

Towards Securing Pocket Hard Drives and Portable Personalities

Nishkam Ravi*, Chandra Narayanaswami, Mandayam Raghunath** and Marcel Rosu
IBM T.J. Watson Research Center
Hawthorne, NY 10532

* Computer Science Department, Rutgers University
nravi@cs.rutgers.edu, {chandras, rosu}@us.ibm.com, **mtr@ieee.org

Abstract

Inexpensive portable storage devices that are available in the market today have made it easier for users to carry data and programs with them and borrow computing platforms when needed. While this model of computing is very attractive, it is promiscuous and thus protection is needed both for the borrower and owner of the computing platform. In this paper, we focus on a subset of this computing model, called portable storage based personalization—where the user boots the borrowed PC from her portable storage device, i.e. pocket hard drive. We analyze the security implications of this model and present a scheme to protect the pocket hard drive from the untrusted platform. The protection scheme includes running tests stored on the pocket hard drive to assess the integrity of the borrowed platform and ensuring that these tests actually get executed untampered.

1. Introduction

Affordable storage media are now available in varying form-factors and in a wide range of capacities and can be accessed almost from any PC. The most popular PC interface is USB, which offers data transfer rates as high as 480Mbps/s (ver. 2.0) and provide enough power to operate storage devices such as 2.5" hard-disk drives. The Firewire interface offers similar capabilities. Hitherto, USB drives and CD-ROMs were mostly used for the purpose of carrying limited amount of personal data. New usage models have emerged. A few vendors offer software that enables users to synchronize their email and folders with USB storage so that users can access these data items from several different computers. The U3 industry consortium promotes a model that enables users to run programs, such as a Firefox web browser, directly from the USB storage device, so that users have access to these applications even on PCs that do not have these applications pre-installed. Similarly, live CDs, which are commonly used, allow a PC to be booted from an OS resident on the CD. Figure 1 shows a 100GB Seagate Hard Drive next to an 80GB Western Digital Hard Drive and a 2GB Micro SD card (which can be plugged in via USB connector).



Figure 1. Top: 2GB Micro SD card. Bottom: 100GB Seagate Hard Drive next to an 80GB Western Digital Hard Drive

IBM Research presented a prototype called SoulPad [6], which enables users to carry their complete computing environment, including the suspended runtime state of all applications, on a USB storage media. When a computer is encountered, a user can *personalize it* by booting from the USB storage media. This system is an example of a new form of mobile computing called: *portable storage based personalization*. In this model, user's portable storage device acts as a *pocket hard drives* that carries user's operating system, applications and sessions. Any machine can be reincarnated into user's PC by plugging the pocket hard drive into it.

Though safer than the normal USB usage model (where the USB is inserted into a computer running its own OS), some security issues could still arise when a public PC is booted from the pocket hard drive. For example, a compromised BIOS could corrupt or copy the personal data of the user stored on the pocket hard drive. Or a virtual machine running on the computer could emulate bare hardware and steal the secrets necessary for decrypting personal data

stored on the pocket hard drive. In order to counter such attacks, the pocket hard drive needs to verify the integrity of the untrusted platform. Since the PC is booted from the pocket hard drive, there is no need to verify the kernel (or any other software layer above the kernel) on the untrusted machine.

One way to accomplish this is to use a hardware-based security mechanism such as Trusted Computing [3]. The main idea behind such an approach is to establish a *chain of trust* starting from a trusted hardware component all the way up to the operating system. Each layer in the software stack *attests* the layer above it based on the hash of its binary. The hashes are stored inside the trusted hardware component and can be retrieved via a trusted interface [10]. Pocket hard drives can potentially exploit this mechanism to verify integrity of the host machine. However, this would require the host machine to have hardware-support for Trusted Computing (such as TPM [3] or Copilot [11]) and an ecosystem would have to be in place for specifying, collecting and storing "good" software hashes. Also, providing attestation capability requires alteration to each layer of the software stack, including BIOS, boot-loader and the kernel. Besides, the pocket hard drive would require a CPU to establish a trusted channel with the host machine and verify the hashes. This undermines the main strength of USB-storage-based computing, which is its ability to "fit in" with the currently existing ecosystem.

The challenge is to verify the integrity of the host machine using software mechanisms without requiring any additional infrastructure. Systems security is an arms race between the attacker and the defender. The one that occupies the lower layer in the system has an upper hand, because lower layers implement the abstractions upon which upper layers are built. In our case, the attacker could occupy the lower layer (i.e., firmware), on top of which the software resident on pocket hard drive runs. We would like to raise the bar for the attacker as much as we can without increasing the cost of the portable device or requiring elaborate infrastructure support. Our goal is to install a very-low-cost but difficult-to-break lock on the door so that breaking into the house requires more time and effort. Also, pocket hard drives are a security threat to borrowed PCs. While this aspect is not the focus of this paper, we address this issue in future work.

With this in mind, we identify and classify the set of attacks that can be launched by a malicious host-machine on the pocket hard drive. We describe a set of software security mechanisms that can be used by the pocket hard drive to verify the integrity of the host machine. Finally, we present some preliminary evaluation results of an early prototype.

2. Attack Model

In our scenario, the pocket hard drive is assumed to carry a base OS which is loaded up at boot-up time and an encrypted partition containing user's data and applications, which is decrypted by the base OS (after the user enters a passphrase) to restore user's session on the borrowed com-

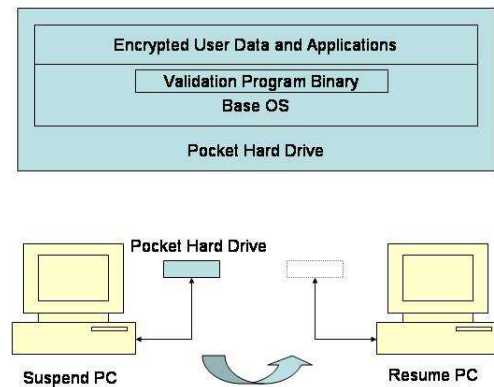


Figure 2. Pocket Hard Drive Usage Model

puter (as shown in Figure 2). The integrity of the borrowed platform has to be established before the sensitive partition is decrypted. We focus on attacks that can be launched on the pocket hard drive from compromised firmware. We ignore attacks that may result from compromised hardware such as CPU, physical memory or peripheral devices (e.g., a keystroke logger), which are harder to launch and require technical expertise on the part of the attacker, unlike firmware attacks which are created by technical experts and made available to others via internet or other public forums.

2.1. BIOS-based Attacks

John Heasman, principal security consultant for *Next Generation Security Software*, noted in January 2006 that *BIOS based rootkits* are possible to implement and will showcase the next generation of threats to systems security [2]. "A collection of functions for power management, known as the *Advanced Configuration and Power Interface (ACPI)*, has its own high-level interpreted language that could be used to code a BIOS rootkit". Here we try to capture some ways in which a malicious BIOS can breach security:

1. BIOS injects malicious code into the base OS of the pocket hard drive before it boots. The code corrupts the base OS so that after it boots up, it corrupts/deletes the sensitive partition or copies the sensitive partition to the native hard-disk drive.
2. Instead of booting the base OS on the pocket hard drive, the BIOS boots up its own OS which emulates the behavior of the base OS and bypasses any security checks that it may have before the user is prompted to enter the passphrase for decrypting the partition. (The malicious OS could be created offline by procuring a copy of the pocket hard drive base OS in advance.) Once the user enters the passphrase, it

steals the passphrase to decrypt the sensitive partition, which it may have copied earlier or may copy in future. In general, once the passphrase is stolen, security is breached, as the encrypted partition is comparatively easier to obtain and distribute.

3. BIOS inspects and analyzes the code/data of the base OS on the pocket hard drive, copies some of it, mixes it with the code of its own OS on-the-fly and boots from there. The mixed code bypasses any security checks that the base OS may have without user's knowledge and steals the passphrase after the user keys it in.
4. Before booting the base OS from the pocket hard drive, the BIOS starts a covert process capable of intercepting instructions (in much the same way as a virtual machine monitor).

Since the BIOS eventually yields control to the OS, the attacks it can launch are limited to code it executes before the OS is booted. Once the OS boots up and verifies the integrity of the BIOS, it is no longer vulnerable to the BIOS modules that may get invoked later. For launching attack 1, the adversary would need write-access to the pocket hard drive. Note that launching attack 3 requires significantly more effort than attack 2. In attack 2, all the work is done before the user walks over to the machine. In other words, the malicious OS to be booted is created off-line. We call such attacks *static attacks*. While attack 3 involves inspection of the base OS code after the user attaches the pocket hard drive. Therefore, the OS to be booted is created after the user has inserted the pocket hard drive. We call such attacks *dynamic attacks*. Similarly, attack 4 requires much more effort than attack 3 as it involves carrying out malicious operations (such as intercepting the instructions) while the OS is executing. We call such attacks *runtime attacks*.

2.2. Virtual Machine based Attacks

Virtual Machine based attacks represent another important threat to pocket hard drives. If there is a virtual machine running on the public PC, when the user tries to reboot the machine she may end up rebooting the virtual machine or not even so. The base OS would then get booted up on top of the virtual machine and thereafter the virtual machine would have complete control over the operations of the OS and the data that gets loaded in memory. We don't list the different kinds of attacks that a virtual machine can launch against the OS running on top of it, as theoretically it has complete control. See Chen et al [13] for a recent overview of this topic and for a description of virtual-machine based rootkits.

3. Solution Sketch

Our solution strives to detect the presence of a malicious BIOS or virtual machine before the OS boots up completely. This is done by invoking a validation program and ensuring that it executes untampered. Although accomplishing this in software is hard, we have made an attempt at raising the bar

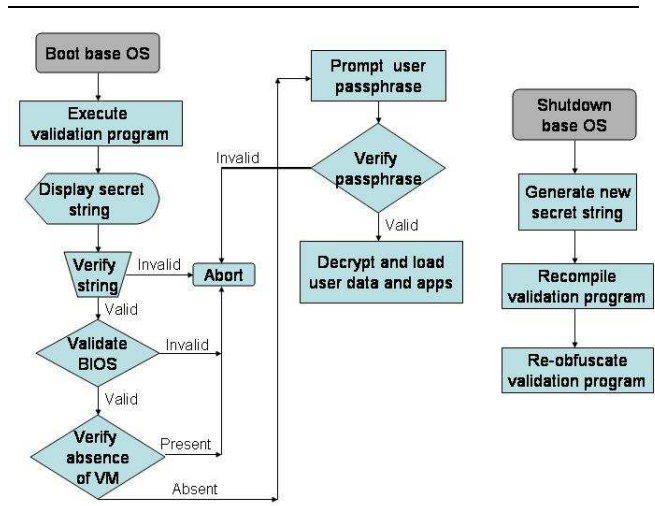


Figure 3. Overview of the validation process

for the adversary. Figure 3 summarizes the validation process. Here we outline our scheme:

- A validation program is invoked during the first stage boot of the base OS before the user is prompted for the passphrase for decrypting the sensitive partition.
- The validation program runs a battery of tests to verify the BIOS and the absence of virtual machine.
- The validation program prints a user identifiable message during execution which ensures that it has been dispatched for execution. The message is a secret shared between the user and the program and it changes each time the validation program is executed.
- Code and address obfuscation [7, 5] is employed to make it hard for the platform to retrieve the secret string or hijack the execution of the program after it has been dispatched for execution. It also makes it hard for the platform to reverse engineer the code using static analysis and modify it. Address obfuscation [5] is a program transformation technique in which program's code is modified so that each time the transformed code is executed, the absolute locations of all code and data objects (including statically linked libraries), as well as their relative distances are randomized. This makes it hard for the platform to overwrite a return address on the stack with the return address of hijack code.
- If an integrity test fails, booting is aborted, otherwise the user is prompted for the secret passphrase and the sensitive partition is decrypted.
- The validation program is recompiled and re-obfuscated after each successful execution; its source code, along with the obfuscator, is stored in the encrypted partition of the pocket hard drive. Only the ob-

ject code of the validation program is stored in the open.

Write-protection is needed to defend against data corruption attacks. For this, we resort to hardware support on the pocket hard drives. USB hard-disk drives with write-protect switch are already available in the market. Security savvy users would have to use such USB HDDs for write-protection, which should be manually disabled after the secret passphrase is entered. Our current scheme is able to defend against *static* and *dynamic* attacks, leaving the adversary with the option of formulating *runtime* attacks which are comparatively harder to launch.

4. Current Implementation

4.1. Untampered Code Execution

In order to make it hard for the adversary to tamper the execution of the validation program, we apply three levels of randomization.

A simple attack is to not execute the validation program at all and simply print the success message, tricking the user into believing that the system is secure. We address this attack by printing a user-identifiable string from the executable, while the integrity checks are carried out. The string is a shared secret between the user and the executable, which changes with every session. This is a *user-in-the-loop* approach. In our current implementation, the one-time session-dependent string is regenerated at the end of a session after the encrypted filesystem containing the guest OS has been unmounted. The user is prompted to enter a text string that will be presented during the next session. In addition to this string, the system adds information such as the date, time and the make of the PC on which the executable is built. We are able to defend against *static* attacks, using this first level of randomization.

Since the shared secret has to be stored on the pocket hard drive, it is possible for the platform to inspect the code and data of the base OS and retrieve the secret string. As mentioned before, this is a *dynamic* attack and is significantly more complex to implement and launch. In order to counter this attack, we apply code obfuscation [7]. The secret string is stored in the executable along with several similar strings and the executable is obfuscated. This makes it hard for the platform to carry out a binary analysis of the executable at runtime and retrieve the secret string. Code obfuscation also makes reverse engineering and modification of the executable hard.

It is also possible for the platform to hijack execution of the executable by analyzing the code and data segments assigned to the executable at runtime, after the shared secret has been printed. This is an example of a *runtime* attack. In order to counter this attack we apply a third level of randomization: *Address Obfuscation* [5], in which program's code is modified so that each time the transformed code is executed, the absolute locations of all code and data objects

(including statically linked libraries), as well as their relative distances are randomized.

4.2. Platform integrity checks

4.2.1. Virtual machine detection: As much as it is hard to detect the presence of a virtual machine, it is hard to achieve perfect virtualization. Garfinkel et al [9] argue that building a transparent VMM (and thus avoiding detection) is fundamentally infeasible. We exploit weaknesses of currently existing virtual machine monitors (VMMs) for detecting their presence from a program running inside them. We do not explicitly detect instruction emulators or software interpreters since the overhead imposed by these are easily detected by timing analysis and sometimes humanly noticeable.

There are certain instructions in the x86 architecture which are extremely hard to virtualize, namely SGDT, SIDT and SLDT [15]. These instructions are *sensitive, unprivileged* instructions, which means that they can be executed in non-privileged mode to get the contents of a sensitive register, without causing a trap to the VMM. All three of these instructions store a special register value into some location. For example, the SGDT instructions stores the contents of the GDTR (Global Descriptor Table Register) in a 6-byte memory location. Since the x86 architecture only has one GDTR, LDTR and IDTR, a VMM must provide each VM with its own virtual set of IDTR, LDTR and GDTR registers. The contents of these virtualized registers differ from that of the actual registers. By executing the SGDT instruction from within an assembly program and comparing the returned value with the expected value, we can detect VMMs. We were able to detect the three well-known VMMs: VMWare, VirtualPC and Qemu, using this test. Since our validation program is obfuscated, it is hard for the VMM to put software checkpoints on these instructions. However, with advances in virtualization, new tests would have to be devised.

We also included tests for detecting VMWare and VirtualPC specifically. These tests exploit the special instructions that VMWare and VirtualPC use in order to establish an interface between the virtual machine and the VMM software. It is possible for a determined attacker to build a customized virtual machine in order to pass all our tests successfully. As part of future work, we plan to add a test to our validation program that detects the presence of a virtual machine based on timing differences [8].

4.2.2. BIOS verification: Several years ago there was a case of a BIOS virus called the CIH Virus that hid on the hard-drive and overwrote the BIOS. Once the BIOS was overwritten the machine generally became non-bootable. To our knowledge there has not been a widespread outbreak of a BIOS virus. However, BIOS based rootkits are gaining popularity [2].

In order to counter BIOS-based attacks, we need to ensure that the BIOS has not been compromised. We borrow the idea of authenticating software based on the hash of its

Table 1. Increase in boot-up and shutdown time due to execution of the validation program

Machine	Avg. increase in boot-up time	Avg. increase in shutdown time
IBM ThinkPad R51	54 msec	2.2 sec
HP Compaq nx6110	51 msec	1.8 sec
Dell Optiplex GX60	52 msec	2.5 sec

binary from the Trusted Computing paradigm. In our solution, the part of the Read Only Memory (ROM) that contains the BIOS is dumped and hashed (using MD5). The hash of the BIOS is matched against the hashes of known BIOSes. The test fails in the event of no match. The BIOS hashes are stored in an encrypted file and the decryption key is hidden as a static variable in the obfuscated executable.

Collecting a set of *good* BIOS hashes is a challenge. To some extent, this challenge is common with trusted computing which assumes a database of good hashes for all software including BIOS, OS and applications. However, unlike OS and applications, BIOS updates are infrequent and, therefore, collecting and maintaining good BIOS hashes should not be as hard. At the same time, in order to have a database of good BIOS hashes, some support is needed from the BIOS manufacturers. This is because the BIOS binary also contains machine specific information, which needs to be filtered out for collecting a set of acceptable BIOS hashes. On-going initiatives to improve BIOS quality, such as EFI [1], could be leveraged to support this feature in the future.

In our current implementation, we build a database of firmware hashes on the pocket hard drive gradually over time. Subsequently when the pocket hard drive comes back to the same machine, the hash comparison approach can detect whether the firmware (i.e., BIOS) has changed in the interim. This can go a long way in restricting BIOS-tampering.

5. Preliminary Evaluation

We modified the first-stage boot sequence to invoke our validation program written in C and assembly. We modified the shut-down sequence to invoke scripts for recompilation, obfuscation and insertion of the secret string. Table 1 shows the execution time of the validation program during boot-up, and the time for recompilation and obfuscation during shutdown, for three different machines. The average increase in boot-up time is around fifty milliseconds while the average increase in shutdown time is around two seconds, which is tolerable. We were able to detect VMWare, VirtualPC as well as Qemu. We constructed a database of "good" BIOS hashes and were able to detect a BIOS whose hash was not in the database. The current evaluation does not quantify or measure the "amount of security" provided by our solution for which we need to identify the current evaluation metrics. Logging timing information from within the validation program may take us closer to this goal.

6. Related Work

The problem of authenticating a host with untrusted BIOS from a *passive* portable storage device using software-mechanisms is unexplored to the best of our knowledge. Gariss et al [10] presented a mechanism for authenticating an untrusted machine using a CPU-enabled mobile device (e.g. mobile phone) by using trusted computing. This solution, though effective, would work with only TPM-enabled machines. Trust-Sniffer [18] is a mechanism for authenticating the software stack on a machine using a passive USB device. Trust-Sniffer assumes that the BIOS is trusted and hence the attack model is orthogonal to the one considered in this paper.

The remainder of this section describes work related to the underlying mechanisms and abstractions used in our solution.

The idea of verifying BIOS by verifying the hash of its executable has been inspired by the idea of attestation used in trusted computing. The key difference is that, in trusted computing, the operation of hashing is carried out by the bootloader which is already attested, while in our case the operation of hashing and verification is carried out from an OS loaded by the BIOS which needs to be verified. As a result, software mechanisms for ensuring untampered execution of the verification code are needed.

Verifiable code execution is an open research problem. Several hardware and software based solutions have been proposed, but none of them provide provable security. Among the popular hardware-based solutions are TPM [3], Aegis [4] and Copilot [11]. A few software-based solutions for verifiable code execution also exist: Genuinity [12], SWATT [17] and Pioneer [16]. All the three approaches assume the presence of an external verifier which can execute code for verification.

In our case, if we assume that the pocket hard drive has a CPU and can thus act as an external verifier, one of these approaches (e.g. Pioneer) can be adopted to solve the problem of verifying untampered execution of the validation program. We have attempted to solve this problem without requiring extra infrastructure (i.e., no CPU on the pocket hard drive). Besides, Pioneer (and similar approaches) assume that the hardware and software configuration of the machine is known a priori, which may be hard to enforce in this case.

User-in-the-loop based authentication mechanism is used in several websites (e.g., Yahoo) where a distorted string is displayed and the user is asked to reenter it. This makes it hard to launch automated attacks.

7. Conclusions and Future Work

This paper explores security issues related with a new form of mobile computing, namely *portable storage based personalization*. We identify the set of attacks that can be launched by an untrusted platform on the pocket hard drive carried by the user. We identify the subproblems that need to be solved in order to solve the problem of platform authentication, without hardware support. We present a preliminary security mechanism that combines known software mechanisms, namely BIOS verification, virtualization tests, and randomization, to authenticate the untrusted platform before personal data is loaded. The challenge is to ensure the untampered execution of this validation program without hardware support, which is partially accomplished by: (1) using the trusted OS on the pocket hard drive to invoke the validation program, (2) increasing the complexity of tampering execution of the validation program using randomization techniques, namely code obfuscation and address obfuscation, and (3) involving the user to help in the verification of the execution of the validation program by verifying the messages displayed on the screen.

Solving this problem completely using only software mechanisms is hard. Each of the subproblems addressed in the paper is an open problem (including BIOS verification, virtual machine detection and untampered code execution). We assembled a set of preliminary techniques to address these problems. Our near future goal is to come up with better and more robust platform validation tests. In particular, tests that detect timing differences [8] in the presence of malicious code hold great promise. By making it computationally hard for the adversary to attack the validation program and detecting the difference in execution time, better security guarantees can be obtained.

A limitation of booting from a portable USB device is that local configuration information (e.g local printers and certain device drivers) may become unavailable. A possible workaround is to request the local configuration information and download device drivers from a local zone provisioning server while the PC boots up [14]. Also, pocket hard drives are a security threat to borrowed PCs. A possible solution for this problem is to modify BIOSes to include a setting that disables access to the native hard-disk drive at a hardware level when the BIOS detects that the computer is being booted from external media. Some older PCs have had physical locks and keys to prevent visitors from opening up the PC. Such mechanisms can also be conceivably extended to control access to physical components on the PC- similar to an ignition key that can be in multiple positions in a car.

References

- [1] Extensible firmware interface.
<http://en.wikipedia.org/wiki/EFI>.
- [2] Rootkits headed for bios.
<http://www.securityfocus.com/news/11372>.
- [3] Trusted computing platform alliance.
<http://www.trustedcomputing.org>.
- [4] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *Proceedings 1997 IEEE Symposium on Security and Privacy*, 1997.
- [5] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
- [6] R. Caceres, C. Carter, C. Narayanaswami, and M. T. Raghunath. Reincarnating pcs with portable soulpads. In *Mobisys '05: Proceedings of the Third International Conference on Mobile Systems, Applications and Services*, 2005.
- [7] C. S. Collberg, C. D. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *International Conference on Computer Languages*, pages 28–38, 1998.
- [8] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. V. Doorn. Towards sound detection of virtual machines. In *Springer Book on Botnet Research*, 2007.
- [9] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *Proceedings of the Eleventh Workshop on Hot Topics in Operating Systems*, 2007.
- [10] S. Garriss, R. Caceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Towards trustworthy kiosk computing. In *Proc. of 8th IEEE Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2007.
- [11] N. L. P. Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th Usenix Security Symposium*, 2004.
- [12] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *12th USENIX Security Symposium*, 2003.
- [13] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.
- [14] M. Raghunath and C. Narayanaswami. Method and system for local provisioning of device drivers for portable storage devices. In *US Patent Application 20070101118*, 2007.
- [15] J. Robin and C. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, 2000.
- [16] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.
- [17] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.
- [18] A. Surie, A. Perrig, M. Satyanarayanan, and D. Farber. Rapid trust establishment for transient use of unmanaged hardware. Technical Report CMU-CS-06-176, School of Computer Science, Carnegie Melon University, December 2006.