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.