A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping

Seunghun Han, Wook Shin, Jun-Hyeok Park, and HyoungChun Kim,
National Security Research Institute

https://www.usenix.org/conference/usenixsecurity18/presentation/han

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping

Seunghun Han Wook Shin Jun-Hyeok Park HyoungChun Kim
National Security Research Institute
{*hanseunghun, wshin, parkparkqw, khche*}*@nsr.re.kr*

Abstract

This paper reports two sorts of Trusted Platform Module (TPM) attacks regarding power management. The attacks allow an adversary to reset and forge platform configuration registers which are designed to securely hold measurements of software that are used for bootstrapping a computer. One attack is exploiting a design flaw in the TPM 2.0 specification for the static root of trust for measurement (SRTM). The other attack is exploiting an implementation flaw in tboot, the most popular measured launched environment used with Intel's Trusted Execution Technology. Considering TPM-based platform integrity protection is widely used, the attacks may affect a large number of devices. We demonstrate the attacks with commodity hardware. The SRTM attack is significant because its countermeasure requires hardware-specific firmware patches that could take a long time to be applied.

1 Introduction

The Trusted Platform Module (TPM) was designed to provide hardware-based security functions. A TPM chip is a tamper-resistant device equipped with a random number generator, non-volatile storage, encryption functions, and status registers, which can be utilized for applications such as ensuring platform integrity and securely storing keys. The Trusted Computing Group (TCG) is an industry consortium whose goal is to specify and standardize the TPM technology, which includes security-related functions, APIs, and protocols. The initial version of the TPM main specification (TPM 1.2) [31] was published in 2003. The revised version, the TPM library specification 2.0 (TPM 2.0) [37] was initially published in 2013.

The TPM technology provides a trustworthy foundation for security-relevant applications and services. TPM is a major component of the integrity measurement chain that is a collection of system components such as the bootloader, kernel, and other components. The chain can either start statically from Basic Input and Output System (BIOS)/Unified Extensible Firmware Interface (UEFI) code modules when the system is booted or dynamically from a specialized instruction set during runtime.

Regardless of how the chain starts, the measurements are "extended" to platform configuration registers (PCRs) inside the TPM. When a value is extended to a PCR, the value is hashed together with the previously stored value in the PCR and then the PCR is updated with the hashed result. A small bit change to a PCR value will affect all the following extended values. The extended values in PCRs can be compared to expected values locally or submitted to a remote attestor. Namely, the integrity measurement chain must be started from a trustworthy entity, also known as the root of trust for measurement (RTM).

The TPM has been widely deployed in commodity devices to provide a strong foundation for building trusted platforms, especially in devices used in enterprise and government systems. The US Department of Defense also considers the TPM to be a key element for dealing with security challenges in device identification and authentication, encryption, and similar tasks.

The TPM chip is designed to cooperate with other parts of the system, e.g., the firmware and the operating system. Mechanisms for cooperation are often complicated and fail to be clearly specified. This may result in critical security vulnerability.

Power management is one of the features which increases complexity of the cooperation. The goal of power management is to save power by putting the system into a low-power state or even cutting off the power when the system is idle. How the power management works is quite complicated because each peripheral device can have its own power state independently from the system-wide power state.

A recent Linux kernel supports the Advanced Config-

uration and Power Interface (ACPI), which is an open industry specification that enables operating system-centric intelligent and dynamic management coordination with power management-aware devices such as CPUs, networks, storage, and graphics processing units.

TPM is a peripheral that supports ACPI. The information stored in the TPM chip such as keys and state values are very important for maintaining the security of the whole system, TPM has to actively and safely save and restore the state as the power state changes.

Unfortunately, the TPM does not safely maintain the state when the power state changes. We found vulnerabilities in both types of RTM that allow an adversary to reset and forge PCRs when the system wakes up. Therefore, the system may look normal even after it has been modified. Considering that TPM has been widely deployed, the impact of our finding is critical, especially when it comes to static measurement. The vulnerability of a static RTM (SRTM) is due to a flawed specification, which means that many products that implement the specification can be affected and patches would not be applicable to all of the products immediately. The vulnerability of the dynamic RTM (DRTM) is due to a bug in the open source project, tboot, which is the most popular measured launch environment (MLE) for Intel's Trusted eXecution Technology (TXT). Patching the bug is relatively simple, and our patch[1] can be found on the tboot project [9]. We also have obtained Common Vulnerabilities and Exposures (CVE) identifiers: CVE-2018-6622 for the SRTM and CVE-2017-16837 for the DRTM attack, respectively.

This paper makes the following contributions:

- We present vulnerabilities that allow an adversary to reset the PCRs of a TPM. The PCRs are resettable whether the RTM processes start statically or dynamically.
- We craft attacks exploiting these vulnerabilities. The attacks extract normal measurements from the event logs recorded during the boot process, and then they use the measurements to perform a replay attack.
- We also address countermeasures for these vulnerabilities. To remedy the SRTM vulnerability that we found, hardware vendors must patch their BIOS/UEFI firmware. We have contacted them and are waiting for releases of the patches. We also produced a patch by ourselves for the DRTM vulnerability that we found. We have obtained the CVE IDs of both vulnerabilities.

In the following sections, we review TPM and ACPI technologies. Then, we introduce their vulnerabilities and exploits against them. The exploits are demonstrated in a variety of commercial off-the-shelf devices. The results of the attacks are presented in this paper. We also suggest different ways of mitigating the vulnerabilities that we found.

[1]The commit hash is 521c58e51eb5be105a29983742850e72c44ed80e

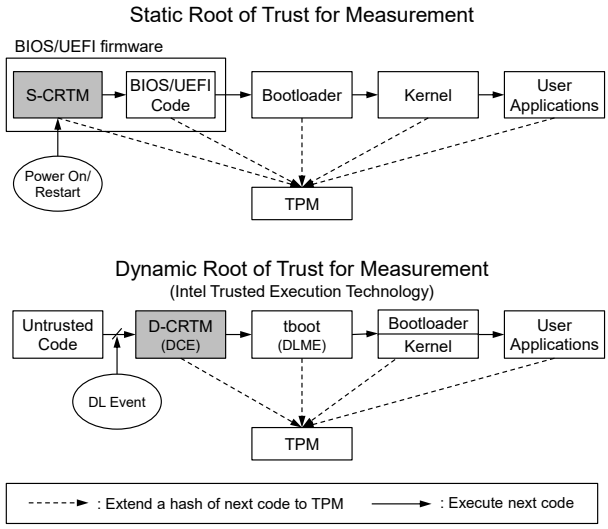

Figure 1: Examples of static and dynamic RTM (SRTM and DRTM, respectively) processes

2 Background

2.1 TPM Technology

A trusted computing base (TCB) [37] is a collection of software and hardware on a host platform that enforces a security policy. The TPM helps to ensure that the TCB is properly instantiated and trustworthy. A measured boot is a method of booting in which each component in the boot sequence measures the next component before passing control to it. In this way, a trust chain is created. The TPM provides a means of measurement and a means of accumulating these measurements. PCRs are the memory areas where the measurements can be stored. When a measurement is "extended" to a PCR, the measurement is hashed together with the current value of the PCR, and the hashed result replaces the current value. As long as the values are updated in this way, it is easy to find an alteration in the middle of the chain. A particular value of a PCR can be reproduced only when the same values are extended in the same order. The trustworthiness of the platform can be determined by investigating the values stored in PCRs. It is also possible to request the PCR values remotely. Remote attestation is a challenge-response protocol that sends PCR values in the form of a digitally signed quote to a remote attestor.

The TPM also functions as a secure storage by provid-

PCR Index	PCR Usage
0	S-CRTM, BIOS, host platform extensions, and embedded option ROMs
1	Host platform configuration
2	BIOS: Option ROM code UEFI: UEFI driver and application code
3	BIOS: Option ROM configuration and data UEFI: UEFI driver, application configuration, and data
4	BIOS: Initial Program Loader (IPL, e.g., bootloader) code and boot attempts UEFI: UEFI boot manager code (e.g., bootloader) and boot attempts
5	BIOS: IPL code configuration and data UEFI: Boot manager code configuration, data, and GPT partition table
6	BIOS: State transitions and wake events UEFI: Host platform manufacturer specific
7	BIOS: Host platform manufacturer specific UEFI: Secure boot policy
8-15	Defined for use by the OS with SRTM
16	Debug
17-22	Defined for use by the DRTM and OS with DRTM
23	Application support

Table 1: Summary of PCR usage (TPM 1.2 and 2.0)

ing "sealing" and "binding" operations that limit access to the storage based on a specific platform state. For example, a TPM's "sealed" data can be decrypted by the TPM only when the PCR values match specified values. "Unbinding" data is done by a TPM using the private key part of the public key used to encrypt the data. Binding can be done by anyone using the public key of a TPM, but unbinding is done by the TPM only because the private key part is securely stored inside TPM and is even locked to specific PCR values.

A chain of trust is an ordered set of elements in which one element is trusted by its predecessor. The trustworthiness of the whole chain depends on the first element. An RTM is the trust anchor of a measurement chain. A TPM is designed to report the platform state securely, but it cannot initiate the measurements by itself. Initiating the measurement is done by another software component that can be trusted called the core RTM (CRTM). Figure 1 shows two different types of RTM: SRTM [32, 39] and DRTM [33]. In addition, Table 1 shows the PCR usage for SRTM and DRTM.

SRTM is the trust anchor that is initialized by static CRTM (S-CRTM) when the host platform starts at power-on or restarts. Often, SRTM is an immutable software program that is stored in ROM or a protected hardware component. In contrast, DRTM launches a

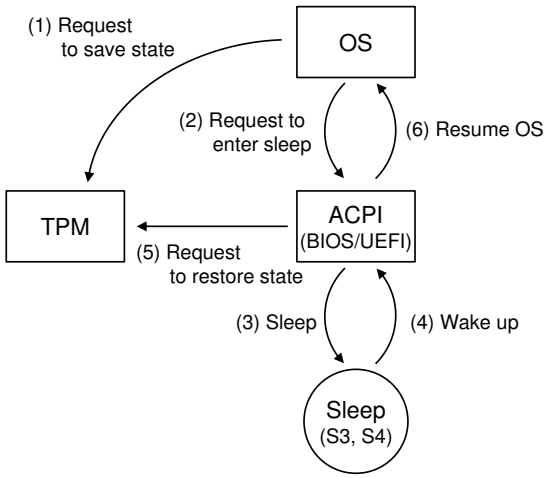

Figure 2: ACPI sleep process with TPM

measured environment at runtime without platform reset. When the dynamic chain of trust starts with a dynamic launch event (DL Event), the DRTM configuration environment (DCE) preamble performs the initial configuration and prepares the DRTM process [33, 43]. As the DRTM process starts, the special code module (the DCE), is executed as a dynamic CRTM (D-CRTM), validates whether the platform is trustworthy, and transfers the control to the initial part of the operating system, called the dynamically launched measured environment (DLME).

A chain of trust can be expanded to user-level applications beyond the operating system kernel. Integrity Measurement Architecture (IMA) [26] measures applications before executing them. IMA is included in the kernel, and therefore its authenticity can be guaranteed by the trust chain.

2.2 ACPI Sleeping States

ACPI [42] is an open standard for architecture-independent power management. It was released in 1996 after being co-developed by Intel, Hewlett-Packard (HP), and other companies.

The ACPI specification defines power states and the hardware register sets that represent the power states. There are four global power states, defined as working (G0 or S0), sleeping (G1), soft-off (G2), and mechanical-off (G3). The sleeping state is divided into four sleeping states:

- S1: *Power on Suspend*. The CPU stops executing instructions, but all devices including CPU and RAM are still powered.
- S2: The same as S1 except the CPU is powered off.

- S3: *Sleep (Suspend to RAM)*. All devices are powered-off except for RAM.

- S4: *Hibernation (Suspend to Disk)*. The platform context in the main memory is saved to disk. All devices are powered off.

Like other devices, a TPM chip is powered off in states S3 or S4. The TCG specifications [32, 39] define how the state is maintained while the power state changes. They also define the roles of the operating system and BIOS/UEFI firmware. The steps defined for saving and restoring the TPM state are summarized in Figure 2. Before sleep, the operating system requests the TPM chip to save the state, and then makes a transition to sleeping states by sending a request to the ACPI in the BIOS/UEFI firmware. All hardware devices are either powered off (in S4) or only the main memory remains powered (in S3). When the platform exits from the sleeping states, the BIOS/UEFI firmware requests the TPM to restore the state and then it starts the operating system.

The TCG specification describes the role of power management over the operating system and the BIOS/UEFI firmware. Power management will be efficient and work as long as the operating system and firmware cooperate well. For the S3 sleep function to work properly, each part must function perfectly without error; however, this state may collapse when one part malfunctions, which is hard to correct using the other parts. Moreover, the power management of a TPM chip needs to be carefully considered when it is partly handled by an operating system that could be compromised by rootkits [29]. In Section 4, we demonstrate how incomplete power management control breaks the chain of trust.

3 Assumptions and Threat Model

3.1 Assumptions

First, we assume that our system measures the firmware and bootloader using TCG's SRTM [32, 39]. Many commodity laptops, PCs, and servers come with TPM support. When their TPM support option is enabled in the BIOS/UEFI menu, the BIOS/UEFI firmware starts the "trusted boot" [25] process, which means that it measures the firmware itself and the bootloader and stores the measurements in the TPM chip.

Second, we assume that our system employs TCG's DRTM architecture [43]. When a DRTM chain starts at runtime, the DRTM itself, kernel file, and initial RAM disk (initrd) file are measured, and the measurements are kept in the TPM. Both Intel and AMD have their extended instructions for supporting DRTM, called TXT and Secure Virtual Machine, respectively. For our experiments, we use Trusted Boot (tboot) [11], which is an open source implementation of the Intel TXT [12].

We also assume that the stored measurements in TPM are verified by a remote attester. These measurements should be unforgeable by an attacker; therefore, any modification in the firmware, bootloader, or kernel will be sent to and identified by an administrative party.

3.2 Threat Model

We consider an attacker who has already acquired the Ring-0 privilege with which the attack can have the administrative access to the software stack of a machine including the firmware, bootloader, kernel, and applications. The attacker might use social engineering to acquire this control or could exploit zero-day vulnerabilities in the kernel or system applications. The attacker may be able to safely upgrade the UEFI/BIOS firmware to a new and manufacture-signed one. However, we assume that he or she cannot flash the firmware with arbitrary code. We also assume that the attacker cannot roll-back to an old version of the firmware, where the attacker can exploit a known vulnerability.

The attacker's primary interest is to hide the breach and retain the acquired privileges for further attacks. TPM and SRTM/DRTM should measure the system and securely leave proof in the PCRs if the bootstrapping software or kernel has been modified. This proof also can be delivered to and verified by a remote administrator.

The attacker may try to compromise the bootloader and kernel by modifying files in the EFI partition and under /boot/. This is feasible because we assume the attacker has privileged accesses to every part of the system software. Moreover, it is easy to obtain, modify, and rebuild the bootloader, kernel, and kernel drivers. The GRand Unified Bootloader (GRUB) and TPM driver that we used in our experiments are accessible via a GitHub repository [5, 19]. Namely, the attacker can boot the system with a modified bootloader or with another boot option if the system has multiple boot options. The TPM and SRTM/DRTM are supposed to securely record and report the fact that the system has not booted with an expected bootloader and configuration. However, they would fail to do that.

We do not consider a denial-of-service attack in this paper. If the attacker has system privileges, he or she can easily turn the system off. We also do not consider hardware attacks that require a physical access to the system circuits. Vulnerabilities of the System Management Mode (SMM) [13] may allow the attacker to remotely and pragmatically alter firmware binary or change the BIOS/UEFI options [6], but we do not consider such vul-

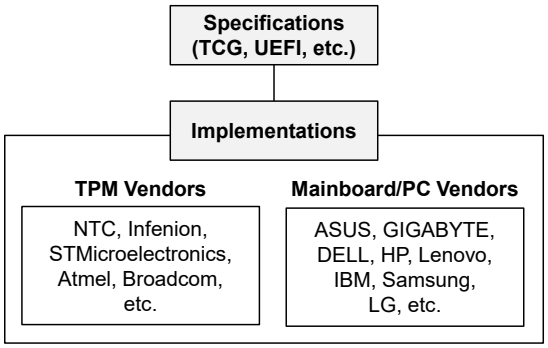

Figure 3: TPM technology entities

nerabilities. Rather, we show that the TPM and SRTM/-DRTM can fail without the need to exploit them.

4 Vulnerability Analysis

4.1 Finding the Security Vulnerabilities

Bootstrapping a system utilizing TPM and SRTM/-DRTM technologies involves many entities, and Figure 3 shows their relationships. Security vulnerabilities can be found when formally analyzing the design and specification of a system, however, it is challenging to formally specify them anyway. Instead, we basically reviewed the specification documents manually and tested real systems. The steps we took to find the vulnerabilities are as follows:

1. While reviewing the TCG specification, we found a change in the TCG specification from TPM 2.0 to TPM 1.2 regarding power management. The difference was regarding restarting TPM when the system resumes [37].

2. Using a real system with support for TPM and SRTM, we tested how a TPM state can be saved and restored as the power state cycles. We found an abnormal behavior when the TPM state is reset. We speculated that the failure was due to the firmware implementations not meeting the specification or ambiguity in the specification [37]. Note that another flaw caused by not meeting the TCG specification has been reported already [3].

3. Based on speculation, we tested other implementation instances of the specification. We could have investigated the firmware source code, but we needed to experiment with a number of products because the firmware of these products is not open. Eventually, the same vulnerability was confirmed in several systems.

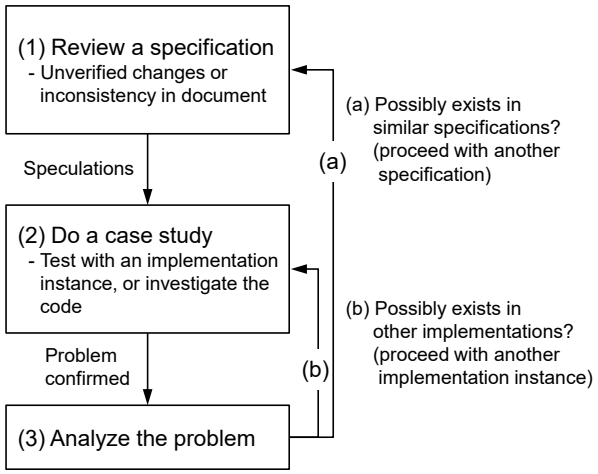

Figure 4: General process of the vulnerability analysis in TPM

4. We investigated the DRTM specifications. At this time, we thought we could apply what we learned to the DRTM, which is similar to the SRTM. In the DRTM, the DCE and DLME are verified, initialized, and launched by hardware support, which means the process is performed by immutable parties.

5. We investigated the open source implementation of DRTM, tboot [11], which is based on Intel TXT. The vulnerability of an authenticated code module (ACM), which is the DCE of Intel TXT, as reported by Wojtczuk and Rutkowska [44, 45] demonstrates that the authenticity and integrity of code are not guaranteed to be flawless. Unlike previous studies, we focus on tboot, which is the DLME, and eventually found mutable function pointers that we were able to exploit.

We summarize this process in Figure 4.

4.2 SRTM Vulnerability: CVE-2018-6622

4.2.1 Problem: The Grey Area

SRTM starts up the chain of trust by measuring each component of the boot sequence including the BIOS/UEFI firmware, bootloader, and kernel. The measurements are extended to the PCRs, from PCR #0 to PCR #15. An alteration of a booting component would leave different values in the PCRs. The alteration can easily be identified when the values are then compared to the correct ones.

It is known that it is difficult for malicious software to become involved in the booting sequence and forge PCR values to hide its involvement. To forge these values, the

malicious software needs to reset the TPM and extend the exact same series of measurements. This is infeasible because the TPM reset requires a host platform to restart.

However, we recently found that PCRs can be initialized when the host platform sleeps. When the platform enters into the S3 or S4 sleeping states, the power to the devices is cut off. TCG specifies how TPM can support power management [32, 37]: TPM is supposed to save its state to the non-volatile random access memory (NVRAM) and restore the state back later. However, the specification does not specify sufficiently how it should be handled when there is no saved state to be restored [39]. As a result, some platforms allow software to reset the PCRs and extend measurements arbitrarily.

A TPM typically has two power states, the working state (D0) and the low-power state (D3). The TPM has a command for saving its state before putting itself into the D3 state and a command for restoring the saved state when getting out of the D3 state. According to the TPM 1.2 specification [32], the operating system may enter into the S3 sleeping state after notifying the TPM that the system state is going to change by sending it the TPM_SaveState command. On exiting from the S3 sleeping state, the S-CRTM determines whether the TPM should restore the saved state or be re-initialized. When S-CRTM issues TPM_Startup(STATE), the TPM restores the previous state. When TPM_Startup(CLEAR) is issued, the TPM restarts from a cleared state.

An unexpected case that could reset the TPM can occur if there is no saved state to restore. How to tackle this problem is specified differently in the TPM 1.2 and 2.0 specifications. In version 1.2 [32], TPM enters failure mode and is not available until the system resets. In version 2.0, TPM2_Shutdown() and TPM2_Startup() correspond to TPM_SaveState() and TPM_Startup(), respectively. Version 2.0 [39] tells TPM to return TPM_RC_VALUE when TPM2_Startup(STATE) even if it does not have a saved state to restore. It also specifies that the SRTM should perform a host platform reset and send the TPM2_Startup(CLEAR) command before handing over the control to the operating system.

Restarting the SRTM and clearing the TPM state is not sufficient to assure the integrity of the platform. It is simply the same as resetting the TPM. An adversary can hence still extend an arbitrary value to the PCRs. This must be forbidden. Otherwise, there should be a way to warn that the TPM state has been reset abnormally.

Although another specification document [37] states that the CRTM is expected to take corrective action to prohibit an adversary from forging the PCR values. However, the specification does not either mandate it or explain how to do this in detail. The incompleteness of this specification may lead to inappropriate implementations and eventually destroy the chain of trust. How an adversary forges the measurements is demonstrated in Section 4.2.2.

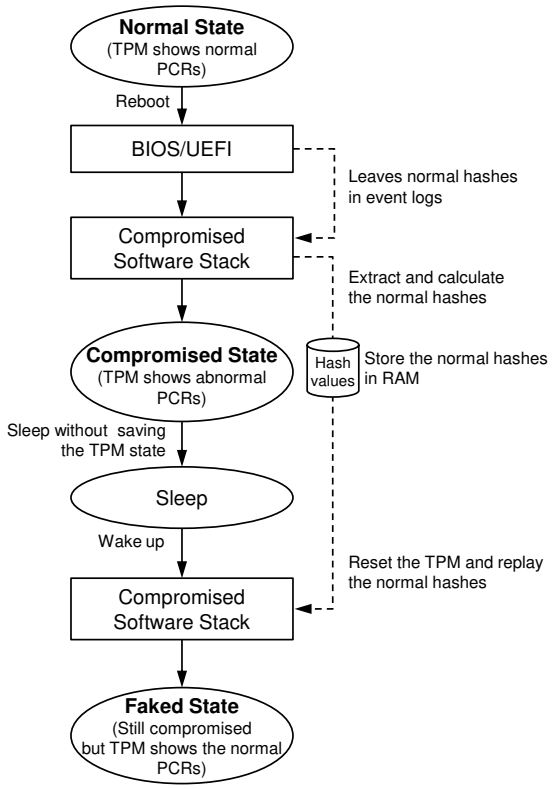

Figure 5: Exploit scenario for the SRTM vulnerability

4.2.2 Exploit Scenario

The aim of an exploit is to conceal the fact that the system has been compromised. By assumption, our attacker has already taken control of the system software including the bootloader and the kernel. Figure 5 depicts the main points of the exploit scenario. The attacker obtains good hash values from the BIOS/UEFI event logs, which are recorded during a normal boot process. Assorted hash values are stored in RAM temporarily, and are finally handed over to the kernel. The attacker can forge PCR values using the obtained hashes after sleep. As a result, the TPM shows that the system is booted and running with genuine software, which is not at all true. The technical details of the exploit are explained in Section 4.2.3.

4.2.3 Implementation in Detail

We explain how to reset the TPM state and counterfeit the PCR values. Figure 6 shows the detailed process of exploiting SRTM vulnerability.

First, before resetting and replaying the TPM, we need

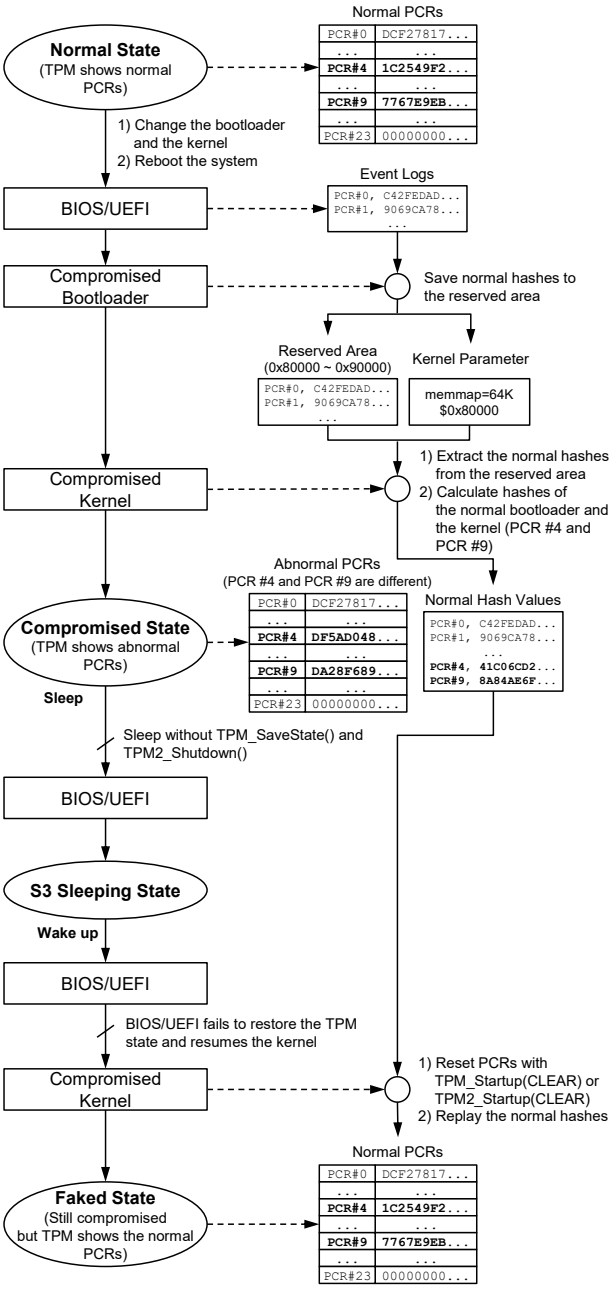

Figure 6: Detailed process of exploiting the SRTM vulnerability

the normal digest values. The normal digests can be extracted from the TCG event logs. When a value is extended to a PCR, the firmware makes an entry in the TCG event logs for later verification. According to TCG ACPI specification [38], the starting address of the preboot event logs is written in the Local Area Start Address field of the Hardware Interface Description Table in the ACPI table. This field is located at offset 42 in TPM 1.2, whereas it is optionally located at offset 68 in TPM 2.0.

Bootloader	BIOS support	UEFI support	TPM 1.2	TPM 2.0
GRUB for CoreOS [5]	✓	✓	✓	✓
Trusted-GRUB1 [40]	✓		✓	
Trusted-GRUB2 [41]	✓		✓	
GRUB-IMA [24]	✓		✓	

Table 2: List of bootloaders with BIOS/UEFI support and TPM version

When the field is not there, there is another option for obtaining the logs. The BIOS/UEFI firmware saves the event logs separately as well for its own use. These logs are accessible until the control is given to the kernel in UEFI mode because they are removed when ExitBootService() is called [36].

To obtain and reuse the normal digests in the logs, we crafted exploits modifying an existing bootloader and the kernel. The bootloader calls the GetEventLog() UEFI interface and collects all event logs. The logs are passed to the kernel through a reserved memory region. The logs are saved in a 64K memory block starting from 0x80000, which is below the 1MB address space. This area should be excluded from the kernel range by setting the kernel's command line parameter "memmap = 64K $ 0x80000" so that the data written in that region can be kept after booting. Our exploit in the kernel resets TPM by making the system enters the S3 sleeping state, and finally extends the measurements, one after another, in the normal order as presented in the logs.

We take the GRUB implementation from the open source Container Linux [4] to implement our exploit. To our knowledge, it is the only existing bootloader implementation that supports UEFI and both versions of the TPM. Table 2 summarizes the bootloaders that have TPM support. Our customized bootloader functions as the SRTM and extracts the event logs for both TPM 1.2 and 2.0. Figure 7 shows an example of the event logs extracted from an Intel mini PC (NUC5i5MYHE).

The normal measurements can be obtained after parsing the event logs. A log entry of the event logs is composed of a PCR index, an event type, a digest, an event size, and event data. The PCR index is the PCR to which a digest is extended. The event type can be either a CRTM version, UEFI firmware variable, initial program loader (IPL), or IPL data. Table 3 summarizes the types needed to parse the event logs. The digest is the hashed result of binary or text values depending on the event type, whereas the event data stores raw data. The event size is the size of the raw data.

The parsed digest values, except for the nor-

```
Dump Address 0xFFFFB8FFC1E40000(Physical Address 0x80000)
TCG Event_version = 1
PCR 0,   Event Type 0x8,         Size 16,  Digest C42FEDAD268200CB1D15F97841C344E79DAE3320
PCR 7,   Event Type 0x80000001, Size 52,  Digest 2F20112A3F55398B208E0C42681389B4CB5B1823
PCR 7,   Event Type 0x80000001, Size 36,  Digest 9B1387306EBB7FF8E795E7BE77563666BBF4516E
PCR 7,   Event Type 0x80000001, Size 38,  Digest 9AFA86C507419B8570C62167CB9486D9FC809758
PCR 7,   Event Type 0x80000001, Size 36,  Digest 5BF8FAA078D40FFBD03317C93398B01229A0E1E0
PCR 7,   Event Type 0x80000001, Size 38,  Digest 734424C9FE8FC71716C42096F4B74C88733B175E
PCR 0-7, Event Type 0x4,         Size 4,   Digest 9069CA78E7450A285173431B3E52C5C25299E473
PCR 5,   Event Type 0x80000006, Size 484, Digest 5C64EDAEA674F708F24B152A79AF26D45990BF65
PCR 4,   Event Type 0x80000003, Size 186, Digest 41C06CD2A38EB0B6208A93D0227E5C49668AA550
PCR 8,   Event Type 0xD,         Size 75,  Digest 3EDC5474CC2D9BDCCAB031E75C6C7C3DF06DF729
... omitted ...
```

Figure 7: TPM event logs of Intel NUC5i5MYHE extracted by the custom bootloader

```
/*****************************************/
/* Skip tpm_savestate and tpm2_shutdown  */
/* in drivers/char/tpm/tpm-interface.c   */
/*****************************************/
int tpm_pm_suspend(struct device *dev)
{
  ... omitted ...
+  printk(KERN_INFO"tpm: tpm_savestate() "
+    "and tpm2_shutdown() are skipped\n");
+  return 0;
+
  if (chip->flags &
    TPM_CHIP_FLAG_ALWAYS_POWERED)
    return 0;

  if (chip->flags & TPM_CHIP_FLAG_TPM2) {
    tpm2_shutdown(chip, TPM2_SU_STATE);
    return 0;
... omitted ...
```

Figure 8: Patch code summary of custom kernel for TPM reset

mal bootloader and kernel (PCR #4 and PCR #9), are the ones to be replayed. The log entry for the bootloader hash can be identified by event type EV_EFI_BOOT_SERVICES_APPLICATION (0x80000003) and the one for the kernel (including the kernel file and the initial RAM disk file) hash is identified by event type EV_IPL (0x0D). Note that the digest originates from our customized bootloader and kernel, not from the original ones. The bootloader and kernel hash values can be obtained from the original bootloader and kernel instead. The bootloader hash value has to follow the Windows Authenticode Portable Executable Signature Format [23, 35]; however, the kernel hash value can be calculated using the sha1sum tool.

To reset the TPM, two tasks must be performed. One is to modify the kernel so that it skips to saving the TPM state and calling TPM_Startup(CLEAR) or TPM2_Startup(CLEAR) after waking up. The code listed in Figure 8 shows how simple this modification is. We add return code at the start of function tpm_pm_suspend() and call function tpm_startup() in the TPM driver using our test kernel module (see include/linux/tpm.h [19]). The other task is to wait until the system sleeps or make the system sleep by giving a suspend command like the ones that systemd or the pm-utils package provides. After resetting the TPM, the normal measurements can be re-extended. We call function tpm_pcr_extend() in the TPM driver to replay the hashes.

4.3 DRTM Vulnerability: CVE-2017-16837

4.3.1 Problem: Lost Pointer

DRTM builds up the dynamic chain of trust at runtime, and it uses the set of PCRs from PCR #17 to PCR #22. These dynamic PCRs [32, 39] need to be initialized during runtime, but the initialization is restricted to locality 4 [34], which means their access is controlled by trusted hardware and not accessible to software. However, in addition to the hardware buttons, there is another chance to reset the PCRs. The dynamic PCRs are initialized when the host platform escapes from the S3 and S4 sleeping states. The DRTM specification [33] explains how DRTM can be reinitialized after the sleeping states.

4.3.2 Exploit Scenario

To undermine a DRTM, some of the extended measurements sent to dynamic PCRs should be forgeable. This is not easy because the DCE, being executed prior to the DLME [33], launches the DLME after extending the measurement of the DLME, as shown in Section 2, however, after the DLME has started, security is a matter of the trustworthiness of the DLME. In other words, it is still possible to break the dynamic trust chain as long as the DLME implementation has own vulnerability.

As shown in Figure 9, the DRTM exploit is mostly similar to the SRTM one. The attacker obtains the good

Event Type	Label and Description
0x00000001	EV_POST_CODE This event must be extended to PCR #0. It is used to record power-on self test (POST) code, embedded SMM code, ACPI flash data, boot integrity services (BIS) code, or manufacturer-controlled embedded option ROMs.
0x00000004	EV_SEPARATOR This event must be extended to PCR #0-PCR #7. It is used to delimit actions taken during the pre-OS and OS environments. In case of TPM 1.2, the digest field must contain a hash of the hex value 0x00000000 for UEFI firmware and 0xFFFFFFFF for BIOS. In case of TPM 2.0, the digest field must contain a hash of the hex value 0x00000000 or 0xFFFFFFFF for TPM 2.0.
0x00000008	EV_S_CRTM_VERSION This event must be extended to PCR #0. It is used to record the version string of the SRTM.
0x0000000D	EV_IPL This event field contains IPL data.
0x80000001	EV_EFI_VARIABLE_DRIVER_CONFIG This event is used to measure configuration for EFI variables. The digests field contains the tagged hash of the variable data, e.g. variable data, GUID, or unicode string.
0x80000003	EV_EFI_BOOT_SERVICES_APPLICATION This event measures information about the specific application loaded from the boot device (e.g., IPL).
0x80000006	EV_EFI_GPT_EVENT This event measures the UEFI GPT table.
0x80000008	EV_EFI_PLATFORM_FIRMWARE_BLOB This event measures information about non-PE/COFF images. The digests field contains the hash of all the code (PE/COFF .text sections or other sections).

Table 3: Summary of event types that are frequently used [39]

hash values left in the logs. After sleep, the values are re-extended to the PCRs by hooking the functions in the DCE and DLME. The result is the same as that of the SRTM exploit.

4.3.3 Implementation in Detail

We explain how to reset the TPM state and counterfeit the PCR values. The tboot [11] is an open source implementation of Intel TXT that employs the notion of DRTM to support a measured launch of a kernel or a virtual machine monitor (VMM). It consists of the secure initialization (SINIT) ACM and tboot, which correspond to the DCE and DLME, respectively. In Intel TXT, the

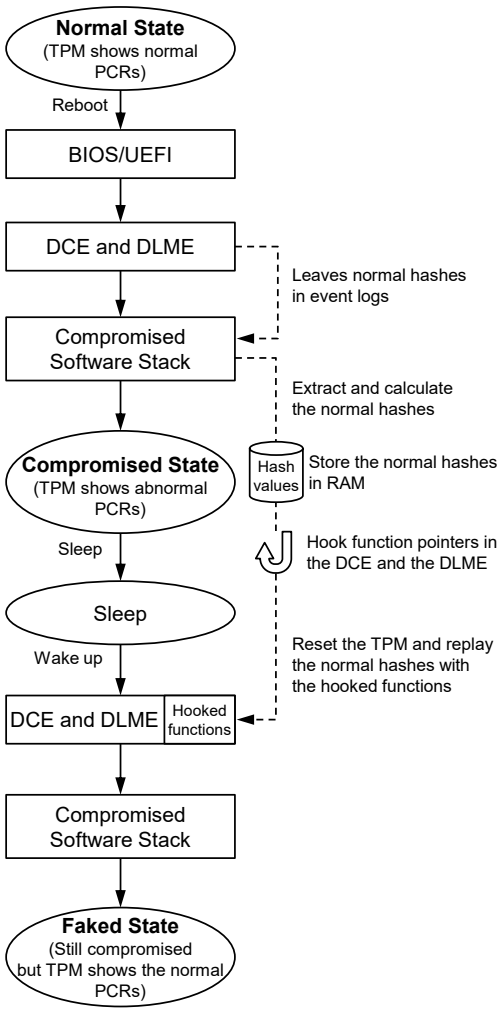

Figure 9: Exploit scenario for the DRTM vulnerability

DLME is called the MLE. The steps of tboot are shown in Figure 10.

The tboot part is loaded by a bootloader, together with a kernel or a VMM. When the bootloader transfers the control to tboot, its pre-launch part starts the SINIT ACM. It measures the MLE (tboot) and extends the measurements to the dynamic PCRs. SINIT ACM starts the post-launch part of tboot, it measures the DRTM components, and extends the dynamic PCRs according to either legacy PCR mappings or details/authorities PCR mappings. Legacy PCR mappings use PCR #17, PCR #18, and PCR #19 for extending the measurements of the launch control policy (LCP), kernel file, and initial RAM disk (initrd) file, respectively. Details/authorities PCR mappings use PCR #17 for the measurements of the LCP, kernel file, and initrd file. PCR #18 is reserved for measurements of the verification key for SINIT ACM and LCP. When exiting the S3 sleeping state, tboot restarts DRTM using the data loaded in the memory at the boot

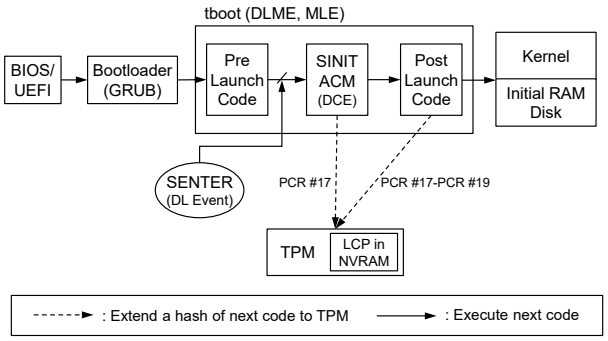

Figure 10: Steps of tboot

time. This means that the process of measuring and extending the kernel or the VMM can be interfered with by compromising the data loaded in the memory.

After reviewing the source code of tboot, we found that some mutable function pointers that are not measured open up a window of attack. Figure 11 shows the detailed process of the exploit for the DRTM vulnerability using mutable function pointers.

According to Intel's specification [14], SINIT ACM obtains a loaded address, a size, and the entry point of an MLE by reading the MLE header. The header should be placed inside the loaded MLE and measured by the SINIT ACM so that unauthorized modification of the header is not allowed. In the latest version of the tboot source code (1.9.6, at the time of this writing), the start and the end of an MLE (_mle_start and _mle_end) are defined in the link script (as shown in Figure 12) including from the start of the code section (.text) to the end of the read-only data section (.rodata). Therefore, any alteration of those sections will be identified by the measurement extended by SINIT ACM.

In contrast to the code and read-only data, the writable data section (.data) and the uninitialized data segment (the .bss section) are not measured. After careful investigations, we found that some variables (g_tpm, tpm_12_if, and tpm_20_if, as shown in Figure 13) exist in the unmeasured sections and could affect the control flow. The mutable variables are function pointers left behind and not measured. By hooking those pointers, we can hook the control flow and eventually forge the dynamic PCRs, bypassing the protections provided by the SINIT ACM.

Similarly to the attack explained in Section 4.2, the normal measurements extended by tboot are recorded in the event logs that reside in the kernel's memory area. The txt-stat tool provided by tboot dumps the kernel memory via /dev/mem and prints out the summary status of TXT and event logs, as shown in Figure A.1 in Appendix.

After obtaining the normal digests, we can forge extended values after tboot takes control by hooking the exposed function pointers. The hook functions reside in the data section of tboot in shellcode form, and the hooking has to be done before the platform enters the S3 sleeping

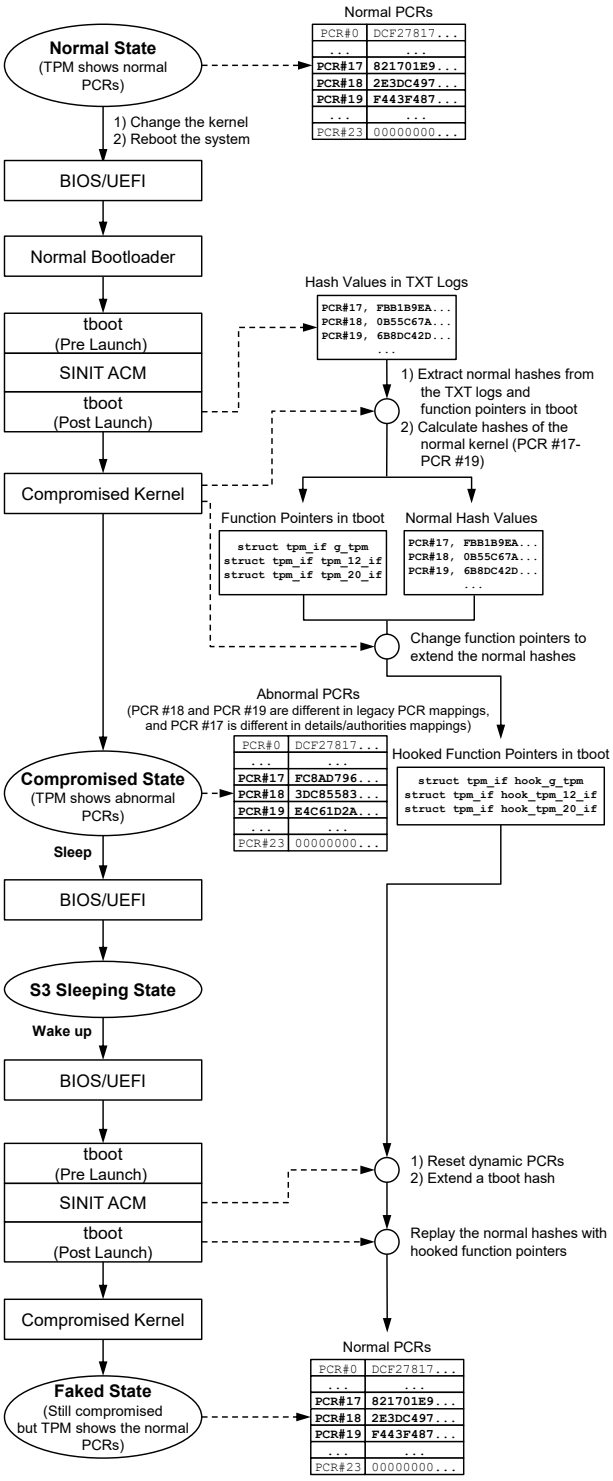

Figure 11: Detailed process of exploiting the DRTM vulnerability

```
SECTIONS
{
  . = TBOOT_BASE_ADDR; /* 0x800000 */

  .text : {
  *(.tboot_multiboot_header)
  . = ALIGN(4096);
  *(.mlept)

  _mle_start = .;      /* Beginning of MLE */
  *(.text)
  *(.fixup)
  *(.gnu.warning)
  } :text = 0x9090

  .rodata : { *(.rodata) *(.rodata.*) }
  . = ALIGN(4096);

  _mle_end = .;        /* End of MLE */

  .data : {            /* Data */
  *(.data)
  *(.tboot_shared)
  CONSTRUCTORS
  }
  ... omitted ...
}
```

Figure 12: Sections in the link script (tboot.lds.x) of tboot

```
/* Beginning of text section (ready-only) */
800000 t multiboot_header
800010 t multiboot2_header
800020 t multiboot2_header_end
801000 t g_mle_pt
804000 T _mle_start    /* Beginning of MLE */
804000 T _start
804000 T start
804010 T _post_launch_entry
... omitted ...
83b000 D _mle_end      /* End of MLE */

/* Beginning of data section (writable) */
83b000 D s3_flag
... omitted ...
83f234 D g_tpm    /* Current TPM interface */
... omitted ...
83f2c0 D tpm_12_if    /* TPM interfaces in */
83f460 D tpm_20_if    /* data section for  */
                      /* TPM 1.2 and 2.0   */
... omitted ...
```

Figure 13: tboot symbols. The TPM interfaces are in the data section

state. The locations of g_tpm, tpm_12_if, and tpm_20_if are as shown in Figure 13. The offsets might differ according to the versions of the implementation, but those function pointers are exposed in the mutable section.

The last step of the attack, likewise, is to reset the TPM state and replay the normal digests. The difference is that, when the platform wakes up, tboot and SINIT ACM are executed. SINIT ACM resets the dynamic PCRs, measures tboot, and extends the measurements to PCR #17. It starts tboot again, and tboot extends the PCRs with the hook functions. The replay should be done by extending the measurements in the designated order for replacing the measurement of the customized kernel with the normal one.

4.4 Evaluation

We tested our exploits on various Intel-based platforms to determine how many devices are exposed to these vulnerabilities. The tested devices are listed in Table 4. Ubuntu 16.04.03 was used as the host operating system. The genuine kernel 4.13.0-21-generic of the operating system was used for our customization, in which we removed the TPM_SaveState() or TPM2_Shutdown() calls. For the SRTM attack mentioned in Section 4.2, we used the source code of CoreOS GRUB 2.0 [5]. For the DRTM attack, we used source code from the tboot project [11]. The devices were UEFI booted from the external hard disk drive, where we installed the customized system with exploits. To replace the normal bootloader and kernel with our customized ones, we put the customized ones under the /boot directory with the same name.

TPM 2.0 supports multiple banks of PCRs, with each bank implementing different hash algorithms. The BIOS/UEFI firmware and the kernel are likely to be extended to separate banks. Although the reported vulnerabilities do not depend on a specific hash algorithm, we used SHA-1 in all evaluations only because the algorithm is supported in both versions of the TPM.

The DRTM exploit requires devices to support Intel TXT and tboot. However, some of them do not support Intel TXT and some of the TXT-supporting devices do not work with tboot, as a result, we could exploit only a few of them. Table A.1 in Appendix shows the tested devices.

4.4.1 SRTM Attack: Grey Area Vulnerability

Table 5 compares all normal PCR values and exploited PCR values except for PCR #10, which is extended by IMA in the kernel. Although the PCR #10 values of all PCs are different, the value of PCR #10 can be extended from PCR #0-PCR #7. We hence attach additional tables in our GitHub repository [10], which lists the PCR values obtained from the normal SRTM-based booting sequence on our tested devices.

Because the static PCRs values are measurements of the SRTM components, most of the values differ according to the manufacturers and model, except for PCR #4 and PCR #9, where the measurements of the boot-

PC No.	Vendor	CPU (Intel)	PC and mainboard model	BIOS Ver. and release date	TPM Ver.	TPM vendor and firmware Ver.	SRTM attack
1	Intel	Core i5-5300U	NUC5i5MYHE	MYBDEWi5v.86A, 2017.11.30	2.0	Infineon, 5.40	Y
2	Intel	Core m5-6Y57	Compute Stick STK2mv64CC	CCSKLm5v.86A.0054, 2017.12.26	2.0	NTC, 1.3.0.1	Y
3	Dell	Core i5-6500T	Optiplex 7040	1.8.1, 2018.01.09	2.0	NTC, 1.3.2.8	Y
4	GIGABYTE	Core i7-6700	Q170M-MK	F23c [2], 2018.01.11	2.0	Infineon, 5.51	Y
5	GIGABYTE	Core i7-6700	H170-D3HP	F20e, 2018.01.10	2.0	Infineon, 5.61	Y
6	ASUS	Core i7-6700	Q170M-C	3601, 2017.12.12	2.0	Infineon, 5.51	Y
7	Lenovo	Core i7-6600U	X1 Carbon 4th Generation	N1FET59W (1.33), 2017.12.19	1.2	Infineon, 6.40	N [3]
8	Lenovo	Core i5-4570T	ThinkCentre m93p	FBKTCPA, 2017.12.29	1.2	STMicroelectronics, 13.12	N [3]
9	Dell	Core i5-6500T	Optiplex 7040	1.8.1, 2018.01.09	1.2	NTC, 5.81.2.1	N [4]
10	HP	Xeon E5-2690 v4	z840	M60 v02.38, 2017.11.08	1.2	Infineon, 4.43	N [3]
11	GIGABYTE	Core i7-6700	H170-D3HP	F20e, 2018.01.10	1.2	Infineon, 3.19	N [3]

Table 4: List of PC and mainboard models and results of the SRTM attack

PC No.	TPM Ver.	PCR No.	PCR values [5] of the ORIGINAL system	PCR values of the COMPROMISED system	PCR values after the SRTM attack
1-7, 9-11	1.2, 2.0	4	`1C2549F2...`	`DF5AD048...`	`1C2549F2...`
		9	`7767E9EB...`	`DA28F689...`	`7767E9EB...`
8 [6]	1.2	4	`849162AD...`	`9966FE5A...`	`849162AD...`
		9	`7767E9EB...`	`DA28F689...`	`7767E9EB...`

Table 5: Forged PCR values after the SRTM attack

loader and kernel are extended. Interestingly, the Lenovo m93p machine (PC #8) has a different value for PCR #4, even though it uses the same bootloader. After looking into the event logs, the m93p machine uses a hash of 0xFFFFFFFF as the event separator (EV_SEPARATOR) while all the other devices use a hash of 0x00000000. It seems 0xFFFFFFFF is used when the firmware is BIOS [32] and 0x00000000 is used for UEFI [35], as long as the TPM version is 1.1 or 1.2. In case of TPM 2.0, the specification [39] allows both of the values to be used. The m93p machine is supposed to use 0x00000000 because it uses TPM 1.2 and a UEFI firmware. This nonconformity does not immediately wreck the security, but it may increase the complexity of resource management, especially in an enterprise where an administrator needs to attest or track down installed software inside the administrative domain.

Table 4 also shows whether the reset and replay attack are possible when each device is booted with the customized bootloader and kernel. All devices with TPM 2.0 are vulnerable to the attack; nevertheless, they are from different manufacturers such as Intel, Dell, GIGABYTE, and ASUS. It seems that all of the manufacturers considered in this study failed to deal with the exception mentioned in Section 4.2 because of the incomplete specification.

On the contrary, all TPM 1.2 devices, except for the Dell Optiplex 7040 mini PC (PC #9), appropriately handle the exception by entering failure mode, in which re-

[2] The EV_S_CRTM_VERSION event is not extended to PCR #0 and the EV_EFI_PLATFORM_FIRMWARE_BLOB event is not extended to PCR #2, which are wrong probably because the software does not comply with the TCG Specification

[3] Entering failure mode

[4] The static PCR values are kept

[5] Only the first eight hexadigits are shown here for the brevity

[6] PC #8 has a different value in PCR #4, which seems incorrect

PC No.	TPM Ver.	PCR No.	Before the intrusion	After the intrusion	After the DRTM attack
1	2.0	17	821701E9...	FC8AD796...	821701E9...
3	2.0	17	257B1024...	E90F27EC...	257B1024...
8, 9	1.2	18	2E3DC497...	3DC85583...	2E3DC497...
		19	F443F487...	E4C61D2A...	F443F487...

Table 6: Forged PCR values after the DRTM attack

```
/* EV_SRTM_VERSION event is not extended to
    PCR #0 */
[1] PCR 0, Event Type 0x8, Digest
0000000000000000000000000000000000000000

[2] PCR 0, Event Type 0x1, Digest
3EBB5D91DA1BC78CB0F206B398AD28520885FEB3

/* EV_EFI_PLATFORM_FIRMWARE_BLOB event is
    not extended to PCR #2 */
[3] PCR 2, Event Type 0x80000008, Digest
0000FF7FB8B600000000169C09B3000000000070

[4] PCR 0, Event Type 0x4, Digest
9069CA78E7450A285173431B3E52C5C25299E473

[5] PCR 2, Event Type 0x4, Digest
9069CA78E7450A285173431B3E52C5C25299E473

... omitted ...
```

Figure 14: Event log summary relate to PCR #0 and PCR #2 in the GIGABYTE Q170-MK

setting and replaying are not possible. We note that the machine (PC #9) maintains the static PCRs even after entering into and exiting from the sleeping states without saving the TPM state. It seems the machine neither cuts the power to the TPM chip off nor restores the TPM state.

It is interesting that the Dell Optiplex 7040 machine with TPM 2.0 (PC #3) is exploitable while the same machine with TPM 1.2 is not, even when the same BIOS/UEFI firmware is used throughout the experiments. This implies that different branches of the same binary are executed depending on the TPM version. The firmware logic may need to be divided according to the TPM versions.

In the case of the GIGABYTE Q170-MK desktop PC (PC #4), the TPM state was successfully reset, but the exploit ended up failing to replay the measurements. The event logs show that the replayed values of PCR #0 and PCR #2 do not match the normal values. Figure 14 shows the series of events that occurred during the boot. Based on the logs, PCR #0 is supposed to be extended with the digests of the event types of 0x8, 0x1, and 0x4. However, it turned out that the SRTM of the PC does not extend the digest of type 0x8 (EV_S_CRTM_VERSION), which does not comply with the TCG standard [39]. Similarly, the SRTM of the Q170-MK (PC #4) omits the digest of type 0x80000008 (EV_EFI_PLATFORM_FIRMWARE_BLOB). This sort of non-conformity may cause a misinterpretation of the device status when a remote attestor expects a particular value according to the standard.

4.4.2 DRTM Attack: Lost Pointer Vulnerability

The normal states of the dynamic PCRs after booting the devices listed in Table 4 are shown in the fourth column of Table 6. The values of the dynamic PCRs are from the measurements of SINIT ACM, tboot, the kernel file, and initial RAM disk file. The states after the DRTM attack are listed in the fifth column of Table 6. The changed values are affected by the customized bootloader and kernel. As explained at the beginning of Section 4.3.3, PCR #18 and PCR #19 differ when the legacy PCR mappings are used, while only PCR #17 is different when the details/authorities PCR mappings are used. The sixth column of Table 6 lists the states to which measurements could be re-extended. Because the vulnerability is due to the software stack, all the devices are exposed to the attack regardless of the TPM version.

5 Discussion and Solutions

5.1 Discussion

Kauer [17] listed the conditions that are required to make a chain of measurements trustworthy.

- Condition 1: The CRTM must be trustworthy. In the case of S-CRTM, the code is stored in ROM. According to the TCG specification [39], S-CRTM only starts when the system is reset and initializes the chain of measurements. For the D-CRTM, the measurement environment is initialized and started only by special hardware instructions, for example, SKINIT of AMD and GETSEC[SENTER] of Intel.
- Condition 2: The PCRs should be resettable only by trusted code.

- Condition 3: The chain has to be contiguous.

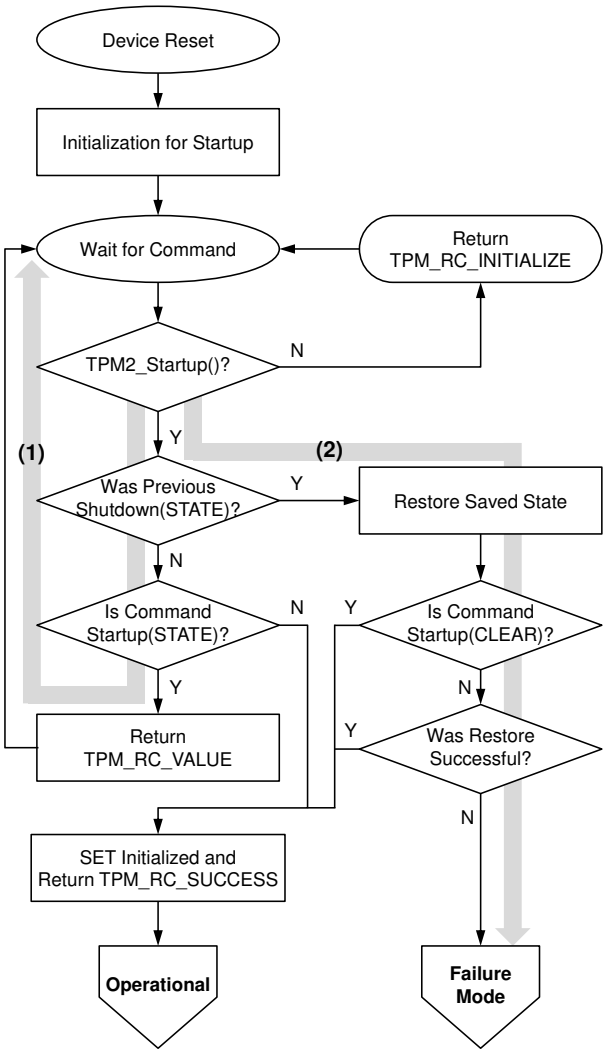

Figure 15: Part of the TPM startup sequences [37]

Our SRTM attack falsifies Condition 2: we are able to reset the TPM without rebooting the system. The attack enabled by the TPM 2.0 specification [37]. Figure 15 shows a part of the "TPM Startup Sequences" diagram taken from the specification document. The vulnerability is due to the absence of a saved state, and it occurs when TPM2_Startup(STATE) is called with no preceding TPM2_Shutdown(STATE) command. As Figure 15 shows, the sequence of transitions (1) ends up with the command-waiting state, which means the TPM is ready to work as usual. As a result, the attacker can reset the PCRs by sending TPM2_Startup(CLEAR) command. The specification expects the CRTM to take "corrective action" in such cases, but does not clearly specify what to do.

The DRTM attack that we discovered does not technically falsify Condition 1. Instead, the attack raises the question whether we can naively assume the correctness of the software in the trust chain. It is difficult to make software free of vulnerabilities. Some studies [17, 20, 21] have proposed designing secure systems using the DRTM supports in order to decrease the size of TCB and remove vulnerable BIOS, OptionROMs, and bootloaders from the trust chain. Unfortunately, even if this issue is addressed, there still is room to find software bugs, as we discovered.

After resetting the TPM, we completed our attack by re-extending the PCRs with good measurements that we obtained from the event logs. According to the TCG specifications [35, 36, 33], prior to passing the control over to the operating system, the BIOS/UEFI firmware and DCE/DLME leave event logs and record measurements. Considering that the operating system can obtain the event logs and the extend operation is provided by the kernel, the specification must address how to protect or remove good measurements recorded in the event logs, in order to prevent the replay attack.

5.2 Solutions

For the SRTM vulnerability, a brutal and desperate remedy is to prohibit the platform from entering the S3 sleeping state, since this power state transition is a vital part of the attacks. Some BIOS/UEFI firmware provides a menu option to disuse the S3 sleeping state.

A better way to address this vulnerability starts with revising the specification. The TPM 2.0 specification should mandate the TPM enter failure mode if there is no state to restore. This approach makes the TPM 2.0 specification consistent with the TPM 1.2 specification. Note that the TPM 1.2 devices in Table 4 were not affected by the attack because they were not resettable when in failure mode. A remote attester can also identify devices in failure mode. The TPM 2.0 devices are already specified to go to failure mode if they cannot successfully start, as shown in Figure 15, path (2). Note that updating the specification has to be followed by updating the TPM firmware.

We have contacted and reported our findings to Intel [16], Dell [7], GIGABYTE [8], and ASUS [1], which are the vendors of the devices we have tested and confirmed to be vulnerable. Intel and Dell are in the process of patching their firmware to take corrective action. We requested a CVE ID regarding the grey area vulnerability, and this ID has been obtained (CVE-2018-6622).

For the specific DRTM vulnerability, we have already sent a patch to the tboot project, which also can be found in the tboot repository [9]. The patch removes the function pointers exposed in the mutable data memory and

protects the APIs inside the measured environment from unauthorized accesses. The CVE ID regarding the lost pointer vulnerability has also been obtained (CVE-2017-16837).

The DRTM vulnerability is due to the exposed function pointers from the virtual function table. To facilitate runtime polymorphism, virtual function tables are often used to dispatch a collection of functions that define the dynamic behavior of an object. These tables need to be included in a section to be measured (e.g., the .text section) or in a read-only data section (.rodata). Otherwise, these tables could be exploited by an attacker who wants to corrupt the pointer and manipulate the behavior of the program. To prevent such attacks, all RTM code must be developed under secure coding standards and audited carefully [27]. Potential flaws could be searched for by source code analysis tools.

6 Related Work

6.1 SRTM Attacks

Kursawe et al. [18] tapped into the Low Pin Count bus signal, which is used for communication between the TPM chip and the CPU. Concealed information such as keys can then be acquired using simple wiretapping attack.

Kauer [17] demonstrated an attack that resets a version 1.1 TPM chip by physically connecting a reset pin to ground. However, this TPM reset attack requires physical access, whilst our discovered attack can be done by software remotely. The author also patched a BIOS TPM driver and flashed the modified BIOS for the purpose of disabling the SRTM. The author implemented a bootloader that uses AMD's DRTM supporting instruction and proposed this bootloader as an alternative to the existing weak SRTM implementations.

Sparks [30] pointed out several vulnerabilities and limitations of the TPM. First, a TPM chip cannot protect programs after it has been loaded because measurements are taken before execution. Second, physical reset is possible. Third, stored keys can be guessed by a side channel attack that measures time differences of RSA calculation. Sparks also summarized the countermeasures against those threats: loaded programs can be protected by hypervisors, the Low Pin Count bus can be protected from attacks by employing tamper-resistant circuits, and the timing attack on the RSA calculation can be prevented by employing the techniques that better hide the statistics of the calculation.

Butterworth et al. [2] exploited a vulnerable BIOS update process to re-flash a BIOS chip with an arbitrary firmware that contains rootkits. After the adversary takes control of the BIOS/UEFI firmware and SMM, IMA [26] and BitLocker [22] cannot protect the TPM. As a mitigation of those attacks, the authors proposed a time-based remote attestation that does not rely on the TPM.

6.2 DRTM Attacks

Wojtczuk and Rutkowska demonstrated an attack against Intel TXT by compromising SMM code [44]. SMM is an operating mode in which code is executed in the most privileged execute mode, which is privileged than a hardware hypervisor. The authors found that SMM code is not measured and were able to infect the system's SMM handler. The authors also found that an arbitrary code can be executed in the SINIT ACM by exploiting an implementation bug within it [45]. The attack even loads an arbitrary MLE and forges the PCR values bypassing protections provided by Intel TXT.

Wojtczuk et al. introduced an attack that exploits a bug in the SINIT ACM [46]. With this attack, they can compromise a hypervisor even when Intel TXT is present. In the attack, they demonstrated that the SINIT ACM cannot protect the Direct Memory Access Remapping ACPI Table, which holds information about the configuration for VT-d (Intel's Virtualization Technology for Direct I/O). VT-d technology [15] is a hardware support for isolating device access and is considered to be a countermeasure against direct memory access attacks, which can bypass the memory protection of a CPU and access system memory.

Sharkey introduced a hypervisor rootkit that emulates the SENTER instruction and TPM using a thin hypervisor [28]. The rogue hypervisor rootkit runs underneath the kernel, compromises Intel TXT, traps access to it, and tricks the system by providing forged PCRs.

7 Conclusion

The TPM is a hardware component found in many modern computers and is intended to provide the root of trust. TPM is specified by TCG and implemented as a tamper-resistant integrated circuit that provides cryptographic primitives and secure storage to hold secret information and reports about the platform state.

The TCG specifications specify how to create and retain a chain of trust based on interactions between the TPM and the RTM. More technologies and manufacturers have become involved as the specification have been updated, as a result, this increased complexity underneath the measurement process. Consequently, logical conflicts and incompleteness in the specifications are obscured and the specification may provide poor guidance to vendors as to its implementation.

In this paper, we addressed the vulnerabilities that allow an adversary to enable a TPM reset and forge PCRs.

One vulnerability comes from a flawed specification, and many commodity devices seem to be affected. The other vulnerability is from an implementation defect in the popular open source implementation of the MLE for Intel TXT.

We crafted attacks exploiting these vulnerabilities and demonstrated them with commodity products. We have informed the hardware manufacturers about our findings, and the vendors are expected to produce and deploy a patch. We also created a patch for correcting the error in the open source project. This patch has already been merged.

8 Acknowledgments

We would like to express our sincere gratitude to Jonathan M. McCune. His constructive and priceless advice greatly helped us improve our manuscript. We also thank the anonymous reviewers and Junghwan Kang for their insightful comments. This work was supported by National IT Industry Promotion Agency (NIPA) grant funded by the Korea government (MSIT) (No.S1114-18-1001, Open Source Software Promotion).

References

[1] ASUS. ASUS product security advisory. https://www.asus.com/Static_WebPage/ASUS-Product-Security-Advisory/.

[2] BUTTERWORTH, J., KALLENBERG, C., KOVAH, X., AND HERZOG, A. BIOS chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013), ACM.

[3] BUTTERWORTH, J., KALLENBERG, C., KOVAH, X., AND HERZOG, A. Problems with the static root of trust for measurement. *Black Hat USA* (2013).

[4] COREOS. CoreOS. https://coreos.com.

[5] COREOS. GRand Unified Bootloader. https://github.com/coreos/grub.

[6] CR4SH. Lenovo ThinkPad System Management Mode arbitrary code execution 0day exploit. https://github.com/Cr4sh/ThinkPwn.

[7] DELL EMC. Dell EMC product security response center. https://www.emc.com/products/security/product-security-response-center.htm.

[8] GIGABYTE. GIGABYTE technical support. http://www.gigabyte.us/Support/Technical-Support.

[9] HAN, S. Fix security vulnerabilities rooted in tpm_if structure and g_tpm variable. https://sourceforge.net/p/tboot/code/ci/521c58e51eb5be105a29983742850e72c44ed80e/.

[10] HAN, S., SHIN, W., PARK, J.-H., AND KIM, H. List of normal pcr values for SRTM. https://github.com/kkamagui/papers/blob/master/usenix-security-2018/appendix-SRTM-pcr-values.pdf.

[11] INTEL. Trusted Boot. https://www.sourceforge.net/projects/tboot.

[12] INTEL. Intel Trusted Execution Technology (Intel TXT) enabling guide. *Intel White Paper* (2015).

[13] INTEL. *Intel 64 and IA-32 Architectures - Software Developer's Manual: Vol. 3B*. Intel, 2016.

[14] INTEL. Intel Trusted Execution Technology (Intel TXT). *Intel White Paper* (2017).

[15] INTEL. Intel Virtualization Technology for directed I/O. *Intel White Paper* (2017).

[16] INTEL SECURITY. Intel security center. https://security-center.intel.com.

[17] KAUER, B. OSLO: Improving the security of trusted computing. In *USENIX Security Symposium* (2007).

[18] KURSAWE, K., SCHELLEKENS, D., AND PRENEEL, B. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH-CRyptographic Advances in Secure Hardware* (2005).

[19] LINUX. TPM drivers. https://github.com/torvalds/linux/tree/master/drivers/char/tpm.

[20] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND SESHADRI, A. Minimal TCB code execution. In *Security and Privacy, IEEE Symposium on* (2007), IEEE.

[21] MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *ACM SIGOPS Operating Systems Review* (2008), vol. 42, ACM.

[22] MICROSOFT. Microsoft BitLocker drive encryption. http://windows.microsoft.com/en-US/windows-vista/BitLocker-Drive-Encryption-Overview.

[23] MICROSOFT. Windows Authenticode portable executable signature format. http://msdn.microsoft.com/en-us/windows/hardware/gg463180.aspx.

[24] OPEN PLATFORM TRUST SERVICES. OpenPTS. https://github.com/openpts/openpts.

[25] PARNO, B., MCCUNE, J. M., AND PERRIG, A. *Bootstrapping Trust in Modern Computers*, 1st, ed. Springer, 2011.

[26] SAILER, R., ZHANG, X., JAEGER, T., AND VAN DOORN, L. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium* (2004).

[27] SEACORD, R. *Secure Coding in C and C++*. SEI Series in Software Engineering. Pearson Education, 2013.

[28] SHARKEY, J. Breaking hardware-enforced security with hypervisors. *Black Hat USA* (2016).

[29] SKAPE, S. Bypassing PatchGuard on Windows x64. http://www.uninformed.org/?v=3&a=3&t=pdf, 2005.

[30] SPARKS, E. R. A security assessment of trusted platform modules. *Dartmouth College, USA, Tech. Rep. TR2007-597* (2007).

[31] TRUSTED COMPUTING GROUP. TPM main part 1 design principles. *TCG White Paper* (2011).

[32] TRUSTED COMPUTING GROUP. TCG PC client specific implementation specification for conventional BIOS. *TCG White Paper* (2012).

[33] TRUSTED COMPUTING GROUP. TCG D-RTM architecture. *TCG White Paper* (2013).

[34] TRUSTED COMPUTING GROUP. TCG PC client specific TPM interface specification (TIS). *TCG White Paper* (2013).

[35] TRUSTED COMPUTING GROUP. TCG EFI platform specification for TPM family 1.1 or 1.2. *TCG White Paper* (2014).

[36] TRUSTED COMPUTING GROUP. TCG EFI protocol specification. *TCG White Paper* (2016).

[37] TRUSTED COMPUTING GROUP. TCG trusted platform module library part 1: Architecture. *TCG White Paper* (2016).

[38] TRUSTED COMPUTING GROUP. TCG ACPI specification. *TCG White Paper* (2017).

[39] TRUSTED COMPUTING GROUP. TCG PC client platform firmware profile specification. *TCG White Paper* (2017).

[40] TRUSTEDGRUB. TrustedGRUB. `https://sourceforge.net/projects/trustedgrub`.

[41] TRUSTEDGRUB2. TrustedGRUB2. `https://github.com/Rohde-Schwarz-Cybersecurity/TrustedGRUB2`.

[42] UNIFIED EXTENSIBLE FIRMWARE INTERFACE. Advanced configuration and power interface specification. *UEFI White Paper* (2017).

[43] WILSON, L. The TCG dynamic root for trusted measurement. *TCG White Paper* (2016).

[44] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking Intel Trusted Execution Technology. *Black Hat DC* (2009).

[45] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking Intel TXT via SINIT code execution hijacking. *Invisible Things Lab* (2011).

[46] WOJTCZUK, R., RUTKOWSKA, J., AND TERESHKIN, A. Another way to circumvent Intel Trusted Execution Technology. *Invisible Things Lab* (2009).

A Appendix

We attach additional pages to present detailed information. This appendix presents results of the DRTM test with Intel TXT and tboot support (Table A.1) and Intel TXT logs (Figure A.1).

PC No.	PC and mainboard model	TPM Ver.	Intel TXT support	tboot support	DRTM test	Note
1	NUC5i5MYHE	2.0	Y	Y	Y	
2	Compute Stick STK2mv64CC	2.0	Y	N	N	The system does not support tboot. It is rebooted while executing the SINIT AC module.
3	Optiplex 7040	2.0	Y	Y	Y	In case of BIOS 1.8.1 version, The system is rebooted while executing SINIT AC module. BIOS 1.4.5 version is used for the DRTM test.
4	Q170M-MK	2.0	Y	N	N	The system does not support tboot. It is rebooted while executing the SINIT AC module.
5	H170-D3HP	2.0	N	N	N	The system does not support Intel TXT.
6	Q170M-C	2.0	Y	N	N	The system does not support tboot. It is rebooted while executing the SINIT AC module.
7	X1 Carbon 4th Generation	1.2	Y	N	N	The system does not support tboot. It is rebooted while executing the SINIT AC module.
8	ThinkCentre m93p	1.2	Y	Y	Y	
9	Optiplex 7040	1.2	Y	Y	Y	For BIOS 1.8.1, The system is rebooted while executing the SINIT AC module. BIOS 1.4.5 is used for the DRTM test.
10	z840	1.2	Y	N	N	The system does not support tboot. It is rebooted while executing the SINIT AC module.
11	H170-D3HP	1.2	N	N	N	The system does not support Intel TXT.

Table A.1: Results of the DRTM test with Intel TXT and tboot support

```
Intel(r) TXT Configuration Registers:
    STS: 0x00018091
        senter_done: TRUE
        sexit_done: FALSE
        mem_config_lock: FALSE
        private_open: TRUE
        locality_1_open: TRUE
        locality_2_open: TRUE
    ESTS: 0x00
        txt_reset: FALSE
    E2STS: 0x0000000000000006
        secrets: TRUE
    ERRORCODE: 0x00000000
    DIDVID: 0x00000001b0058086
        vendor_id: 0x8086
        device_id: 0xb005
        revision_id: 0x1
    FSBIF: 0xffffffffffffffff
    QPIIF: 0x000000009d003000
    SINIT.BASE: 0xa2ef0000
    SINIT.SIZE: 196608B (0x30000)
    HEAP.BASE: 0xa2f20000
    HEAP.SIZE: 917504B (0xe0000)
    DPR: 0x00000000a3000041
        lock: TRUE
        top: 0xa3000000
        size: 4MB (4194304B)
    PUBLIC.KEY:
        2d 67 dd d7 5e f9 33 92 66 a5 6f 27 18 95 55 ae
        77 a2 b0 de 77 42 22 e5 de 24 8d be b8 e3 3d d7
************************************************************
     TXT measured launch: TRUE
     secrets flag set: TRUE
************************************************************
... omitted ...
TBOOT:   pol_hash: ce 78 8c 7b 47 b2 91 85 b8 8c 3c a0 7d f7 02 e3 a1 e4 60 03
TBOOT:   VL measurements:
TBOOT:     PCR 17 (alg count 1):
TBOOT:         alg 0004: fb b1 b9 ea b0 c9 2a c0 9c 28 14 f5 38 b5 ad 02 af e0 ee af
TBOOT:     PCR 18 (alg count 1):
TBOOT:         alg 0004: fb b1 b9 ea b0 c9 2a c0 9c 28 14 f5 38 b5 ad 02 af e0 ee af
TBOOT:     PCR 17 (alg count 1):
TBOOT:         alg 0004: 0b 55 c6 7a d3 89 03 8e 2c d3 99 17 c0 06 8f 20 68 d4 b1 50
TBOOT:     PCR 17 (alg count 1):
TBOOT:         alg 0004: 6b 8d c4 2d 1f 54 aa 6b 60 98 13 b8 f2 0e 89 2a 5d 14 5c e9
TBOOT:           Event: /* The hash of a policy control field and policy hash */
TBOOT:               PCRIndex: 17
TBOOT:                   Type: 0x501
TBOOT:                 Digest: fb b1 b9 ea b0 c9 2a c0 9c 28 14 f5 38 b5 ad 02 af e0 ee af
TBOOT:                   Data: 0 bytes
TBOOT:           Event:
TBOOT:               PCRIndex: 18
TBOOT:                   Type: 0x501
TBOOT:                 Digest: fb b1 b9 ea b0 c9 2a c0 9c 28 14 f5 38 b5 ad 02 af e0 ee af
TBOOT:                   Data: 0 bytes
TBOOT:           Event: /* The hash of a kernel file (vmlinuz) and command lines */
TBOOT:               PCRIndex: 17
TBOOT:                   Type: 0x501
TBOOT:                 Digest: 0b 55 c6 7a d3 89 03 8e 2c d3 99 17 c0 06 8f 20 68 d4 b1 50
TBOOT:                   Data: 0 bytes
TBOOT:           Event: /* The hash of a initial RAM disk file (initrd) */
TBOOT:               PCRIndex: 17
TBOOT:                   Type: 0x501
TBOOT:                 Digest: 6b 8d c4 2d 1f 54 aa 6b 60 98 13 b8 f2 0e 89 2a 5d 14 5c e9
TBOOT:                   Data: 0 bytes
... omitted ...
```

Figure A.1: List of the txt-stat logs and extended hashes in Intel NUC5i5MYHE. Details/authorities PCR mappings are used.