BootKeeper: Validating Software Integrity Properties on Boot Firmware Images

Ronny Chevalier, Stefano Cristalli, Christophe Hauser, Yan Shoshitaishvili, Ruoyu Wang, Christopher Kruegel, Giovanni Vigna, Danilo Bruschi, Andrea Lanzi

BootKeeper: Validating Software Integrity Properties on Boot Firmware Images

Ronny Chevalier
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
ronny.chevalier@centralesupelec.fr

Stefano Cristalli
Università degli Studi di Milano
stefano.cristalli@studenti.unimi.it

Christophe Hauser
University of Southern California
hauser@isi.edu

Yan Shoshitaishvili
Arizona State University
yans@asu.edu

Ruoyu Wang
Arizona State University
fishw@asu.edu

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

Danilo Bruschi
Università degli Studi di Milano
bruschi@di.unimi.it

Andrea Lanzi
Università degli Studi di Milano
andrea.lanzi@unimi.it

ABSTRACT

Boot firmware, like UEFI-compliant firmware, has been the target of numerous attacks, giving the attacker control over the entire system while being undetected. The *measured boot* mechanism of a computer platform ensures its integrity by using cryptographic measurements to detect such attacks. This is typically performed by relying on a Trusted Platform Module (TPM). Recent work, however, shows that vendors do not respect the specifications that have been devised to ensure the integrity of the firmware's loading process. As a result, attackers may bypass such measurement mechanisms and successfully load a modified firmware image while remaining unnoticed. In this paper we introduce BootKeeper, a static analysis approach verifying a set of key security properties on boot firmware images before deployment, to ensure the integrity of the measured boot process. We evaluate BootKeeper against several attacks on common boot firmware implementations and demonstrate its applicability.

CCS CONCEPTS

• **Security and privacy → Systems security**.

KEYWORDS

firmware, TPM, SCRTM, binary analysis

ACM Reference Format:
Ronny Chevalier, Stefano Cristalli, Christophe Hauser, Yan Shoshitaishvili, Ruoyu Wang, Christopher Kruegel, Giovanni Vigna, Danilo Bruschi, and Andrea Lanzi. 2019. BootKeeper: Validating Software Integrity Properties on Boot Firmware Images. In *Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19), March 25–27, 2019, Richardson, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3292006.3300026

CODASPY '19, March 25–27, 2019, Richardson, TX, USA

ACM ISBN 978-1-4503-6099-9/19/03…$15.00
https://doi.org/10.1145/3292006.3300026

1 INTRODUCTION

One of the most critical components of every computer is the boot firmware (e.g., BIOS or UEFI-compliant firmware), which is in charge of initializing and testing the various hardware components, and then transfer execution to the Operating System (OS). As a result of its early execution, the boot firmware is a highly privileged program. Any malicious alteration of its behavior can have critical consequences on the entire system. An attacker that can control the firmware can control any parts of the software and undermine the security of the entire OS. Without any protection of the integrity of the boot firmware, we cannot assure any security properties of the software executing on the system.

To guarantee the software integrity of the machine, the Trusted Computing Group (TCG), an industry coalition formed to implement trusted computing concepts across personal computers, designed a new set of hardware components, the aim of which is to solve various hardware-level trust issues. In their specification, they define the TPM, which is composed of a co-processor that offers cryptographic functions (e.g., SHA-1, RSA, random number generator, or HMAC) and a tamper-resistant non-volatile memory used for storing cryptographic keys [33]. The TPM along with other software components together form a root of trust, which is leveraged as part of several security mechanisms, including the *measured boot* process. With measured boot, platforms with a TPM can be configured to measure every component of the boot process, including the firmware, boot loader, and kernel. Such measurement process is also called the Static Root of Trust for Measurement (SRTM).

The core of trust of the entire process is established based on the integrity of the first piece of code inside the boot firmware which is doing the first measurements, also called the Static Core Root of Trust for Measurement (SCRTM) [41]. In the event where a

malicious modification of the SCRTM successfully hides from the self-measurement technique, the whole chain of trust and, consequently, the integrity of the entire system may be broken. In this direction, Butterworth et al. [4] described different examples of attacks against the SCRTM component. In particular, the authors show how a novel "tick" malware, a 51-byte patch to the SCRTM, can replay a forged measurement to the TPM, falsely indicating that the BIOS is genuine. These attacks take advantage of the fact that some vendors do not measure the SCRTM code, thus allowing an attacker to modify it and to forge measurements without being detected.

Recent platforms incorporate an immutable, hardware protected SCRTM [11, 23]. Intel Boot Guard and HP Sure Start are immutable SCRTMs which measure and verify, at boot time, the integrity of the BIOS image before executing it. Such technologies are not directly vulnerable to the aforementioned attacks, since their code cannot be modified by an attacker.[1] Both technologies, however, are only available in recent Intel and HP platforms, *leaving previous hardware implementations, or devices of other vendors, vulnerable against forged measurements*. In such implementations, since the SCRTM is not hardware protected, *it is usually attached to the firmware image itself during the firmware update process*. Even when the firmware image is signed, attacks may compromise this process [14], and consequently allow an attacker to modify both the firmware code and the SCRTM.

In order to solve these challenges in validating the SCRTM code, we design a self-contained approach based on static analysis at the binary level, which is able, starting from a boot firmware image, to validate the correctness of the measurement process. Our system verifies software properties on the SCRTM code embedded in firmware images, including: (1) the completeness of firmware code measurements in terms of fingerprinted memory regions, (2) the correctness of cryptographic functions implemented[2] inside the SCRTM (e.g., SHA-1), and (3) the correctness of the SCRTM's control flow. More in detail, the first property ensures that the code of the entire firmware is measured correctly by the SCRTM, i.e., that none of the instructions to be executed at runtime will be missed by the measurement process. The second property ensures that the implementation of cryptographic functions inside the SCRTM is correct. The third property validates the correctness of the measurement operations performed by the SCRTM in terms of execution order. It also guarantees the atomicity of operations occurring between memory fingerprinting and write operations performed on the TPM component (i.e., ensuring that what is measured is what is written to the TPM). Altogether, this set of properties can prevent attacks aiming to elude the measurement process, and it guarantees that the integrity of a firmware image is properly verified during the *measured boot* process.

We implement a prototype of our system, dubbed BootKeeper, based on the angr program analysis framework [6, 29]. We evaluate our system on different open source boot firmware images, and we implement different attacks against the firmware to show the efficacy of our approach. Our paper makes the following contributions:

- We devise a set of software properties that can be used for validating the measurement process and mitigate firmware attacks aimed to subvert the entire system.
- We design and implement BootKeeper, a binary analysis approach to detect and prevent measurement boot firmware attacks in different attack scenarios.
- We perform experimental evaluation against different attacks and several boot firmware implementations to demonstrate the effectiveness of our approach.

2 BACKGROUND

In this section, we introduce the background technology needed to understand our approach. We first describe the principles of the TPM, then we describe the UEFI specifications and some of the software/hardware components involved in the boot measurement process.

2.1 Trusted Platform Module (TPM)

The TPM specification defines a co-processor offering cryptographic features (e.g., SHA-1, RSA, random number generator, or HMAC), and tamper-resistant storage for cryptographic keys [33].

The TPM provides a minimum of 16 Platform Configuration Register (PCR) which are 160-bit wide registers used to store the measurements done by the SCRTM (usually SHA-1 hashes). The design of these registers allows an unlimited amount of measurements and prevents an attacker from overwriting them with arbitrary values. In order to do this, the only possible operation is *extend*:

$$PCR_i = H(PCR_i \mid\mid m)$$

Where PCR_i is the ith PCR register, H is the hash function and m the new measurement. The TPM *concatenates* each new measurement sent with the previous value of the register, then it hashes the result, which becomes the new value of the register. This mechanism is crucial to establishing a chain of trust, since the only way to obtain a given measurement value from a PCR is to reproduce the same series of measurements in the same order. The measured boot process relies on this mechanism to guarantee that a given software platform is valid and has not been tampered with.

The TPM also relies on these measurements to provide specific features (e.g., secure storage or remote attestation). For instance, with the sealing operation, the TPM offers the ability to encrypt data, with a key only known to the TPM, and it binds the decryption to the PCRs values. During the decryption (i.e., unsealing), the TPM only decrypts the data if the PCRs values match the ones used during the encryption. One common use case for the sealing operation is to store the disk encryption key. It ensures that the disk is decrypted only if the platform has booted with the expected hardware and software, and if no attacker tampered with the boot process (e.g., an evil maid attack).

2.2 Static Core Root of Trust for Measurement

The SCRTM is responsible for the first measurement sent to the TPM in the PCR0 register and it is considered trusted by default on the system. Since the default values of the PCRs are known

[1] Nonetheless, Intel Boot Guard has been shown to be vulnerable to some attacks as well [19].

[2] Vendors typically implement the cryptographic functionalities used as part of the measurement process in software.

(either `0x00...0x00` or `0xFF...0xFF`), the entire trust in the SRTM relies on the SCRTM. If it is possible for an attacker to modify the SCRTM, then it is also possible for the attacker to forge the first measurement, and the next one, etc.

Therefore, the TCG PC client specific implementation [31] states that the SCRTM must be an immutable portion of the firmware. The specification defines immutability such that only an approved agent and method can modify the SCRTM. Most firmware fulfill this requirement by using signed updates, because the SCRTM can only be modified if the update is coming from the vendor. Recent firmware fulfill the requirement using an immutable hardware protected SCRTM [11, 23]. Unfortunately, legacy platforms do not provide signed updates, or do not require them. Furthermore, Kallenberg et al. [12], and Wojtczuk and Tereshkin [37] have successfully exploited multiple vulnerabilities in the implementation of signed firmware updates by vendors, allowing an attacker to update the firmware with a malicious one. Finally, if the private key of the vendor is compromised, the platform is vulnerable.

2.3 Unified Extensible Firmware Interface

In 2005, 11 industry leading technology companies created the Unified Extensible Firmware Interface (UEFI) forum which defines specifications for interfaces [36] used by the OS to communicate with the firmware [42] and Platform Initialization (PI) specifications [35] which define the required interfaces for the components in the firmware, allowing multiple providers to create different parts. UEFI specifications are about the interfaces, while UEFI PI specifications are about building UEFI-compliant firmware. Manufacturers are now providing, as boot firmware replacing the Basic Input/Output System (BIOS), UEFI-compliant firmware images. UEFI and PI specifications define seven phases, as illustrated in Figure 1, which describe the boot process of a platform:

(1) The Security phase (SEC) is the initial code running, it switches from real mode to protected mode, initializes the memory space to run stack-based C code, and discovers, verifies and executes the next phase.
(2) The Pre-EFI Initialization phase (PEI) initializes permanent memory, handles the different states of the system (e.g., recovery after suspending), executes the next phase.
(3) The Driver Execution Environment phase (DXE) discovers and executes drivers which initialize platform components.
(4) The Boot Device Selection phase (BDS) chooses the boot loader to execute.
(5) The Transient System Load phase (TLS) handles special applications or executes the boot loader from the OS.
(6) The Runtime phase (RT) is when the OS executing, but there are still runtime services of firmware available to communicate with the OS.
(7) The After Life phase (AL) takes control back over the OS when it has shutdown.

The TCG specifies requirements for the measurement of UEFI-compliant firmware in TPM PCRs [34]. The SCRTM in UEFI-compliant firmware is generally formed by the SEC and PEI phases [42], although no strict requirements about its location are specified in the TCG specification as it can also be the entire BIOS [31]. Moreover, in recent platforms, the SCRTM is a hardware-protected component,

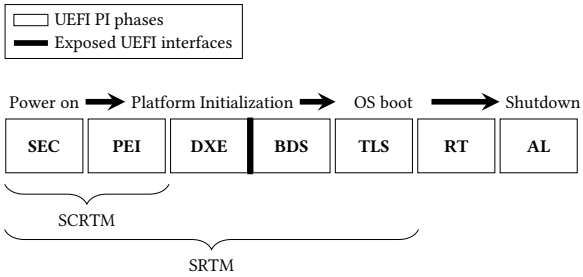

Figure 1: UEFI PI phases with the ones corresponding to the SCRTM and SRTM

outside of the BIOS, that performs measurements on the BIOS before its execution [11, 23]. In our work, however, we only consider a non-hardware protected SCRTM.

3 APPROACH OVERVIEW

BootKeeper is an offline analysis approach leveraging state-of-the-art binary analysis techniques to evaluate the validity and correctness of boot firmware images.

3.1 Threat Model

Our approach targets systems that implement measured boot protection mechanisms by using Trusted Computing technology. From a high-level perspective, an attacker may attempt to tamper with a system's firmware in two ways:

- By exploiting weaknesses of the SCRTM, e.g., a buggy or incorrect SCRTM may only perform partial measurements. In this case, an attacker may be able to inject code within the vendor's firmware image in the non-measured portions of the memory.
- By directly injecting a malicious SCRTM, the attacker may spoof the vendor's golden measurement values to pretend that the legitimate firmware is in place, while executing a malicious version of it.

By successfully circumventing the measurement process, an attacker may not only compromise the integrity of the system while tricking the attestation procedure into reporting a legitimate software platform, but it may also leak secret information from the TPM such as cryptographic keys used for full disk encryption (as used by Microsoft Windows's BitLocker, among other software products relying on this mechanism).

In the remainder of this paper, we assume the following attacker model:

(1) The attacker does not have physical access to the system.
(2) The attacker does not have any form of privileged access to the system (neither local or remote, i.e., no control over the OS).
(3) The system itself has not been infected prior to the attack and is non-malicious (i.e., the SCRTM is invoked from a non-malicious environment).
(4) The SCRTM's code does not implement user input mechanisms (but such mechanisms may be implemented as part of later stages of the EFI boot process).

(5) The attacker has the ability and sufficient knowledge about the target platform to craft malicious firmware images, i.e., access to the vendors' official firmware images, and knowledge about the platform's golden values (i.e., correct measurement values), or the ability to obtain those by reverse engineering.
(6) The attacker may spread malicious images online (e.g., by compromising the vendor's website or through third party websites such as user forums).
(7) Optionally, the attacker may remotely interfere with the automated firmware update process that comes with some systems by compromising the download site, or by mounting a man in the middle attack when applicable (e.g., if this process does not check the validity of SSL certificates), to trick the remote system into applying a firmware update using an attacker-chosen malicious image.

3.1.1 Signature Verification. In order to successfully install a malicious firmware image in the target system, an attacker needs to bypass eventual signature verification mechanisms in place. While this aspect is outside of the scope of this paper, we briefly demonstrate the practicality of this assumption as follows.

A good practice when releasing software updates is to rely on cryptographic signatures in order to guarantee the integrity of the new software image before or during the installation process. Unfortunately, this process is often imperfect, as several vendors do not implement proper signature verification mechanisms, leaving gaps for an attacker to use a forged firmware image. In other situations, attackers may use a stolen certificate [14] to sign malicious firmware images, or may remotely exploit a vulnerability in the firmware update routine, to bypass the signature checks. In the remainder of this paper, we assume that the signature verification mechanism is either absent, or vulnerable.

3.2 Analysis Steps

Our analysis approach relies on the verification of a set of key properties, which we describe in more detail the remainder of this section.

3.2.1 Code Integrity Properties (CIP). The SCRTM code is always implemented with two main fundamental operations: (1) an operation of fingerprinting, which scans the code regions in memory (e.g., using SHA-1) and (2) a TPM write operation, storing the computed fingerprints in the TPM. We define these two operations as the building blocks of any SCRTM measurement process performed on the platform.

In order to elude the measurement process, an attacker may act at two different levels.

- Firstly, as illustrated in Figure 2, the attacker may modify the fingerprint function (e.g., code or parameters) to generate spoofed measurement values which correspond to valid golden values (i.e., values corresponding to correct measurements on the vendor's firmware) even though the original firmware code is modified.
- Secondly, the attacker may modify the results of the fingerprint function just before these are written to the TPM.

The *tick* and the *flea* attacks, described by Butterworth et al. [4], are concrete examples of such attacks.

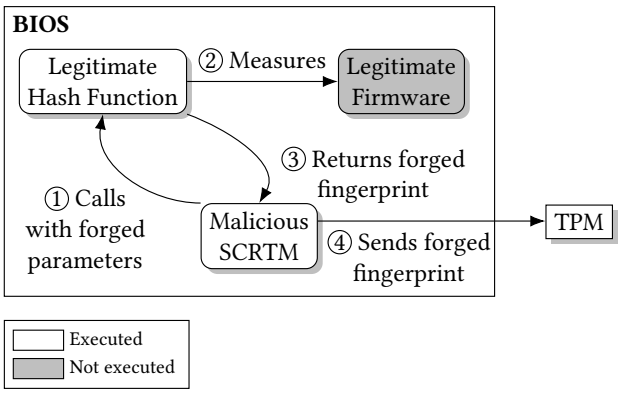

Figure 2: Example of a measurement-spoofing attack where an attacker sends a legitimate fingerprint of non-executed firmware.

In order to prevent these attacks, our system verifies the three following properties:

(1) *The authenticity of cryptographic hash functions.* Regardless of any potential implementation variants, our system must be able to verify the authenticity of the code used as part of the fingerprinting measurement process. BootKeeper leverages binary analysis techniques to verify that the correct hash function is indeed used as part of the firmware's fingerprinting code.
(2) *The atomicity of the measurement process.* A correct SCRTM implementation should also guarantee the atomicity of its measurement process, i.e., that the fingerprinting and TPM write operations are invoked sequentially in the correct order, and that the integrity of the measurement values if preserved between these two operations. In order to verify this property, BootKeeper constructs a Control-Flow Graph (CFG) of the SCRTM, and detects eventual operations modifying the measurement results before those are written to the TPM.[3]

These two properties ensure the correctness of the SCRTM's code measurements process. In addition to these, BootKeeper also ensures that the firmware under analysis does not present risks of certain classes of runtime attacks, as described below.

3.2.2 Code Execution Integrity Property (CEIP). Even if properties (1) and (2) are guaranteed, an attacker may attempt to alter the control-flow of the SCRTM by redirecting the execution to malicious code hidden in the binary firmware image. Fortunately, UEFI firmware runs in an execution environment protected by Data Execution Prevention (DEP). In other words, an attacker cannot trivially execute code injected in arbitrary sections of the binary image.

[3]In practice, such operations may either correspond to malicious code attempting to forge measurement values, or to benign buggy code reporting erroneous measurements.

In the execution context of the SCRTM, an attacker does not have the ability to inject code at runtime since the SCRTM's code does not implement user input mechanisms and the SCRTM code is invoked from a non-malicious environment (see rules 3 and 4 of our attacker model in subsection 3.1). However, it remains possible for the attackers to hide code within the binary firmware image, and to attempt to trigger its execution at runtime.

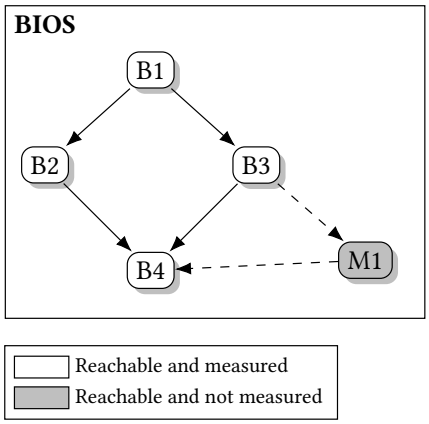

Figure 3: Example of an incomplete measurement of the firmware where reachable (malicious) code is not measured.

BootKeeper addresses this family of attacks as well by relying on an additional property:

(3) *Completeness of the measurements.* The SCRTM must guarantee the completeness of the measurements of the firmware code memory regions. More in detail, every memory region belonging to the CFG of the SCRTM must be measured and reported to the TPM component. By doing so, attempts to hide malicious code within non-measured memory regions is detected, as represented in Figure 3, showing benign (B) and malicious (M) basic blocks forming a CFG.

We emphasize that BootKeeper does not rely on a-priori knowledge of the legitimate CFG of firmware images. Instead, the goal of BootKeeper is to ensure that *all reachable code* will be *correctly* measured and reported to the TPM at runtime. As such, the detection of malicious code is a two-stage process: a static part (BootKeeper) which ensures that the verification code is correctly implemented, and a dynamic part (the measured boot process), which relies on those mechanisms.

Recall the attacker model presented in subsection 3.1: (3) the SCRTM executes in a non-malicious environment (i.e., the initial state of the system is non-malicious, only firmware images are) and (4) it does not implement I/O mechanisms. As a consequence, dynamic code injection attacks are specifically excluded.

4 SYSTEM DESIGN AND IMPLEMENTATION

In this section, we present the algorithmic properties of our analysis components.

4.1 Code Integrity Properties (CIP) Validation

The process of validating the Code Integrity Properties presented in subsubsection 3.2.1 relies on four analysis steps: (1) Detecting TPM write operations inside the firmware code, (2) Detecting hash functions inside the firmware code, (3) Validating the authenticity of hash functions, (4) Validating the atomicity of the measurement process. We now describe how the system achieves each step of the analysis.

4.1.1 TPM Write Operation Detection. The first step of our analysis is to identify the TPM write operations inside the firmware code. Such operations can be identified by searching for standard API prototypes. According to the TPM specification version 1.2 [31, 32], such functions must use a fixed address (`0xFED40000` is the default), along with an offset used for distinguishing among different operations (e.g., read and write) on the TPM registers.

Unfortunately, we cannot predict how the compiler organizes the instructions or how the code of the firmware performs the write accesses. Developers tend to create abstraction layers (e.g., to avoid redundancy or to have a modular code), and may use different optimization flags for the compilation of the firmware. As a result of this, TPM read and write operations do not straightforwardly appear as an offset from the specified constant address value in the binary executable version of the firmware. Therefore, in order to tackle this problem and find a state of the program where a write access to the TPM known address happens, we leverage symbolic execution, starting at the entry point of the firmware, and record every instruction writing at the specified address (`0xFED40024` in our case). This step of symbolic execution allows BootKeeper to resolve computed addresses for which it would be extremely difficult to reason about in a purely static setup. We leverage the angr [6] platform to perform symbolic execution. During symbolic execution, the system tracks the state of registers and memory throughout program execution along with the constraints on those variables. Whenever a conditional branch is reached, the execution follows both paths while applying constraints to the program state to reflect whether the condition evaluates to True or False. At the end of this analysis step, BootKeeper has obtained a list of addresses in the firmware corresponding to the TPM write operations, if any such operation is present. If the system does not find any TPM write operation, it flags the firmware image as non-valid.

4.1.2 Detecting Hash Functions. The second step of our analysis is to identify hash functions among the functions used by the firmware image to elaborate the measurement values. We apply the following algorithm. The analyzer starts from the TPM write operations and works iteratively on the instructions flow in reverse order. To this end, we leverage a static backward approach [15]. where for each identified TPM write operation, our analyzer computes a backward slice starting from the sensitive data. Sensitive data are the parameters of the TPM write operations, in our case the measurements value of the hash function stored in a particular memory region. The aim of this backward slicing analysis step is to identify modifications on sensitive data, and to find all the data sources from which the modified data is derived.

The backward slice technique allows us to focus our analysis only on the instructions that lead to a single TPM write access,

which is important for two reasons. First, for performance reasons, focusing on a subset of the program greatly improves the time needed to perform further analyses. Second, and more importantly, it ensures that the TPM write access is connected to a measurement computed earlier in the execution (i.e., not inserted in an ad-hoc manner). Hence, given a set of instructions related to the TPM write access, BootKeeper detects if an actual measurement is present.

The output of this analysis step is a set of traces corresponding to instructions correlated with the TPM write operations parameters. Such traces could also be leveraged to locate the hash function code as well. However, it is worth noting that the backward slicing algorithm returns an over-approximation of the instructions leading to the program point where the TPM measurements are sent, and therefore, it might include unrelated instructions. For example, functions which simply move computed hash values from one memory structure to another may also be whitelisted by this analysis. For this reason, we cannot simply rely on this technique to accurately locate the hash functions themselves, and we leverage a different approach to precisely locate those, as presented in the following paragraph.

4.1.3 Validating the Authenticity of Hash Functions. After BootKeeper has identified the set of instructions related to a particular TPM operation, it extracts the corresponding blocks and attempts to recognize one of the possible valid hash function (e.g., SHA-1). In order to automatically identify cryptographic functions within binary code, we leverage the approach presented by Lestringant et al. [17] which relies on Data Flow Graph (DFG) isomorphism. From a high-level perspective, this approach compares the code structure of known functions with the code structure of unknown functions, to determine if these implement the same algorithm. This approach first employs a normalization step (based on a code rewrite mechanism), which is designed to increase the detection capability by erasing the peculiarities of each instance of an algorithm. Then, by relying on a sub-graph isomorphism algorithm, the normalized DFG is compared to that of known reference functions. We chose this approach to recognize functions of well-known libraries which are used in real-world boot firmware images (including Crypto++ and OpenSSL [21]).

BootKeeper leverages this technique starting from the instruction traces highlighted in the previous analysis step. From there, it creates a DFG for each corresponding function of the firmware image involved in the trace, and attempts to match the signature of a standard hash function, such as the SHA-1 implementation of libOpenSSL. The output of this analysis step is the set of basic blocks belonging to the identified hash function. If no known hash function is found, then *BootKeeper flags the firmware image as non-valid.*

4.1.4 Validating the Atomicity Property. The final step involved in validating code integrity is to ensure that no modification of the computed measurements occurs before those are written to the TPM. From the instruction traces obtained during the backward slicing step (subsubsection 4.1.2), BootKeeper executes the corresponding code paths symbolically, from the hash function's return instruction to the TPM write operation, and rules out the presence of instructions modifying the computed hash value. In order to detect such instructions, BootKeeper performs a last step of forward reaching definition analysis on each identified code path to ensure that the value stored in the TPM indeed corresponds to the return value of the correct hash function. If any instruction on a given path modifies the measurement value, the atomicity property is violated. In this case, the firmware image is reported as invalid, and BootKeeper reports the faulty instruction and program path.

4.2 Code Execution Integrity Validation

Recall that, in addition to verifying the correctness of the measurement process, BootKeeper also evaluates the risks of runtime attacks through a Code Execution Integrity Property (CEIP) described earlier in Section 3.2.2, which consists in the completeness of measurements.

Conceptually, if the measurement process is incomplete, i.e., leaving out portions of the code section, it becomes possible for an attacker to "hide" malicious code in the non-fingerprinted areas, hence the importance of this property.

4.2.1 Completeness of the Measurements. BootKeeper ensures that every function in the CFG of the SCRTM is measured. The acute reader may wonder what becomes of basic blocks of code which is deemed non-reachable by the control-flow recovery analysis: this point is discussed in section 6. The CFG is computed by a recursive algorithm which disassembles and analyzes each basic block, identifies its possible exits (i.e., successors) and adds them to the graph. It repeats this analysis recursively until no new exits are identified. CFG recovery has one fundamental challenge: indirect jumps. Indirect jumps occur when the code transfers control flow to a target represented by a value in a register or a memory location (e.g., `jmp %eax`). Indirect jumps can be categorized in two main classes: (1) Computed, an example could be an application that uses values in a register or memory to determine an index into a jump table stored in memory; (2) Runtime binding, i.e., function pointers which jump targets are determined at runtime. BootKeeper leverages state-of-the-art analysis techniques for control-flow recovery, as available in the `angr` framework [6]. This process leverages a combination of forced execution, backwards slicing, and symbolic execution to recover, to the extent possible, all jump targets of each indirect jump. While it may not always be possible to recover all jump targets in complex software projects, it is practical in the context of a boot firmware's SCRTM due to the minimal aspect of such a code base. A more detailed discussion of this point and the practical limitations that it may involve is provided in section 6.

Once BootKeeper has obtained a CFG of the SCRTM, it proceeds to verify the coverage of the SCRTM in terms of code fingerprinting. In order to do so, BootKeeper analyses all input values passed to the fingerprinting functions (i.e., hash function identified in subsubsection 4.1.3) to determine the address and size of the memory areas used to compute the measurements. At this point, BootKeeper has statically identified the addresses and sizes of the memory regions that will be fingerprinted by the SCRTM at runtime. Its next step is to ensure that all reachable code in the SCRTM's CFG does indeed fall within the measured memory regions. In order to do so, BootKeeper verifies that the address of each basic block belonging to the CFG falls within that range. If it is not the case, it means that a part of the firmware's code will not be measured correctly, and *flags the firmware image as non-valid.*

By ensuring that all the reachable code in the CFG is indeed measured by the SCRTM, BootKeeper prevents firmware modification attacks. where malicious code is inserted within the executable paths of the firmware's code.

5 EXPERIMENTAL EVALUATION

This section presents our experimental results. Our evaluation metrics cover multiple attack vectors representing a large span of the attack surface against the SCRTM. These include the ability of our prototype to identify and recover the location of TPM write instructions, the effectiveness of our approach to detect the presence of forged TPM measurements, and to detect possible hidden code areas which are left unmeasured. In addition to this, we also evaluate the robustness of our analysis against various compiler optimization settings.

5.1 Experimental Setup

We evaluate BootKeeper on two real-world implementations of boot firmware used in the industry, as well as a custom-crafted malicious firmware implementing state-of-the-art attacks.

(1) SeaBIOS [20], a native x86 BIOS implementation with TPM support. It also supports standard BIOS features and calling interfaces that are implemented by a typical proprietary x86 BIOS implementations. This project is meant to provide an improved and more easily extendable implementation in comparison to the proprietary counterparts which come as stock firmware on standard x86 hardware, and can be deployed as replacement firmware on a variety of motherboards.
(2) EDK II [30], a modern, cross-platform firmware platform supported by a number of real Intel and ARM hardware platforms. It is a component of Intel's TianoCore (Intel's reference implementation of UEFI). Major vendors (e.g., Apple, ARM, or HP) contribute to its development, and it serves as a basis for a number of proprietary UEFI-compliant firmware implementations.
(3) A custom-crafted malicious firmware image which we implemented to reproduce multiple variants of the state-of-the-art attacks introduced by Butterworth et al. [4].

For validating cryptographic functions, we are using firmware (EDK II) that implements the SHA-1 function of the OpenSSL [21] libraries and `gcc` as a compiler.

5.2 TPM Write Access Detection

We evaluated BootKeeper against SeaBIOS, EDK II, and our custom firmware. Our hypothesis is that complex firmware implementations, like SeaBIOS and EDK II, use abstraction layers which tend to obfuscate TPM write instructions, i.e., these do not exhibit such instructions in the form of a fixed address corresponding to the specification (e.g., `mov 0xfed40024, al`). Additionally, these abstraction layers may invoke several functions to initialize data structures, perform hardware tests, use loop structures or handle different types of errors, thus creating many paths, potentially leading our analysis to path explosion during its symbolic execution phase. In order to test the resiliency of our approach against the aforementioned intricacies, but also against compiler optimizations, we evaluated

Table 1: Detection of the function writing the measurements

Optimization flags	Custom firmware	EDK II	SeaBIOS
`-O0`	●	○	✗
`-O1`	●	○	●
`-O2`	●	●	●
`-O3`	●	●	●
`-Os`	●	●	●

● : detected ○ : not detected ✗ : error

BootKeeper in the context of multiple compiler optimization settings. For each firmware implementation, we produced 5 variants using different optimization flags: `-O1` to `-O3` which optimize for speed, `-Os` which optimizes for size, and `-O0` for no optimization. This process generates 15 firmware variants. We present the results in Table 1.

BootKeeper can successfully detect the TPM write access in 80% of all cases (12 out of 15). During this evaluation phase, we set an analysis timeout threshold of 10 minutes.

It is worth noting that BootKeeper found all the write access instructions within all the tested variants, with the notable exception of *unoptimized* EDK II images. We explain this observation by the fact that a lesser optimization level means more instructions to execute, more loops and therefore more likelihood of path explosion during symbolic analysis. While extending the analysis timeout threshold would be a viable option, we argue that vendors typically compile their code with size optimizations (using `-Os`) when releasing firmware for production use, to reduce memory footprint, in which case our approach succeeds under the 10 minutes threshold.

In the case of SeaBIOS, the analysis results are missing in the situation where no optimization is used (`-O0`). Unfortunately, a bug in SeaBIOS prevented the code from compiling with this optimization level at the time of our evaluation, hence we could not test it. Nonetheless, BootKeeper can successfully detect TPM write operations in SeaBIOS in all other optimization settings.

In summary, these experiments demonstrate that our approach can correctly detect the TPM write operations even in intricate cases, in real-world firmware implementations, from legacy BIOS to recent UEFI-compliant firmware.

5.3 Forged TPM Measurements Detection

In order to evaluate the effectiveness of our approach against state-of-the-art attacks, we reproduced multiple variants of the *tick* attack, described by Butterworth et al. [4]. Each developed attack variant attempts to forge the measurements sent to the TPM by relying on several techniques: (1) attacking the SHA-1 function, (2) leaving out non-measured code, (3) modifying the TPM parameters. We now describe how BootKeeper performs against these attacks.

Hash Function Validation. A first attack attempts to subvert the hash function results. In order to test this attack, we create two firmware versions: one without a SHA-1 function, and another with a modified version of SHA-1 which returns a tampered hash value. In either case, BootKeeper reports no match when generating

signatures for the function defined inside the instruction traces (compared against a generated database of common cryptographic implementations from Crypto++ and libOpenSSL) and reports a violation of the property "valid hash function".

Completeness of the Measurement. We evaluate this property by implementing an attack that can modify the firmware's code by including new code to the CFG of the program. More specifically, we add new malicious functions to a non-measured code area that is invoked before returning from the SCRTM code. Since this code is not included in any measurement performed by the hash function, BootKeeper can detect it by ensuring that every part of the CFG is correctly measured (as described previously in subsubsection 4.2.1).

Atomicity Property. In order to validate the atomicity property, we define a scenario where an attacker properly measures the firmware with the correct SHA-1 function, but then tampers with the results by overwriting the measurements with other values before sending those to the TPM, thus violating the atomicity property of the measurement operations. During the analysis, BootKeeper can fetch the parameters of the hash function, and more importantly the address of the pointer where the hash is stored. Then, by performing the symbolic execution step described in subsubsection 4.1.4, it can detect the malicious instruction responsible for overwriting the measurements.

In order to fully reflect practical analysis challenges, we also consider the case where an attacker attempts to trick the analysis by incorporating a real measurement using the correct hash function, and later writing the measured value somewhere else before sending a forged value to the TPM. We reproduce this attack scenario to evaluate the resilience of our analysis step presented in Section 4.1.4 against the presence of false positives involved by the backward slicing algorithm. In our evaluation, false positives indeed occur, but these are effectively filtered by the following step of forward reaching definition analysis (as expected). In this situation again, BootKeeper can detect the attack, and report the malicious instruction responsible for overwriting the measurements.

5.4 Performance

While performance is not critical in the context of an offline analysis setting, we demonstrate the practicality of BootKeeper in terms of analysis time and memory usage. In the following, we use a valid firmware for evaluation, and measure the required amount of time BootKeeper takes to proceed, while monitoring the peak of RAM usage. In this experiment, we use a single-thread on an Intel Xeon E312 with 64GB of RAM. The results presented in Table 2 were obtained by running this experiment 100 times, and represent the mean, minimum, maximum values and the standard deviation of both the runtime (in milliseconds) and the RAM usage (in megabytes). Executing the entire analysis (i.e., validating the firmware) takes on average 1 minute and 48 seconds. The memory usage peaks at 522 megabytes.

Table 2: Time and RAM usage for firmware analysis

Type		Minimum	Maximum	Mean	Standard deviation
Time	(ms)	103 950	108 760	105 704	849
RAM	(MB)	522.3	522.8	522.7	0.1

6 DISCUSSION

BootKeeper is a purely static approach relying on advanced binary program techniques to analyze firmware images. As any static approach, it comes with some challenges. In this section, we describe in more detail the nature of these challenges, and how BootKeeper addresses those. Finally, we point to some practical limits of our approach and propose alternative research directions of interest.

6.1 Theoretical Limitations

In order to detect violations of the properties introduced in Section 3, BootKeeper relies on a combination of state-of-the-art static analysis techniques, which together provide the basis for implementing the verification algorithms presented in Section 4. These techniques, however, are subject to theoretical limits, which prevents our approach from reasoning about certain classes of properties in all possible situations.

BootKeeper relies in particular on:

- Static CFG recovery to determine the set of possible execution paths of a firmware image. The CFG obtained from binary analysis is neither sound nor complete (the general problem of deciding if an arbitrary path in a program is executable is undecidable [22]).
- Symbolic execution and constraint solving, to reason about the possible concrete values of memory and registers at arbitrary points of the execution. Symbolic execution is subject to the well-known state explosion problem due to its exponential growing nature, and the general problem of constraint (SMT) solving is NP complete.
- Data-flow analysis, to isolate program paths involving measurement values, to generate program slices in order to isolate the instructions affecting these values, and more generally, to detect faulty operations. Reasoning about data-flow at the binary level requires accurate models of data structure recovery, and is subject to the pointer aliasing problem [22].

Our approach inherits from these general limitations. We discuss the practical impact of these theoretical limitations in subsection 6.2 below.

6.2 Practical Impact

The following is a discussion of the practical impact of the aforementioned limitations in our approach.

6.2.1 False Positives. In the case where the CFG is too conservative and includes an overestimate of possible code paths in the graph, BootKeeper will accordingly operate conservatively during the verification of property 3, i.e., the completeness of measurements. While this may lead to false positives in certain circumstances, we stress that the CFG we obtain from a binary has a basic-block level granularity. In comparison, vendors typically scan entire memory regions corresponding to sections from the firmware binary. *Why not just measure entire sections then?* Our approach is more fine-grained, and aims to ensure that the vendor conforms to at least a

defined minimal code coverage corresponding to (an estimation) of the possible execution paths.

In the case of an incomplete CFG between the return value of the hash function used in the measurements and the subsequent TPM write operation, BootKeeper will not be able to compute a backward slice and therefore will not be able to validate property 2, i.e., the atomicity of the measurement process. In this case, *it will flag the image as malicious*, thus generating a false positive.

6.2.2 False Negatives. Similarly, the CFG may miss edges in the graph corresponding, such as indirect jumps caused by complex instances of runtime binding which cannot be resolved even with symbolic execution. This may happen, among other possible cases, when external information (i.e., external to the program) is required to compute the jump target. The presence of such obfuscated or evasive code in early stages of firmware execution is, by itself, an excellent indicator of maliciousness which our approach could be extended with.

When such constructs are benign and part of the official vendor's firmware, an attacker may succeed in hiding a payload P if: 1) the SCRTM omits one portion of executable memory M_1 during the measurements, 2) BootKeeper is missing a part of the CFG corresponding to a set of basic blocks mapped in memory during runtime as M_2 (in practice, M_2 may not be contiguous), and the following holds true:

$$M1 \cap M2 \neq \emptyset \wedge |M_1 \cap M_2| \geq sizeof(P)$$

Without ruling out the possibility of strong attacker specifically challenging state-of-the-art static analysis techniques, we estimate that BootKeeper significantly raises the bar for an attacker to circumvent the measurement process, and we consider our approach practical in the context of a large span of possible attacks.

The presented limitations are intrinsic to any static approach, and cannot trivially be addressed without additional knowledge of the runtime environment. In the next section, we discuss possible alternatives to overcome these limitations.

6.3 Alternative Solutions

In order to analyze the program paths involved in firmware during execution in a dynamic setting, an emulation of all the hardware components involved during the platform initialization process would be necessary. Implementing such a system is a cumbersome engineering task, especially if numerous targets need to be supported. An alternative approach is to directly instrument the hardware to dump the state of registers and memory as the firmware executes, the knowledge of which would ease offline analysis. Similar to this is the Avatar approach [39] which selectively switches between different execution models, in a setup which is backed by the physical hardware.

While relevant to this discussion, neither of these approaches fits within the scope of this paper. In comparison, BootKeeper requires no hardware nor custom hardware models.

6.4 Obfuscation

Static binary program analysis techniques are vulnerable to the presence of obfuscation, and it is possible that a malicious firmware author could attempt to attack BootKeeper in this manner. For instance, Sharif et al. [27] obfuscate conditional code by using the result of a hash function as a condition replacement. Since cryptographic hash functions have the pre-image resistance property, it is impossible for constraints solvers to solve all the constraints generated by the operations of the hash function.

These weaknesses are inherent to any tool relying on static program analysis [5, 28, 38].

In the context of boot firmware, the problem related to obfuscation is two-fold. First, genuine vendors could use obfuscation techniques to protect their code against reverse engineering. Secondly, by relying on obfuscation techniques, an attacker could attempt to defend against automated program analysis. While the former would affect BootKeeper, the latter may be used as an indicator malice.

7 RELATED WORK

To the best of our knowledge, John Heasman developed the first public BIOS rootkit by modifying Advanced Configuration and Power Interface (ACPI) tables stored in the BIOS [9], and he also showed how to make a persistent rootkit by re-flashing the expansion Read-Only Memory (ROM) of a Peripheral Component Interconnect (PCI) device [10]. Other attacks have been performed since then, Anibal Sacco and Alfredo Ortega discussed how to inject malicious code in Phoenix Award BIOS [24] and Jonathan Brossard showed the practicability of infecting different kinds of firmware [2]. In addition to papers and proof of concepts of attacks, some malware is also taking advantage of the lack of security of the boot firmware. For example, the Chernobyl virus [8], which appeared in 1999, tried to overwrite the BIOS to make it unbootable. In 2011, the malware called Mebromi [18] re-flashed the BIOS of its victims to later write a malicious Master Boot Record (MBR) which infected the OS even when it was re-installed from scratch.

All these attacks can be detected if the vendor is trustworthy, a TPM device is present and used correctly. Several misconfiguration and design issues, however, show that the TPM can be attacked as well. In this direction, Butterworth et al. [4] demonstrated a replay-attack that forges the measurement sent to the TPM to fake an uncorrupted BIOS in case of non-respect of the specifications and recommendations. Bruschi et al. [3] also showed a replay-attack in an authorization protocol of the TPM. Sadeghi et al. [25] and Butterworth et al. [4] revealed that some TPM implementations do not meet the TCG specifications which may have critical security implications. Kauer [13] also demonstrated a TPM reset attack which allows an attacker to forge the PCR values.

Several approaches have been proposed to improve the TPM technology and the boot firmware integrity techniques. For example, Bernhard Kauer proposed a counter measure [13] to the reset attack on the TPM by using a Dynamic Root of Trust for Measurement (DRTM). In the direction of firmware security, dynamic analysis using symbolic execution has been extensively used to find vulnerabilities in firmware [1, 7, 16, 28, 40]. More related to our work, Bazhaniuk et al. [1] used an approach to detect vulnerabilities in boot firmware. Our work is orthogonal to such approach and focuses on boot firmware phases where vulnerabilities are not

detected or fixed by the vendor and they can be used by an attacker to tamper with the boot process (e.g., to forge PCRs values).

Butterworth et al. [4] designed a timing-based attestation at the BIOS level as an alternative to the hashing of the firmware. Such a technique provides a reliable way to attest the integrity of a platform even if the attacker has the same privilege level as the SCRTM. The idea, adapted from previous work on timing-based attestation [26], is that in the absence of an attacker the time required to perform a checksum of the firmware will be constant. When an attacker tries to fake the checksum, she requires additional instructions that increase the execution time, hence it can be detected by the system. While this work greatly improves the trust in the remote attestation, and fixes the vulnerabilities discovered in their paper, it requires a complicated architecture for being deployed. In fact, it needs to set up a remote server for the attestation phase and to modify the interrupt signal handling in the OS to obtain a precise measurement of the code execution. On the contrary, our approach works without having an attestation architecture and it only performs static checks on the firmware boot image.

Recent platforms incorporate immutable, hardware protected SCRTMs, called Intel Boot Guard [23] and HP Sure Start [11]. They are immutable SCRTMs that measure and verify at boot time the BIOS before its execution, thus providing firmware integrity and a trusted boot chain with a Root-of-Trust locked into hardware. Such technologies ensure that the first measurement cannot be forged, since the attacker cannot modify their code. Both technologies, however, are only available in recent Intel and HP platforms. In addition, Intel Boot Guard has been showed to be vulnerable to a certain class of attacks [19]. The advantage of our approach with respect to those new technologies is twofold. First, it can be used to protect architectures that are not equipped with such hardware features. Second, our approach is orthogonal to such hardware protections, since BootKeeper can be used as a standalone analyzer from the vendor side for validating the SCRTM code as the last step of the deployment process. The main contribution of BootKeeper is related to the software properties that we devise for validating the measurement process. When BootKeeper is used by the vendor, our analyzer can perform the same analysis (e.g., enforcing software properties) at the source code level, and verify that no one tampers with the measurement task during the developing process.

8 CONCLUSION

In this paper, we introduce BootKeeper, a binary analysis approach to validate the measurement process of a boot firmware. Our system uses static analysis and symbolic execution to validate a set of software properties on the measurement process implemented as part of the UEFI *measured boot* specification. BootKeeper detects incorrect implementations of UEFI firmware which do not exhaustively or correctly implement the measured boot process, as well as malicious images crafted with the intention of bypassing the measured boot process. More specifically, BootKeeper focuses on the SCRTM, which is the most critical component in the verification chain. An incomplete SCRTM implementation leaves room for an attacker to hide code in subsequent parts of the firmware, whereas a malicious SCRTM voluntarily ignores specific regions where malicious payloads are hidden, or attempts to forge measurements in order to match the measured values of a legitimate vendor firmware (i.e., golden values), among other possible attacks.

This approach can greatly improve trust in boot firmware update procedures. We evaluate BootKeeper against real-world firmware used in the industry as well as custom malicious firmware images, and show that our system is able to detect multiple variants of a variety of attacks from the state-of-the-art in the literature.

REFERENCES

[1] Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R Tuttle, and Vincent Zimmer. 2015. Symbolic execution for BIOS security. In *9th USENIX Workshop on Offensive Technologies (WOOT '15)*.

[2] Jonathan Brossard. 2012. Hardware backdooring is practical. *BlackHat, Las Vegas, USA* (2012).

[3] Danilo Bruschi, Lorenzo Cavallaro, Andrea Lanzi, and Mattia Monga. 2005. Replay attack in TCG specification and solution. In *Proceeding of the 21st Annual Computer Security Applications Conference (ACSAC '05)*. IEEE, 127–137.

[4] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. 2013. BIOS Chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security (CCS '13)*. ACM, 25–36.

[5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[6] University California Santa Barbara Computer Security Lab. [n. d.]. angr, a binary analysis framework. https://angr.io/ Accessed: 2018-11-30.

[7] Drew Davidson and Benjamin Moench. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the USENIX Security Symposium*.

[8] F-Secure. [n. d.]. Virus:DOS/CIH Description | F-Secure Labs. https://www.f-secure.com/v-descs/cih.shtml Accessed: 2018-11-30.

[9] John Heasman. 2006. Implementing and detecting an ACPI BIOS rootkit. In *Black Hat Europe*.

[10] John Heasman. 2007. Implementing and detecting a PCI rootkit. In *Black Hat DC*.

[11] HP Inc. 2018. HP Sure Start. https://www8.hp.com/h20195/v2/GetPDF.aspx/4AA7-2197ENW.pdf Accessed: 2018-11-30.

[12] Corey Kallenberg, John Butterworth, Xeno Kovah, and C Cornwell. 2013. Defeating Signed BIOS Enforcement. (2013). EkoParty, Buenos Aires.

[13] Bernhard Kauer. 2007. OSLO: Improving the Security of Trusted Computing. In *Proceedings of the USENIX Security Symposium*.

[14] Doowon Kim, Bum Jun Kwon, and Tudor Dumitraş. 2017. Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, 1435–1448.

[15] Akos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. 2003. Interprocedural static slicing of binary executables. In *Proceeding of the 3rd International Workshop on Source Code Analysis and Manipulation (SCAM '03)*. IEEE, 118–127.

[16] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. 2010. Testing Closed-Source Binary Device Drivers with DDT. In *USENIX Annual Technical Conference*.

[17] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. 2015. Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 203–214.

[18] Ge Livian. 2011. BIOS Threat is Showing up Again! https://www.symantec.com/connect/blogs/bios-threat-showing-again Accessed: 2018-11-30.

[19] Alex Matrosov. 2017. Who Watch BIOS Watchers? https://medium.com/@matrosov/bypass-intel-boot-guard-cc05edfca3a9 Accessed: 2018-11-30.

[20] Kevin O'Connor. [n. d.]. SeaBIOS. https://www.seabios.org/SeaBIOS Accessed: 2018-11-30.

[21] OpenSSL Foundation, Inc. [n. d.]. OpenSSL. https://www.openssl.org/ Accessed: 2018-11-30.

[22] Ganesan Ramalingam. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471.

[23] Xiaoyu Ruan. 2014. Boot with Integrity, or Don't Boot. In *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, Berkeley, CA, USA, Chapter 6, 143–163.

[24] Anibal L Sacco and Alfredo A Ortega. 2009. Persistent BIOS infection. In *CanSecWest Applied Security Conference*.

[25] Ahmad-Reza Sadeghi, Marcel Selhorst, Christian Stüble, Christian Wachsmann, and Marcel Winandy. 2006. TCG inside?: a note on TPM specification compliance. In *Proceedings of the first ACM workshop on Scalable trusted computing*. ACM, 47–56.

[26] Dries Schellekens, Brecht Wyseur, and Bart Preneel. 2008. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming* 74, 1 (2008), 13–22.

[27] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the Network and Distributed System and Security symposium (NDSS)*.

[28] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the Network and Distributed System and Security Symposium*.

[29] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*. 138–157.

[30] The TianoCore Community. [n. d.]. EDK II. https://github.com/tianocore/tianocore.github.io/wiki/EDK-II Accessed: 2018-11-30.

[31] Trusted Computing Group 2005. *PC Client Specific Implementation Specification for Conventional BIOS*. Trusted Computing Group.

[32] Trusted Computing Group 2005. *PC Client Specific-TPM Interface Specification*. Trusted Computing Group.

[33] Trusted Computing Group 2011. *TPM Main, Part 1 Design Principles*. Trusted Computing Group.

[34] Trusted Computing Group 2014. *EFI Platform Specification*. Trusted Computing Group.

[35] UEFI Forum 2017. *UEFI Platform Initialization Specification* (version 1.6 ed.). UEFI Forum.

[36] UEFI Forum. 2017. *Unified Extensible Firmware Interface Specification*. Version 2.7.

[37] Rafal Wojtczuk and Alexander Tereshkin. 2009. Attacking Intel BIOS. (July 2009). Black Hat USA.

[38] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, 732–744.

[39] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.. In *NDSS*.

[40] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems Firmwares. In *Proceedings of the 21st Symposium on Network and Distributed System and Security (NDSS '14)*.

[41] Shiva Dasari Zimmer, SR Dasari, and SP Brogan. 2009. *Trusted Platforms: UEFI, PI, and TCG-based firmware*. Technical Report. Intel and IBM.

[42] Vincent Zimmer, Michael Rothman, and Suresh Marisetty. 2010. *Beyond BIOS: developing with the unified extensible firmware interface*. Intel Press.