BIOS Chronomancy: Fixing the Core Root of Trust for Measurement

John Butterworth
jbutterworth@mitre.org

Corey Kallenberg
ckallenberg@mitre.org

Xeno Kovah
xkovah@mitre.org

Amy Herzog
aherzog@mitre.org
The MITRE Corporation

ABSTRACT

In this paper we look at the implementation of the Core Root of Trust for Measurement (CRTM) from a Dell Latitude E6400 laptop. We describe how the implementation of the CRTM on this system doesn't meet the requirements set forth by either the Trusted Platform Module (TPM) PC client specification[12] or NIST 800-155[20] guidance. We show how novel *tick* malware, a 51 byte patch to the CRTM, can replay a forged measurement to the TPM, falsely indicating that the BIOS is pristine. This attack is broadly applicable, because all CRTMs we have seen to date are rooted in mutable firmware. We also show how *flea* malware can survive attempts to reflash infected firmware with a clean image. To fix the untrustworthy CRTM we ported an open source "TPM-timing-based attestation" implementation[17] from running in the Windows kernel, to running in an OEM's BIOS and SMRAM. This created a new, stronger CRTM that detects *tick*, *flea*, and other malware embedded in the BIOS. We call our system "BIOS Chronomancy", and we show that it works in a real vendor BIOS, with all the associated complexity, rather than in a simplified research environment.

Categories and Subject Descriptors

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

Keywords

Trusted Computing, Firmware, BIOS, TPM, Malware, Timing-based Attestation

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[12]. 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.

The TPM is a passive chip that relies on code running on the main CPU to direct its actions. In this paper we do not focus on the roots of trust for storage and reporting that reside physically within the TPM. Instead we examine the Static Root of Trust for Measurement (SRTM) that is rooted 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 boots the SRTM will measure itself as well as other parts of the BIOS, the master boot record, etc. and store the measurements in the TPM. The component that specifically performs self-measurement is considered the *Core* Root of Trust for Measurement (CRTM). If the CRTM can be modified without the self-measurement detecting the change, the chain of trust is fundamentally broken, and all subsequent elements in the chain can be corrupted without detection. In order to be reported in a trustworthy way, the SRTM stores measurements into the TPM Platform Configuration Registers (PCRs). Measurement appraisers that want to evaluate the boot measurements can ask the TPM for a copy of the PCRs signed by a key only the TPM has access to.

The flexibility with which the TPM PC client spec is written at times leads to ambiguity. This has lead to implementations inadvertently not measuring components that require change detection to be adequately secure. In some respect, the NIST 800-155[20] special publication (which is entirely advisory, and not associated with the official TCG specs) can be seen as an attempt to decrease ambiguity of the TCG specs 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 found in every revision of the TPM PC client spec. Therefore we hope this paper will serve as a cautionary tale of why today's SRTM implementations need to not

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516714.

[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[8].

be blindly trusted, and why future implementations should closely follow the more detailed 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[31][14][15].

We therefore believe it is necessary to apply techniques explicitly designed to combat attackers at the same privilege level[17, 18, 24, 25, 26, 27]. We call our implementation BIOS Chronomancy because our additional trust is *divined from timing*. The application of this technique to the BIOS provides many advantages over higher level implementations, because the environment in which both the attacker and defender are working is more constrained. This dramatically improves the likelihood of the CRTM detecting an unauthorized modification of itself. In this paper we will show a proof-of-concept custom BIOS that detects our CRTM-targetting malware, as well as other stronger attacks. We have run this custom BIOS on 17 Dell Latitude E6400 laptops for months without issue.

This paper makes the following contributions:

1. We analyze the implementation of the existing Latitude E6400 SRTM, how it measures itself, and how it deviates from the TPM PC client spec.
2. We describe the implementation of a *tick*, a CRTM-subverting BIOS parasite.
3. We describe the implementation of a *flea*, a CRTM-subverting, reflash-hopping, BIOS parasite that shows why enforcing signed updates is insufficient to protect currently deployed systems.
4. We discuss our design for the application of TPM timing-based attestation at the BIOS level. We also evaluate this implementation's performance against a worst case attacker, who is adding only the minimal timing overhead necessary to hide from detection, but not achieve any other goal.
5. We found and fixed a problem with the existing TPM timing-based timing attestation systems[24][17] that would allow an attacker to always forge his runtime to be within acceptable limits.

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 the SRTM implementation, and in Section 4 we analyze how the Latitude E6400's SRTM implementation was found lacking. In Section 5 we discuss implementation details of our timing-based attestation system, and in Section 6 we evaluate it against various attacks. We detail our conclusions in Section 7.

2. RELATED WORK

There have been a number of papers and proof of concept attacks that took advantage of deficient 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[13]. 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[22], 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[32] that would render a machine unbootable by writing zeros to the BIOS flash chip. In a much more recent attack, the Mebromi malware[11] 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[31]. They achieved this by exploiting a buffer overflow in the processing of the BIOS splash screen image. Given the prevalence of legacy, presumably un-audited, 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[16] 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, OptionROMs 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 [29] ITL 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; the SRTM was part of the root of trust for the DRTM. In [30] ITL 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[21], HyperCheck

[2] We believe that the security community should either attempt to create a truly secure CRTM, 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.

[28], 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[23] and DynIMA[9]. 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[6].

While the timing-based attestation presented in this paper is adapted from the open source reference implementation provided by Checkmate[17], which is derived from Pioneer[26], in spirit this work is much more closely tied to earlier application of software-based attestation to embedded systems such as SWATT[27] or SBAP[18]. This is because the environment in which a BIOS executes is more constrained, allowing for better coverage. While PioneerNG[25] also ran in SMM, it did not work in conjunction with a real world BIOS, but instead was implemented within CoreBoot. Further, it derived timing-based trust from attestation to a USB device. As our work uses the TPM for trusted timing, it serves as a more stand-alone trusted computing system, allowing for easier adoption.

Our system is therefore the first which has been shown to work within a real environment; which takes into account which portions of the SRTM should remain with Original Equipment Manufacturer (OEM) code; and which shows how existing SRTM code's self-protection is lacking and in need of timing-based attestation. Our system does currently lack a countermeasure for malicious code running on peripheral processors parallel to the BIOS execution. We believe that the use of a system like VIPER[19] to verify peripherals' firmware would mesh well with ours in the future.

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. The intention is that an organization should investigate any PCR change that does not result in an expected golden measurement value. 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. 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

While the weaknesses described below are in terms of the Dell Latitude E6400, it is believed that due to known BIOS code reuse[14], that the same problems occur in 22 Dell models. The E6400 was released prior to NIST 800-147 and 800-155 publication, and was only designed with the TPM PC client specification as a a guide. We picked the E6400 for analysis in 2010 only because it was readily available and it fit our Arium debugging hardware. This system is still supported as the latest BIOS update was released in July 2013. We acknowledge that there have been subsequent changes in BIOS implementation by vendors, most notably adoption of the Unified Extensible Firmware Interface (UEFI). However, we have analyzed newer machines in less depth, and universally the SRTM and CRTM are still implemented in mutable firmware. The problems of open BIOSes and inadequate SRTM coverage we discuss in this paper are not automatically solved in newer UEFI systems, as has also been shown by Bulygin[6].

4.1 Overwritability

As pointed out by [16], being able to freely modify the SRTM completely undercuts its function as the root of trust for measurement. Indeed, the TPM PC client spec [12] 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.

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

Unlike NIST 800-155, NIST 800-147[7]provides BIOS configuration guidelines for end users to minimize exposure to malicious changes; most importantly setting a BIOS password and requiring signed BIOS updates.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 only 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 signed updates are not a panacea. Methods to bypass signed updates have been shown by ITL[31], Bulygin[6], and ourselves[14][15]. Although [6] did not give specifics of the misconfigurations checked for, we can infer that it was similar to the below.

On systems with Intel IO Controller Hub 9, like the E6400, 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. This security mechanism was missing from the E6400 in revision A29, but fixed in A30. Misconfigurations like this that can generically undercut signed BIOS updates could be common across all vendors, but no one has the data on this yet. We are currently working to collect this data across large deployed populations.

4.2 Inaccuracy

PCR0 is the primary PCR value that we are concerned with as it captures the measurement of the CRTM. However, it is worth noting the 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![3] 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 (.2%) in the BIOS range! The bulk of the measurements consist of measuring the first 64 bytes of each of the 42 compressed modules. 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.

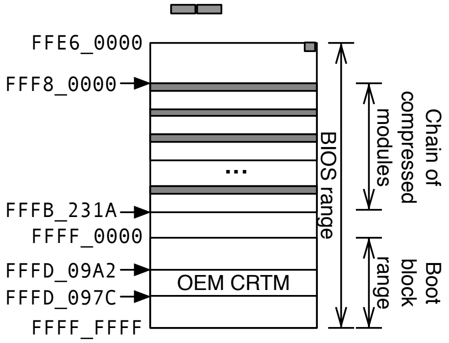

Figure 1: Some components found in the BIOS range (not to scale). Dark grey is memory measured by the SRTM, white is unmeasured.

The intent of the SRTM is to provide trust that no critical BIOS contents have been modified; something this implementation cannot do. 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.

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

[3] Specifically, $PCR_{1,2,3} \leftarrow SHA1(0x00_{20}||SHA1(0x00))$; a fact that is trivially independently verifiable.

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.

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.[4] 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 [16] did not constitute a tick, because there was no forgery of PCR0, only disabling TPM commands (which leads to trivially detection). 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. By controlling the BIOS update process, a flea is able to persist where a tick would be exterminated. On the E6400 it achieves this by hooking the SMRAM runtime executable code that performs the BIOS update process. The flea can then detect when an update is occurring, identify what revision of BIOS is being flashed, and, based on that revision, appropriately patch itself into the prospective update image. 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 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. BUILDING A BETTER CRTM WITH BIOS CHRONOMANCY

We want to make it clear that we are not completely replacing or re-implementing all the functionality of existing SRTM code. Given our proof of concept attacks' ability to subvert CRTM self-measurement, we outline a mechanism that can be added to commercial CRTMs to provide trustworthy evidence that the CRTM specifically has not been tampered with.

5.1 Timing-based Attestation

As described in Section 2, there has been much work in the area of timing-based attestation. In [17] the phrase "timing-based attestation" was coined to be a superset of software-based attestation, including techniques that require dedicated hardware such as the TPM to perform trustworthy timing measurement. In all such systems, the general principle is to use a specialized software construction that has an explicit timing side-channel built into it. The side-channel is meant to allow the software to have a consistent runtime in the absence of an attacker, and an increased runtime in the presence of an attacker who is manipulating its operation. The crux of such software construction is to create a looping system that checks itself, and any other security/measurement code the defender wishes to protect. If the attacker would like to manipulate the behavior of the security code, he must do so in a way that still generates a correct self-checksum. Doing this requires modifying the looping code that generates the checksum. Each additional instruction of logic added to the loop code to help the attacker subvert security code will increase the runtime of a single loop. This time increase is multiplied by the number of loops, leading to a detectable timing change when compared to the expected baseline runtime. As described in previous work, baseline runtimes will be specific to a CPU and its frequency.

We have adapted the open source TPM tickstamp-based attestation code from [17] to run within a customized E6400

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

BIOS and serve as an improved CRTM. Unlike other CRTMs, ours is actually capable of detecting an attacker at the same privilege level who is explicitly attempting to subvert our measurement. Our timing is specifically derived from the timing delta between two TPM "tickstamp" requests. A tickstamp can be thought of as just a signed value of the TPM microprocessor's clock tick count. The TPM 1.2 specification provides a TPM_TickStampBlob command which takes a nonce, data blob, and a TPM signing key as arguments. The TPM then returns a structure containing a signature, the current TPM ticks value, and the current Tick Session Nonce (TSN), denoted as $(signature, ticks, TSN) \leftarrow TickStamp_{key}(data, nonce)$. The $signature$ is given by $Sign_{key}(data, nonce, ticks, TSN)$. On TPM reset (typically a reboot), a new 20 byte TSN is generated by the TPM hardware random number generator and the TPM tick counter is reset to zero, beginning a new timing session. The TPM 2.0 specification has moved the tickstamp functionality to the new TPM2_GetTime command and we anticipate our attestation model will work with 2.0 compliant TPMs.

Our self-check protocol was inspired by the Schellekens et al. protocol and begins in the first stage of system boot. BIOS Chronomancy operates as follows:

1. The current TSN is read using the TPM_GetTicks operation. $TSN_{curr} \leftarrow GetTicks()$
2. The tickstamp 1 (TS1) structure is generated by requesting a tickstamp with the TSN_{curr} from above used as both the data and the nonce. The AIK key is discussed in the next section.
   $TS1 \leftarrow TickStamp_{AIK}(TSN_{curr}, TSN_{curr})$
3. Data from the signature of TS1 is used as a nonce to generate the self-check function checksum (CS).
   $CS \leftarrow SelfCheck(TS1.signature)$
   The self-check function is broken down into the following stages:
   (a) The self-check function measures *the self-check code proper* by performing a pseudorandom traversal over its own memory.
   (b) The self-check function performs linear sweeps over static portions of SMRAM, the BIOS, and Interrupt Vector Table.
4. The result of the checksum is then tickstamped.
   $TS2 \leftarrow TickStamp_{AIK}(CS, CS)$
5. TS1, TS2, and checksum are then stored to SMRAM to report back to an external appraiser upon request.

One key difference between the BIOS Chronomancy protocol and the Schellekens protocol is that our nonce has to be derived internally, as opposed to being received from an outside appraiser. Because of this requirement, our protocol differs slightly in both the attestation and the appraisal process. We chose to perform BIOS Chronomancy as early as possible in the boot process so that the system state has changed as little as possible, and therefore is more easily verified. Our appraisal process is detailed in Section 5.3.

5.1.1 Tickstamp Forgery Attack

While implementing our BIOS attestation code we discovered and implemented an attack on the Schellekens et al. protocol implemented by Kovah et al. Both groups describe using a TPM signing key in the tickstamp operations in their protocols. However, the use of a TPM signing key in this way allows the TPM to behave as a signing oracle on behalf of an attacker. An attacker can abuse this signing oracle to request that the TPM sign artificially crafted tickstamp structures with a TPM_Sign operation. This allows the attacker to forge a tickstamp with an arbitrary tick value. Thus an attacker can beat the existing systems by forging tickstamp structures with either ticks added to TS1 or subtracted from TS2. The delta between the two tickstamps will then always fall within the acceptable timing limits. We have implemented this attack and verified that it works. We utilize the following improvement to the protocol in order to fix this vulnerability. According to the TPM specification it is possible to use an *Attestation Identity Key* (AIK) as opposed to a signing key to perform the TPM Tickstamp operation. AIKs are restricted by the TPM to be used only with specific operations like TPM_TickStampBlob or TPM_Quote. They can not be used to sign arbitrary data with the TPM_Sign operation, and thus can not be abused to forge tickstamps. We have implemented this improved version of the protocol in BIOS Chronomancy and confirmed it to be safe against the tickstamp forging attack.

5.2 What to Measure?

To build a proper self-check, we must provide evidence that neither our code, operating as the CRTM, nor the OEM SRTM that will be handed off to has been subverted. The high-level measurement process is as follows. As shown in Figure 2 (a), BIOS code (1) instantiates SMRAM (2), which contains the BIOS Chronomancy code (3)-(6). Immediately after instantiating SMRAM, the BIOS sends a signal to the SMI handler requesting that a measurement take place. The SMI handler fields this request and begins executing the BIOS Chronomancy (BC) code. First, the BC code requests a TPM tickstamp. Then the BC self-measurement code measures the entire BC range (Section 5.2.1). Next it measures the entire SMRAM ((4) - Section 5.2.2) and the BIOS ((5) - Section 5.2.3).

5.2.1 Self Measurement

The self-measurement portion of the attestation code is logically divided into 8 blocks. The self-measurement traverses pseudorandomly through these blocks as it measures its own code. Each block incorporates the following components into the checksum:

1. EIP_DST - This is the address of the block being transferred *to*. This provides evidence of the self-check's location in memory, so that if the code *executes* from an unexpected address, it cause a different checksum.
2. DP - The data pointer for the self-check's own memory being read. This provides additional evidence of the code's location in memory so that if it is *read* from an unexpected address (such as an attacker copy), it causes a different checksum result. A 32 bit pointer.
3. *DP - This is the 4 bytes of data read from memory by dereferencing DP. This causes the final checksum to change if there are any code integrity attacks on the self-check.
4. PRN - A pseudorandom number that is seeded from the TPM tick session nonce, and updated after every use per the below pseudocode. It determines such choices as what the next DP and EIP_DST should be.

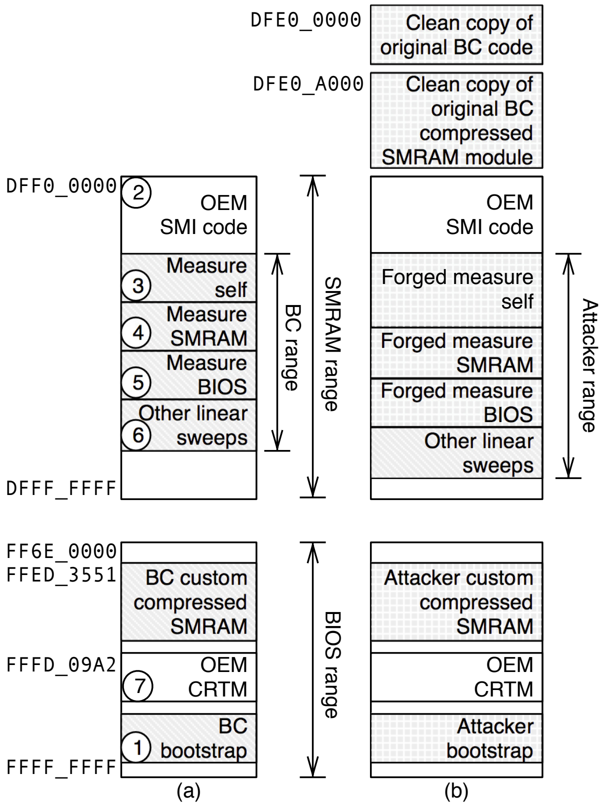

Figure 2: BIOS Chronomancy (BC) memory layout by default (a), and when an attacker is present (b)

The following presents the general construction of one of the blocks composing our self-measurement code.

```
blockOne:
  PRN += (PRN*PRN | 5);
  accumulator ^= PRN;
  accumulator += DP;
  accumulator ^= *DP;
  DP = codeStart + (PRN % codeSize);
  DiffuseIntoChecksum(accumulator);
  iterations--;
  if (iterations == 0)
    goto linearSweeps;
  //start of inter-block-transfer
  EIP_DST = blockAddressTable[PRN & 7];
  accumulator ^= EIP_DST;
  goto EIP_DST;
```

In the above C-style pseudocode, *codeStart* is the address of the start of the self-checksum code and *codeSize* defines the total size of the BIOS Chronomancy code. Each of the 8 self-checksum blocks follows the general layout presented above, but differ in both the order and logical operators used to incorporate the self-checksum components into the *accumulator* value. This is so that an attacker must follow the exact same series of pseudorandomly chosen blocks in order to compute the same checksum. The *accumulator* value is then diffused into the bits of the checksum at the end of each block with an add and rotate in order to counter the attack described in [26]. Our implementation of the self-checksum blocks were programmed by hand in x86 assembly to ensure maximum optimization. The lack of a proof of optimality of a given assembly sequence is a well known caveat of work in this field. At the end of each block, the PRN is used to determine which block should be executed next. This area is referred to as the *inter-block transfer*. A description of how an attacker can attempt to produce the correct self-checksum by modifying the block construction to forge the checksum components is given in Section 5.4.

5.2.2 SMRAM Measurement

After the BC code checks itself, in order to cede minimal space in which an attacker can hide, the entirety of SMRAM must be measured. This provides a measurement of the OEM SMM code which could have been modified by attacks such as [10]. It also provides another measurement of the self-check code, which resides in SMRAM. We use a simple linear sweep snippet of code to incorporate 4 bytes at a time from the SMRAM into the self-checksum. We also note that it is important to find and measure the SMBASE register, as this tells the processor what the base address for SMRAM should be the next time SMM is entered.

Because there is a single SMM entry point, it is required that an attacker (like a flea) residing in SMRAM will have to modify existing code or data. The entry point to SMM always has a series of conditional checks to determine the reason SMM was invoked. To gain control flow, the attacker would have to insert inline code hooks, or find a function pointer in data that could be pointed at his own code.

Immediately upon being decompressed from the BIOS firmware image, the E6400's SMRAM has a lot of "placeholder" values that get overwritten. For example, the base address of SMRAM is calculated dynamically for the given system, and this base value is then added to locations within the decompressed SMRAM executable. This is analogous to "relocations" in normal executables. A large portion of the SMRAM memory range is uninitialized and unused. Initializing this space with a pseudorandom sequence is necessary to effectively measuring SMRAM. If a static value were used, an attacker could hardcode that value into his code while residing in and forging the contents of this region. That would lead to an attacker speed up.

At the time our attestation code is invoked, SMRAM has its EBP pointing to address 0x20E and its ESP pointing to 0xDFF0_421E. Our code is compiled to expect the stack frame to point to 0xDFFF_FF00, so we have to relocate EBP for the duration of our measurements. Our local variables are large enough that, if we didn't, we would end up underflowing EBP and writing data to read-only high memory. To help optimize cache accesses, the stack pointer was modified so that our data was located on a 64-byte aligned boundary, 0xDFF0_3A80.

5.2.3 BIOS Measurement

The BIOS range is measured after the SMRAM. Because the BIOS sets up SMRAM, measuring the BIOS range will cover the compressed version of the SMRAM module. This further raises the bar for the attacker to provide consistent lies. The BIOS is mapped to high memory at 0xFFFF_FFFF minus the size of the BIOS, in bytes. So for a BIOS image of size 0x1A_0000 bytes, its measurable range in RAM would be 0xFFE6_0000 to 0xFFFF_FFFF. We use a simple linear sweep snippet of code to incorporate 4 bytes at a time from the BIOS into the self-checksum.

It has been observed on the Latitude E6400 that the entire flash chip remains static for a given BIOS revision. Therefore, the entire contents of the chip could also be incorporated into the self-check. This would include the management engine, platform data, gigabit ethernet, and flash descriptor regions in addition to the BIOS region which is mapped to memory. Some of these regions could be used by malware for the storage of malicious data and/or code. Strictly speaking, any security relevant portions should already be measured by the OEM SRTM, therefore we would be duplicating functionality, while unnecessarily increasing runtime. However, we intend to experiment with inclusion of the entire flash chip data in the future, to understand the full performance implications.

5.3 Attestation Appraisal

Appraisal of the BIOS timing based attestation proceeds similarly to the original Schellekens et al. paper. An appraiser is for instance code running on a separate server that verifies the provided attestation, and a reporting agent can be an OS application or kernel driver.

1. The appraiser sends an attestation request that includes a nonce to the reporting agent.
2. The reporting agent requests attestation results by invoking an SMI with an appropriate input and the appraiser's nonce.
3. The reporting agent generates the current tick stamp (CTS) by performing $CTS \leftarrow TickStamp_{AIK}(nonce, nonce)$.
4. The reporting agent presents to the external appraiser: CTS, TS1, TS2 and the checksum (CS).
5. The appraiser verifies the signature on TS1, TS2, and CTS to confirm that they were produced by an authentic TPM.
6. The appraiser verifies $TS1.TSN = TS2.TSN$ to confirm that the TPM tick counter was not reset during the attestation.
7. The appraiser verifies that $TS1.TSN = TS2.TSN = CTS.TSN$ to ensure that the current attestation reflects the most recent reboot of the system and a replay attack is not taking place.
8. The appraiser confirms that the nonce used to generate the checksum is a function of the signature on TS1.TSN. This enforces the requirement that TS1.TSN was calculated before the checksum calculation started.
9. The appraiser confirms that the data signed by TS1 is equal to the current tick session nonce. This precludes precalculation attacks because the attacker can not know what a future TSN will be.
10. The appraiser confirms that the data signed by TS2 is equal to the checksum returned by the agent. This enforces the requirement that TS2 was calculated after the checksum calculation was complete.
11. The appraiser extracts $\Delta = TS2.ticks - TS1.ticks$, (the time needed to perform the attestation) and confirms it is within acceptable limits.
12. The appraiser confirms that the checksum received from the agent is equal to the checksum it has calculated based on the same nonce.

If all of the above steps are completed successfully the appraisal is considered successful. To facilitate the above appraisal, a method to retrieve the measurement data is required. In our proof-of-concept implementation, the timing-based attestation code copies the measurement data from its storage address at physical address 0xA_0000 to 0x9_0000. This destination address was chosen for the proof-of-concept because it is accessible by the Windows kernel.

5.4 Attestation Forgery Attack

In this section we discuss our reference attacker under the best-case scenario for the attacker, and worst-case scenario for the defender. The attacker has full knowledge of the BIOS Chronomancy technique, and coexists in SMM alongside the measurement code. The attacker is assumed to have the minimum timing overhead because he is not trying to achieve any objective beyond forging the self-checksum to hide the fact that he is resident in the BIOS. Also, we assume the attacker knows what ranges we are measuring and has stored his copies of original binaries in locations we are not measuring, eliminating the added step of concealing them. Additionally, we ensure that the attackers code, like the original measurement code, is cache-optimized to minimize the frequency of cache collisions. The goal was to grant the attacker every advantage possible to minimize the overhead incurred from performing the forgery attack.

To hide his modifications to the self-check code, the attacker must forge the *DP values measured in each block. This can be performed by adding additional arithmetic instructions that apply an offset to DP before dereferencing it, to point it to a clean copy of the self-check code. These additional arithmetic instructions cause the block addresses of the attackers code to become misaligned with the block addresses of the original measurement code. Since these block addresses influence the EIP_DST values and are incorporated into the checksum, these too must also be forged by the attacker, adding additional timing overhead.

The original measurement code incorporates the EIP_DST component during the inter-block transfer process, requiring just a single table containing each of the measurement-block base addresses. Forging EIP_DST, however, requires the attacker to use two tables. One table stores the locations of the attacker's blocks, while an additional table is required to keep track of what the original block locations were in the un-modified code. The use of two tables instead of one introduces extra instructions and therefore extra timing overhead for the attacker. Alternatively, the attacker could use the block sizes in the unmodified code in combination with the location of the first block in the unmodified code to calculate the original EIP_DST during each inter-block transfer. However, we determined that table lookup was more efficient than on the fly calculation. The general construction of an attacker self-checksum block is presented here:

```
forgeryBlockOne:
  PRN += (PRN*PRN | 5);
  accumulator ^= PRN;
  //apply an offset "offToClean" to DP
  //to ensure it reads clean data
  accumulator += DP
  accumulator ^= *(DP-offToClean);
  DP = codeStart + (PRN % codeSize);
  iterations--;
  if (iterations == 0)
    goto forgeryLinearSweeps;
```

```
tmp = PRN & 7;
//attacker forges EIP_DST incorporation
//but incurs an additional table lookup
accumulator ^= originalAddressTable[tmp];
EIP_DST = forgeryAddressTable[tmp];
DiffuseIntoChecksum(accumulator);
goto EIP_DST;
```

In total, our original self checksum blocks were comprised of 43 x86 assembly instructions, 11 of which were memory accesses. Our most optimized attacker self checksum blocks were comprised of 47 instructions, 12 of which were memory accesses. This yields an attacker overhead of 4 instructions and 1 memory access. This overhead is incurred when the attacker attempts to maintain the correct checksum during the self-checksum's measurement of itself. However, our attestation code is designed to measure other parts of the BIOS as well during the linear sweeps that occur during Section 5.3's phase 3.b of the self-checksum algorithm. The attacker incurs additional overhead while attempting to forge the checksum during these linear sweep phases. Since this malware has achieved persistence by modifying the BIOS firmware, the attacker must hide the changes in the BIOS range from detection. Also, the attacker must hide its active runtime presence in SMRAM. To hide modifications to these regions, the attacker must provide an alternate version of the linear sweeps measurement code. The attacker's linear sweep code substitutes original good values wherever the attacker has modified code or data. We implemented a simple if/then statement to check whether the DP falls within a range where the malicious code is stored. If DP does fall within this range, then it is modified to point to the equivalent known-good *DP stored in the clean copy.

When we went to optimize for our 3rd revision of the attack, we found from performance testing that the forgery of linear sweeps accounted for a negligible amount of the total attacker overhead, as described in the next section. So we did not make a variant attacker that has code to sweep the original measurement area for known good ranges, and then switches to alternate code reading from a different DP when reading known bad ranges. But we do expect such an attacker would have slightly lower overhead and we will test that in a future revision.

6. EVALUATION

For our first experiment, we wanted to see if it was possible to have a single baseline for the expected number of TPM ticks that a good measurement should take. It was observed in [17] that different instances of the same TPM model had different baseline tick count times. The experiment outlined in that paper used STMicro TPMs, and we wanted to see if our Broadcom TPMs would behave similarly.

This experiment was performed on 17 Latitude E6400 laptops running Windows 7 32 bit on a 2.80 GHz Core 2 Duo CPU with a Broadcom BCM5880KFBG TPM. Two versions of our BC code were created: an unmodified (hereafter referred to as "clean") version and a version containing hide-only malware (hereafter referred to as the "forged" version). Both the clean and forged version of the code were run twenty times at each of 625K, 1.25M, and 2.5M iterations of the BC measurements. Each measurement recorded 8 separate timing data points that will be discussed later. To support the experiments, additional code was added to both the clean and forged versions to record the timing data points to SMRAM memory. To collect the stored data, a kernel driver was written to signal the SMI handler to make the data available for reading and outputting to file. Additionally, a windows batch file was written to execute on Windows startup to load and execute the kernel driver and then reboot the machine. The batch file repeats this process 20 times.

We found that the Broadcom TPMs did provide an equivalent baseline for the runtimes across multiple hosts for the clean and forged code. The results are shown in Table 2.

Table 2: Clean and forged BIOS Chronomancy runtimes for different numbers of iterations. 1 Broadcom TPM tick is about 64μs.

Iterations	Clean average TPM ticks	Clean std. dev. TPM ticks	Forged average TPM ticks	Forged std. dev. TPM ticks
625K	16723	15.98	16771	29.13
1.25M	21069	16.42	21793	156.75
2.5M	29756	16.90	31817	207.78

Additionally, if this technique is to be used in commercial BIOSes, we need to set a fixed number of iterations for the BC code that show a clear difference between the attacker runtime and defender runtime. This means setting some range relative to the average runtime which if exceeded indicates that an attacker has influenced the runtime. We have chosen to set the baseline range to $avg. \pm 3 * std.dev.$, ensuring a less than .3% chance that a normal measurement will fall outside of the expected range. In Figure 3 we have taken the attacker runtimes and normalized them against the upper bound of the acceptable range by subtracting it from each attacker measurement; displaying the difference in time the attackers code took to execute. This figure shows that a few of the 625K iteration attacker measurements are negative, indicating that they fall within the acceptable range. However, even the machine which had the best results for an attacker only had 6 of 20 measurements fall into the acceptable range; therefore in practice the attacker would be detected. But for all measurements with 1.25M and 2.5M iterations, the attacker data is well into positive territory, and therefore can clearly be distinguished. To guard against possible future optimization by the attacker, it would make sense to use a number of iterations of 1.25M or 2.5M in practice.

For experimental purposes we collected eight more-granular measurement time points to show what subcomponents of measurement or forgery dominate runtime.

1. RDTSC_OUT - Time stamp obtained from the RDTSC time stamp counter; referred to as "out" because it occurs "outside" of the TPM tickstamp.
2. TPM Tickstamp 1(TS1) - Tickstamp used for reporting run start
3. RDTSC_IN start - Time stamp obtained from the RDTSC time stamp counter; referred to as "in" because it falls "inside" the TPM tickstamp measurement and the actual measurement code.
4. Linear Sweeps start - Measured with an RDTSC
5. Linear Sweeps end - Measured with an RDTSC
6. RDTSC_IN end

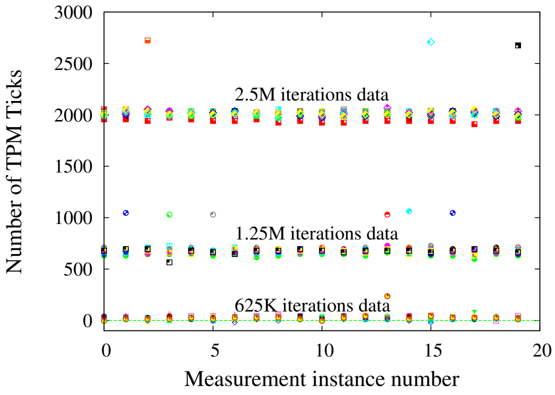

Figure 3: Attacker overhead runtimes, normalized against upper bound of acceptable clean runtime. Positive values indicate that the attacker failed to hide runtime overhead. Negative values indicate the attacker succeeded in running in an acceptable amount of time. 1 tick is about 64μs.

7. TPM Tickstamp 2 (TS2)
8. RDTSC_Out End

We will refer to these times by their number. So for instance time 3 minus time 1 is the time it takes to receive the tickstamp result from the TPM. Time 4 minus 3 is dominated by the self-check code running. Time 5 minus 4 is the time for all the linear sweeps over SMRAM, BIOS, and IVT. The total runtime, time 8 minus 1 gives us the total runtime for BIOS Chronomancy in CPU cycles, which can then be converted to an absolute runtime, as shown in Table 3. Note that runtimes do not double as the number of iterations double, in part because of overhead such as the 2 TPM tickstamp operations taking about 166ms each. The difference on top of the baseline runtime *is* seen to double as the number of iterations doubles.

Table 3: Average total runtime for BIOS Chronomancy on a single host for a 20 runs, for a given number of iterations, in milliseconds.

Iterations	Runtime (ms)
625K	1376
1.25M	1646
2.5M	2188

When we examined the runtime for clean code within the time range given by times 3 to 6, we found that 72.9 % of the time is spent between time 3 and 4, and only 6.9% of the time is spent between time 4 and 5. When we examine the hide-only adversary's 20 run average time overhead on a single host at 1.25M iterations, we see that he takes 47.1ms more than the clean code, for a total of 2.9% overhead. By comparing the RDTSC measurements again from times 3 to 6 we found that 98.6% of the overhead occurs between time 3 and time 4. Only .02% of his overhead occurs between times 4 and 5 (where even a hide-only adversary must forge the sweep of SMRAM and BIOS.) This same comparison can be seen in Table 4. Ultimately, this tells us that even for more complicated malware like a flea, the overwhelming majority of differences in timing are caused by forging the self-checksum itself, not by any other miscellaneous cleanup the attacker must do.

Table 4: Percentage overhead for a sub-section of runtime, as a portion of total overhead for the given attacker. Data taken from average of 20 runs using 1.25M iterations.

Attack Type	Percentage of total overhead incurred from time 3 to 4	Percentage of total overhead incurred from time 4 to 5
Hide-only	98.6	.02
Tick	73.55	6.73
Flea	73.80	6.75

7. CONCLUSION

As the *core* root of trust for measurement, proper implementation of the CRTM is critical. However, this work is the first detailed examination of a real implementation of a CRTM & SRTM and finds that implementation to be untrustworthy.

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.

Because all CRTMs, including those on the latest UEFI systems, are rooted in mutable firmware, the applicability of these attacks is very broad. PC vendors seem unwilling to root their trust in truly immutable ROM, likely because modern firmware is large and complex (as multiple BIOS updates attest to). Therefore, in the absence of immutable roots of trust, we have found and fixed problems with existing work on TPM tickstamp timing-based attestation, and adapted it to run in BIOS and SMM. Our BIOS Chronomancy system provides robust protection under the demonstrably real assumption that an attacker can be operating at the same privilege level as the CRTM in the BIOS. Although BIOS vendors place a premium on speed of boot time, some customers such as governments require much higher assurance that their firmware is not modified. We believe our evaluation has shown that BC is a practical technique and can be incorporated into existing OEM BIOS codebases as an optional configurable option for those customers who need trustworthy detection of BIOS infection.

8. ACKNOWLEDGEMENTS

Special thanks to Jonathan McCune for helpful feedback on an early version of this paper. And to Rafal Wojtczuk and Yuriy Bulygin for detailed ideas on future attacks.

9. REFERENCES

[1] Coreboot. http://www.coreboot.org/. Accessed: 5/01/2013.

[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. Accessed: 5/01/2013.

[6] Y. Bulygin. Evil maid just got angrier: Why full-disk encryption with TPM is insecure on many systems. In *CanSecWest*, Vancouver, Canada, 2013. Accessed: 5/01/2013.

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

[8] I. 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: 5/01/2013.

[9] 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*.

[10] L. Duflot, O. Levillain, B. Morin, and O. Grumelard. Getting into the SMRAM: SMM Reloaded. Presented at CanSec West 2009, http://www.ssi.gouv.fr/IMG/pdf/Cansec_final.pdf. Accessed: 02/01/2011.

[11] 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: 5/01/2013.

[12] T. C. Group. TPM PC Client Specific Implementation Specification for Conventional BIOS. Version 1.21 Errata version 1.0, Feb. 24 2012.

[13] J. Heasman. Implementing and detecting a ACPI BIOS rootkit. In *BlackHat Europe*, Amsterdam, Netherlands, 2006. Accessed: 5/01/2013.

[14] C. Kallenberg, J. Butterworth, and X. Kovah. Dell BIOS in some Latitude laptops and Precision Mobile Workstations vulnerable to buffer overflow. http://www.kb.cert.org/vuls/id/912156. Accessed: 08/01/2013.

[15] C. Kallenberg, J. Butterworth, and X. Kovah. Under submission. http://www.kb.cert.org/vuls/id/255726. Accessed: 08/01/2013.

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

[17] 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*.

[18] 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)*.

[19] 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)*.

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

[21] J. Rutkowska and R. Wojtczuk. Preventing and detecting Xen hypervisor subversions. In *BlackHat*, Las Vegas, USA, 2008. Accessed: 5/01/2013.

[22] A. Sacco and A. Ortega. Persistent BIOS infection. In *CanSecWest*, Vancouver, Canada, 2009. Accessed: 5/01/2013.

[23] 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*.

[24] 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.

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

[26] 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.

[27] 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*.

[28] 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*.

[29] 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: 5/01/2013.

[30] R. Wojtczuk and J. Rutkowska. Attacking Intel TXT. In *BlackHat Federal*, Washington D.C., USA, 2009. Accessed: 5/01/2013.

[31] R. Wojtczuk and A. Tereshkin. Attacking Intel BIOS. In *BlackHat*, Las Vegas, USA, 2009. Accessed: 5/01/2013.

[32] M. Yamamura. W95.CIH - Symantec. http://www.symantec.com/security_response/writeup.jsp ?docid=2000-122010-2655-99. Accessed: 5/01/2013.

APPENDIX

A. TERMINOLOGY REFERENCE

There are numerous acronyms related to low level firmware, trusted computing, and timing-based attestation technologies. Although all terms are defined when first used in the paper, this appendix can serve as an easier reference.

AIK - Attestation Identity Key - A TPM key with more restricted capabilities than a general signing key.
BIOS - Basic Input Output System - The firmware that executes at system reboot on an x86 PC.
BC - BIOS Chronomancy - The name of our timing-based attestation system hosted in the BIOS.
CS - (BC-specific) CheckSum
CTS - (BC-specific) Current TickStamp
CRTM - Core Root of Trust for Measurement - The initial component of the SRTM that must be trusted implicitly, and that must measure itself.
DP - (BC-specific) Data Pointer
***DP** - (BC-specific) A pseudo-C language style indication of dereferencing the DP
DRTM - Dynamic Root of Trust for Measurement - A root of trust that is instantiated on demand. While technically BC is a DTRM, we use this term only to refer to the more commonly understood implementations such as Intel TXT or AMD SVM.
EEPROM - Electronically Erasable Programmable Read Only Memory - The storage type often used for BIOS.
EIP_DST - (BC-specific) The x86 extended instruction pointer (EIP) that is the destination for the next BC block jump.
flash chip - Synonym of EEPROM for our purposes.
ICH - I/O Controller Hub - Responsible for mediating access to slow peripherals
LPC Bus - Low Pin Count Bus - The bus the TPM is on.
MCH - Memory Controller Hub - Responsible for mediating access to RAM and fast peripherals
OEM - Original Equipment Manufacturer
PCR - Platform Configuration Register - A storage area on a TPM that can only be set by SHA1 hashing the current value concatenated with the input value.
PRN - (BC-specific)Pseudo-Random Number - Updated in every BC block, and seeded by the TSN.
SMI - System Management Interrupt - Dedicated x86 interrupt to invoke SMM code
SMM - System Management Mode - The most privileged general execution domain on x86 systems
SMRAM - System Management RAM - The memory where SMM code and data is stored
SPI Bus - Serial Peripheral Interface Bus - The bus the BIOS EEPROM is located on.
SRTM - Static Root of Trust for Measurement - A root of trust that indicates the configuration that the system booted into.
TCG - Trusted Computing Group - Responsible for the TPM specification.
Tickstamp - A tick count *according to the TPM's internal clock*, that has been signed by a TPM key.
TPM - Trusted Platform Module - A dedicated chip that provides some cryptographic capabilities.
TS1,TS2 - (BC-specific) TickStamp1,2
TSN - Tick Session Nonce - A part of a TPM TickStamp data that is pseudo-randomly generated on each TPM reset.
UEFI - Unified Extensible Firmware Interface - A specification for firmware design.

B. ADDITIONAL FIGURES

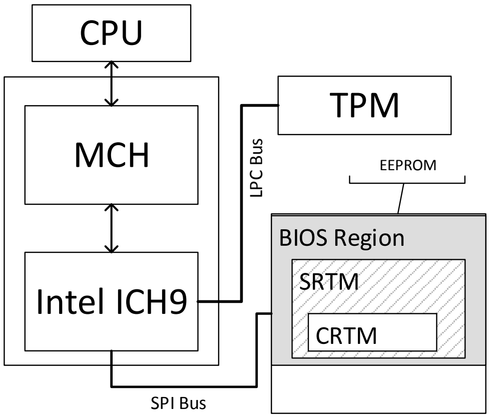

Figure 4: Relation of major components discussed in this paper

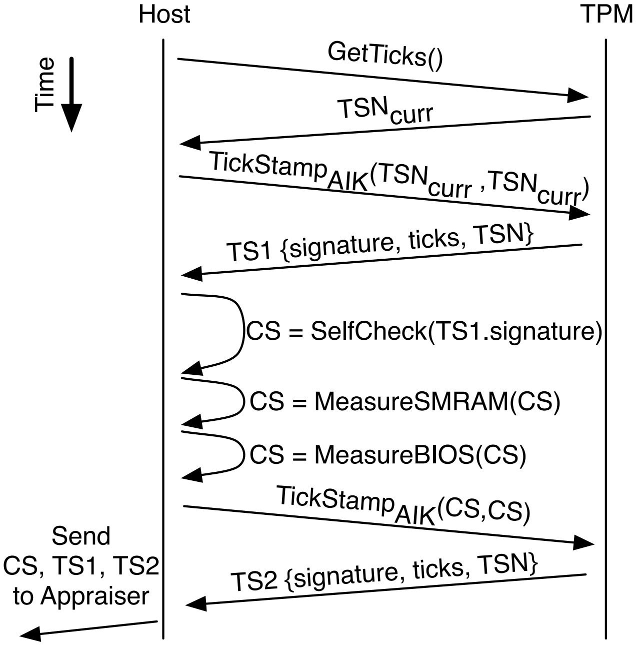

Figure 5: Graphical representation of protocol described in section 5.1