Problems with the Static Root of Trust for Measurement

John Butterworth *Corey Kallenberg* *Xeno Kovah*
Amy Herzog
jbutterworth@mitre.org, ckallenberg@mitre.org
xkovah@mitre.org, aherzog@mitre.org
The MITRE Corporation

Abstract

In 2011 the National Institute of Standard and Technology (NIST) released a draft of special publication 800-155. This document provides a more detailed description than the Trusted Platform Module (TPM) PC client specification for content that should be measured in the BIOS to provide an adequate Static Root of Trust for Measurement (SRTM). In this paper we look at the implementation of the SRTM from a Dell Latitude E6400 laptop. We describe how the implementation of the SRTM on this system doesn't meet all the requirements set forth by both the TPM PC client specification and the NIST guidance. We also show how a 51 byte patch to the SRTM can cause it to provide a forged measurement to the TPM indicating that the BIOS is pristine. If a TPM Quote is used to query the boot state of the system, this TPM-signed falsification will then serve as the root of misplaced trust. We also show how reflashing the BIOS may not necessarily remove this trust-subverting malware.

1 Introduction

The Trusted Computing Platform Alliance began work on the Trusted Platform Module (TPM) specification in 2000. In 2003 the Trusted Computing Group (TCG) was founded, and adopted the initial TPM 1.1 specification, before announcing the 1.2 specification in 2004[10]. Today, most enterprise-grade laptops and desktops contain a version 1.2 TPM, and the TPM 2.0 specification is under active development, with Windows 8 supporting draft compliant commands.

TPMs serve two main functions. First, a TPM can serve as a Root of Trust for Storage (RTS), by providing a Storage Root Key (SRK) used only to encrypt other keys. This allows for a key hierarchy to be built, eventually allowing user data to be encrypted by TPM-protected keys. The TPM's RTS function is not the focus of this paper.

The TPM can also provide a Root of Trust for Reporting (RTR) by providing tamper-evident measurement storage and reporting functions. The TPM has Platform Configuration Registers (PCRs), which contain SHA1 hashes that represent measurements of security-critical portions of the computer. These PCR values are only changeable with an *extend* operation. The extend operation adjusts a PCR in the following manner: $PCR_n \leftarrow SHA1(PCR_n||data)$, where $||$ is concatenation and *data* can only be 20 bytes (the size of a SHA1 hash). Because of SHA1's preimage resistance, it is infeasible for an attacker to set a PCR to an arbitrary value, such as that which represents a clean measurement. The TPM uses these values in one of a small number of ways to provide a remote appraiser with the ability to securely determine the system's configuration. The TPM provides the capability to "Seal" a key in storage such that the key cannot be used unless the PCRs are in a particular (specified) state; it can also report PCR contents signed with a special-use private key only available to the TPM via the "Quote" operation.

In this paper we will focus exclusively on a different root of trust: one that depends on the TPM, but does not actually reside within it. The Static Root of Trust for Measurement (SRTM) instead resides within the BIOS.[1] The SRTM is not used for on-demand runtime measurements, but rather to achieve a trusted boot. Per the TPM PC client spec, when the system starts the SRTM will measure itself as well as other parts of the BIOS and extend PCR0 with the resulting measurement. It is this self-measurement property that makes it the *core* root of trust. If the SRTM can be modified without the self-measurement detecting the change, the chain of trust is broken from the beginning and all other elements in the trust chain can be corrupted without detection. In this paper we demonstrate an undetected modification to the SRTM that allows us to forge PCR0 and thereby subvert

[1] The SRTM is in contrast to the Dynamic RTM (DRTM), a mechanism that can instantiate a trusted environment at some later time, even if the system booted in an untrusted state. An example implementation of a DRTM is Intel's Trusted Execution Technology[7].

the entire trust chain rooted at the SRTM.

As with many specifications, the flexibility with which the TPM PC client spec is written at times leads to ambiguity. This can lead to implementations inadvertently not measuring components that require change detection to be adequately secure. In some respect, the NIST 800-155[16] special publication can be seen as an attempt to decrease ambiguity by providing specific areas that should be measured to ensure a secure boot. However, in the case of the particular SRTM code we analyzed in this paper, we found it does not even adhere to some of the clear recommendations from the TPM PC client spec. Therefore we hope this paper will serve as a cautionary tale of why existing SRTM implementations need to be evaluated, and why future implementations need to closely follow the NIST guidance.

While the NIST 800-155 guidance is an excellent starting point, in our opinion it is insufficient because it relies too heavily on access control to keep the attacker out of the BIOS. We believe attackers will always find a way to achieve the same privileges as the defender. The history of exploits is the history of access control being bypassed, even to the point of subverting requirements for signed BIOS updates[27].

This paper makes the following contributions:

1. We evaluate the current and historical protection of the BIOS against reflashing, which is generally the protection against SRTM modification.
2. We analyze the implementation of the existing Latitude E6400 SRTM, how its sets PCR values, and how it deviates from the TPM PC client spec.
3. We describe the implementation of a *tick*, a PCR-forging BIOS parasite.
4. We describe the implementation of a *flea*, a PCR-forging, reflash-hopping, BIOS parasite that shows why enforcing signed updates is insufficient to protect currently deployed systems.

This paper is organized as follows. In the next section we discuss related work in the area of BIOS security and trusted computing. In Section 3 we describe how we have extracted information about SRTM implementations, and in Section 4 we analyze how the Latitude E6400's SRTM implementation was found lacking. In Section **??** we discuss implementation details of our timing-based attestation system, and in Section **??** we evaluate it against various attacks. We detail our conclusions in Section 5.

2 Related Work

There have been a number of papers and proof of concept attacks that took advantage of the lack of access control on the BIOS reflashing procedure to introduce malicious code into the BIOS. One of the first attacks claiming to be a "BIOS Rootkit" was described by Heasman[11]. This attack did not target the BIOS code itself, but rather modified the ACPI tables set up by the BIOS. Subsequent ACPI table interpretation caused beneficial effects for the attacker, like arbitrary kernel memory writes. Later attacks by both Core Security[18], and Brossard[5] relied on the open source CoreBoot[1] project. This project was meant to serve as an open source BIOS alternative, although at the time of writing the newest Intel chipset (ICH7) supported by CoreBoot is approximately 6 years old. For in-the-wild attacks, there is the famous example of the CIH or Chernobyl virus[28] that would render a machine unbootable by writing zeros to the BIOS flash chip. In a much more recent attack, the Mebromi malware[9] rewrote the BIOS of a machine with code that would then write a typical Master Boot Record infection routine to the first sector of the disk. This allowed the malware to persist even if the hard drive was replaced or reformatted. Both of these attacks were limited in their spread because they supported only one chipset configuration.

All of the preceding attacks on the BIOS relied on the BIOS being unprotected and easily writeable, and only having security through the obscurity of the knowledge needed to reflash a BIOS. The most noteworthy exception to this assumption of an unprotected BIOS is the attack by Invisible Things Lab (ITL) which reflashed an Intel BIOS despite a configuration requiring all updates be signed[27]. They achieved this by exploiting a buffer overflow in the processing of the BIOS splash screen image. Given the prevalence of legacy, presumably unaudited, code in BIOSes, we expect there are many other similar vulnerabilities lurking. This is a key reason why we advocate for designing under the assumption that access control mechanisms protecting the SRTM will fail.

In 2007 Kauer[12] reported that there were no mechanisms preventing the direct reflashing of the BIOS of a HP nx6325, and he specifically targeted manipulation of the SRTM. He decided to simply replace the SRTM with an AMD-V-based DRTM to "remove the BIOS, Option-ROMs and Bootloaders from the trust chain."[2] While the intent of a DRTM is to not depend on the SRTM, as was acknowledged by Kauer, the DRTM can in fact depend on the SRTM for its security. ITL has shown this through multiple attacks. In [25] they described an attack where manipulation of the ACPI tables generated by the BIOS and parsed by the DRTM could lead to arbitrary code execution within the context of the DRTM;

[2] We believe that the security community should either attempt to create a truly secure SRTM, as we are trying to do in this paper, or should push for its removal everywhere so that no one falsely believes it to be providing trust it cannot actually provide.

the SRTM was part of the root of trust for the DRTM. In [26], they showed how an attacker with SMM access could execute in the context of a TXT DRTM thanks to the lack of a System Management Mode (SMM) Transfer Monitor (STM). Given the BIOS's control over the code in SMM, and the longstanding lack of a published Intel STM specification, it is expected that most systems attempting to use TXT will be vulnerable to attacks originating from SMM for quite some time.

Because the BIOS sets the SMM code, it is worth pointing out that the lack of a trustworthy SRTM undermines security systems relying solely on SMM's access control to achieve their security, such as HyperGuard[17], HyperCheck[24], HyperSentry[4],and SICE[3]. If such systems were using timing-based attestation to detect changes to their SMM code, they would be much harder to subvert even by a malicious BIOS flash. Similarly, a subverted SRTM undercuts load-time attestation systems such as IMA[19] and DynIMA[8]. It also subverts systems like BitLocker[2] that rely on sealing a key against PCRs that are *expected to change* in the presence of an attacker, but that don't here.

3 Journey to the Core Root of Trust for Measurement

To analyze a system SRTM, a BIOS firmware image from that system must be obtained to identify both where and how the SRTM is instantiated. There are three primary ways to obtain a BIOS image for analysis. One is to desolder the flash chip from the mainboard and dump the image to a binary file using an EEPROM flash device. The EEPROM device is invaluable when having to recover a "bricked" system resulting from an experiment to modify a BIOS gone awry. The second way to get the BIOS is to use a custom kernel driver that reads the firmware image from the flash chip and writes it to a binary file. The third is to extract and decode the information from vendor-provided BIOS update files. In all cases, the binary in the obtained file can be statically analyzed using software such as IDA Pro. However in situations where you want to investigate "live" BIOS/SMM code, e.g. a routine that reads an unknown value from an unknown peripheral, a hardware debugger such as the Arium ECM-XDP3 is very useful.

NIST 800-155 uses the term "golden measurement", to refer to a PCR value provided by a trusted source (such as the OEM) indicating the value that should exist on an un-tampered system. However, currently no SRTM golden measurements are provided by OEMs. This leads to a situation where organizations must simply measure a presumed-clean system, and treat the values as golden measurements. The intention is that an organization should investigate any PCR change that does not result in an expected golden measurement value. Table 1 displays the "presumed-good" PCR hashes for our E6400.

We discovered that the SRTM measurement in PCR0 in Table 1 is derived from a hash provided to the TPM from a function which is executed during the early BIOS POST process. The function is called from within a table of function pointers. Each pointer is part of a structure which includes a 4-byte ASCII name. The name of the function that initially serves to instantiate PCR0 is "TCGm", presumably for "Trusted Computing Group measure".

This function uses a software SHA1 computation (as opposed to the TPM's built in SHA1 function) to hash slices of the BIOS and then presents that hash to the TPM for extension to PCR0. A hash is constructed from the first 64 bytes of each compressed module contained within the BIOS ROM (there are 42 of these modules in total); two small slices of memory; and the final byte of the BIOS firmware image. Within the first 64 bytes is a data structure containing the size, and therefore the SRTM developers most likely are assuming that measurement of the first 64 bytes will be sufficient to detect any changes within the compressed module. After all of these locations have been hashed and combined, the final hash is extended into PCR0 of the TPM. But we also found that a second extend is done on PCR0 with the single last byte of the BIOS, similar to what was done with the other PCRs as described in section 4.2.

4 SRTM Implementation Weaknesses

The Dell Latitude E6400 was released in 2008 before NIST 800-147 and 800-155 publication, but it is upgradeable to NIST 800-147 protections via a BIOS update. We picked the E6400 for analysis in 2010 because it was readily available and it fit our debugging hardware. The E6400 BIOS is based on Dell's legacy BIOS core, which is representative of legacy BIOS that do not have some of the protections called for by recent standards. This makes these BIOS versions more accessible targets for evaluation and modification than recent systems with a modern UEFI-based BIOS for example. It is assumed that more recent systems may also include bug fixes, updates, and protections that to some degree harden the platform against similar modifications. However, at this time legacy systems rather than UEFI make up the majority of deployed PCs.

4.1 Overwritability

As pointed out by [12], being able to freely modify the SRTM completely undercuts its function as the root of trust for measurement. Indeed, the TPM PC client spec

Table 1: Dell Latitude E6400 presumed-good PCR's (BIOS revision A29)

hexadecimal value	index	TCG-provided description
5e078afa88ab65d0194d429c43e0761d93ad2f97	0	S-CRTM, BIOS, Host Platform Extensions, and Embedded Option ROMs
a89fb8f88caa9590e6129b633b144a68514490d5	1	Host Platform Configuration
a89fb8f88caa9590e6129b633b144a68514490d5	2	Option ROM Code
a89fb8f88caa9590e6129b633b144a68514490d5	3	Option ROM Configuration and Data
5df3d741116ba76217926bfabebbd4eb6de9fecb	4	IPL Code (usually the MBR) and Boot Attempts
2ad94cd3935698d6572ba4715e946d6dfecb2d55	5	IPL Code Configuration and Data

[10] says: "The Core Root of Trust for Measurement (CRTM) MUST be an **immutable** portion of the Host Platform's initialization code that executes upon a Host Platform Reset." (Emphasis ours.) Unfortunately this immutability is not per the dictionary definition. Instead, "In this specification, immutable means that to maintain trust in the Host Platform, the replacement or modification of code or data MUST be performed by a Host Platform manufacturer-approved agent and method." There are therefore a number of reasons why the CRTM may in practice be quite mutable. In our experience, the CRTM is the same as the SRTM.

Unlike NIST 800-155, NIST 800-147[6] lays out guidelines on how the BIOS of systems should be configured by end users to minimize the exposure to malicious changes. The most important changes are setting a BIOS password, and turning on the capability to require all BIOS updates be signed. This signed update process would thereby provide the immutability specified by the TPM PC client spec. Like many other legacy systems, ours shipped without signed updates being required, leaving the SRTM vulnerable. But beyond this, we found that the revision A29 BIOS original on our system was not unsigned, it did not even have an option to turn on signed updates! Only beginning in revision A30 was the BIOS signed, and a configuration option requiring signed updates available. But even when signed updates are available and enabled, implementation weaknesses (that existed in the A29 but which were fixed in the A30) could allow an attacker to bypass signed updates.

For instance, on systems with this chipset, the BIOS flash chip can be directly overwritten by a kernel module unless provisions are implemented by the BIOS manufacturer to prevent this from occurring. The mechanism to prevent direct overwrite has two components: proper configuration of the BIOS_CNTL register's BIOSWE and BLE bits, and a routine in SMM to properly field the System Management Interrupts (SMI) that subsequently occur.

When properly configured, the BIOS_CNTL register causes an SMI to be triggered whenever an application attempts to enable write-permission to the BIOS flash. This provides SMM the opportunity to determine whether this is a sanctioned write to the flash chip or not and, in the latter case, reconfigure the BIOS_CNTL register to permit read-only access to the BIOS flash. All this occurs prior to the application having any opportunity to perform any writes to the flash chip.

By default on the E6400, firmware updates are validated only by a simple 32-bit Cyclic Redundancy Check (CRC) checksum. Legitimate updates are of the "capsule" type described by NIST[16]. "The OS uses SMM to modify the BIOS, then continue without rebooting the system." This method allows the user to update their BIOS without disrupting the current operation of the system. However the BIOS update will not be applied in the EEPROM by SMM code until the next reboot.

4.2 Inaccuracy

PCR0 is the primary PCR value that we are concerned with as it captures the measurement of the SRTM. However, it is worth noting the obvious duplication of values among PCRs 1, 2, 3 in Table 1. Projects attempting to implement TPM-supported trusted boot capabilities are often puzzled by what is actually being measured by the SRTM to set those values. We determined the origin of such PCRs as we had previously noted similar duplicate PCR values among many of our enterprise systems. After observing the BIOS's interaction with the TPM it was determined that the oft seen duplicate value in our PCR values was simply an extend of the single last byte of the BIOS! Specifically, $PCR_{1,2,3} \leftarrow SHA1(0x00_{20}||SHA1(0x00))$; a fact that is trivially independently verifiable. This complete failure to measure the important parts of the system associated with these PCR values contravenes the TPM PC client spec.

As shown in Figure 1, the OEM SRTM excludes the overwhelming majority of the BIOS memory from measurement. To generate PCR0, it only measures the dark gray portion of the BIOS, which amounts to only 0xA90 out of 0x1A_0000 bytes in the BIOS range! The bulk of the measurements consist of measuring the first 64 bytes of each of the 42 compressed modules. (The mod-

ules range in size from 0x9E to 0x29583 bytes.) There are also two slices of 8 bytes, at 0xDF45_13C0 and 0xDF45_13C7 that we have been unable to determine the purpose of measuring.

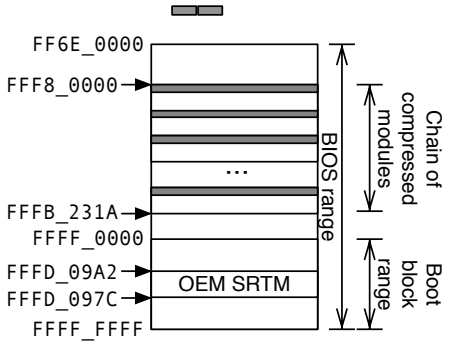

Figure 1: Some components found in the BIOS range (not to scale). Dark grey is memory measured by the SRTM, white is unmeasured.

The intent of the SRTM is to provide trust that no critical BIOS contents have been modified. In short, this implementation can not achieve that goal. This is an important discovery, and we are not aware of any related work validating the functioning of an SRTM, rather than blindly trusting it. We have conducted cursory examinations of other SRTMs and observed similar problems with incomplete coverage. This suggests the need for more validation going forward to ensure SRTMs are properly implementing NIST 800-155 guidance going forward.

4.3 Proof of Concept Attacks

4.3.1 Naive

We describe a naive attack as one that reflashes the BIOS but which can be trivially detected by PCR0 changing. We would call this naive even in the presence of an SRTM which had more complete coverage. In this paper we are primarily concerned with advanced attackers who are seeking to bypass existing trusted computing technologies that are assumed to be provisioned correctly for use in their respective organization.

4.3.2 The Tick

We define a *tick* to be a piece of parasitic stealth malware that attaches itself to the BIOS to persist, while hiding its presence by forging the PCR0 hash. A tick has to exist in the same space as the SRTM. Regardless of whether the entirety of the BIOS is hashed to generate PCR0, a tick can perform the same process on a clean copy of data, or simply replay expected SHA1 hash values to the TPM for PCR0 extension. On the E6400 this later strategy is easily performed at the end of the "TCGm" function just before the hash is passed to the TPM. For example, to forge the known-good PCR0 hash shown in Table 1 for BIOS revision A29 running on a Dell E6400, a hardcoded hash value of "F1 A6 22 BB 99 BC 13 C2 35 DF FA 5A 15 72 04 30 BE 58 39 21" is passed to the TPM's PCRExtend function.[3] So even though the BIOS has been tampered with in a way that would normally change PCR0, the change goes undetected since the dynamic calculation of the hash to extend PCR0 with has been substituted with a hardcoded constant of the known-good hash. It is worth pointing out that the BIOS modifications made by Kauer in [12] did not constitute a tick, because there was no forgery of PCR0, only disabling TPM commands. Our tick implementation is only a 51 byte patch to the BIOS. After a tick is attached to the BIOS, it can make other changes go undetected by traditional trusted boot systems.

4.3.3 The Flea

We define a *flea* as parasitic stealth malware that, like a tick, forges PCR0 but is additionally capable of transferring itself ("hopping") into a new BIOS image when an update is being performed. A flea is able to persist where a tick would be wiped out, by controlling the BIOS update process. On the E6400 it does this with a hook in the SMRAM runtime executable. A BIOS update is written to memory, a soft reboot occurs, and then SMRAM code writes the image to the EEPROM. The flea detects when an update is about to take place and scans the installation candidate to identify which revision of BIOS is being installed. Once the flea has identified the revision, it patches the installation candidate in memory to maintain and hide its presence, and then permits the update to continue. At a minimum the flea must modify the update candidate with the following patches: a patch to enable the new image to forge PCR0; the compressed module that defines SMRAM containing the flea SMM runtime portion that controls the update process; and the necessary hooks required to force control flow to the flea's execution.

Our flea residing on an E6400 with BIOS Revision A29, forges the known-good PCR0 value as described in the previous section. A BIOS update is about to occur which the flea has determined will be to BIOS revision A30. The flea retrieves and applies the A30 patches, among which will be one that provides the necessary constant so that PCR0 will provide the known-good value for BIOS revision A30.

One challenge for the flea is that it must find storage

[3]For independent verification purposes, Table 1's $PCR_0 \leftarrow SHA1(SHA1(0x00_{20}||SHA1(0xF1A6...21))||SHA1(0x00))$

for its patches. We ultimately chose to use unused portions of the flash chip. In our current implementation these patches can consume upwards of 153KB per revision and there can be many BIOS revisions to support across the lifetime of a system. However our current implementation inefficiently stores the data uncompressed, because we did not have time to utilize a compression method that could use the BIOS's built in decompression routine. Our flea code implementation absent the patches is only 514 bytes. An open question is what a flea should do if it is not able to identify the incoming BIOS revision. We preface this discussion by asserting that in practice we believe this will be an uncommon situation. Any attacker that cares to persist in a system's BIOS will likely have the resources to analyze BIOS updates and deploy updates for the flea's patch database well before a compromised organization can deploy BIOS updates. However, as a stalling strategy, our flea will begin to act as if it is updating the BIOS, but then display an error and not make any changes if it cannot identify the pending update.

The key takeaway about our creation of a flea is that it mean to underscore the point that simply following the NIST 800-147 guidance to lock down a system and enable signed updates on existing deployed systems *is not enough to protect them.* Once a system is compromised, the presence of a flea means it will stay compromised. This is why we advocate for confronting the problem head-on with BIOS Chronomancy.

5 Conclusion

As the *core* root of trust for measurement, proper implementation of the SRTM is critical. However, this work is the first (public) examination of a real implementation of the SRTM and finds that implementation to be untrustworthy. So long as the core root of trust for measurement is mutable (not implemented in a physical ROM chip), there will always be opportunities for attackers to subvert its measurements.

We believe that both NIST 800-147 and 800-155 are important guidelines which should be followed by their respective audiences. To demonstrate the danger of putting false trust in opaque SRTM implementations, we implemented a proof of concept attack that we called a *tick*, that modified the BIOS on our Dell Latitude E6400 while not causing a change to any of the PCRs. A common remediation that might be employed to try to remove a tick is to reflash the BIOS with a known-clean version. However this is not sufficient to protect against existing BIOS malware maintaining a foothold on the system. To show this we implemented a *flea*, which can jump into and compromise the new BIOS image during the update process.

References

[1] Coreboot. `http://www.coreboot.org/`. Accessed: 11/01/2012.

[2] Microsoft bitlocker drive encryption. `http://windows.microsoft.com/en-US/windows-vista/BitLocker-Drive-Encryption-Overview`. Accessed: 2/01/2013.

[3] A. Azab, P. Ning, and X. Zhang. SICE: A Hardware-Level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *Proceedings of 2011 ACM Conference on Computer and Communications Security*.

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

[5] J. Brossard. Hardware backdooring is practical. In *BlackHat*, Las Vegas, USA, 2012.

[6] D. Cooper, W. Polk, A. Regenscheid, and M. Souppaya. BIOS Protection Guidelines). NIST Special Publication 800-147, Apr. 2011.

[7] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer Manual, Vol. 3b, Part 2. `http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software_developer-vol-3b-part-2-manual.pdf`. Accessed: 11/01/2012.

[8] L. Davi, A.-R. Sadeghi, and M Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*.

[9] M. Giuliani. Mebromi: the first BIOS rootkit in the wild. `http://blog.webroot.com/2011/09/13/mebromi-the-first-bios_rootkit-in-the-wild/`. Accessed: 11/01/2012.

[10] Trusted Computing Group. TPM PC Client Specific Implementation Specification for Conventional BIOS. Version 1.21 Errata version 1.0, Feb. 24 2012.

[11] J. Heasman. Implementing and detecting a ACPI BIOS rootkit. In *BlackHat Europe*, Amsterdam, Netherlands, 2006.

[12] B. Kauer. OSLO: improving the security of trusted computing. In *Proceedings of 2007 USENIX Security Symposium on USENIX Security Symposium*.

[13] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*.

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

[15] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *Proceedings of the 2011 ACM Conference on Computer and Communications Security (CCS)*.

[16] A. Regenscheid and K. Scarfone. BIOS Integrity Measurement Guidelines (Draft). NIST Special Publication 800-155 (Draft), Dec. 2011.

[17] J. Rutkowska and R. Wojtczuk. Preventing and detecting Xen hypervisor subversions. In *BlackHat*, Las Vegas, USA, 2008. Accessed: 11/01/2012.

[18] A. Sacco and A. Ortega. Persistent BIOS infection. In *CanSecWest*, Vancouver, Canada, 2009.

[19] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 2004 conference on USENIX Security Symposium - Volume 13*.

[20] D. Schellekens, B. Wyseur, and B. Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Electron. Notes Theor. Comput. Sci.*, 197:59–72, February 2008.

[21] A. Seshadri. *A Software Primitive for Externally-verifiable Untampered Execution and its Applications to Securing Computing Systems*. PhD thesis, Carnegie Mellon University, 2009.

[22] 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 *Proceedings of the ACM symposium on Operating systems principles*, SOSP, pages 1–16, 2005.

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

[24] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: a hardware-assisted integrity monitor. In *Proceedings of the 2010 international conference on Recent advances in intrusion detection*.

[25] R. Wojtczuk and J. Rutkowska. Attacking Intel TXT via SINIT code execution hijacking. `http://invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf`. Accessed: 11/01/2012.

[26] R. Wojtczuk and J. Rutkowska. Attacking Intel TXT. In *BlackHat Federal*, Washington D.C., USA, 2009.

[27] R. Wojtczuk and A. Tereshkin. Attacking Intel BIOS. In *BlackHat*, Las Vegas, USA, 2009.

[28] M. Yamamura. W95.CIH - Symantec. `http://www.symantec.com/security_response/writeup.jsp?docid=2000-122010-2655-99`. Accessed: 11/01/2012.