PRISM: Enabling Personal Verification of Code Integrity, Untampered Execution, and Trusted I/O on Legacy Systems

or

Human-Verifiable Code Execution

Jason Franklin Mark Luk Arvind Seshadri Adrian Perrig

February 3, 2007
CMU-CyLab-07-010

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

PRISM: Enabling Personal Verification of Code Integrity, Untampered Execution, and Trusted I/O on Legacy Systems

or

Human-Verifiable Code Execution

Jason Franklin Mark Luk Arvind Seshadri Adrian Perrig
Carnegie Mellon University

Abstract

Today's computer users receive few assurances that their software executes as expected. The problem is that legacy devices do not enable personal verification of code execution. In addition, legacy devices lack trusted paths for secure user I/O making it difficult to ensure the privacy of data.

We present PRISM, a software-only human-verifiable code execution system that temporally separates a legacy computer system into a trusted component and an untrusted component. PRISM enables a user to securely interact with applications by establishing a trusted path and enables personal verification of untampered application execution.

PRISM enables the development of a new class of applications which we term personally verifiable applications (PVAs). PVAs have the property that a user can both securely interact with and execute these applications even in the face of a kernel-level compromise. We develop a personally verifiable digital signature application that assures the user that the password-protected private key is not misused and that neither the private key nor the password are disclosed to malware on the device. We describe an implementation of this application on a personal device, and evaluate the usability of our approach with a user study.

1 Introduction

An open research challenge is how to empower humans to *personally verify* what code is executing on their computational devices. Even trusted computing hardware cannot provide this property - as Balacheff et al. state in their book on TCG technology: "Unfortunately, the [code verification] mechanisms are inherently complex, and cannot be directly used by people – they require cryptographic operations and complex comparisons." [7].

We present a first step towards enabling a user to personally verify the execution of software, without the assistance of specialized hardware or an additional trusted computational device. Enabling a human to personally verify which code is executing on a device is an important problem. Currently proposed attestation technology, for example TCG [29], requires that a trusted device is used to verify an untrusted device. In this scenario and in most approaches to the untrusted computer problem [2, 8, 10, 12, 20, 22], the a priori existence of a trusted device is assumed. However, techniques to initialize trust in a device are rare, making the a priori existence of a trusted device questionable. By leveraging personal verification of code execution, we can overcome these problems: a trained user can initialize trust by personally verifying a device, then utilizing this newly trusted device to verify other devices. Hence, personally verifiable code execution addresses the problem of how we *initialize trust*.

This paper presents PRISM, a software-only verifiable code execution system that enables *personal verification* of untampered code execution. PRISM enables the creation of a trusted path completely in software, making it possible to ensure the integrity, authenticity, and confidentiality of user input and output (I/O) on legacy devices.

PRISM builds on tamper-evident software primitives [13, 24, 25]. These primitives enable runtime tamper-evidence through *optimality* and *enforced atomicity*. Tamper-evident software provides sufficient information in the form of output and timing for a verifier to detect tampering during execution, however, they require extensions to enable human-verifiable guarantees and trusted I/O functionality. Recent work has proven the existence of

secure tamper-evident functions based on the concepts of optimality and enforced atomicity [13]. The proof is architecture-dependent and based on a simple micro-controller. However, the result suggests that secure tamper-evident functions exist for other simple devices. This paper assumes the existence of a secure tamper-evident function for our architecture and provides a proof-of-concept implementation in Appendix A.

PRISM enables an important class of applications which we term *personally verifiable application* (PVAs). PVAs are applications which protect sensitive data and computation by utilizng personal verification of application execution and leveraging a software-only trusted path. We develop an example digital signature PVA. The digital signature PVA uses a software-only trusted path to maintain the confidentiality of program I/O and uses personal execution verification to ensure the authenticity and integrity of user input.

Usage Scenario. We detail a particular scenario where the infrequent execution of security-critical applications merits the use of a PVA. Consider Alice, the administrator for a security conscious organization such as a government national laboratory. Part of Alice's daily tasks include sending email notifications concerning the latest security patches relevant to her clients. As part of this task, Alice composes an email including a hyper-link to the appropriate patches, signs the email with her private key, and transmits the message. Upon receiving the message, Alice's clients verify its signature and now, trusting that Alice indeed sent the message, follow the included link to download and install the latest patches.

Alice uses a digital signature application to sign her message. Her private key is stored in a file and encrypted under a password. She invokes the application and passes the file to sign as an argument. The application prompts for a password, decrypts the private key, and signs the file. Trusting that her software operates as expected, Alice transmits her signed message. Even though Alice uses encryption and a password to protect her private key, malware on her device can compromise her security in a number of ways.

Malware on Alice's device might attempt to capture her private key or sign arbitrary messages, potentially modifying the included link to point to malicious patches. Malware could capture the password used to decrypt the private key, capture the unencrypted private key in memory, or sign an entirely different file. How can Alice obtain assurance that none of these malicious activities have occurred? By using a personally verifiable digital signature application in conjunction with PRISM, Alice can obtain a guarantee that the correct signature application has executed without the intervention of any malware that may be present on her device. PRISM provides these properties even if the OS kernel is compromised.

Summary of Contributions. We design, implement, and evaluate PRISM, a software-only human-verifiable code execution system which allows a human to personally verify untampered code execution. PRISM includes a software-only trusted path for user I/O.

2 PRISM System Architecture and Protocol Design

To personally verify code execution on a computing device, we propose that the user undertake the role of the verifier in a verifiable code execution protocol. Before presenting the details of our design, we state our assumptions and threat model, and describe verifiable code execution.

2.1 Assumptions

We assume that the hardware of the computing device is not malicious and that it matches the manufacturer specification. We assume that the computing device has a single CPU without virtualization support and that the interrupt handlers we install during the process of verifiable code execution are free of any software vulnerabilities. We assume that the computing device can be prevented from asking a faster computing platform (proxy) to perform computation on its behalf by disabling wired (disconnecting cables) and wireless network connections (by removing cards). Devices with built-in wireless communication interfaces cannot be used with PRISM unless the wireless interface can be disabled in a externally verifiable manner (e.g., power button). We assume that the user has access to trusted challenge-response pairs for their device; we discuss their generation and management in Section 2.7.

2.2 Threat Model

We consider a strong active adversary who may have complete control over all software on the device and complete control of the highest privilege level including control over the OS. The attacker, however, does not modify the hardware of the device. For example, the adversary does not load malicious firmware on peripherals to perform malicious DMA writes to the memory region containing the PVA. Also, the adversary does not program a benign peripheral device to overwrite PVA memory via a DMA write.[1]

2.3 Background: Verifiable Code Execution

Verifiable code execution is a challenge-response protocol between an external *verifier* and an untrusted device. Through the protocol, the verifier obtains the guarantee that an arbitrary piece of code, called the *target code*, on the device executes untampered from any malicious software that may be present on the device.

Malicious software can tamper with the execution of the target code in two ways: (1) it can modify the target code image before the target code is invoked for execution, and (2) it can modify the execution state of the target code, or the target code image, or both while the target code is executing.

To obtain the guarantee of verifiable code execution, the verifier needs to be able to detect if the execution of the target code is tampered using attacks (1) or (2). Attack (1) can be detected by asking the device for a guarantee of integrity of the target code image before the target code is invoked for execution. For detecting Attack (2), it is sufficient if the device provides to the verifier a guarantee of what code can execute concurrently with the target code.

The verifier can use the integrity guarantee of the target code image to ensure that the target code image was not modified before invocation for execution. The guarantee of what other code can execute concurrently with the target code can be used by the verifier to decide whether or not any concurrently executing code could tamper with the execution of the target code. In conjunction, these two guarantees enable the verifier to determine whether or not the execution of the target code on the device will be untampered by malicious code.

Consider an execution agent that exists on the device and constructs the above two guarantees. The execution agent on the device sends *integrity measurements* of the target code and all other code that can execute concurrently with the target code to the verifier. The integrity measurements need to be computed as a function of an unpredictable challenge sent by the verifier to prevent pre-computation and replay attacks.

An integrity measurement of a piece of code is either a cryptographic hash or a checksum of the executable image of the code. The integrity measurement constitutes a signature unique to the measured code image. Therefore, the verifier can use the integrity measurements both to determine what other code on the device can execute concurrently with the target code, and to determine if the integrity of the target code is intact.

In the rest of this section we give a detailed description of how the execution agent generates the two guarantees required for verifiable code execution. Readers who are familiar with earlier work on software-based verifiable code execution [24], namely the concept of an Untampered Execution Environment (UEE), may wish to skip directly to Section 2.4.

There are two questions that need to be answered to construct the required guarantees: one, how can we make an exhaustive list of all software on the device that can execute concurrently with the target code, and two, how is the execution agent on the device protected from malicious code?

Untampered Execution Environment. To make an exhaustive list of all code that can execute concurrently on a device with a single CPU, we examine how multiple threads can execute concurrently

[1] DMA write attacks are easily dealt with on most personal computing devices, which are the target of PRISM. Such devices use low-end CPUs whose cache controllers do not provide cache coherency for DMA writes. That is, the cache controllers do not automatically update the cache contents when there is a DMA write to the corresponding location in main memory. Since DMA writes do not propagate into the CPU caches, we can defend against DMA write attacks by ensuring that there is at least one piece of the PVA which is loaded into the cache at the start of execution and never leaves the cache till the execution of the PVA finishes. This piece can be used to load other pieces of the PVA into the cache from main memory, and to measure their integrity. We do not discuss this approach further in this paper, since the focus of the paper is not on designing primitives for externally-verifiable code execution.

on a single CPU. A single CPU can support only virtual concurrency, wherein the execution of different threads are time multiplexed on the CPU. The time multiplexing is accomplished with the help of interrupts. Once a thread of execution starts to execute on the CPU, the only way the thread relinquishes its control of the CPU is due to an interrupt, delivered either synchronously or asynchronously with respect to the thread's execution. If, before invoking the target code for execution, we modify all the interrupt handlers on the device so that they (A) do not call any other code on the device to handle the interrupt, and (B) transfer control back to the target code once they finish handling the interrupt, then the interrupt handlers are the only pieces of code that can execute concurrently with the target code. We call interrupt handlers that satisfy the conditions (A) and (B) *self-contained handlers* since their execution is guaranteed to not invoke any other code on the device.

Once we replace all interrupt handlers with self-contained handlers, we can guarantee that no other code on the device can execute concurrently with the target code other than the self-contained handlers. Therefore, the execution agent on the device only needs to send to the verifier integrity measurements of the target code and the self-contained handlers to prove untampered execution of the target code, along with a proof that all interrupt handlers have been replaced by self-contained handlers.

Two types of interrupts exist on any device: maskable and non-maskable. Maskable interrupts can be disabled by software whereas non-maskable interrupts cannot be disabled. In our implementation of verifiable code execution, instead of installing self-contained handlers for all interrupts, we mask all the maskable interrupts and only replace the non-maskable interrupt handlers with our own self-contained handlers. This is an optimization that reduces the size of the trusted computing base (code that can execute concurrently with the target code). For this optimization to be secure, we impose an additional restriction on the self-contained handlers: that they do not enable maskable interrupts when they execute; otherwise an unmeasured maskable interrupt handler could obtain control of the CPU during the execution of the self-contained handlers.

We call an environment in which maskable interrupts are disabled and the non-maskable interrupt handlers have been replaced by self-contained handlers an *untampered execution environment* (UEE). Setting up the UEE involves modifying the device's CPU state to disable maskable interrupts and modifying the Interrupt Vector Table (IVT) of the CPU to replace non-maskable interrupt handlers with the self-contained handlers. The UEE limits what other code on the device can execute concurrently with the target code. By doing so, the UEE allows us to precisely determine the trusted computing base required for untampered execution.

Tamper-Evident Function. To operate correctly, the execution agent on the device must be protected from malicious tampering. Various options exist to protect the execution agent, for example, the execution agent can be protected by specialized hardware. This, however, prevents verifiable code execution on legacy machines, which is why we utilize a software-only approach. Unfortunately, a software execution agent can be tampered with by malicious code on the device. To guarantee verifiable code execution of the target code, the verifier requires a guarantee of verifiable code execution for the software execution agent itself. To address this cyclic-dependence, we build a tamper-evident function into the execution agent which constructs a proof of its own integrity (checksum of its instructions), proof of the integrity of the rest of the execution agent (checksum of the rest of the execution agent image), and proof that the UEE is correctly set up (CPU state required for UEE set up).

We implement our tamper-evident function as an iterative checksum function which constructs the UEE, and then computes a checksum over its instruction sequence, the instructions of the rest of the execution agent, CPU state which needs to the modified to create the UEE, and the self-contained handlers. This checksum is computed as a function of a random challenge that is sent by the verifier to prevent pre-computation and replay attacks. The resulting checksum is a probabilistic argument that (I) the code of the checksum function and the rest of the execution agent are unmodified, (II) the CPU state for creating the UEE is correctly set up, and (III) the self-contained handlers are unmodified.

An attacker can attempt to forge the checksum result when conditions (I), (II), or (III) do not hold. We design our checksum function such that if an at-

tacker attempts to forge the checksum, the execution time of each checksum iteration is increased. In other words, we use time as a side channel to detect checksum forgery attacks. Since the checksum function is iterative, the time overhead of a checksum forgery attack is directly proportional to the number of iterations performed by the checksum function. Therefore, the time overhead can be increased or decreased depending on the requirements of the verifier.

If the checksum code is the fastest (time-optimal) code which fulfills the required conditions, then any modification of the code which also fulfills the required conditions will require more time to execute. If the verifier receives the correct checksum within the expected amount of time, then it is guaranteed that conditions (I), (II), and (III) above hold. Thereby, the verifier knows that the execution agent on the device will execute untampered.

After computing the checksum, the execution agent computes a cryptographic hash of the target code, returns the hash value to the verifier, and immediately invokes the target code by jumping to it. The target code in turn inherits the UEE of the execution agent. Once the verifier obtains the correct hash value, it obtains the guarantee that the target code image is unmodified and that the target code will execute in the UEE on the device. In this manner, the verifier obtains the guarantee of verifiable code execution of the target code.

2.4 Design Challenges and PRISM Overview

To verify code execution on a computing device, we propose that the user undertake the role of the verifier in a verifiable code execution protocol. Several challenges need to be addressed: (1) users cannot create random challenges, (2) users cannot compute the correct checksum, (3) users cannot accurately time the checksum computation, and (4) users cannot securely interact with an application without a trusted I/O path.

We propose the following design to address these challenges. We provide the user with access to trusted random challenges and their corresponding responses (Section 2.7 describes the details of how these trusted challenge-response pairs are generated). The user uses one challenge-response pair at a time to verify untampered code execution. The user also knows the *detection threshold D*, which is the maximum allowed time for the checksum function to run before the user declares the checksum invalid (Section 2.6 describes the details on how D is derived). The user employs a timer (e.g., their wristwatch) to measure the execution time. We design our checksum function to exhibit a high run-time overhead if the execution is tampered by malicious code, such that a user can detect the delayed checksum computation based on a coarse-grained time measurement. On our implementation platform, the best known software attack on the checksum computation experiences at least a 33% slowdown over the legitimate checksum, as we describe in Appendix A.

2.5 PRISM Details

In this section, we describe the PRISM protocol that is executed by the user and the computing device. Figure 1 shows the memory layout of the execution agent and the PVA on the computing device. The execution agent consists of six parts: the iterative checksum function $\mathcal{F}$, a function O that writes output to the display, a hash function H, a function P that sets up a trusted I/O path for the PVA, K a self-contained keyboard interrupt handler and device driver, and I the self-contained handler for all non-maskable interrupts. The self-contained handler is a dummy interrupt handler, consisting of an `interrupt-return` instruction, that unconditionally returns control every time it is invoked. Having a simple dummy handler for non-maskable interrupts makes it easy to verify that the handlers are self-contained and also reduces the size of the TCB for untampered code execution.

Figure 2 presents an overview of the PRISM protocol. The user holds a list of challenge-response pairs and uses a wristwatch to measure the execution time of the checksum function. The list contains the detection threshold D and pairs $\langle C_i, R_i \rangle$, where C_i is a challenge value and R_i is the expected checksum value.

To verifiably execute a PVA, the user starts PRISM by passing a pointer to the PVA to the execution agent. The execution agent sets up memory as Figure 1 shows, placing the PVA adjacent to itself in memory. The execution agent then prompts the user to enter a challenge. The user picks a fresh challenge-response pair $\langle C_i, R_i \rangle$ from the list, enters the challenge C_i into the computing device via the

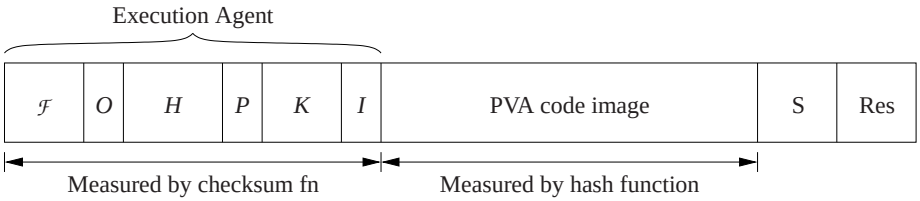
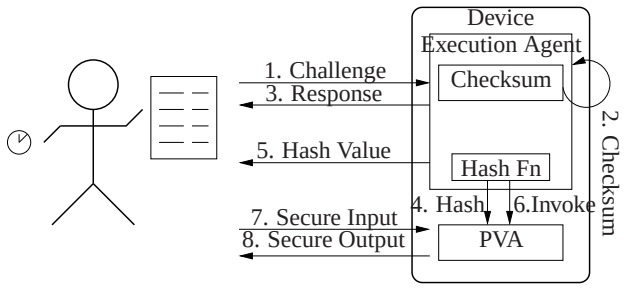


Figure 1: Memory layout of the execution agent and PVA. $\mathcal{F}$ is the iterative checksum function, O writes data to the display, H is a hash function, P is the function that sets up a trusted I/O path, K is the keyboard driver, I is the self-contained interrupt handler, memory area S is the PVA's heap and stack, and Res is the memory region where the PVA stores its results. The memory region from the checksum function to the self-contained handler is checked by the checksum while the hash checks the PVA code image. The PVA arguments and results are not checked.



(a) Functional diagram.

	$A:$	$t_1 \leftarrow$ current time, pick$\langle C_i, R_i, D\rangle$
1.	$A \rightarrow U:$	$\langle C_i \rangle$
2.	$U:$	$R' \leftarrow$ Checksum$(C_i,$ Exec Agent$)$
3.	$U \rightarrow A:$	$\langle R' \rangle$
	$A:$	$t_2 \leftarrow$ current time
	$A:$	if $(t_2 - t_1 \geq D \vee R' \neq R_i)$ Failure
4.	$U:$	$h \leftarrow$ Hash(PVA)
5.	$U \rightarrow A:$	$\langle h \rangle$
	$A:$	verify measurement result h
6.	$U:$	Invoke PVA.
7.	$A \rightarrow U:$	$\langle$Secure Input$\rangle$
8.	$U \rightarrow A:$	$\langle$Secure Output$\rangle$

(b) Protocol diagram. A represents user Alice, and U represents the untrusted device.

Figure 2: Overview of PRISM's execution. The numbers represent the temporal ordering of events.

device's keyboard, and records the current time t_1. Note that interrupts are enabled when the user enters C_i. Therefore, the user can interact with the device via the keyboard as usual.

As soon as C_i is entered, the execution agent invokes $\mathcal{F}$ with C_i as input. $\mathcal{F}$ sets up the UEE by disabling the maskable interrupts and replacing the non-maskable interrupt handlers with the self-contained handlers, as described in Section 2.3. Then, $\mathcal{F}$ iteratively computes a checksum over the memory region of the execution agent and the CPU state necessary to create the UEE, seeded with the challenge C_i. Upon completion of the checksum computation, $\mathcal{F}$ invokes O to display the checksum R' on the device's screen. Writing to the display does not require interrupts to be enabled since it can be accomplished by writing the output directly to the video framebuffer. When the user sees R', she notes the current time t_2, and verifies that the elapsed time $t_2 - t_1$ is less than the detection threshold D, and that the displayed checksum R' is equal to the expected result R_i. If $R' = R_i$ and $t_2 - t_1 < D$, the user receives the guarantee that the checksum function has executed untampered.

After displaying the checksum, O invokes H to compute the hash of the PVA. After H computes the hash of the PVA, O displays the hash value. The user verifies that the hash value displayed matches the expected value. After displaying the hash of the PVA to the user, O invokes P to set up a trusted I/O path between the PVA and the user.

The trusted I/O path guarantees that no software on the device other than that which has already been measured will handle the I/O between the PVA and the user, and provides confidentiality and authenticity of all I/O between the user and the PVA. The trusted I/O path uses the device's screen for output and the keyboard for input. It uses O to display output to the device's screen. Unlike displaying information on the screen, which can be done efficiently without interrupts, using the keyboard requires interrupts to be enabled for efficiency. Assuming that maskable interrupts cannot be selectively reenabled, using the keyboard for trusted input requires that all maskable interrupts be reenabled and that a mea-

sured interrupt handler exist for the keyboard interrupt. The function *P* registers the self-contained keyboard interrupt handler *K* and device driver that is part of the execution agent to handle the keyboard interrupt. It also registers the self-contained handler, *I*, as the interrupt handler for all other maskable interrupts. *P* then reenables all maskable interrupts and invokes the PVA for execution. There now exists a trusted I/O path through which the user can interact with the PVA.

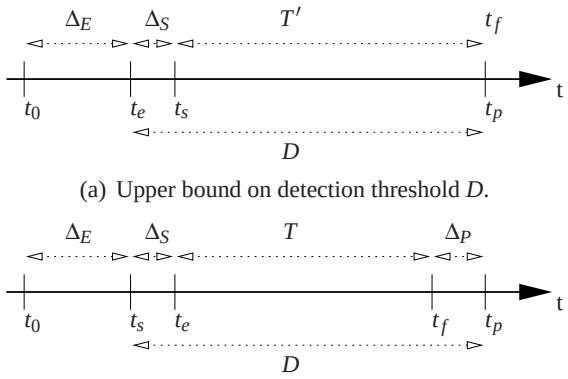
After the PVA terminates, it returns control back to the execution agent. The execution agent enters the clean-up phase, overwriting all PVA and execution agent memory, except the memory region Res, to protect potentially sensitive PVA data and restores the CPU of the device to its original state.

2.6 Timing Considerations

Given the hardware configuration of the computing device (i.e., processor and clock speed), the required number of iterations is selected such that a compromised checksum function exhibits a human-detectable slowdown. The detection threshold D is thus picked as described in this section. Let T be the running time of the checksum function and let T' be the running time of the fastest malicious checksum function, where $T' = (1+\delta)T$ and δ is the minimum performance overhead of a compromised checksum function (on our implementation platform, $\delta = 0.33$ as we describe in Appendix A).

Let Δ_E be the time required for the user to enter the challenge value, Δ_S be the time required to get the current time (this could be negative if the user notes the current time before entering the final character of the challenge value), and Δ_P be the time to get the time after the device displays the response. Figure 3 depicts these events and time durations.

To derive an upper bound on the detection threshold D, we need to ensure that no false negatives are possible. A false negative is the event that a malicious checksum function had been accepted by the user as being legitimate. This could happen if the perceived execution time of the malicious checksum function is less than the detection threshold D. Thus, the detection threshold D cannot be greater than a realistic estimate of the shortest perceived execution time of a malicious checksum function. In the worst case, we need to assume a maximum value of Δ_S, which we write as $\max(\Delta_S)$, and that $\Delta_P = 0$, such that the perceived time duration for the checksum computation is minimal. Figure 3(a) depicts this case. (This is quite a pessimistic estimate, since in reality, Δ_S is very small [the user types the last character of the challenge value at the same time as getting the time] and the value of Δ_P is large due to the user's reaction time.) Hence, $D < T' - \max(\Delta_S)$.



(a) Upper bound on detection threshold D.

(b) Lower bound on detection threshold D.

Figure 3: Timing diagrams. t_0 and t_e represent the start and end time for entering the challenge value, respectively. t_s and t_p represent the start and stop times for the timer. t_f represents the finish time for the checksum function. Given these timings, we obtain $\Delta_S = t_s - t_e$ and $\Delta_P = t_p - t_f$.

To derive a lower bound on the detection threshold D, we need to ensure that false positives are infrequent. A false positive is the event that a legitimate checksum function had been perceived to be malicious. This could happen if a legitimate checksum function's perceived runtime is longer than the detection threshold D. Thus, D cannot be less than a realistic estimate of the longest perceived execution time of a legitimate checksum.

In the worst case, we need to assume that Δ_P is at the maximum value ($\max(\Delta_P)$) and that Δ_S is at the minimum value, which can be a negative value ($\min(\Delta_S)$), i.e., the user has a slow reaction time but notes the time before entering the last character of the challenge value. Figure 3(b) depicts this case. Hence, $D > T + \max(\Delta_P) + |\min(\Delta_S)|$.

Since $T' = (1+\delta)T$, where δ is the performance overhead of the malicious checksum function, we obtain the following inequality for the detection threshold D by combining the upper and lower bound: $T + \max(\Delta_P) + |\min(\Delta_S)| < D < (1+\delta)T - \max(\Delta_S)$

Since Δ_S and Δ_P depend on the user, we will need to set T appropriately to enable a solution. T depends on the number of iterations of the checksum loop, thus, we need to iterate the checksum loop sufficiently many times to satisfy the inequality. Although we could optimize T based on the user, for simplicity we have chosen to use a conservative value that covers the majority of users. As we describe in our implementation section, we have chosen a value for T that can handle $\min(\Delta_S) = -1$s, $\max(\Delta_S) = 1$s, and $\max(\Delta_P) = 2$s, which suffices for user reaction times of maximally 2 seconds. From our implementation we obtain $\delta = 0.33$ and derive from the inequality that $T = 12$s, $T' = 16$s, and $D = 14$s. Given the magnitude of these timing values, differences in temperature that affect the clock speed of the device add negligible variance.

2.7 Challenge-Response Pair Generation and Management

In this section we discuss how the user obtains a trusted list of challenge-response pairs. The requirements for a trusted pair are that the challenge value is unpredictable, and that the challenge-response values remain secret. Two main security vulnerabilities exist in this context: (1) the pair creator itself could be malicious, and generate challenge-response pairs for malicious code, and (2) a legitimate challenge-response pair had been leaked to the device, perhaps by eavesdropping on the channel by which the challenge is delivered.

All techniques to generate challenge-response pairs must rely on some initial trust assumptions. Our approach to challenge-response pair generation is as follows. First, we discuss how we can use one trusted challenge-response pair to securely bootstrap generation of new pairs. This reduces the problem significantly. Instead of requiring the user to gain access to a list of challenge-response pairs, we only require the user to gain access to one such pair. Next, we discuss different methods of providing the user with the initial challenge-response pair, in such a way to minimize the trusted computing base.

Extending lists of pairs. The user can use one challenge-response pair to generate new challenge-response pairs. This can be achieved by including the challenge-response pair generation program as a PVA. The user issues the initial challenge-response pair, and if it verifies correctly, the user has a guarantee that the challenge-response pair generation program is executing in an untampered execution environment. Any newly generated challenge-response pairs can be displayed to the user from within the untampered execution environment with trusted output. These challenge-response pairs can be trusted, and can be used later to bootstrap future challenge-response pair generation.

Trusted manufacturer generates initial pair. The scheme of extending challenge-response pairs still requires the existence of one authentic challenge-response pair in the beginning. We propose to have a trusted manufacturer provide unique challenge-response pairs. The manufacturer could distribute pairs along with the device or provide a telephone or fax service. Defending against eavesdropping attacks is beyond the scope of the paper.

We note that trusting the manufacturer does not require trusting additional entities. We already trust the manufacturer to guarantee the hardware configuration of the device. We do require the user to now trust the manufacturer to create secure challenge-response pairs. A potential scenario for challenge-response pair generation follows. Each manufacturer could utilize a dedicated device equipped with a field programmable secure co-processor system like the IBM 4758 [11]. Such devices would be disconnected from the network and would not interact with any other machines or execute any other programs except the challenge-response pair generation program. Furthermore, since there are very few of these dedicated machines, it is plausible to physically secure them (i.e., inside a locked room). This approach reduces the trusted computing base for pair-generation to these few challenge-response pair generators.

3 Implementation

We describe the implementation of the execution agent and the digital signature PVA. A detailed description of the implementation can be found in Appendix A.

3.1 PRISM Implementation

We implement the execution agent on the Sharp Zaurus SL6000 Personal Digital Assistant (PDA). This PDA has a 400 MHz Intel XScale-PXA255 processor, 64 MBytes of memory, and runs the XScale port

of version 2.4.18 of the Linux Kernel. The XScale PXA255 is a 32-bit RISC processor, with 16 general purpose registers (including a program counter), and two processor status (flags) registers [28].

Checksum Function. Our checksum function, $\mathcal{F}$, is an iterative function that uses all general-purpose registers of the PXA255 to generate a 68-byte checksum, which we represent as a vector of seventeen 32-bit checksum pieces. Each checksum piece uses one 32-bit CPU register. Each iteration of our checksum code performs the following: (1) derives a pseudo-random number using a 32-bit T-function [18], (2) reads the memory word based on a PC-relative address generated by the pseudo-random number, and (3) updates one checksum piece based on the memory word read and CPU state information from the CPU status (flags) registers.

We implement the execution agent as a Linux kernel module. The checksum function, $\mathcal{F}$, is written in assembly, while the rest of the execution agent is written in C. The user invokes PRISM by running a shell script that loads the kernel module. After the kernel module loads, the script prompts the human for the challenge, which is an 80-bit nonce, in the form of 16 alphanumeric characters. The 80-bit nonce is used to seed the T-function, and to initialize the seventeen checksum pieces. The kernel module then executes the PRISM protocol on the Zaurus as described in Section 2.5.

If the protocol terminates successfully, the human obtains the guarantee of untampered execution of the PVA and can interact with the PVA via a trusted I/O path.

3.2 Personally Verifiable Digital Signature Application

We now describe an implementation of a PVA in the context of a secure digital signature application that provides the guarantee that only the correct specified message is signed by the user's private key. To obtain this guarantee, the digital signature application executes as a PVA in the untampered code execution environment instantiated by PRISM.

In this application, a user stores a private signature key encrypted under a password on the file system. To digitally sign a message, the user starts the signature application, enters the password to decrypt the private key, and the application uses the decrypted private key to digitally sign the message. Without knowing the password, an attacker is unable to sign arbitrary messages.

Many avenues of attack exist. Malware can attempt to steal the password used to decrypt the private key, or capture the decrypted private key in memory, or even pass a different message to be signed. In current computer systems, when we use such an application, we can never be sure that the correct message has been signed and that none of these attacks has occurred. By implementing the signature application as a PVA, we guarantee to the user that none of these attacks has occurred.

Implementation Details. The digital signature application consists of two modules: an AES-CBC module and an RSA module. The AES module decrypts the private key file, while the RSA module generates digital signatures. For this prototype implementation, both modules were implemented inside the same kernel module as the checksum code.

The protocol proceeds as follows. Before invoking PRISM, the user specifies the filename of the encrypted private key and the filename of the message to be signed. Initialization code loads these files into memory and makes the corresponding memory pages available to the PVA as illustrated in Figure 1. Next, the user initiates PRISM and loads the corresponding digital signature PVA into memory. If the user is satisfied with the computation time and response of PRISM, the user would interact with the digital signature application and enter the password to decrypt the RSA private key stored on the device. The private key is encrypted under a 128-bit AES key that is derived from a hash of the password.

Next, the user specifies the message to be digitally signed by typing the message in its entirety or using the previously loaded file. To ensure that the correct message is being signed, the application displays the message before generating the signature. The PVA then stores the digital signature in the memory region Res as shown in Figure 1. Finally, the clean-up phase of the signature application overwrites all intermediate state information, such as the password and the decrypted private key.

As previously described, the TP/UEE module ensures that the untampered execution environment has been correctly set up and that malware cannot gain control to tamper with the execution of the signature

application. The keystrokes are read in entirely by verified handlers, thus, malicious code cannot eavesdrop them. Since no malicious code can interfere with the execution, the private key will remain secret, and only the desired message will be signed.

4 Security Analysis of Human Factors

The security analysis in this section encompasses only the extensions to tamper-evident software primitives required to enable personal verification. A general security analysis of tamper-evident primitives can be found in previous work [13, 24, 25].

4.1 Attacks on User Timing Verification

To attack timing verification, malicious code on the user's computing device attempts to guess possible challenge values based on the user's partially typed challenge. By precomputing with possible values, the malicious code hopes to gain a time advantage that would allow for the execution of malicious code within the detection threshold.

There exist multiple ways for an attacker to guess possible challenge values. One possible technique is for an attacker to start a thread for each possible challenge value. Each time an additional character of the challenge is input, the attacker can kill threads that correspond to challenge values that do not match the real challenge. Each additional character of the challenge reduces the search space by a factor of the number of possible values for a single character of the challenge values, resulting in an exponentially decreasing search space. The end result is that the attacher obtains an early start on computing the response.

If the malicious code correctly guesses the challenge within α seconds of the input of the final character of the challenge, then the attacker has gained an precomputation advantage of α. This precomputation advantage is only useful in masking execution tampering if it meets or exceeds the overhead of the malicious checksum function. Otherwise, the execution tampering will be detected.

Let k_i denote the ith challenge character keypress. When a user inputs an n character challenge, we obtain a sequence $k_1, k_2, \ldots, k_n$. Assume we are able to bound the time between keypresses below by T_{min} and above by T_{max} and the attacker needs to correctly guess the correct challenge value Δ_A seconds before k_n. In the worst case, the user delays T_{max} between each keypress and the attacker needs to guess $s = \lceil \frac{\Delta_A}{T_{max}} \rceil$ keypresses. Given c possible characters for each keypress, the attacker succeeds with probability $\frac{1}{c^s}$.

A promising defense against probabilistic guessing attacks is to perform mandatory postcomputation after each keypress. This postcomputation would take as input the partially input challenge and produce a value which is input to the subsequent postcomputation after a specified amount of time. The intuition behind this defense is that minimizing the time available to the attacker increases the number of keystrokes ahead that are required to be guessed. If we perform T_{pc} seconds of mandatory post computation after keystrokes $k_1, k_2, \ldots, k_{n-1}$, the attack must guess $s' = \lceil \frac{\Delta_A}{T_{max} - T_{pc}} \rceil$ keypresses. Since the attacker's usable time for computation is reduced, the attacker's probability of success is correspondingly reduced to $\frac{1}{c^{s'}}$ where $s' > s$.

The probability of a successful guessing attacks can also be decreased by increasing the rate of user input. If a human user types in a challenge arbitrarily slowly (T_{max} tends to ∞), the attacker's precomputation advantage can be unbounded. In protocols which rely on time, even coarse-grained timing like that of PRISM, the user must enter the characters of the challenge at a reasonable rate to minimize the likelihood of a successful guessing attack.

4.2 Attacks on Challenge-Response Verification

The security of our scheme relies on the assumption that the challenge-response pairs used to check a computing device are fresh. In addition, the challenge must be kept secret until input into the device, and the response must be kept secret until the checksum function finishes executing and displays a response. If these assumptions are violated, the undetectable execution of malicious code is possible.

4.3 Attacks on User Attention

Humans are susceptible to coercion and confusion. A malicious device may attempt to confuse users in a number of ways. For example, a malicious device may attempt to gain a precomputation advantage by requiring a user to press `return` after entering the challenge or by displaying prompts requesting spurious user input. A malicious device may also attempt

to delay the start of the timer by explicitly instructing the user to start timing, or may signal the completion of the checksum function early as a way to minimize perceived execution time. Social engineering attacks like these are common in protocols which involve humans [16]. Since PRISM targets sophisticated users like administrators, the impact of these attacks can likely be countered with appropriate user training or other verification mechanisms [16]. An open research is to increase the robustness of human-in-the-loop protocol robustness to such attacks.

5 Usability Analysis

In this section, we evaluate the PRISM design and the usability of PVAs. Our user study evaluates the usability of the personal verification process.

5.1 Potential User Errors

Our user study evaluates the following user errors: a device returns an incorrect response and the user fails to correctly verify the response; the checksum computation is delayed, but the user fails to identify the delay.

5.2 User Study

Participants. We tested the digital signature PVA on a total of 12 users. Other than the prerequisite of having some level of experience with mobile devices, we recruited users from a diverse technical background and age range. Seven users were male and five were female. There were five undergraduate students, three graduate students, three staff and one faculty. Five users were between 18 and 22 years of age, five were between 23 and 30 years of age, and two were older than 31 years. Among the users, two currently use a PDA, and ten do not.

Experiment Setup. Each user was given a Sharp Zaurus PDA, a watch, and a sequence of challenge-response pairs printed on a sheet of paper.

Three verification functions are present on the Zaurus, one correct one and two malicious ones. One malicious function displays an incorrect checksum within the detection threshold D, while another displays the correct checksum but takes longer than D to execute.

There are a total of four test scripts presented to the user. Two of them correspond to the correct execution agent, while the other two scripts correspond to the two incorrect execution agents. We purposely avoid a one-to-one correspondence in an effort to prevent the user from guessing which script invokes which execution agent via a process of elimination.

The four test scripts are presented to the user in a pre-determined order. The user is asked to determine the correspondence between the test scripts and the execution agent based on two factors. One, the execution time as recorded by the watch, and two, whether the sixteen character checksum response matches with the response on the printed sheet. The user is asked to perform two trials for each test script using a different challenge each time.

Procedure. Before the experiment begins, we explained to the users the purpose of our study, and encouraged them to think aloud and voice their opinions on any features. Then, each user is presented with the PDA and a watch. We present to the user the following scenario where she needs to input her password in a secure manner in order to securely sign a message. Next, the four test programs are executed with fresh challenge-response pairs randomly drawn. Finally, we asked the user to complete a brief demographic survey and solicited for general comments. In some cases, we would redirect the user's attention to portions of the interface that she may have overlooked during testing.

5.3 Experimental Results

We consider the trial to be a false positive if the user perceived the device to be malicious even though the correct execution agent was executed: false positives do not endanger the secrecy of the password or private key, however they negatively affect the user experience. As we see from our results, we achieve a false positive rate of 0.

Error Type	Rate
False Positive	0 / 24
False Negative (delayed execution)	0 / 12
False Negative (incorrect checksum)	1 / 12

Figure 4: Results of user study

A false negative occurs when the user fails to detect the execution of a malicious execution agent. The false negative rate is more important because a single case could potentially leak the secrecy of the password and/or private key.

We implemented two malicious execution agents: delayed execution with correct checksum output, and

on-time execution with incorrect checksum output. A false negative in the former implies an error in the user's measurement of the execution time, while a false negative in the latter implies that the user failed to recognize the incorrect checksum response. As our user study shows, none of the twelve users had any problem with the maliciously delayed response, while one user failed to verify the checksum.

Our most important task is to study whether users are able to use our scheme to verify software integrity. As our results show, our users were able to successfully classify whether or not the program was malicious in 47 out of 48 tests despite the users' lack of familiarity and minimal training.

Out of our 48 tests, the only error occurred when a user failed to realize that the 16 character checksum response was different from the corresponding challenge-response pair. All other test cases, including all test cases relying on time-delayed execution, were correctly answered. This finding supports the use of time as a side-channel to verify software and that the detection threshold we calculated is reasonable. The only error was due to a human error in verifying a sixteen character checksum. A potentially alternative to character checksum verification would be to use a visual hash function [21] to simplify this task.

After the user study, we asked our participants if they would in fact use such a verification system for their security-sensitive tasks. Half of the respondents responded that they would happily use PRISM, a promising percentage! Amongst negative responses, the long execution time was cited as the most significant drawback.

6 Related Work

Related work falls into three categories: load-time attestation, verifiable code execution, and attacks on software-based integrity mechanisms.

Load-time Attestation. Load-time attestation [3–5, 17, 19, 23, 25, 29] is a primitive that allows an external verifier to verify the integrity of the memory contents of an untrusted platform. In general, load-time attestation primitives cannot be used for verifiable code execution because of the time-of-check-to-time-of-use (TOCTTOU) problem. Since load-time attestation only provides load-time guarantees, an attacker who modifies the target executable after attestation, but before it is invoked for execution, would remain undetected.

Verifiable Code Execution. Several methods exist to support verifiable code execution by an external device, however, none of them can be easily used by a human. AMD's Pacifica [1] and Intel's LT [14] are two next-generation hardware platforms which provide the guarantee of verifiable code execution. They allow a computing device to prove to an external verifier that a particular executable had been invoked for execution. Unfortunately, Pacifica and LT are inadequate for our needs because the external verifier cannot be a human. Balacheff et al. propose a mechanism for verifying that the correct document is being signed by a private key [6]. They suggest to leverage a special TPM with a secure path to the display as well as a secure link to a smart card. The TPM would then display a secret image on the screen to indicate to the user that the secure output mode is on and display the message to be signed within that image. Chen and Morris propose Cerium, an approach that relies on a physically tamper-resistant CPU with an embedded public-private key pair and a micro-kernel that runs from the CPU cache to provide a remote host with the guarantee of verifiable code execution [9]. Shi et al. propose BIND as a technique to verifiably bind the output of a code to the code that generated it [27]. BIND uses LT or Pacifica-style CPU extensions to obtain an assurance that a Secure Kernel (SK) was correctly loaded into memory. BIND then uses the SK to obtain a guarantee of what code was executed to generate a given output.

Attacks on Software-based Integrity Mechanism. Genuinity is software-based attestation techniques that relies on a verification function to generate a checksum of the memory contents [17]. The verification function is designed so that if an attacker tampers with the checksum computation, the time taken to compute the checksum increases. Genuinity is vulnerable to a data-substitution attack described by Shankar et al. [26]. The Split-TLB attack is a powerful attack against software-based integrity mechanisms such as self-checksumming software primitives [30]. Since self-checksumming code reads its own instructions as data, during execution the I-TLB and D-TLB point to the same physical page. The attacker sets up the TLB such that the same virtual

address points to two different physical addresses. Since only the virtual address is used in the checksum computation, malicious code can execute and still produce the correct checksum. Our implementation defends against the Split-TLB as we describe in Appendix A.

7 Concerns

We address a number of potential concerns which may remain.

Applicability. We have described only a narrow well-motivated application of PRISM. This is in part to convince the reader that there is an important scenario where the assumptions that underly PRISM's design are fulfilled. Numerous additional applications exist that satisfy our assumptions, in particular, whenever a trusted personal device is required and can be disconnected from the network.

Human Intelligence. PRISM is a human-in-the-loop protocol which relies on the intelligence of the user. Humans, however, are not perfect and user error is inevitable. With this in mind, it is important to note that while user verification errors reduce the security of PRISM, the reduction only returns the user to the security level they were at before using PRISM.

Practicality. We do not claim that our system is a panacea for human assessment of the trustworthiness of a system. Today's users have few techniques to evaluate the untampered execution of code on their devices. PRISM provides this ability as a technique to initialize trust in personal devices.

Security. It is not a goal of this paper nor is there sufficient space to reproduce the security analysis done in previous work [24]. Instead, we specifically treat the human factor-based extensions made to previous work and provide a proof-of-concept implementation in Appendix A.

Simplicity. PRISM is a simple protocol; this simplicity is necessary when imposing verification on a human. Even if the reader disagrees with the approach described in this paper, the contributions of PRISM are not just its realization, but also the principles and ideas this paper develops in relation to trust and what we believe is a unique perspective on the positioning of humans in trustworthy computing.

Usability. Using PRISM imposes a greater burden on the user than not using PRISM at all, however PRISM enables important functionality which improves the security of systems even in the face of a kernel-level compromise. As evidence of user interest, half of the users in our user study stated they would be happy to use PRISM for their security-sensitive tasks. To improve the usability of PRISM and better understand the concerns of users, we intend to further study the feedback of users who felt the overhead of PRISM outweigh its benefit.

Widespread Adoption. This paper does not evaluate if PRISM will achieve widespread adoption nor make any such claims. We believe that this question is difficult to predict and hence cannot be a necessary criteria for research. The primary goal of this paper is to develop reasonable techniques to allow humans to personally verify code execution, a task which was previously difficult. We believe PRISM is a promising approach to enable new opportunities to improve the security of systems.

8 Conclusion

PRISM is a first step towards the goal of human-verifiable code execution on legacy devices. Much work remains including an extended user study and a thorough evaluation of PRISM's applicability, practicality, and potential for widespread adoption. These assessments are not likely to be successful without a prototype implementation and preliminary evaluation, which is exactly what this paper provides. PRISM and our PVA are available for download at: http://www.ece.cmu.edu/∼mluk/hvce.taz

9 Acknowledgments

First and foremost, we thank John Bethencourt for useful discussion early in the design of PRISM. We thank Sachin Kulkarni for designing and implementing an early version of the verification function for the XScale architecture. We also appreciate the useful feedback of Michael Bailey, Dan Boneh, Evan Cooke, Virgil Gligor, Jason Hong, Yoshi Kohno, Jon McCune, Robin Sommers, and Nick Weaver.

References

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

[2] Martín Abadi, Michael Burrows, Charles Kaufman, and Butler Lampson. Authentication and delegation with smart-cards. *Science of Computer Programming*, 21(2):91–113, October 1993.

[3] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 65–71, May 1997.

[4] William A. Arbaugh, Angelos D. Keromytis, David J. Farber, and Jonathan M. Smith. Automated recovery in a secure bootstrap process. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, pages 155–167, March 1998.

[5] Reflection as a Mechanism for Software Integrity Verification. Diomidis spinellis. In *Proceedings of ACM Transactions on Information and System Security, Vol. 3, No. 1,*, February 2000.

[6] Boris Balacheff, Liqun Chen, David Plaquin, and Graeme Proudler. A trusted process to digitally sign a document. In *Proceedings of Workshop on New Security Paradigms*, pages 79–86, 2001.

[7] Boris Balacheff, Liqun Chen, David Plaquin, Graeme Proudler, and Siani Pearson. *Trusted Computing Platforms: TCPA Technology In Context*. Prentice Hall, 2002. ISBN 0-13-009220-7.

[8] Dirk Balfanz and Edward W. Felten. Hand-held computers can be better smart cards. In *Proceedings of the 8th USENIX Security Symposium*, Washington, D.C., USA, August 1999. USENIX.

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

[10] Dwaine Clarke, Blaise Gassend, Thomas Kotwal, Matt Burnside, Marten van Dijk, Srinivas Devadas, and Ronald Rivest. The untrusted computer problem and camera-based authentication. In *International Conference on Pervasive Computing*, 2002.

[11] Joan Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[12] Howard Gobioff, Sean Smith, J.D. Tygar, and Bennet Yee. Smart cards in hostile environments. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, Oakland, California, November 1996. USENIX.

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

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

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

[16] Chris Karlof, Naveen Sastry, and David Wagner. Cryptographic voting protocols: A systems perspective. In *Proceedings of the Fourteenth USENIX Security Symposium (USENIX Security '05)*, 2005.

[17] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th USENIX Security Symposium*, pages 295–308. USENIX, August 2003.

[18] A. Klimov and A. Shamir. A new class of invertible mappings. In *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2003.

[19] Jr. Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot, a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[20] Alina Oprea, Dirk Balfanz, Glenn Durfee, and D. K. Smetters. Securing a remote terminal application with a mobile trusted device. In *20th Annual Computer Security Applications Conference (ACSAC'04)*, 2004.

[21] Adrian Perrig and Dawn Song. Hash visualization: A new technique to improve real-world security. In *Proceedings of the 1999 International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*, pages 131–138, July 1999.

[22] Andreas Pfitzmann, Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Trusting mobile user devices and security modules. *IEEE Computer*, 30(2):61–68, February 1997.

[23] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*. USENIX, August 2004.

[24] 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 1–15, October 2005.

[25] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[26] Umesh Shankar, Monica Chew, and J.D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th USENIX Security Symposium*. USENIX, August 2004.

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

[28] A. Sloss, D. Symes, and C. Wright. *ARM System Developer's Guide*. Morgan Kaufmann Publishers, 2004.

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

[30] G. Wurster, P.C. van Oorschot, and A. Somayaji. A Generic Attack on Checksumming-Based Software Tamper Resistance. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

A Implementation Details

We have implemented PRISM on the Sharp Zaurus SL6000 Personal Digital Assistant (PDA). This PDA has a 400 MHz Intel XScale-PXA255 processor, has 64 MBytes of RAM, and runs a 2.4.18 Linux Kernel specialized for embedded devices. The PXA255 processor is based on the ARM Version 5TE architecture, excluding float point instructions.

A.1 Design and Implementation of Checksum Code on XScale

We base our checksum code on previous work in verifiable code execution on the x86 architecture [24]. However, differences between the XScale and x86 required that we completely redesign the execution agent. As we detail below, the simplicity of XScale made it easier to implement a secure execution agent. We first describe the design of our checksum code. Then, we explain how an adversary tampering with its execution would either result in a wrong checksum or a noticeable time delay. Finally, we discuss how we set up an untampered execution environment.

Architecture Primer. The XScale PXA255 is a 32-bit RISC processor, with 16 general purpose registers (including a Program Counter), and two processor status registers [28]. The two status registers are the current process status registers, `cpsr`, and the saved process status register, `spsr`. There are seven processor modes in total: six privileged modes (*abort, fast interrupt request, interrupt request, supervisor, system, and undefined*), and one non-privileged mode (*user*).

There are significant differences between the XScale architecture and the x86 architecture that are particularly relevant to crafting the checksum code. In XScale, the general purpose registers are orthogonal. This means the user can write and read the PC (`r15`) like any other general purpose register, which greatly simplifies the checksum code. Also, the way the processor manages different privilege levels and interrupts is very different from the x86 architecture, and a thorough understanding of its intricacies were needed to set up the untampered execution environment.

Checksum Design

The checksum code needs to be constructed in such a way that tampering with its execution would either generate the wrong checksum or a runtime delay. The input to the checksum function is (1) a 68-byte output generated by feeding the typed challenge into a pseudo-random number generator and (2) a memory region to verify (which is the execution agent, as Figure 1 shows). The checksum code computes a checksum over the specified memory region, and returns a 68-byte response.

Our checksum code is a time-optimal iterative function that uses all general-purpose registers on our platform (15 in our case), and generates a 68-byte checksum, which we represent as a vector of seventeen 32-bit checksum pieces. Each checksum piece uses one CPU register: twelve in general purpose registers, and five in status registers.Each iteration of our checksum code performs the following: 1) derive a pseudo-random number, 2) read the memory word based on PC-relative address, 3) update one checksum piece based on the memory word read and other state variables, 4) update program state. Below, we explain the purpose of each step.

Initialization. Before checksumming begins, we first initialize the checksum with the challenge. This is to prevent the adversary from pre-computing the checksum.

Step 1) The first operation in a checksum loop is to generate a pseudo-random memory address to perform a PC-relative memory read. The key insight behind this pseudo-random memory traversal is that an attacker cannot predict what memory location will be read next. Thus, if the attacker alters the memory, it would need to check whether the current memory access falls within the modified region. Even if the attacker only modified one byte, the increased runtime of the memory check will be noticeable.

Step 2) We derive the 32-bit address of the memory location based on the output of a 32-bit T-function, $x = x + (x^2 \vee 5) \ mod \ 2^n$, where $\vee$ is the bitwise or operator [18]. A T-function is a function from n-bit words to n-bit words that has a cycle length of 2^n. Such a function acts as a pseudo-random number generator. The T-function's initial value as well as initial checksum value are derived from the challenge issued by the user.

Step 3) The checksum function updates one checksum piece based on the pseudo-random memory location and other state variables. State variables include the address of the memory read, the current PC, and number of words read so far.

Step 4) Finally, the state variables are updated, and the process iterates for a different checksum piece. The small code size of the checksum loop is required in order for the addition of instructions to exhibit a noticeable runtime overhead.

The checksum code needs to run at the highest privilege level with maskable interrupts disabled. On the XScale platform, the checksum code will run at *supervisor* mode. This information is included in the checksum by incorporating the status register `cpsr` into the checksum.

Checksum Implementation

We implemented the entire system as a kernel module on the Sharp Zaurus PDA. The checksum loop itself was written in assembly, while the remainder of the execution agent was written in C. In an effort to strive for time optimality in the actual implementation, we unrolled the checksum loop twelve times, such that one unrolled loop modifies one checksum register.

Defense Against the Memory Copy Attack. In the memory copy attack, the attacker uses a malicious checksum function to compute the checksum over a good copy of the checksum function as described in the Pioneer work [24]. The XScale CPU permits using the the PC as a general purpose register in arithmetic and logic instructions and has PC-relative addressing mode. Therefore, the memory copy attack can be detected by reading data using PC-relative addressing and incorporating the PC in the checksum.

Defense Against Split-TLB Attack. The Split-TLB attack is a powerful attack against self-checksumming software primitives [30] that allows an attacker to setup the TLB in order to forge the checksum. By virtue of the fact that the self-checksumming code reads its own instructions as data, executing such functions imply that the I-TLB and D-TLB are pointing to the same physical page. For example, under normal circumstances, the virtual address $vaddr_1$ has an entry in the I-TLB as well as D-TLB, both of which are pointing to the same physical page $paddr_1$. Under the Split-TLB attack, the attacker sets up the TLB such that the same virtual address points to two different physical addresses. As illustrated in Figure 5, $vaddr_1$ in the D-TLB points to the correct code at physical page $paddr_1$, while $vaddr_1$ in the I-TLB points to malicious code at $paddr_2$. Executing the malicious code yields the correct checksum, since only the virtual address $vaddr_1$ is used in the checksum computation.

For ease of exposition, we present our defense against the Split-TLB attack in three phases, with increasing attack overhead.

In our first defense, we augment the self-checksumming loop with self-modifying code. The intuition behind this defense is to force the attacker into additional computation since it needs to modify two physical pages instead of one. Under normal circumstances, self-modifying code modifies one physical page, and both the I-TLB and D-TLB point to the same page. Under the Split-TLB attack, self-modifying code would only modify one physical page (e.g., $paddr_1$ from D-TLB). Therefore, the attacker needs to perform an additional write to its malicious code residing in $paddr_2$. This also requires an additional entry in the D-TLB such that a different virtual address, $vaddr_2$, points to $paddr_2$.

Self-modifying code does defend against the Split-TLB attack, since for every iteration, the attacker needs to perform an additional write. Unfortunately, the time overhead is not sufficiently high to be detectable by a human. An additional write manifests in about 2 extra cycles per iteration, or an 8% overhead. This is a far cry from the 33% overhead we achieved in the memory copy attack. Therefore, to defend against the Split-TLB attack, we need to magnify this time overhead.

In addition to self-modifying code, the second defense uses up all TLB entries, which forces the attacker's one additional TLB entry to experience a TLB miss. We alias 32 different virtual addresses $vaddr_1$, $vaddr_2$, ... $vaddr_{32}$ to all point to the same $paddr_1$ such that all 32 entries in the I-TLB and D-TLB are filled. The attacker needs its malicious 33^{rd} D-TLB entry, which causes the D-TLB to experience TLB misses.

Unfortunately, the attack overhead is not significantly higher than before. Since the attacker has 31 available D-TLB entries for 32 different virtual addresses, the expected number of TLB misses an at-

tacker faces per iteration is 1/32. The amount of time to load in a TLB entry is around 4 cycles. Thus, the expected time overhead per iteration is 2 cycles from the first defense, plus 4/32 cycles from this defense.

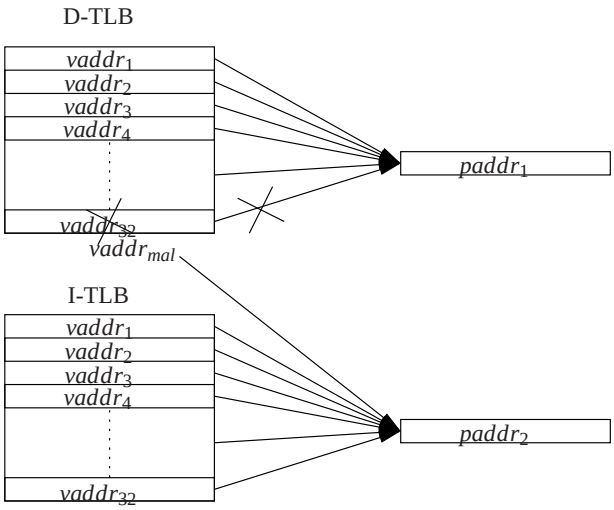


Figure 5: **Split-TLB attack and defense.** To defend against the Split-TLB attack, we fill all 32 entries in both the D-TLB and I-TLB to point to the same physical page. A Split-TLB attacker would setup the TLB in a way such that the same virtual page points to two physical pages $paddr_1$ and $paddr_2$. Because of self-modifying code, the attacker also needs to replace one TLB entry in the D-TLB with a malicious entry from $vaddr_{mal}$ to $paddr_2$. Therefore, the number of TLB misses are different between a legitimate checksum function and a malicious checksum function employing the Split-TLB attack.

The insight behind our final defense is that the number of TLB misses experienced by an attacker is different than that experienced by a legitimate checksum function. This can be exploited by including the performance counter of TLB misses into the checksum. Since the performance counter is incremented automatically by hardware upon each TLB miss, a software-based attacker cannot tamper with this counter.

According to our second defense, the legitimate checksum function experiences 0 TLB misses. An attacker would simply replace the performance counter reads with an immediate of 0. The solution is to induce TLB misses even in legitimate verification functions. Therefore, we setup 33 TLB aliases $vaddr_1$, $vaddr_2$, ... $vaddr_{33}$. The attacker still needs its additional malicious 34^{th} D-TLB entry. Thus the attacker's expected number of TLB misses per iteration is 2/34, while the expected number of TLB misses by the legitimate function is 1/33. Since TLB misses are not predictable, the attacker has no trivial way of forging the performance counter aside from simulating the entire TLB in software. In this way, the Split-TLB attack is detected.

A final note to this defense is that an attacker may avoid TLB misses completely to gain a time advantage. The attacker achieves this by creating 1 MB pages instead of the default 4K pages. Therefore, the attacker only needs 1 TLB entry to map all of $vaddr_1$, $vaddr_2$, ... $vaddr_{33}$ to $paddr_1$, in addition to one malicious TLB entry. Since the attacker only requires two TLB entries, he does not experiences any TLB misses, and gains an expected time advantage of 4/33 cycles per iteration. However, this time advantage is insignificant since the attacker still cannot predict the correct number of TLB misses experienced by a legitimate checksum function. The attacker still needs to simulate the entire TLB in software, which would require more than 4/33 cycles per iteration.

Pseudo-code and assembly code. Figure 6(a) shows the pseudo-code of our checksum code. Figure 6(b) shows one unrolled loop of the checksum function. The code shown is not the optimized version, but a verbose version to aid readability.

Each iteration of our checksum code has 24 assembly instructions and takes 28 CPU cycles. Because of the small code size and the regular instruction set architecture of the XScale processor, we were able to hand-optimize our checksum code to obtain the assurance of having the fastest possible implementation. In our future work we plan to demonstrate code optimality through the use of automated optimization tools such as Denali [15].

A.2 Empirical Analysis of Attack Overhead

In this section, we analyze different attacks against the execution agent and show why these attacks significantly slow down the checksum computation.

Benefits of XScale Architecture. First, because of the inherent properties of the XScale architecture, attacks based on the properties of the x86 platform are no longer possible. For example, multiprocessors and hyper-threading remain as possible attacks against any software-based attestation work on PCs. The absence of such capabilities on the XScale archi-

```
//Input: y number of iterations of the checksum function
//Output: Checksum C
//          daddr - address of current memory access
//          x - value of T function
//          status - status register
for l = y to 0 do
  //T function updates x where 0 ≤ x ≤ 2^n
  x ← x + (x^2 ∨ 5) mod 2^n
  //Modifies daddr based on PC and x from T function
  daddr ← PC + x[4 : 0]
  //Read from memory address daddr, modify checksum.
  //Let C be the checksum vector and j be the current index.
  C_j ← C_j + C_{j−1} ⊕ C_{j−2} + C_{j−3} ⊕ C_{j−4} + C_{j−5} ⊕ C_{j−6} +
  C_{j−7} ⊕ C_{j−8} + C_{j−9} ⊕ C_{j−10} + C_{j−11} ⊕ PC + daddr ⊕
  mem[daddr] ⊕ status + l
  C_j ← rotate_right(C_j)
  //Update checksum index
  j ← (j + 1) mod 12
end for
```

(a) Checksum Function Pseudocode

Assembly Instruction

```
smulbb r11, r10, r10
add r3, r3, r2
eor r3, r3, r1
orr r11, r11, #0x5
adds r10, r10, r11
eor r11, r9, r10, LSR #27
add r3, r3, r11
and r11, r11, #0xfffffffc
ldr r11, [r11]
add r3, r3, r0
eor r3, r3, r15
add r3, r3, r11
mrs r11, cpsr
eor r3, r3, r14
add r3, r3, r13
eor r3, r3, r9
add r3, r3, r8
eor r3, r3, r7
add r3, r3, r6
eor r3, r3, r5
add r3, r3, r4
eor r3, r3, r12
add r3, r11, r3, ROR #1
subs r12, r12, #1
```

(b) Checksum Assembly Code. r3 is the current checksum register.

Figure 6: Implementation of checksum function.

tecture rules out these attacks. Moreover, the absence of an System Management Mode (SMM) or dynamic clock scaling on the XScale processor greatly simplifies the design and results in a more robust and secure system.

Best Known Attack: Checksum Forgery. Other than hardware attacks, the best known attacks against the execution agent are checksum forgery attacks carried out in software. In a checksum forgery attack, the attacker manipulates the checksum computation to generate the correct checksum even though the attacker has modified the execution agent. We now discuss why such attacks result in a high runtime overhead.

Checksum forgery attacks can be classified into *data substitution attacks* and *memory copy attacks*. In a data substitution attack, the attacker modifies a certain memory region, and redirects memory reads for this region to another location that contains a correct copy the data. A description of memory copy attacks and a novel defense has been previously discussed. We describe the overhead of such attacks below.

Intuition behind Attack Overhead. Our checksum code occupies all 15 CPU general purpose registers (12 for the checksum, one for the loop counter, one for the data offset from the PC and one for the T-function), and 6 status registers. Hence, the attacker has no free registers to implement their attacks. The only way the attacker can obtain registers is to keep some of the checksum computation state in memory. However, accessing memory is slow. As we show, slow memory accesses are the reason why such attacks achieve a detectable time overhead.

In addition to performing memory access to free up registers, the attacker often needs to access certain immediate values to implement its attacks. The attacker has two ways to do so: 1) store and load such values from memory, or 2) encode immediate operands within the instruction. On the XScale architecture, immediate operands are encoded within each instruction, and can only be 8 bits in length because all instructions have a fixed length of 32 bits. If a program wants to operate on immediate operands that are longer than 8 bits, the program first has to move the immediate from memory into a register and then operate on the register. Thus, any attack where the attacker needs to use immediates that are longer

than 8 bits will experience a dramatic slowdown.

Data Substitution Attack. To perform a data substitution attack, the attacker must check every memory read by the checksum code to detect if the read is accessing modified memory locations. This is due to the fact that the checksum code accesses memory in a pseudo-random manner. The compare instruction modifies the state of the CPU's status register. Since we incorporate the status register into the checksum, the attacker has to save the status register before the compare instruction, and use the saved copy of the status register to compute the checksum. Furthermore, the compare instruction needs another operand—a register that contains the address of the malicious content. This requires the attacker to obtain an immediate value, which we already showed to be slow.

Memory Copy Attack. Performing a memory copy attack on the XScale is also expensive. To defend against this attack, we incorporate both the PC and the random offset from the PC used to read data into the checksum. This requires the attacker to forge either one of these values.

The attacker can forge the PC value by using an immediate or by keeping the correct PC value in memory. To forge the random offset, the attacker has to use a memory location.

Evaluation of Attack Overheads. When we implemented the different checksum forgery attacks mentioned above, we found the fastest checksum forgery attack to be the memory copy attack whereby the attacker forges the PC using a value in memory. This attack has an overhead of 33% per iteration of the checksum loop in contrast to the 2% per iteration of the checksum loop overhead in previous work. There are two reasons for the significantly higher overhead in our case. One, the number of CPU cycles per iteration for our checksum code is much smaller than previous work (28 cycles versus 52 cycles) due to the absence of pseudo-random jumps in our code. Two, the lack of x86-style segmentation support in the XScale architecture and the limit on the size of immediates that can be used in instructions limits the attacker's flexibility in constructing memory copy attacks. These two restrictions force the attacker to use memory operands for performing memory copy attacks, increasing the attacker's time overhead.

A.3 Untampered Code Execution Environment

We ensure untampered code execution by setting up an untampered execution environment. The key point is that we do not execute code unless it has been verified. The only exception is when we execute the checksum code itself. Instead of verifying before execution, the user executes the checksum code, and verifies its execution based on whether it generates the correct checksum in the expected amount of time.

Figure 7 shows how the untampered execution environment is setup in three phases. In the verification phase, we receive the challenge from the user and run the execution agent. In the setup phase, we run the code required to establish a trusted path and set up the untampered execution environment. At the end of the setup phase, the response is printed to the screen. The runtime measured by the user consists of summation of the verification and setup phases. In the execution stage, after the user verifies the checksum and runtime, she may run the verified application.

Verification Phase. During verification, we ensure that the checksum loop is the only code running by explicitly disabling all maskable interrupts and executing in *supervisor* mode. This is signified in Figure 7 by the symbol $-$. We verify this by incorporating the CPU status register (`cpsr`) into our checksum. The status register has an interrupt-enable bit. Thus, if the attacker runs our checksum code without disabling interrupts, the interrupt enable bit would be different and hence the checksum will be incorrect. Also, the `cpsr` encodes the processor mode in its least significant five bits, preventing an attacker from running our code in a different processor mode.

We also prevent malicious exception and non-maskable interrupt handlers from tampering with the execution of the execution agent by implicitly disabling them, signified in Figure 7 by the symbol $\sim$. This is done on the XScale by the `spsr` trick. The key insight is that during an exception or interrupt, the processor automatically switches processor mode based on the cause of exception, and saves the `cpsr` to the banked `spsr` of the exception mode. This is done by the hardware, and cannot be prevented by the attacker. Thus, if we store checksum pieces onto the banked `spsr` registers of all five possible exception modes (*Undef, IRQ, FIQ, Abort, and super-*

visor), an exception or unmaskable interrupt would corrupt the checksum.

	Verify	Setup		Exe	
Checksum Loop	E		V		V
UEE Setup		E	V		V
Trusted Path	–	E	V	E	V
Maskable Int	–	–		–	
Unmask. Int/Expt.	∼	∼	V	E	V
PVA			V	E	V

Figure 7: Verify, Setup, and Exe represents the three chronological time periods: verification phase, the Untampered Execution set up code, and execution of the PVA. The rows represent different pieces of code that could be executed. *E* means this code may execute at this time. *V* means this code had been verified to be correct. – means this code is explicitly disabled to run at this time (i.e., interrupts turned off). ∼ means this is implicitly disabled (i.e., it may execute, but will return incorrect response).

At the end of the 12 unrolled loops of the checksum function, we select one of these five processor modes and update their corresponding banked `spsr` based on the current checksum. Banked registers are hidden from all other modes except for the corresponding mode. Thus, we need to explicitly switch into the mode by writing to the corresponding bits in `cpsr`. Then, we modify the checksum in `spsr`, and switch back to *supervisor* mode to continue regular checksum computation.

Setup Phase. After the verification phase, we set up the trusted path and the untampered execution environment. Note that we can trust the code to set up the untampered execution environment because this code had been incorporated into the checksum in the first phase. From this point on, we would like to enable two interrupts that are needed for the trusted path: keyboard interrupt and display interrupt. We replace the interrupt handler for the keyboard and display driver with our own self-contained interrupt handler that has been previously verified. This is accomplished by freeing the corresponding IRQ handler and registering our own with the IRQ number of the keyboard and display driver. All other interrupts remain disabled by writing the corresponding bits into the Interrupt Mask Register (ICMR). In other words, we have selectively enabled only the keyboard and display interrupts as a means to establish a trusted path for user I/O.

Next, we address the unmaskable interrupts and exceptions. The `spsr` trick only temporarily protects against exceptions and unmaskable interrupts, since no protection is guaranteed once the checksum is returned to the user. Therefore, during this phase, we need to replace the unmaskable interrupt/exception handlers with a return instruction. This is accomplished as follows. In the XScale architecture, the processor services interrupts and exceptions by walking a 32 byte Interrupt Vector Table and executing the instruction present in the table entry that corresponds to the number of the interrupt of exception. This instruction is typically an unconditional branch to the corresponding interrupt or exception handler. There are two Interrupt Vector Tables (at addresses `0x00000000` and `0xFFFF0000`), and the processor decides which one to use based on a certain bit in the status register. Our checksum code modifies both Interrupt Vector Tables so that every entry is a return instruction. If any interrupt or exception occurs after this point, control returns back directly to the running code. Thus, malicious exception and interrupt handlers never get a chance to execute. In order words, exceptions and unmaskable interrupts are merely dropped. Hence, such faking on part of the attacker will either lead to an incorrect checksum value or a longer checksum computation time. After the untampered execution environment is set up, the response is displayed to the user via the trusted path previously established.

Execution Phase. In the final phase, the only code that could possibly execute are: the PVA, the keyboard/display driver, and the "return" instruction that occurs upon an exception or unmaskable interrupt. All such code had been previously verified. Thus, we achieve a environment free of unverified malicious code.