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 design, 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.

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 checksum 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.

Evaluation Results. Figure 16 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.

7. 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 4.1. 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 6) 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 [22]. This level of variability may complicate the process of establishing baseline, or expected, behavior for VIPER on a particular platform. Additional investigation is warranted.

The Keyboard Conundrum. Passwords and bank account information frequently enter systems via the keyboard. Unfortunately, it is among the most resource-impoverished, and thus the most susceptible to being impersonated by other devices. It may be more practical to empower keyboards with some level of cryptographic awareness, than to truly verify every last legacy peripheral that gets dragged along in today's SuperIO chips.

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.

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.

8. RELATED WORK

To identify malware on devices within a computer system, Li et al. propose using software-based attestation to verify the firmware of an Apple aluminum keyboard that runs an 8-bit microcontroller [18]. The current work represents a significant extension in the same spirit as this work.

Duflot et al. [6] 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 [27]. 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.

We now chronologically review previous work on software-based attestation. Spinellis proposes "reflection" as an approach to verify the software running on a system [34]. 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 [17], 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 cache misses. By validating the number of cache misses the verifier can inspect whether the code was correctly executed. Shankar, Chew, and Tygar, however, point out several issues with their approach [33].

Seshadri et al. introduced software-based attestation and

developed SWATT, a system to verify the software of an embedded device [31]. 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 [29], verification of code running on an Intel Pentium IV processor through the Pioneer function [30], and code running on an AMD Opteron K8 architecture through the Outpost function [28]. Castelluccia et al. point out weaknesses in the specific SWATT and ICE functions [4], triggering significant discussion [25, 8]. 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 [9]. 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 [24]. Shaneck et al. explore the use of encrypted and self-modifying code to verify software on sensor nodes [32]. Their approach also relies on randomized traversal and timing. Jakobsson and Johansson have studied new approaches to software-based attestation on mobile devices [15, 16].

9. CONCLUSIONS

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.

Acknowledgements

We are particularly grateful to Loïc Duflot, Arrigo Triulzi, and Fernand Lone Sang for their help with peer-to-peer PCI transfers. We also wish to thank the (surprisingly many) anonymous reviewers for their time, attention, and valuable suggestions. Finally, we wish to thank Michael Stroucken for providing a Sun Fire V20z 1U rack-mount server to us, Michael Farb and Julie Bowman for improving the writing in the paper, and Virgil Gligor and Jim Newsome for stimulating conversations about new approaches for software-based attestation.

This research was supported in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389, MURI W 911 NF 0710287, and W911NF-09-1-0273 from the Army Research Office, 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, CMU, LMC, or the U.S. Government or any of its agencies.

10. REFERENCES

[1] Advanced Micro Devices, Inc. AMD I/O virtualization technology (IOMMU) specification. Publication No. 34434, Revision: 1.26, Feb. 2009.

[2] Advanced Micro Devices, Inc. AMD64 architecture programmer's manual volume 2: System programming. Publication No. 24593, Revision: 3.17, June 2010.

[3] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the ACM conference on Computer and Communication Security*, 2010.

[4] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the ACM Conference on Computer and Communications Security*, Nov. 2009.

[5] K. Chen. Reversing and exploiting an Apple firmware update. In *Black Hat*, 2009.

[6] L. Duflot, Y.-A. Perez, and B. Morin. Run-time firmware integrity verification: what if you can not trust your network card? In *CanSecWest*, 2011.

[7] L. Duflot, Y.-A. Perez, G. Valadon, and O. Levillain. Can you still trust your network card? CanSecWest, 2010.

[8] A. Francillon, C. Castelluccia, D. Perito, and C. 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`, Oct. 2010.

[9] V. Gratzer and D. Naccache. Alien vs. quine, the vanishing circuit and other tales from the industry's crypt. In *Proceedings of Eurocrypt*, May 2006.

[10] IEEE Computer Society: 802.3 Working Group. IEEE standard 802.3x-1997, 1997.

[11] A. N. Inc. Tigon Open Firmware. `http://alteon.shareable.org`.

[12] A. N. Inc. Tigon/PCI Ethernet Controlller (revision 1.04). `http://alteon.shareable.org`, 1997.

[13] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture. Order Number: 253665-073US, Jan. 2011.

[14] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual volume 3b: System programming guide, part 2. Order Number: 253669-037US, Jan. 2011.

[15] M. Jakobsson and K.-A. 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.

[16] M. Jakobsson and K.-A. Johansson. Assured detection of malware with applications to mobile platforms. In *Proceedings of the Workshop on Hot Topics in Security (HotSec)*, Aug. 2010.

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

[18] Y. Li, J. M. McCune, and A. Perrig. SBAP: Software-Based Attestation for Peripherals. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, 2010.

[19] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Apr. 2008.

[20] Mindshare Inc., R. Budruk, D. Anderson, and T. Shanley. *PCI Express System Architecture*. Addison-Wesley Professional, Sept. 2003.

[21] MindShare Inc., T. Shanley, and D. Anderson. *PCI System Architecture (4th Edition)*. Addison-Wesley Professional, June 1999.

[22] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs. In *Proceedings of the European Conference on Computer systems (EuroSys)*, 2011.

[23] NIST. Recommendation for key management. Special Publication 800-57 Part 1, Mar. 2007.

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

[25] A. Perrig and L. 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.

[26] F. L. Sang, Èric Lacombe, V. Nicomette, and Y. Deswarte. Exploiting an I/OMMU vulnerability. In *Proceedings of IEEE Conference on Malicious and Unwanted Software (Malware)*, 2010.

[27] F. L. Sang, V. Nicomette, Y. Deswarte, and L. 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.

[28] A. 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.

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

[30] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. In *Proceedings of ACM Symposium on Operating Systems Principles*, Oct. 2005.

[31] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[32] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *Proceedings of the European Workshop on Security and Privacy in Ad Hoc and Sensor Networks (ESAS)*, 2005.

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

[34] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security*, Feb. 2000.

[35] A. Triulzi. Project Maux Mk.II, I Own the NIC, now I want a shell. In *The 8th annual PacSec conference*, 2008.

[36] A. Triulzi. The Jedi Packet takes over the Deathstar, taking NIC backdoor to the next level. In *The 12th annual CanSecWest conference*, 2010.

[37] J. Wang, A. Stavrou, and A. K. Ghosh. HyperCheck: A Hardware-Assisted Integrity Monitor. In *Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2010.