Verified/Secure Boot: Formal Verification of Firmware & Hardware in a large SoC

David J. Gilhooley
Adviser: Sharad Malik
Second Reader: Aarti Gupta

May 8, 2017

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Electrical Engineering
Princeton University

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.

David J. Gilhooley

Verified/Secure Boot: Formal Verification of Firmware & Hardware in a large SoC

David J. Gilhooley

Abstract

A system can only be considered truly secure if there exists a proof that the system meets its security specification. This is done with the use of Formal Verification tools, which can analyze systems and provide formal proofs of correctness. However, security systems are becoming larger and more complicated, making it harder to provide such proofs. These systems have also started to offload calculations to hardware modules. In the past, tools for Formal Verification were focused on proving correctness for either a hardware system or a software system. Traditional verification tools focus on a single type of system and are not equipped to handle current systems that depend heavily on interactions between hardware and software. New tools have been developed to perform verification on these systems, but the techniques are tested on small, academic systems. This thesis investigates the use of new formal verification tools on the industry-sized security program Verified Boot. Google's Verified Boot is a secure bootloader used in the chromebook laptops. Verified Boot uses cryptographic functions to ensure that only Google's code is run on the laptop. This process involves many hardware-software interactions, making it a difficult system for traditional verification techniques. In order to perform the verification, the Instruction Level Abstraction (ILA) toolchain is used to model Verified Boot's required hardware of a SHA Accelerator and a Trusted Platform Module. The ILA toolchain allows formal models to be created from Hardware. These models are verifably equivalent to the Hardware models and can be used in conjunction with the software verification. A formal model was created for parts of a Trusted Platform Module (TPM). Additionally, a software framework was created around Verified Boot in order to run it entirely on open sourced hardware. Finally, multiple software abstraction techniques are used in order to run Formal Verification tools on Verified Boot's large codebase in a reasonable time. Twenty security proofs are generated from the codebase, verifying the claims originally stated by Google. Program correctness, array-bounds, and program flow are verified against all possible user inputs.

Acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computing systems are becoming larger and more complicated, making security vulnerabilities harder to find. At the same time security breaches are having a larger impact on both personal and societal levels. As the computer industry begins to produce laptops, smart phones, servers, and embedded systems, computer hardware becomes more diverse and specialized. Diverse hardware increases the potential for vulnerabilities, and a hardware vulnerability is difficult or impossible to fix without replacing the device.

The current industry standard is to use Unit Testing to discover errors in programs. Unit Testing is the process of dividing a program up into functional chunks and verifying each function as a single unit. These units are used as "black boxes", meaning that they are fed a range of test inputs, and the behavior is compared against expected results. However, Unit Testing has a problem with scaling. Inputs for tests scale exponentially, meaning computers can only check a unit against every possible input if the space of inputs is relatively small. Most Unit Tests only test against a few inputs that are thought to be representative. However, this means that there is always the possibility of bugs on the set of inputs that were not used in testing. It also leads to the possibilities of false positives when testing on inputs that cannot appear in a real system.

Formal Verification can be seen as an enhancement to Unit Testing. Formal Verification is the act of proving or disproving the correctness of a function by using mathematical proof techniques. This verification requires that the function's correctness is specified in a series of formal properties. Once these properties are created, current verification methods will produce proofs that the properties hold under all possible inputs. Unlike Unit Testing, Formal Verification tools look at the code itself to quickly find inputs that would violate a user-defined property.

Formal Verification guarantees that a program is working correctly. However, today's security programs are relying both on software programs and hardware modules working together, and there are no industry standards for verification across this boundary.

1.1 The Hardware-Software Boundary

Originally a computer was a general purpose CPU, and software would describe how the CPU should perform various tasks. Today, computer engineers realize that specialized hardware can perform a single task much more efficiently than generalized hardware. This lead to the creation of a System on a Chip, or SoC. An SoC is the combination of CPUs with smaller hardware modules that perform specialized tasks. The CPU is still the main processing unit, meaning it is responsible for communicating with each hardware module. However, the CPU no longer performs all processing in the system.

Hardware modules are taking some responsibility away from software. If software wants to decrypt a message, then it can pass the message to an RSA hardware module and receive the decrypted message back. This means that the software does not need to implement a decryption function. However, the software now needs to know how to communicate with the Hardware module. Software that is aware of specific Hardware is known as Firmware. Software interacts with Hardware by reading and writing to specific registers. Firmware is the Software that controls these register interactions. The remainder of this thesis will not use the term Firmware, and will instead refer to the boundary of Software and Hardware. This naming convention provides more clarity for the purpose of this thesis.

The boundary between Software and Hardware is more important now that SoCs are becoming larger and including more specialized Hardware. More computers are including security based hardware, like hashing and encryption/decryption modules. It is no longer enough to verify software and hardware separately, as the interactions between the two are an important part in modern security algorithms.

1.2 Existing Verification Techniques

Formal Verification is defined as the application of mathematical methods to prove or disprove the specification of a system [1]. To use Formal Verification, the user specifies properties of a system to be verified. Propositional logic is one of the main ways that

properties are specified. For example, a programmer could express that given the fact that a password check failed, the user should not be allowed to access bank statements. If A is access and P is correct password, then the following statement describes the property that, "If access is given, then the Password is correct".

$$A \rightarrow P \tag{1.1}$$

Propositional Logic statements can be described completely through Boolean Logic and can then be evaluated using Boolean Satisfiability Solvers (SAT solvers) [2]. SAT Solvers are an important component of verification but they have known limitations with scaling. Additional systems have been used to extend the capabilities of SAT solvers, allowing them to describe more complicated properties and systems. One system is known as Satisfiability Modulo Theories (SMT), where Boolean variables can represent predicates such as inequalities between real valued variables (eg: $x < y$) [3].

SMT and SAT Solvers are two tools that perform Formal Verification. Automated tools exist to transform software programs with assertions into SMT statements. This means that software can be easily verified with SMT solvers. One popular tool that was used in this research is the C Bounded Model Checker (CBMC) [4]. This tool automatically transforms software into SMT statements and verifies user assertions. More information about Verification tools can be found in Section 4.

1.3 Problem Statement

The trend in modern Computer Architecture is to offload computation to specialized hardware for either speed or power savings [5]. As this becomes more common the job of verifying functionality becomes more difficult. Software and Hardware can be verified as separate units but this misses bugs that exist as a result of the interaction between the two.

In past research, small SoCs have been created for the purpose of testing formal verification techniques [6]. These SoCs have been built by connecting numerous open source components where the hardware specifications are readily available. Formal verification has been seen to be successful in analyzing security protocols such as Secure Boot [6], Remote Attestation [7], and Firmware Loading [8]. However, most research papers create their own SoCs from open source hardware and develop their own firmware for verification. While these papers are useful, they do not say much

about the application of Formal Verification to industry SoCs or industry Firmware. SoCs and Firmware used by industry are typically much larger and more complicated, which makes them more challenging for Formal Verification. Additionally, other Formal Verification papers design techniques specific to a single application, without considering the extension of a framework for general Hardware Software verification.

A standard tool for verification across the Hardware Software boundary would greatly speed up the verification of different systems. Throughout this paper, the Instruction Level Abstraction (ILA) toolchain will be used to define Hardware Software interactions. An ILA defines Hardware as a collection of instructions that can be used by Software through accessing specific registers [9]. The ILA toolchain provides a way to "precisely capture updates to all firmware-accessible states spanning cores, accelerators, and peripherals [10]." This technique claims to allow software and hardware to be verified simultaneously and at scale using formal methods.

This thesis will be applying formal verification techniques to Google Chrome's Verified Boot. Verified Boot is a large scale, security based program that spans the Hardware Software boundary. It is an implementation of a secure bootloader that has been designed by Google to run on their Chromebook laptops [11]. A bootloader is responsible for initializing the computer's hardware and loading the rest of the Operating System. A secure bootloader will only load an OS that passes specific security checks [12]. Verified Boot checks that the OS it is loading has been signed by Google and is unmodified.

Verified Boot is a good candidate for this report because its execution relies on both Hardware and Software. The Verified Boot code is also hard-wired into the Chromebook laptops, meaning that any bugs must be found before laptop production starts. The importance of finding bugs makes Formal Verification guarantees more appealing. Additionally, Google has released both Verified Boot and Chrome OS as Open Source software, meaning verification can be run directly on industry code.

1.3.1 Contributions

The main contribution of this thesis is the application of Formal Verification tools to a large industrial project that spans the Hardware Software boundary. In order to perform Verification, specific abstractions are used, and these techniques can be extended to other large projects.

This report also takes advantage of the Instruction Level Abstraction tool chain to show its effectiveness in a large scale system. In doing so, a hardware model is created

for parts of the Trusted Computing Module (TPM). The universal nature of the ILA toolchain means this hardware abstraction can be used in other projects. The use of the ILA toochain in this paper also displays its effectiveness in large projects.

Finally, this report ends with security recommendations for Verified Boot, which would harden the boot system based on the results of the Formal Verification.

1.4 Overview

This thesis moves from a description of the Verified Boot system to the Formal Verification techniques. Chapter 2 describes the Hardware used (and not used) by Verified Boot. Chapter 3 describes the Software implementation and organization. It describes any software deviations from Verified Boot that were necessary. Chapter 4 describes the Formal Verification techniques and abstractions used. Chapter 5 outlines the results of Formal Verification and outlines security recommendations for Verified Boot. Chapter 6 concludes the thesis by reiterating results and exploring possibilities for future work.

Chapter 2

Hardware Design

This section describes the Hardware design of Google's Verified Boot (Vboot).

Vboot runs before the chromebook's main operating system and verifies that the operating system has originated from Google and has not been modified. The main operating system is packaged together with meta-data and security information such as signatures and checksums. This package will be referred to as Google's Image.

Vboot attests the Image by the cryptographic functions of hashing and signing. These functions can be computationally expensive and they are used often in a modern computer, so they are good candidate for hardware acceleration.

The hardware modules used by Vboot can be broken into two groups: secure storage and hardware accelerator. For secure storage there is a Flash memory module that provides Read-Only non-volatile storage, and a Trusted Platform Module (TPM) that provides non-volatile storage that can be locked until the next boot. For hardware accelerators there is the cryptographic accelerator that implements the Secure Hash Algorithm (SHA).

While the code for Vboot is open source and therefore available online, the laptop's hardware does not have source code available. However, Google has released documents describing specifications for the Hardware modules required. These specifications were used to pick Open Source hardware that could be made to work with the Vboot code. Verification is not done on Chromebook hardware because source code access for the hardware is not available. However, if it becomes available, then the techniques outlined will be easily applicable to the real hardware.

2.1 Flash Memory

The Flash Memory is an amount of extra non-volatile memory connected to every Chromebook. The Flash Memory contains both a Read-Only section and a Read-Write section of memory. The Read-Only section is arguably the most important piece of hardware because it is the Root of Trust of the Vboot system. The Root of Trust is the set of Software and Hardware elements that are inherently secure [13]. This means the Root of Trust is assumed to be secure by a security analyst, because if it is not secure then none of the properties will hold. The code within the Read-Only section is the root, or base, that is responsible for verifying the rest of the Vboot process. The Read-Only section is enforced by a physical screw, called the Write Protect screw, which is located in the laptop. Removing this screw removes the Root of Trust, but doing so voids the laptop's warranty.

The Read-Only memory is flashed into the laptop when the Chromebook is being built in the factory, before the Read-Only screw is added to the casing. The Reset Vector of the CPU points to the Read-Only flash memory, and the code located in the RO section provides information to verify the rest of the system. Verified Boot claims to be secure as long as the Root of Trust holds.

Each Chrome book comes with 8 Mb of non-volatile Flash memory [14]. The memory is organized according to the Flashmap which is included in the appendix. The Flash memory is connected to the CPU by the Serial Peripheral Interface (SPI). The SPI driver handles the protocol and provides an interface for reading multiple bytes of of memory at once.

2.2 TPM

The Trusted Platform Module or TPM is a general purpose cryptographic processor that is connected to a computer system for Hardware based security[15]. A cryptographic processor is a processor that implements a variety of cryptographic functions, like hashing, encryption and decryption, and random number generation. The full list of cryptographic functions is provided in Figure 2.1. Additionally the TPM contains its own storage (volatile and non-volatile), that can be accessed from the main CPU in specific ways. For the purpose of this report, the TPM is used as a module that can securely store keys and measurements made by the bootloader.

The physical hardware of a TPM can be created by any company, but it must conform to the standards released by the Trusted Computing Group (TCG) [16].

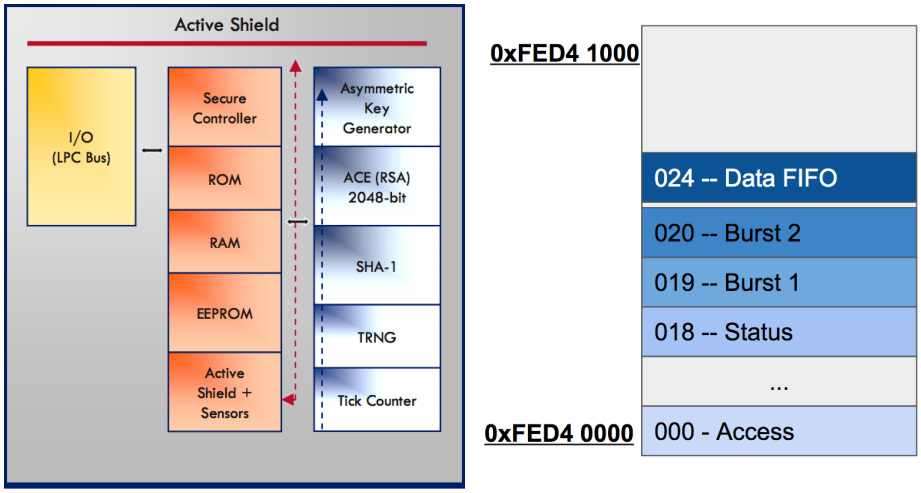

Figure 2.1: On the left we can see a HW block diagram of the TPM itself [20]. On the right we can see the registers of the TPM that are accessed by the CPU.

The TCG has released two specifications of TPMs and they are known as TPMs version 1.2 and 2.0. For the scope of this project we looked at the TPM 1.2 specifications [17][18][19].

The TPM specification outlines the commands, structures, and interface for the system. The TPM commands describe the full functionality of the system by describing each command the main CPU can request. The TPM structures describe what data structures can be passed back and forth from the CPU to the TPM, as well as what information the TPM can hold within. The TPM interface describes the registers that are exposed to the CPU and how they should be used to send data back and forth.

2.2.1 TPM Command Interface

The CPU communicates to the TPM by reading and writing specific memory addresses. These addresses allow the CPU to send multi-byte commands to the TPM and receive multi-byte responses, even though a single byte is read or written at a time. The addresses and names of the registers can be seen in Figure 2.1. The registers are described below.

- `Status` — Reading this register returns a value with flags that show if the TPM is accepting a command, sending a command, or busy. Writing specific values to this register tells the TPM to start accepting a command, or to run a command that has recently been sent.

- `Burst` — Reading this register lets the CPU know how many bytes can be read (if sending a command) or written (if receiving a command) at a single time. This register cannot be written to.

- `Data` — Reading this register reads one byte of data from the TPM . Writing this register sends one byte of data to the TPM .

Sending a command serially to the TPM works in the following way. First, the `Command_Start` value is written to `Status`. Second, the `Burst` register is read to see how many bytes the TPM can accept. Next, the Data register is written to up to `Burst` times, until the command has been sent. If the command is longer then `Burst` amount, `Burst` can be polled again until it becomes non-zero. Finally, the `Command_Go` value is written to `Status` and `Status` is polled until the TPM says that it has finished processing the command.

Receiving a command follows the process above, except that `Data` is read instead of written to. Now that it is understood how commands are passed to and from the TPM, we can discuss the commands used by Google's Vboot.

2.2.2 Platform Configuration Registers

One component of the TPM is called Platform Configuration Register, or PCR. The goal of a PCR is to store measurements of the system state in a secure way. PCRs are secure storage because they cannot be written to directly, they are either extended or reset. PCRs are extended by concatenating the current PCR value with the input, and then storing the SHA1 hash of the result. Resetting the PCR changes the value to all zeros, although the PCRs used by VBoot have the reset functionality disabled. The TCG states that only three commands affect PCRs: `PCR_Extend`, `PCR_Read`, and `PCR_Reset`. PCRs can also be locked, meaning that they are then unable to be extended or reset for the remainder of the system's power cycle. The TCG specifications of these commands are included in the Appendix.

The hardware diagram of `PCR_Extend` can be seen in Figure 2.2. Because the PCRs store a SHA1 Hash, they are useful for taking the measurement of a system. Hashing is a one way cryptographic function that is used to protect the integrity of a message. We can think of a hash as the fingerprint of a message, because it takes an arbitrary length of data and outputs a fixed length (20 bytes). The PCRs are iteratively extended with measurements of the system, for example, the hash of the Read Only Firmware state, then the hash of the Read Write Firmware state, and

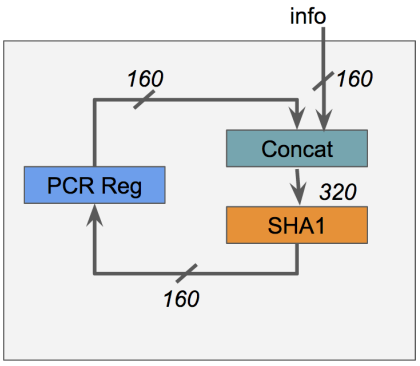

Figure 2.2: `PCR_Extend` Hardware Diagram

then the hash of the Kernel state. The final result will be identical each time if and only if there were no changes in each of the states being measured. Because SHA hashing is collision resistant, it is difficult to find a fake configuration of the system that will hash to the same value as a correct configuration. This means that if the hashes are equal it is reasonable to assume that the system is unchanged.

Under specification 1.2, a TPM is required to have 21 PCRS, each PCR is 20 bytes long and they are stored in the TPM's non-volatile storage. Each power cycle the PCRs are reset to all 0's.

2.3 SHA

While the TPM implements SHA hashing, Vboot also takes advantage of a dedicated SHA hashing accelerator. The TPM is designed to be secure, but the SHA module is designed to be faster than implementing the Hashing algorithm in software. The SHA-1 hashing function can operate on any length of input. It will iteratively hash the input in blocks of 64 bytes and output a 20 byte hash result.

The SHA module is much simpler to use than the TPM in terms of Memory Mapped Registers. Writing to the `sha_rdaddr` tells the module where to read the input data. The module will read `sha_len` bytes of the input data. Writing to the `sha_rdaddr` tells the module where to put the final 20 bytes of output data. Writing to the `sha_start` register causes the module to start the Hashing process. Reading the `sha_state` register returns the state that the module is in. This register is how a software program knows the module has completed its operation.

2.4 RSA

Many computer systems include RSA hardware accelerators, however Vboot does not require one. The Vboot library implements all RSA algorithms in software. These software algorithms will be used and verified instead of a hardware module. It is possible to add RSA hardware support to Chromebooks relatively easily, and this may help speed up boot times dramatically.

RSA is a cryptographic platform that allows messages to be encrypted and decrypted through the use of public and private keys. The properties of RSA are used to send messages where the message receiver can be sure of the identity of the message's sender. For example, Google encrypts the Image with a private key known only to Google. In order to recover the Image, the Chromebook must decrypt the data with Google's public key, which is publicly available. The RSA algorithm asserts that only Google has access to the private key, and that only Google's public key can decrypt the data. This assertion provides a reasonable guarantee that the data arrived from Google.

In reality, the RSA algorithm is unable to encrypt large amounts of data. For this reason, only the hash of the data is encrypted. The entire image can be attested from the encrypted hash because the end user can hash the image again and compare the two. This is the attestation process that will be used throughout the rest of the paper. The encrypted hash value of a data structure is known as a "Signature." Encrypting with an RSA private key is known as "signing" data. Decrypting with an RSA public key is known as "unsigning" data. Using a signature is referred to as "attesting" a data structure and the outline of this process is given in Figure 3.3 in Chapter 3.

2.5 QEMU Emulation

QEMU [21] is an open source emulator that is used in this research project to emulate the SoC running Vboot. For this project it is used to emulate a 32 bit i386 CPU with a Memory Mapped TPM and SHA module. Starting Vboot relies on QEMU's ability to boot any image that starts with a defined multi-boot header [22]. QEMU then supplies its own bootloader, which is responsible for initializing RAM, putting the processor in 32 bit mode, and loading the image off of disk. As we will see in Section 3.5.1, for Chromebook this job is normally taken care of by Coreboot, but for the purposes of this project it was easier to use QEMU's built-in bootloader.

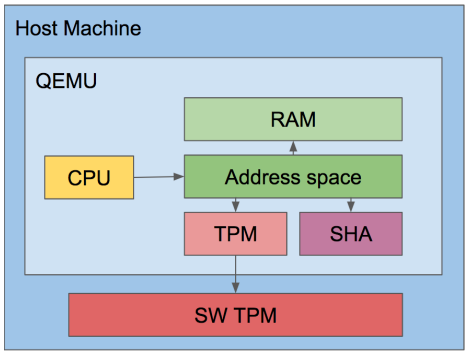

Figure 2.3: The QEMU layout. We can see here how the CPU and other hardware modules write RAM, and how some QEMU modules are interfaces to external software libraries.

The boatloader code that interacted with QEMU (included in the appendix and the github link) was based off of a project developed for "Open Source at Princeton" by Lance Goodridge and me. The project allowed students to build their own Operating System from scratch and develop it within QEMU. The project was repurposed by this thesis for its well-organized Makefile, linker script, and multi-boot header. These tools facilitated the creation of a standalone C file that could hook into the Vboot repository and run Google's Verified Boot within QEMU's virtualized Hardware.

The QEMU emulator was modified to add a TPM and a SHA module. However, it does not include the Flash Memory. While the Flash Memory is important as a Root of Trust in the System, its purpose is very simple. Its function is to provide read-only storage, and this is a property inherent to the Hardware. Instead of adding Flash Memory to the QEMU emulation, the Image is loaded directly into RAM using a specialized linker script. The firmware image would normally be located in the Flash Memory, but it is now located in RAM. The Read-Only properties of this section of RAM are not enforced, but will be assumed to be correct when the system is being analyzed.

2.6 Instruction Level Abstraction (ILA)

The Instruction Level Abstraction (ILA) toolchain was used to abstract the hardware modules so they could be verified in conjuction with the Vboot software. Verifying Vboot without the hardware modules would lead to incorrect results. Vboot's software interacts with hardware by reading and writing to memory-mapped registers that trigger hardware events. Without including the hardware modules in the verifi-

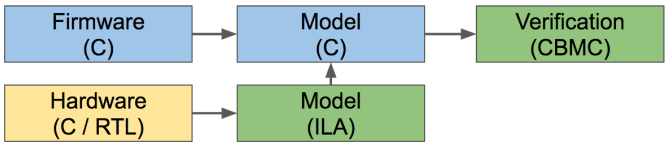

Figure 2.4: A flowchart of how ILA is used in the Verification process. In this report, ILA is used to combine the hardware model with the C program for CBMC verification.

cation, a software only model checker like CBMC will not understand that a write to a specific address corresponds to an entire hardware operation like SHA hashing.

ILA provides a way to "precisely capture updates to all firmware-accessible states spanning cores, accelerators, and peripherals [10]." An ILA is created from either a C or Verilog program that describes the Hardware. To create an ILA, a programmer defines a "template" in python describing the hardware. The ILA library then verifies this template against the given C or Verilog program to ensure that they exhibit the same behavior. This is known as "synthesizing" the template into an ILA model. The template describes each element of the hardware's state and gives a list of how each state can be updated. The synthesis method uses the C or Verilog program to determine when the update functions defined by the template are activated.

The template is defined using standard update terminology for hardware, which uses the idea of separate Decode and Execute stages. The template's Decode stage specifies which state elements correspond to unique instructions. For any modules with memory mapped registers, the address of the register being accessed, and any write information will be included in the Decode stage. The Execute stage involves the state update functions, and it depends on the values in the Decode stage. For example, the TPM ordinal determines if the TPM will execute a hashing function or a locking function. Executing the function is the purpose of the Execute stage.

The ILA synthesis is responsible for automatically mapping the Decode stage to the Execute stage, and it uses the C or Verilog program to perform this mapping such that the behavior of the hardware and the ILA model are identical. The ILA template that is created captures the interface of the hardware. It produces an abstraction that is aware of how the hardware reacts to reads and writes of specific registers. This template can be connected back with the software such that verification is aware of the Hardware-Software boundary.

The benefits of the ILA toolchain are clearly seen when looking at its expressiveness in capturing hardware behavior. The QEMU TPM interface is defined in 3,077

lines of C code. This does not include the functionality of the TPM itself. This same functionality can be captured with an ILA template of the TPM interface that is 144 lines of python code. Additionally, a TPM simulator was created in 253 lines of python code. The same TPM simulator was converted to 307 lines of C code. These simulators can be verified to have identical behaviors using the ILA toolchain. They can also be verified against the ILA's synthesized TPM, which combines the template and one implementation and creates a 1,880 line C file.

2.6.1 Connecting the ILA

An ILA model is converted to C code through the ILA toolchain. This allows the model to be added to the Verified Boot software. As seen in the Appendix, the Software for the TPM relies on the `read8` and `write8` functions to make reads and writes to hardware. In a real system, these reads and writes would connect directly to the TPM. In the system created for this report, the function connects to the `tpmUpdate` function generated by the ILA. Using the ILA toolchain means that C function representing the hardware model can be verified to be functional equivalent to the actual hardware. The ILA toolchain can perform equivalence checking between two C models, two Verilog models, and a C and a Verilog model [9].

Chapter 3

Software Design

This section will discuss the Software implementation of Verified Boot. Verified Boot (Vboot) is a secure boot implementation only loads an Image that has been signed by Google and is untampered. Images that have been recognized as security risks will not be loaded and the system will restart into a Recovery Mode. This chapter discusses the security promises made by the system as well as the attack vectors that are not considered. It will also discuss relevant program flow and data structures.

3.1 Security Promises

The main purpose of Chrome OS's verified boot is to provide relative security to the end user without sacrificing usability or functionality. In its mission statement, verified boot is designed against an "opportunistic hacker [23]." Vboot protects against vectors of attack that are relatively quick to exploit. Vboot will realize any attack or modification to the Image on the next boot cycle. Once an attack is realized Vboot is responsible for forcing the system back into a secure state. For example, installing a malicious kernel driver to act as a keylogger would alter the Image on the Chromebook. The next time the Chromebook boots, Vboot will recognize that the Image changed, and it will force the user to enter Recovery Mode and secure the device.

There are many attacks that Vboot is not able to recognize or protect against. For example, Vboot does not make any promises about the safety of the system once the kernel is running and the user has control. Any attacks that are outside of the Image (in userspace) would remain undetected because they would not modify the kernel. For example, changing the Web browser to use a proxy that collects information would not be considered a kernel attack. Userspace programs by definition have less access to the system and this reduces the severity and influence of the attack. Possibly one

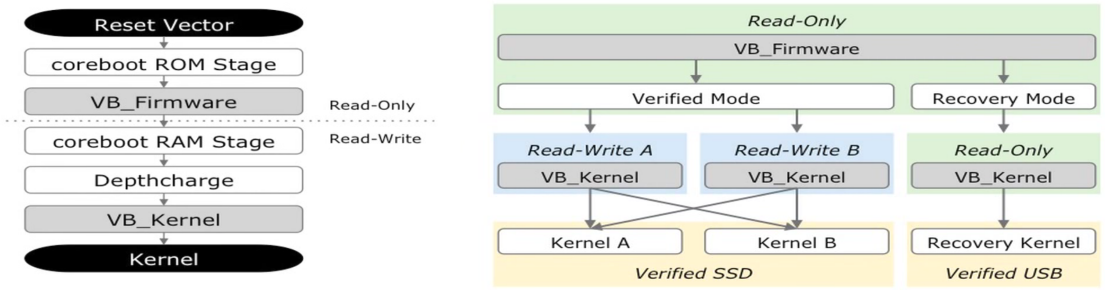

Figure 3.1: The left image shows the stages of Vboot Firmware [14]. The right image shows the different boot paths, and data locations [14]. Both make the distinction between Read-Only and Read-Write memory.

of the biggest flaws is that any attack to the kernel during runtime would remain until the system was powered down. In certain situations this could provide plenty of time for an attack to steal valuable data. However, such an attack is outside the scope of this paper.

3.2 Verified Boot Flow

In order to make Vboot modular and extensible for future platforms, Google has split the process into three main sections as seen in Figure 3.1. This split allows Google to make the Read-Only stage as small as possible, which is desirable because bugs in this stage cannot be fixed. These sections are referred to as Read-Only (RO) Firmware, Read-Write (RW) Firmware, and Kernel. These sections are run in sequence with each section attesting the section that comes later. The Root of Trust begins in the Read-Only section. This section is assumed to be unmodified because it is read-only. Then each section attests that that next section is signed by Google and unmodified. In this way, the Root of Trust builds until all of the system's code is attested, and the laptop can boot.

For the scope of this thesis only the attestation the Read-Only Firmware's attestation of the Image will be discussed. The Read-Only attestation contains more interesting control flow, and the attestation process is essentially repeated for the Read-Write Firmware.

The Read-Only Firmware of Vboot is responsible for attesting Google's Image. Vboot contains all of the code and hardware drivers it needs to accomplish this task. It verifies the Image signature using Google's main public key. This public key is also read-only so it is guaranteed to be secure. The Read-Only Firmware is also

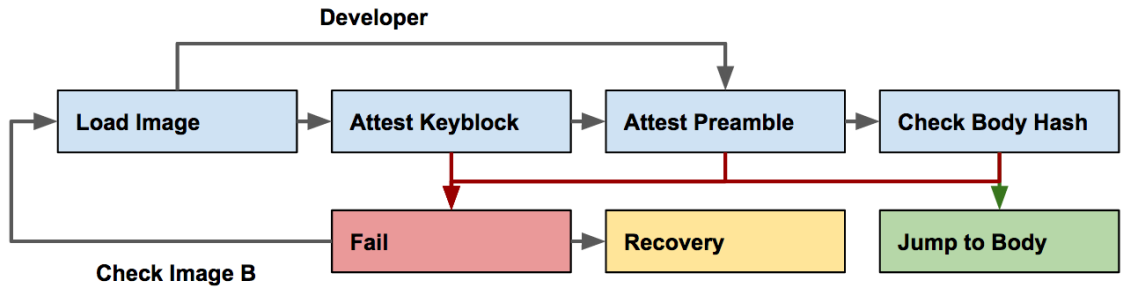

Figure 3.2: The logical flow for attesting the Image. Notice the arrow Developer mode from Loading to attesting the Preamble.

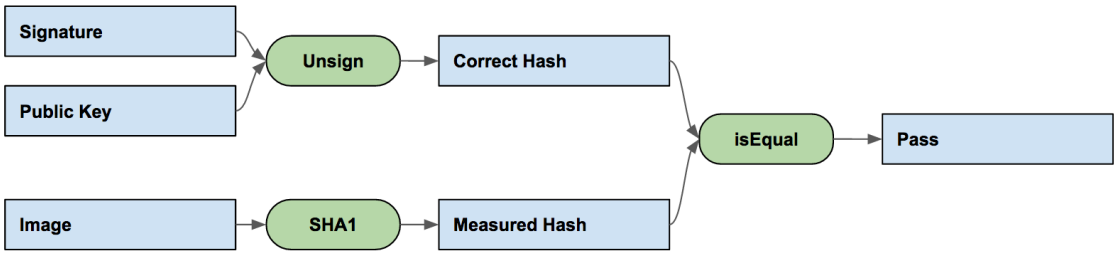

Figure 3.3: The logical flow for attesting any data structure.

responsible for the transitions between Developer Mode, Safe Mode, and Recovery Mode. These transitions will be described in more detail in Section 3.3.

Vboot's process for attesting the Image is described as follows:

- Vboot loads the Image. This consists of copying the image out of Flash Memory and into RAM.
- Vboot attests the Keylock. The Keyblock's signature is a hash of the Keyblock that is signed by Google's Private key. Google's Public Key is used to decrypted the signature. Once decrypted the signature is is now a hash of the Keyblock. Finally, Vboot hashes the Keyblock, and this value is compared with the decrypted signature. If the hashes are not equal, then Vboot can be sure that the Keyblock has been modified in some way.
- Vboot attests the Preamble. The Keyblock holds the key that is used for the Preamble's signature. The Keyblock is used so that Google has the ability to change the encryption type of the Preamble and Firmware if they wish. Google's Main Public Key is stored in Read-Only memory, so it cannot be changed.
- Vboot attests the Image. The Image is hashed and compared against the hash stored in the preamble. The hash in the preamble has already been attested

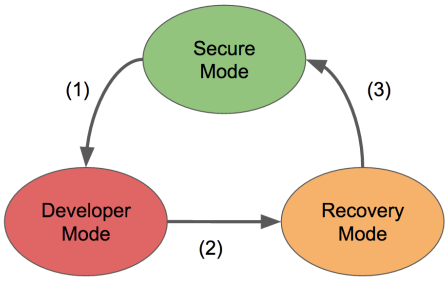

Figure 3.4: The FSM for the boot modes.
1. The TPM's nonvolatile storage and keys are wiped. The user's data is erased.
2. The laptop is booted into Recovery, Vboot installs a Chrome USB.
3. The TPM's storage goes through reconfiguration. The entire disk is wiped.

because the preamble was signed, so this attests that the Image is correct and unmodified.

- Vboot executes the Image. The system jumps to the location of the Image in RAM and begins to execute the Operating System code located there.

These stages are also seen in Figure 3.2.

3.3 Boot Modes

Vboot has three major boot modes that influence the program's behavior. The first is the Secure mode which goes through the full process of Verified Boot. The second mode is the Recovery mode which allows for a broken machine to format all memory and transition to Secure Mode. The third mode is the Developer mode and in this mode the RSA signature is not verified. The Developer mode exists so that hobbyists can write their own firmware boot code and run operating systems other than Chrome OS.

The transitions allowed between the modes can be seen in Figure 3.4. On each transition, Vboot is responsible for taking steps to ensure that security is preserved. These responsibilities are also seen in the figure.

3.3.1 Developer Mode

Developer mode poses interesting security questions to Verified boot because it disables Vboot's security guarantees that the OS has been signed by Google [24]. To allow the existence of Developer mode, but still maintain Secure Mode, there are

various requirements around the Developer mode transition. First, a physical presence is required to fully complete the Developer mode transition. This means that a user must press a certain key combination (Control + D) when the developer mode screen appears at boot time. The physical presence exists such that developer mode cannot be enabled through an off-site software attack without the user's knowledge. The developer mode screen is referred to internally as the "Scary Screen", and its purpose is also to prevent users from being tricked into enabling developer mode by an external phishing party.

In order to prevent an attacker from enabling Developer Mode so that they could read or write to secure storage, Vboot wipes all secure storage on the transition into Developer Mode. The secure storage that is wiped includes the RSA keys and various secrets stored in the TPM and the partion on disk where user data is stored. Wiping this data on the transition is necessary for the confidentiality of the system and failure to do so is a security risk.

These precautions are also taken on the transition from Developer Mode to Safe Mode. If secure storage is left untouched moving from Developer Mode to Safe Mode, then an attacker would be able to place potentially malicious information in secure storage. Vboot assumes that the secure storage can only be written to and read from by Google, and this assumption is violated in Developer Mode. Wiping all secure storage on this transition is the easiest way to ensure that the platform has full control. The precautions mean that on the transition from Developer to Safe Mode, the system has to go through Recovery Mode.

3.3.2 Recovery Mode

Recovery Mode is responsible for getting the system back to a secure state. Recovery Mode will be activated automatically when an error is recognized by Vboot. This error could range from hardware failures to a corrupted image to a detected attack on the system. When the system boots into Recovery it requests a Recovery Image stored on external memory, either an SD card or a USB drive. Recovery Mode uses Google's Recovery RSA keypair. Google's Recovery public key is stored in Read-Only Memory for security. Recovery Mode does full attestation of the Recovery Image, and if the Image passes then it will overwrite the Images stored in main memory.

Recovery mode is responsible for wiping secure memory, mainly the user's disk partition and TPM's secure storage. If Recovery completes successfully, the system is in a secure state and Vboot can then continue safely. Recovery is also the only

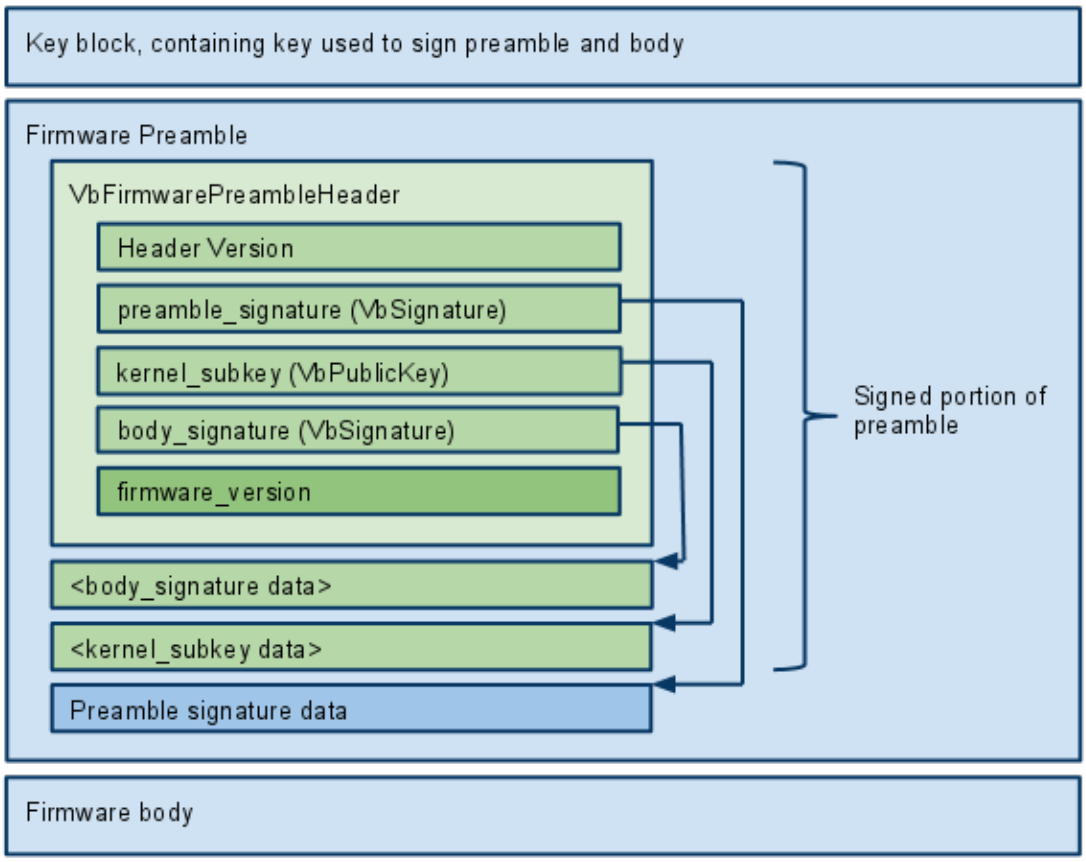

Figure 3.5: Layout of Google's Image [25]

mode that disables Vboot's rollback detection. Often if there are problems with an Image update, Recovery mode is the only way to install an older Image.

3.4 Data Structures

To start understanding the Vboot process of attesting the Image, it is necessary to talk about the Image's data structures.

3.4.1 Google's Image

The actual Image is not a data structure but a chunk of data that is loaded contiguously into RAM. The image structure, as seen in Figure 3.5, consists of three parts: the Key Block, the Preamble, and the main body of the image. The Key Block is attested first using Google's public key stored in Read-Only memory. Once the Key Block has been shown to be safe, the Key Block's public key will be used to verify

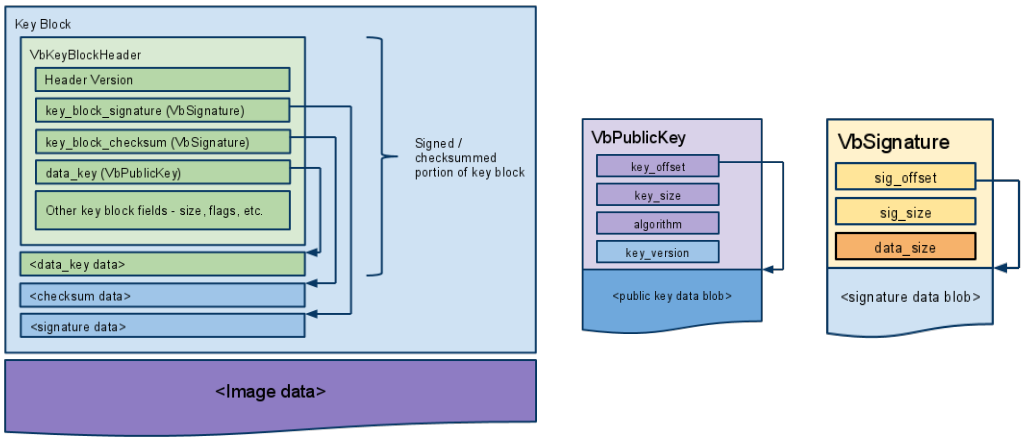

Figure 3.6: The Key Block data structure and the data structure for Keys and Signatures [25]

the preamble. The Preamble contains a hash of the Image's body. The Image's body contains the code that is going to be run next. Vboot attests the body by hashing it and comparing that value against the hash stored in the Preamble.

3.4.2 Key Blocks

The Key Block is the data structure that allows a hierarchy of RSA keys to be used during Vboot. Figure 3.6 shows the structure of the Key Block. The Key Block flags are used to determine which mode of Vboot the Keyblock is valid in. There are 4 possible boot modes corresponding to the combination of the two binary options, Developer and Recovery.

Within the Key Block there exists data structures for a public key and a signature. Google has added the ability to change their encryption strength. They have added support for RSA 1024, 2048, 4096, 8192 and for SHA 1, 256, 512, for a total of 12 different possible algorithm combinations.

3.4.3 Google Binary Block

The Google Binary Block (GBB) is a part of Vboot's Root of Trust. It is a data structure stored in Read-Only memory that is initialized and configured in the factory at the laptop's creation. It contains the Root and Recovery RSA keys, a host of flags that affect boot operation, and the bitmaps used for the Vboot screen displays.

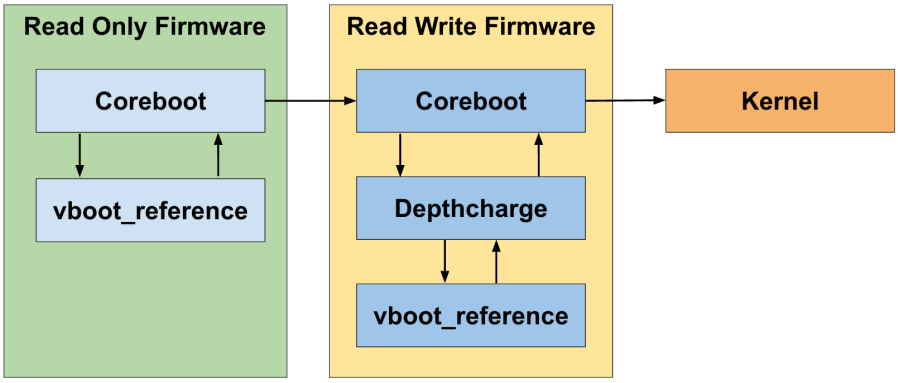

Figure 3.7: ChromeOS's boot flow goes through Coreboot, Depthcharge, and the Vboot Library twice for Firmware and Kernel verification

3.5 Code Organization

Like most modern operating systems, Chrome OS is written in C. Like Linux, Chrome OS is maintained using Git for version control. Git is a diff-based, non-centralized version control system that makes it easy for programmers to share code, rollback changes, and maintain separate branches of the same codebase [26]. Google has built a tool called "repo" that is used on top of git [27]. Repo is a tool to manipulate multiple code repositories. It's primary benefit is that it allows a company to specify how multiple git repositories should be installed and placed within a given file-system. This is both helpful and necessary as Chrome OS consists of over one thousand different external and internal repositories.

Coreboot, vboot_reference, and depthcharge, are the repositories used by Google for Verified Boot. The flow of Verified Boot through the repos can be seen in Figure 3.7 and the purpose of each repo is described below.

3.5.1 Coreboot and Depthcharge

Coreboot is a fully Open-Sourced alternative to traditional BIOS implementations. [28] It is lightweight and is configured to implement the full standard of the Unified Extensible Firmware Interface (UEFI). Google has chosen Coreboot because of its small code footprint, full extensibility, and the fact that it is available freely as an Open Source project.

The Coreboot code is responsible for doing very early initialization code on the main CPU. Coreboot is setup so that once a baseline level of initialization is complete, it passes control to another section of code called a "payload" [29]. This payload is

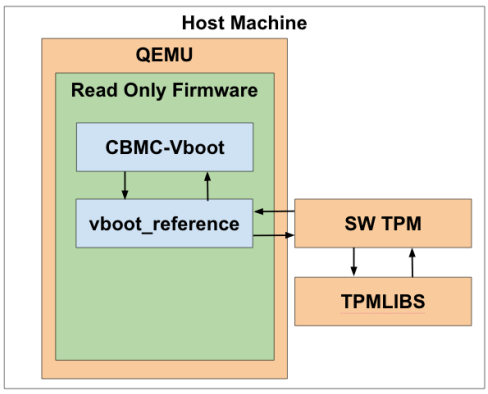

Figure 3.8: This project's boot flow uses a created C wrapper for interfacing with QEMU and the TPM, and an unmodified Vboot for the verification

responsible for initializing the more specialized drivers, and the concept of a payload means that Google can keep support more hardware without altering the Coreboot source code.

The payload that Coreboot calls to further initialize the Chromebook is Depthcharge. Depthcharge contains platform specific functions like reading and writing commands to the TPM, or interacting with the SHA accelerator. The vboot_reference library will make calls back into Coreboot for platform specific access. As this report was run on an emulated Chromebook and not a real platform, the Coreboot + Depthcharge repositories were replaced by new code in the CMBC-Vboot repo.

3.5.2 Vboot_reference

The vboot_reference repo contains all of the control and algorithms for the vboot process [30]. The repo is designed such that it does not rely on any knowledge about the platform. If a function requires usage of a driver or something that is board specific, it will make a callback into Depthcharge which will provide the relevant information. Again, for this report, Depthcharge will be replaced by the CBMC-Vboot repository.

One of the more helpful features of the vboot_reference repo is that it can be built in a stand-alone fashion as a C archive file. More information about building the project can be found in the appendix, but this feature allowed the Vboot functions to be placed into the QEMU environment easily and be adapted for many different types of hardware.

3.5.3 Software TPM

The Software TPM was taken from a series of repositories created by IBM's Stefan Berger [31]. These repositories emulated the TPM functions [32], the TPM Command Fifo [31], and the TPM Register Interface [33]. appendix.

This emulated TPM implements all of the TPM's functionality as defined by the TCG Standards. Software emulation was desirable over a physical chip for many reasons. First, for security reasons, the state of a hardware TPM is difficult to modify. If testing required forcing the TPM into a particular state, it may involve manual power restarts and multiple calls to the TPM library. The Software TPM allows state to be saved, restored, and modified on the fly without going through the TPM API. Second, a Hardware TPM's operations would be a black box. The Software TPM's code is open source and can be inspected manually.

When the Software TPM is passed to QEMU, it is accessed through the Registers outlined in Section 2.2. This is the same way that the Vboot TPM library accesses the TPM, so these functions can be used unmodified.

3.5.4 CBMC-Vboot

The CBMC-Vboot is a repository created to connect Vboot to the emulated hardware used in this project [34]. The layout of this repository is discussed in the Appendix. We can see in Figure 3.8 that it is used as a wrapper to tie Vboot together with QEMU and the Software TPM libraries.

Chapter 4

Verification Setup

This section will discuss the security properties used by the Read-Only section of Verified Boot. It will also verify these properties using the Hardware and Software elements that were outlined in Section 2 and Section 3. All properties are checked using the C Bounded Model Checker (CBMC). To check some properties, Hardware modules are converted into C using the ILA toolchain.

4.1 Vboot Properties

The properties that will be verified can be separated into 3 categories: array accesses, program flow, and correctness. The array access property restricts the writeable addresses in memory to valid array boundaries. Program flow properties refer to the program's execution path. Correctness properties refer to facts about the code such that, if things were designed successfully, certain states should not be reached.

These properties will be specified in Computation Tree Logic (CTL). CTL adds temporal distinctions over propositional logic and this allows one to easily describe how a system changes over time. The boolean connectives in propositional logic are $\neg$ for NOT, $\wedge$ for AND, $\vee$ for OR, and $p \rightarrow q$ for p implies q. CTL describes how propositional logic changes through time. Table 4.1 describes the CTL operators and syntax.

4.1.1 Memory Access Locations

Memory accesses in C are very important because C allows for arbitrary addresses to be accessed. Addressing arrays before their starting address or after their final address is known as a buffer overrun and it is one of the most common security vulnerabilities.

Table 4.1: A description of the CTL syntax

Syntax	Description
Xp	p is true in the next state
Fp	p is true eventually
Gp	p is always true

Buffer overruns allow a program to write or read to arbitrary memory. In Vboot, a buffer overrun could be used to modify the RSA public keys after they have been read in from secure storage, or it could modify the firmware image after it has been verified as secure. Such modifications would defeat the security claims of Vboot and for this reason it is important to prevent the accessing of arrays past the array boundaries.

The property below describes correct array accesses in CTL.

$$G(a \rightarrow (i > 0 \wedge i < size(ptr))) \tag{4.1}$$

In this property a is any array access, ptr is the start of the array and i is the access offset. This property states that for every memory access, the index is greater than 0 so it cannot access memory before the array, and the index is smaller than the size of the array so memory after the array cannot be accessed.

Array indexing is common enough that CBMC provides checking by default. However, this check on its own is not strong enough to provide array access security for Vboot. This is because Vboot uses pointer manipulation in order to access structures and data within the Image. Vboot reads the Image from disk as one contiguous binary. After the Image has been loaded, the data structures within the Image use size and offset variables to locate more data. CBMC is not able to automatically detect if this pointer manipulation results in an access outside of the Image.

The following property defines correct pointer manipulation in Vboot.

$$G(\text{offset(struct)} + \text{size(struct)} < \text{size(img)} \wedge \text{offset(struct)} > 0) \tag{4.2}$$

In this property *struct* is the structure being created from a pointer dereference, and *img* is the image that it is being pointed to. The property states that the offset and size of the struct do not go past the edge of the image and that the offset of the struct does is not negative. We can assume that the offset is not negative because Vboot offsets should only point forward in the Image.

Together these two properties state that array accesses must be in bounds and that structure dereferencing must be in bounds of the image that is being referenced. As these are the two memory referencing techniques that have the potential for overrun, these properties protect the system against overrun attacks.

4.1.2 Program Flow properties

There needs to be an ordered constraint on the flow of the program. If the program flow is described formally then it will be easy to check that all stages of secure boot were called and in the correct order. This will catch incorrect programs that skip steps and therefore skip verifying the image, or programs that execute steps out of order and therefore access data that has not been verified yet. The graph for program flow is shown in Figure 4.1. When verifying the flow of Vboot it is necessary to note that there are two Images stored in the system, Image A and Image B, and either can be used in the Vboot process. It is also necessary to note that Developer mode will skip a step in the verification process, and that this skipped step is a valid part of the Vboot flow.

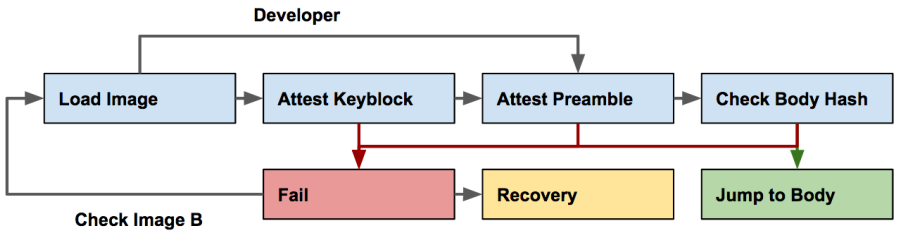

Figure 4.1: The logical flow for Vboot.

Let s_i correspond to the ith stage from Figure 4.1 and s_{ai} or s_{bi} correspond to that stage for Image A or Image B respectively.

The property below states that at any time, we must be in at least one of five stages for either image A or image B.

$$G(\bigvee_{i=0}^{4} s_{ai} \vee s_{bi}) \tag{4.3}$$

The formula below states the transitions available for Image A. Each i$^{\text{th}}$ stage can transition either to itself or the following stage. Image A has the ability to fail and transition immediately to the first stage of attesting Image B.

$$G(s_{ai} \rightarrow (s_{ai} \vee s_{ai+1} \vee s_{b0})) \tag{4.4}$$

Now we will describe the state transitions for Image B. They are similar to Image A but Image B is always tried second and can only transition to the next stage. Image B cannot transition back to verifying Image A or it would be possible to have an infinite loop of attesting incorrect images.

$$G(s_{bi} \rightarrow (s_{bi} \vee s_{bi+1})) \tag{4.5}$$

The final property refines the transition for state s_0 for Image A and B. In Figure 4.1 we can see that state s_0 can transition to s_1 but it also has a special transition to s_2 that is only valid if Developer Mode is enabled.

$$G(s_0 \rightarrow (s_0 \vee s_1 \vee (M_D \wedge s_2))) \tag{4.6}$$

All together these properties fully describe the program flow graph in Figure 4.1. If these hold then steps will not be skipped in the verification process and we can be assured that verification of the image will be attempted in order.

4.1.3 Correctness properties

Correctness properties refer to the code itself and help to determine if it has been written correctly. For Verified Boot, correctness properties focus on the correct attestation of the image and the guarantee that an incorrect Image cannot be marked as correct and loaded for execution.

The following property states that if the system is not in developer mode and the signature verification fails, then the system will eventually reach the failure state for that image. Here *img* can refer to either Image A or Image B. It is important that failure and passing is tracked on an image basis because it is possible for Image A to fail but Image B to pass.

$$\neg M_d \wedge \neg \text{verifySignature(img, rootKey)} \rightarrow XF(\text{img.pass} = F) \tag{4.7}$$

The following property states that if the hash of the image data does not match the hash stored in the image, then that specific image will eventually fail.

$$\text{hash(img.data)} \neq \text{img.hash} \rightarrow XF(\text{img.pass} = F) \tag{4.8}$$

The next property checks against the rollback attack. Rollback states that all image versions must be greater than or equal to the last seen image version that is kept in secure storage. This property claims that if the image version is lower than what is seen in secure storage, then the image will eventually fail. Rollback is disabled on Recovery Mode and the property below reflects this fact.

$$\neg M_r \wedge (\text{img.version} < \text{prevVersion}) \rightarrow XF(\text{img.pass} = F) \tag{4.9}$$

The next property checks that if both images fail, then the Vboot process will fail and the system will eventually reboot into recovery mode.

$$\neg \text{imgA.pass} \wedge \neg \text{imgB.pass} \rightarrow XF(\text{pass} = F) \tag{4.10}$$

In the formulas above, img refers to the Image to be verified (either Image A or Image B) and rootKey is Google's public key that has been loaded out of the GBB. M_d refers to developer mode being active and M_r refers to recovery mode being active. Img.version is the Image version number and prevVersion is the last seen version that is stored in the TPM.

$$(a \rightarrow b) = (\neg a \rightarrow \neg b) \tag{4.11}$$

The contrapositive of these properties will be proved. The contrapositive of a conditional is the inverse of the condition where the antecedent and consequent are negated. For example, instead of proving that "if the hash fails then the image will fail" it will be proved that "if the image passes then the hash must have passed". The formula for a contrapositive is expressed in Equation 4.11. The contrapositive is easier to check through software assertions and is logically equivalent to the original property.

4.2 CBMC

CBMC is a Bounded Model Checking program for the C language that is released by Carnegie Mellon. A Bounded Model Checker is a tool that performs Formal Verification. Model checking is a way to exhaustively prove whether a given model meets its specification. Model checking typically uses propositional logic or temporal logic. At its core, CBMC transforms a C program into binary logic and then uses a SAT solver to check it against the user's specification. The specification is user defined, using an API of C functions defined by the CBMC library.

The CBMC API is very simple. The list below describes the 3 functions in the CBMC API.

- `assert(bool e)` – takes in a boolean expression and throws exception if false.
- `nondet_int()` – returns a non-deterministic integer.
- `assume(bool e)` – takes in a boolean expression and forces any non-deterministic variables to conform.

If the `assert()` function throws an exception then CBMC will produce a "counterexample." A counterexample lists the program inputs and steps taken that caused the false assertion. The `nondet_int()` function is very helpful to model user input to the Vboot process. Any data that could be modified by the user could feasibly take any value, and this is represented cleanly by a non-deterministic integer. CBMC exhaustively checks user assertions against all possible non-deterministic integer values.

4.3 Abstractions

CBMC was able to be successfully run on portions of the verified boot library and provide information on the satisfiability of various assertions. Running CBMC on the Read-Only section of Vboot with a full Firmware Image is not possible as the program will consume the full 126 GB of RAM on the computing sever and then be killed. In order to check properties, abstractions needed to be found such that parts of the Vboot flow could be checked at a single time.

4.3.1 Function Over-Abstractions

The key abstraction used is that individual functions are self contained. This leads to a natural abstraction at the function definition. Each Vboot function returns 0 if successful and an error code if it is not successful. When a function is abstracted away, it is programmed to return a non-deterministic value. If the non-deterministic value is zero (success), then the minimal amount of external changes (if any) should be applied to the larger function. If the abstracted function returns non-zero (error), then the external changes should be replaced with non-deterministic values.

This method is an over-abstraction of a functions found in the Vboot library. An over-abstraction contains the full set of states found in the original function, plus

additional states that would not be possible. Because the over-abstraction is a superset of the original states, any properties that are proved with the over-abstraction will also be proved on every state of the original function. Therefore any properties proved with abstracted functions will also hold if the real functions were left in place.

4.3.2 Loop Removal

Another helpful abstraction is to remove long loops that are present in Verified Boot. CMBC performs loop unrolling before it performs any model checking. In a program with long loops, the unrolling adds unnecessary complexity and state to the model and it increases checking time dramatically. All of Vboots long loops are comparing or copying arrays. An array compare can be abstracted into a single non-deterministic read on both arrays. An array copy can be abstracted into a single non-deterministic write on the array. Because the index into the array is non-deterministic in both cases, CBMC will still perform verification on every acceptable index and find possible counterexamples. This abstraction was taken from an earlier thesis that also performed Formal Verification [6].

4.3.3 Memory Allocation

The final abstraction is the creation of a simple memory allocation function so CBMC will handle pointer manipulation correctly. Vboot data structures contain offsets to other data structures, not full pointers. Vboot uses offsets because the data structures are loaded from disk and there is no guarantee that the data will be loaded into a specific address. (There is however, a guarantee that the data will be loaded contiguously.) The following code shows the simple pointer arithmetic needed for offsets.

```
uint32_t offset = &data - &image;
void *data_ptr = &image + offset;
```
(4.12)

However, CBMC will not perform pointer arithmetic unless the original pointer was set to a specific value. A small memory management function was created to assign pointer values to data structures so CBMC would perform the correct arithmetic. In this abstraction memory addresses start at 0x0 and increase as needed.

Memory is never reclaimed. The idea for this abstraction originated from the work done in modeling memory by Franjo Ivancic [35].

4.3.4 Google's Assertions

Google has already created a Unit Test Framework for the Verified Boot process. This framework tests various properties of each function. The testing framework also relied on propositional logic and assert statements. With minor changes to the framework each test could also be verified using Formal Verification tools.

The verification work done in this paper is still strictly better than the existing tests. The existing assertions are now proven formally on any input, and more advanced properties could be proven. Google's framework also stubs out any calls to hardware, meaning that it will not catch any errors across the Hardware Software boundary.

Chapter 5

Verification Results

This section describes the verification tests on Vboot using CBMC. Each subsection begins with a diagram showing the functional call graph of the section of code being verified. The diagrams are color-coordinated as followed. The blue functions are fully software functions that are implemented in the Vboot Library. The purple functions connect to the SHA accelerator on a real platform, but use a software abstraction for verification. The red functions connect to the TPM on a real platform, and are verified on either the TPM ILA or the TPM C abstraction.

The following sections focus on verification at the function level. The sections are organized so the larger functions are in the beginning and the smaller, more specific functions are at the end. The function being verified in a specific section is left unmodified, but some of the connecting functions are replaced with abstractions. These abstractions all follow the idea of an "over-abstracted" function discussed in the last section. These abstractions are acceptable because the real functions are also verified in later sections. The abstracted functions can be seen in the code repo [34].

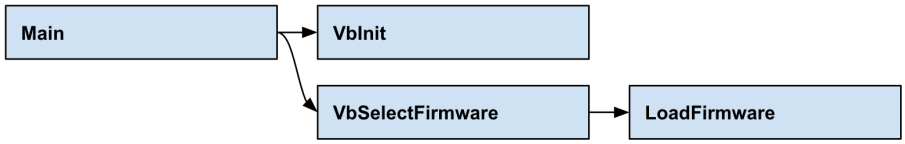

Figure 5.1: The general call graph of functions for Vboot. A full overview is available in the Appendix.

At the end of each section there will be a table describing the results of each CBMC run. CMBC uses an implementation of a SAT solver known as the MiniSat solver [36]. This solver outputs unSAT if the assertions hold and SAT if they do not. The "steps" column is the number of C lines that were stepped through to reach

the proof conclusion. The "VCC" column is the number of user assertions that were being verified. The "time" column is the total clock time that the verification ran for. The "memory" column is the maximum memory usage in Megabytes.

All results were gathered on Princeton's SAT8 server. The SAT8 server contains an Intel Xeon E5645 CPU and has 128 Gigabytes of RAM.

5.1 VbSelectFirmware

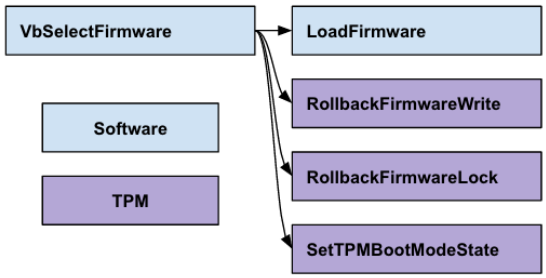

Figure 5.2: The call graph of functions for VbSelectFirmware

The VbSelectFirmware function mostly acts as a wrapper to LoadFirmware. It is responsible for loading data and accessing the TPM. VbSelectFirmware is responsible for loading a chosen Image's version number into the TPM and locking the TPM's data. This functionality is what prevents rollback attacks across multiple boots. VbSelectFirmware is also responsible for loading the TPM's PCR0 with a hash of the system's boot flags. This functionality prevents the system from changing boot flags mid-boot without detection.

Here are the properties verified on VbSelectFirmware:

- TPM locking – checks that it is not possible for VbSelectFirmware to return success if it is unable to lock the TPM
- LoadFirmware – checks that it is not possible for control to pass to an Image if it is not selected by LoadFirmware
- Array accesses – check for all possibilities of array bounds errors.

The CBMC checks run successfully and in a short time. The VbSelectFirmware function does little work on its own, so there is not much to verify. As we will see in the next section, most of the work is done in LoadFirmware.

Table 5.1: Verifying the VbSelectFirmware function

test name	steps	VCCs	time (s)	memory (Mb)	sat/unsat
TPM locking	5269	6	2.42	164	unsat
LoadFirmware	5264	2	2.64	164	unsat
Array Accesses	5257	10	2.67	165	unsat

5.2 LoadFirmware

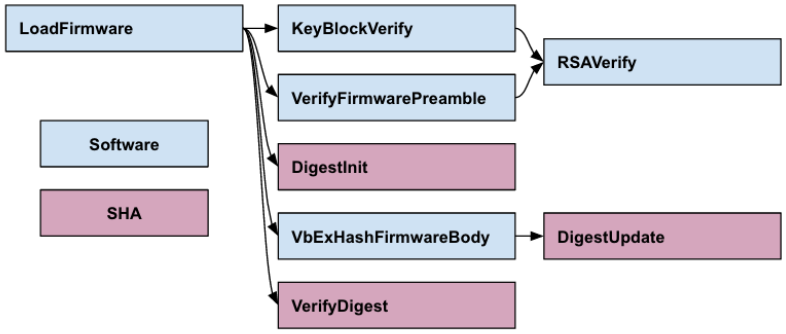

Figure 5.3: The call graph of functions for LoadFirmware

LoadFirmware drives most of Vboot's Image attestation process. LoadFirmware is responsible for attesting the Keyblock, Preamble, and Image. When LoadFirmware returns, it will have set the pass or fail statuses of both Images, and control will pass to a working Image.

The CBMC verification results for this function can be seen in Table 5.2. The following is an itemized list of what each check is verifying.

- Google Assertions – this check replaces Google's assertions with CBMC assertions and checks that none of them are violated under any inputs
- Rollback test – checks that under no input conditions will LoadFirmware choose an Image with a lower version number (see Property 4.9).
- Rollback/rec – this check is supposed to find a counter-example. It checks for rollback conditions without realizing that rollback is allowed under recovery.
- Hash Failure – checks that it is impossible to verify an image as correct if hashing the Image Body returns an error (see Property 4.8).

- RSA Failure – checks that is impossible to verify an image as correct if verifying either the Keyblock or Preamble returns an error (see Property 4.7).
- Array accesses – checks for all possibilities of array bounds errors.

Table 5.2: Verifying the LoadFirmware function

test name	steps	VCCs	time (s)	memory (Mb)	sat/unsat
Google Assertions	6460	34	123	2312	sat
Rollback	4125	1	4.27	259	unsat
Rollback/rec	4123	1	4.68	394	sat
Hash Failure	4182	2	4.45	258	unsat
RSA Failure	4180	2	4.45	258	unsat
Array Accesses	4130	57	126	2487	unsat

Table 5.3: Verifying the LoadFirmware function (with memory addressing)

test name	steps	VCCs	time (s)	memory (Mb)	sat/unsat
Google Assertions	6460	34	3799	60496	unsat
Rollback, Hash, RSA	2858	6	3733	59874	unsat

We can see from these checks that CBMC is working as expected. The counter example of the "rollback/rec" check quickly pointed the user to the counterexample where the Recovery Mode allowed rollback. All of the assertions were checked in a short time without the memory allocating technique outlined in Section 4.1.1. However, CBMC was able to find counterexamples to the Google Assertions because array addressing was not working correctly. The counterexamples involved Preamble and Keyblock addresses that were outside the bounds of the original Image. These addresses allowed Vboot to behave insecurely because the array addressing was not working correctly. However, we can see that Vboot checks for this when the memory addressing technique is applied in Table 5.3

Table 5.3 shows the same checks with the memory allocating technique implemented. We can see that CBMC takes much longer to prove the assertions, but the Google Assertions are working correctly. Without the memory allocation technique, it would be reasonable to assume that a generic personal computer could run Formal Verification with CBMC. The first checks finished in under two minutes and the most memory used was roughly 2 Gb of RAM. However, with CBMC, the checks take up to an hour and use up to 60 Gb of RAM. It is extremely unlikely that a personal

computer would have 60 Gb of RAM, meaning that CBMC can only be run on a large server for this type of verification.

5.3 TPM TLCL functions

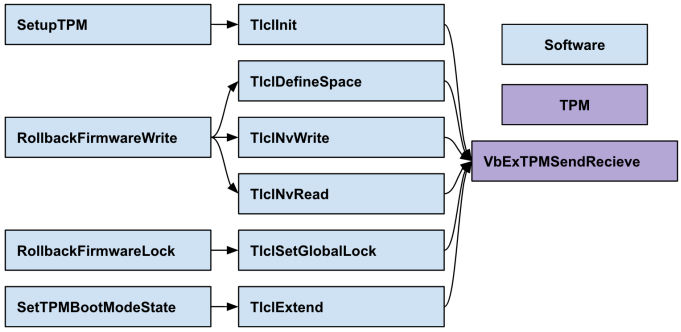

Figure 5.4: This shows the Vboot call graph for the TPM functionality

The TPM Lightweight Command Library (TLCL) is Google's library for building and sending TPM commands. Each function corresponds to a single TPM command and a single call to the TPMSendReceive function. These checks verify correct functionality for the TLCL commands that are used by Vboot.

At this point, the TPM abstraction created in Section 2.2 is added to the verification program. The TPM abstraction connects to Vboot as described in Section 2.6.1.

- Error Handling – the TPM returns success only if the TPM command completes successfully
- SetupTPM – the TPM is initialized successfully with well-defined states
- RollbackFirmwareWrite – the RollbackFirmwareWrite function sends the write value to the TPM unmodified and no TPM assertions are broken
- RollbackFirmwareLock – the RollbackFirmwareLock function will always lock the correct section and no TPM assertions are broken
- SetTPMBootModeState – the SetTPMBootModeState function sends the PCR value to the TPM unmodified and no TPM assertions are broken

The longest check run in this section finishes in thirty minutes and uses 6.7 Gb of RAM. It is likely that these checks could be run on a high-end laptop. All checks completed successfully. In addition to the individual check's assertions, the checks also include the TPM assertions outlined in the next section.

Table 5.4: Verifying the TLCL Library

test name	steps	VCCs	time (s)	memory (Mb)	sat/unsat
Error Handling	34639	10	11	1579	unsat
SetupTPM	613418	140	1674	6960	unsat
RollbackFirmwareWrite	478639	108	952	5447	unsat
RollbackFirmwareLock	17450	4	7	369	unsat
SetTPMBootModeState	34621	8	11	397	unsat

5.4 TPM Send and Receive

Each TLCL function passes through the single function TpmSendReceive. TpmSendReceive sends data to and from the TPM with the TPM command registers outlined in Section 2.2.1. The TpmSendReceive function must be verified against all possible inputs to ensure that the TPM cannot be put into an unwanted state. The assertions outlined in this section are also verified against the TLCL library functions in the previous section. While the previous section verified TPM correctness during well-defined commands, this section verifies the TPM against sending and receiving random data.

- Liveness – Checks that TpmSendReceive will not enter a state where it polls on an unchanging TPM register
- Data Send – Checks that TpmSendReceive will only return success if the TPM has read each sent byte successfully
- Data Receive – Checks that TpmSendReceive will not return if the TPM still has data to send

Table 5.5: Verifying the VbExTpmSendReceive function

test name	steps	VCCs	time (s)	memory (Mb)	sat/unsat
Liveness	20369	2217	359	56434	unsat
Data Send	21703	259	34	3991	unsat
Data Receive	20231	2	391	61040	unsat

Again, everything is verified successfully. The longest command finished in 6 minutes and used 15 Gb of RAM. These commands would likely have to be run on a high-end Desktop or Server because of their high RAM usage.

The Liveness property was verified in a unique way compared to other properties in this report. The Liveness property ensures that the TPM firmware will not be stuck in an infinite loop. An infinite loop will result if the firmware polls a TPM register that will never change, or if the TPM returns an infinite amount of data. These situations are checked against using CBMC's loop unrolling. Each `while` loop is unrolled a pre-defined number of times. CBMC inserts assertions that the loops will not be accessed more than their unroll limit. Because the loop unrolling assertions hold for every possible input, we can be sure that each loop is run a finite number of times.

Chapter 6

Security Recommendations

This chapter uses the results in the last chapter to provide security recommendations for Verified Boot. While no security holes were found in Verified Boot, specific additions to the Vboot process would reduce the surface of attack in the future. Reducing the surface of attack would also allow other, stronger properties to be proven.

6.1 Memory Locking

Adding hardware support for Memory Locking would greatly improve Verified Boot. Memory Locking is the ability to lock sections of RAM as Read-Only. This feature would help the security of Verified Boot by isolating it from any external repositories. As mentioned in the Software chapter, Verified Boot makes function calls into the Coreboot and Depthcharge repositories. If the Firmware Image is not locked before these function calls are made, then it is possible that the image is corrupted before the control returns back to Verified Boot. Additionally, Coreboot and Depthcharge are in charge of the SoC's interrupts, so it is possible that the interrupt handlers could manipulate the Vboot data structures at any time.

At the moment Coreboot and Depthcharge must be trusted. However, it would be beneficial to remove the assumption that they will not interfere with the boot process. The Coreboot and Depthcharge repositories together contain 415,035 lines. For comparison the Vboot repository contains only 16,000 lines. As the other projects are an order of magnitude larger, it is unlikely that they can be verified in the same manner.

The flow in Image 6.1 shows an example of how the hardware locking would be implemented. This would prevent any Time of Check Time of Use (TOCTOU) Vulnerabilities that would exist in the Verified Boot library. A TOCTOU attack is

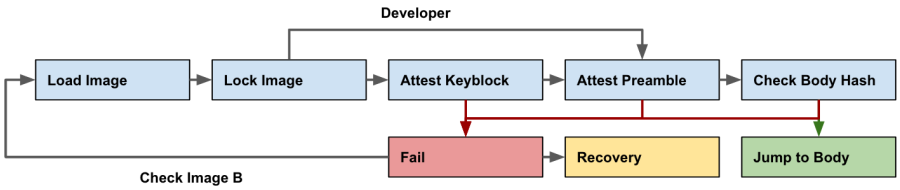

Figure 6.1: The Vboot flow for locking the Image.

when the attacker modifies data after it has been checked, but before it is being used. In Vboot's case a TOCTOU attack would modify the Image after it has been attested, but before Vboot jumps to the start of the Image's code.

TOCTOU vulnerabilities are very common in secure programs [37]. Implementing memory locking would make it trivial to prove that the system has no TOCTOU attack vectors. Until this feature is implemented, there will always be the possibility of a TOCTOU attack. This recommendation was brought to the attention of Google Engineer's through the Chromium-OS email group. Their response was that it is possible to implement under current hardware, and it was recommended under NIST's "Bios Protection Guidelines," but it has not been implemented yet [38]. There are examples of Secure Boot implementations that enable memory locking [6].

Chapter 7

Conclusions

This paper investigated the feasibility of formally verifying a large scale system. Google's Verified Boot program was shown to be a good candidate for formal verification because of its security guarantees and its interactions across the Hardware-Software boundary. VBoot's security claims were expressed formally using CTL and were verified against the C implementation using the model checker CMBC. Model checking abstractions were discussed, such as abstracted functions, memory allocation, and non-deterministic array writes. These abstractions were necessary for Formal Verification tools to produce results on the Verified Boot software. It was observed that the high RAM usage of CBMC prevents verification from being run on personal computers.

7.1 Future Work

The analysis on Verified Boot is helpful in the way that it can be applied to other systems. Verified Boot was chosen because it is a widely used code base that crosses the Hardware-Software boundary. However other systems exhibit more interesting behavior like concurrent hardware accesses.

Verified Boot uses hardware, but there are very little opportunities for problems involving hardware concurrency. Other systems include interweaving uses of non-blocking Hardware[8] and applying the ILA toolchain to these systems would showcase more strengths of the ILA.

Another direction for the ILA toolchain that has been considered but not explored is its use to verify that two hardware modules have the same functionality. This type of verification is important as a TPM can be produced by various Hardware Vendors [39][40][41], and the different implementations have been found to vary

significantly[42]. A full ILA description of a TPM would be very useful for comparing two hardware chips from different vendors, as well as verifying that a software implementation matches the Hardware implementation. Once the entire TPM functionality exists in an ILA description, more verification could be done on the TPM API itself [43][44].

Additionally, there are more areas in Google's Verified Boot that deserve more attention. The Read-Write section of Verified Boot is ignored because it shares the majority of its code with the Read-Only section, but it is still worth verifying. The Update functionality of Verified Boot is also be worth investigating. Problems in the Update functionality will not affect the properties outlined in this thesis. This report already assumes that an attacker can upload arbitrary updates.

Appendix A

Code Organization

The code for this thesis is based off of Google's Verified Boot reference repository [30] and the modifications can be accessed in a cloned and edited repository [34]. Changes to the Verified Boot repo were mostly restricted to the main Makefile and the unit test folder `tests/`.

In order to have the results reproducible, all work was done inside of Vagrant Virtual Machine running Fedora 23. To get a working environment, clone the repository, initialize the Vagrant box, and run the following command:

```
$PATH/CBMC-Vboot/scripts/install-fedora.sh
```
(A.1)

This script will install the 49 packages required to run all of the code. Next, restart the Vagrant box and run the initialization script with:

```
$PATH/CBMC-Vboot/scripts/init-fedora.sh
```
(A.2)

This script will compile the 6 packages that need specific configurations.

The installation process takes roughly an hour on my 2015 Macbook Pro and the resulting VM image requires 4 Gbs of Hard Drive space.

The `scripts/` folder contains the configuration scripts. The main scripts are for installation, but there are also scripts to manage the SW TPM library, and to build a Vboot Image with related Public/Private keys.

The `qemu/` folder contains all of the code to run Vboot within QEMU with the hardware setup described in Section 2.5. The `tpm-pcr.c` file tests only the TPM PCR functionality. The `tpm-badcmds.c` file stress tests the TPM behavior with bad commands. The `vboot-full.c` goes through the full process of Chrome's Verified

Boot. The `README.md` file contains information on building the software and using GDB testing.

The `CBMC/` folder contains all of the code for running Formal Verification on Vboot. The python files with the naming structure `run_{$FILE}.py` will build and run the CBMC command for `$FILE`. The C files with the naming structure `test_{$FILE}.c` contain the main code being tested by CBMC. The `README.md` file contains information on repeating the tests outlined in this report.

A.1 CBMC Command Builder

The code below shows the python script that builds a CBMC command for Formal Verification.

```
1  import os
2
3  # the location of Vboot library
4  vboot_dir = "../_vboot_reference/"
5
6  # all Vboot directories with relevant .h files
7  includedDirs = [
8    vboot_dir + "host/lib/include",
9    vboot_dir + "firmware/include/",
10   vboot_dir + "firmware/lib/cryptolib/include/",
11   vboot_dir + "firmware/lib/include/",
12   vboot_dir + "tests"
13   ]
14
15 # The main file being tested
16 testFile = "test_keyblockVerify.c"
17
18 # all Vboot files that need to be included
19 includedFiles = [
20   # include file below to do Google's unit test assertions
21   # "test_common.c",
22
23   vboot_dir + "firmware/stub/utility_stub.c",
24   vboot_dir + "firmware/lib/vboot_common.c",
25   vboot_dir + "firmware/lib/vboot_nvstorage.c",
26   vboot_dir + "firmware/lib/vboot_common_init.c",
27   vboot_dir + "firmware/lib/crc8.c",
28   vboot_dir + "firmware/lib/vboot_firmware.c",
29   vboot_dir + "firmware/lib/cryptolib/rsa_utility.c",
30   vboot_dir + "firmware/lib/utility.c"
31   ]
32
33 extras = [
34   " -D NONDET_VARS",
35   " -D CBMC",
36   "--little-endian", "--64"
37   ]
38
39 #-------------------------------------------------------------
40 # Build and Run the command
41 #-------------------------------------------------------------
42 commandString = "cbmc "
43 for s in includedDirs:
44   commandString += "-I " + s + " "
45 commandString += testFile + " "
46 for s in includedFiles:
47   commandString += s + " "
48 for s in extras:
49   commandString += s + " "
```

```
50
51  print (commandString)
52  os.system(commandString)
```

A.2 TPM Firmware

This code shows the TPM read and write functionality.

```
1   #include <stdint.h>
2   #include "tlcl.h"
3   #include "utility.h"
4   #include "vboot_api.h"
5   #include "util.h"
6
7   /* macros to access registers at locality ''l'' */
8   #define ACCESS(l)                          (0x0000 | ((l) << 12))
9   #define STS(l)                             (0x0018 | ((l) << 12))
10  #define DATA_FIFO(l)                       (0x0024 | ((l) << 12))
11  #define DID_VID(l)                         (0x0F00 | ((l) << 12))
12  /* access bits */
13  #define ACCESS_ACTIVE_LOCALITY             0x20 /* (R)*/
14  #define ACCESS_RELINQUISH_LOCALITY         0x20 /* (W) */
15  #define ACCESS_REQUEST_USE                 0x02 /* (W) */
16  /* status bits */
17  #define STS_VALID                          0x80 /* (R) */
18  #define STS_COMMAND_READY                  0x40 /* (R) */
19  #define STS_DATA_AVAIL                     0x10 /* (R) */
20  #define STS_DATA_EXPECT                    0x08 /* (R) */
21  #define STS_GO                             0x20 /* (W) */
22
23  /* TPM main address */
24  #define ADDR FED40000
25
26  int locality = 0;
27
28  /*
29   * Sends len number of bytes over to the TPM.
30   * Returns the number of bytes it sent succcessfully
31   */
32  uint32_t send(const unsigned char *buf, uint32_t len) {
33    int status, burstcnt = 0;
34    uint32_t count = 0;
35
36    // tell the TPM that we will be writing a command now
37    write8(STS_COMMAND_READY, STS(locality));
38
39    // loop to write command
40    while (count < len - 1) {
41      // get the burst count
42      burstcnt = readBurstCount();
43
44      if (burstcnt == 0){
45        delay(); /* wait for FIFO to drain */
46      } else {
47        // write (len - 1) data to the TPM
48        for (; burstcnt > 0 && count < len - 1; burstcnt--) {
49          write8(buf[count],
50              DATA_FIFO(locality));
51          count++;
52        }
53
54        // wait while we are not in a valid state (overflow)
55        while ( ( (status = read8(STS(locality)))
56              & STS_VALID) == 0) {}
57
58        // break if the TPM is not expecting more data
59        if ((status & STS_DATA_EXPECT) == 0)
60          return -1;
61      }
```

```
62    }
63
64    /* write last byte */
65    write8(buf[count], DATA_FIFO(locality));
66
67    // wait until valid state
68    while ( ( (status = read8(STS(locality)))
69          & STS_VALID) == 0) {}
70    // check that no more data is expected
71    if ((status & STS_DATA_EXPECT) != 0)
72      return -1;
73
74    // Tell the TPM to execute the command
75    write8(STS_GO, STS(locality));
76
77    return len;
78  }
79
80  // Receive helper function
81  // It will copy received data into buf.
82  // It *attempts* to read count bytes but it
83  // will return early if the TPM has no data
84  // Returns: number of bytes received
85  int recv_helper(unsigned char *buf, int count) {
86    int size = 0, burstcnt = 0;
87
88    while (is_data_aval(locality) && size < count) {
89      // Get the burst count (amount we can read at once)
90      if (burstcnt == 0)
91        burstcnt = readBurstCount();
92
93      // if burst count is zero then there is no data to read
94      if (burstcnt == 0)
95        delay();
96      // otherwise read the data
97      else {
98        for (; burstcnt > 0 && size < count; burstcnt--) {
99          buf[size] = read8(DATA_FIFO(locality));
100         size++;
101       }
102     }
103   }
104   return size;
105 }
106
107 /*
108  * Receive takes a buffer of size count.
109  * It receives one command return from the TPM,
110  * as long as command_size < count
111  */
112 int recv(unsigned char *buf, int count) {
113     int command_size;
114     int size = 0;
115
116     if (count < 6)
117       return 0;
118
119     // Check that data is available
120     if (!is_data_aval(locality))
121       return -2;
122
123     // Read first 6 bytes
124     // (All commands are larger than 6 bytes and
125     //   these bytes include the command size
126     //   and the tpm tag)
127     if ((size = recv_helper(buf, 6)) < 6)
128       return -1;
129
130     // Get the command size
131     command_size = (*(buf + 5));
132     if (command_size > count)
133       return -1;
134
135     /* read all data, except last byte */
```

```
136       if ((size += recv_helper(&buf[6], command_size - 6 - 1))
137           < command_size - 1)
138         return -1;
139
140       /* check for receive underflow */
141       if (!is_data_aval(locality))
142         return -1;
143
144       /* read last byte */
145       if ((size += recv_helper(&buf[size], 1)) != command_size)
146         return -1;
147
148       // Make sure no data is left
149       if (is_data_aval(locality))
150         return -1;
151
152       write8(STS_COMMAND_READY, STS(locality));
153       return 0;
154     }
```

A.3 TPM ILA

This code shows the ILA for the TPM Hardware

```
1  import ila
2  import fifo_def
3  from fifo_sim import fifo
4
5  # Helper function to synthesize a single state
6  def synth(m, state, sim):
7    print "Synthesizing: " + state
8    m.synthesize(state, sim)
9    ast = m.get_next(state)
10   m.exportOne(ast, "ast/" + state + ".ast")
11
12 # create and synthesize the entire TPM ILA
13 def createTPMILA():
14   m = ila.Abstraction("tpm")
15
16   # ------------------------------------------------------------
17   # Inputs
18   # ------------------------------------------------------------
19   cmd     = m.inp("cmd", 3)
20   cmdaddr = m.inp("cmdaddr", 64)
21   cmddata = m.inp("cmddata", 8)
22
23   # ------------------------------------------------------------
24   # Constants
25   # ------------------------------------------------------------
26   ZERO = m.const(0x0, 8)
27   ONE = m.const(0x1, 8)
28   THIRTY = m.const(0x1e, 8)
29
30   # These are the flags that status can output
31   STS_VALID             = m.const(fifo_def.STS_VALID, 8)
32   STS_DATA_AVAIL        = m.const(fifo_def.STS_DATA_AVAIL, 8)
33   STS_DATA_EXPECT       = m.const(fifo_def.STS_DATA_EXPECT, 8)
34
35   # these are the commands that can be written to status
36   STS_GO                = m.const(fifo_def.STS_GO, 8)
37   STS_COMMAND_READY     = m.const(fifo_def.STS_COMMAND_READY, 8)
38
39   # ------------------------------------------------------------
40   # Variable Definitions
41   # ------------------------------------------------------------
42   # Fifo State
43   fifo_state = m.reg("fifo_state", 8)
44   m.set_next("fifo_state", ila.choice("fifo_state_choice",
45     [ZERO, ONE, ONE+1, ONE+2, ONE+3]))
```

```
46
47      # Status register
48      fifo_sts = m.reg("fifo_sts", 8)
49      m.set_next("fifo_sts", ila.choice("fifo_sts_choice",
50        [STS_VALID, STS_VALID | STS_DATA_AVAIL, STS_VALID | STS_DATA_EXPECT, ZERO]))
51
52
53      # internal index to the FIFO,
54      # is amount written so far
55      fifo_in_amt  = m.reg("fifo_in_amt", 8)
56      m.set_next("fifo_in_amt", ila.choice("fifo_in_amt_choice",
57        [fifo_in_amt, fifo_in_amt+1, ZERO]))
58
59      fifo_in_cmdsize = m.reg("fifo_in_cmdsize", 8)
60      m.set_next("fifo_in_cmdsize", ila.choice("fifo_in_cmdsize_choice",
61        [fifo_in_cmdsize, cmddata, ZERO]))
62
63      # the fifo memory.
64      # 256 8 bit registers
65      fifo_indata = m.mem("fifo_indata", 8, 8)
66      m.set_next("fifo_indata",
67        ila.choice("fifo_indata",
68          [fifo_indata,
69            ila.store(fifo_indata, fifo_in_amt, cmddata)]))
70
71      # internal index to the FIFO,
72      fifo_out_amt = m.reg("fifo_out_amt", 8)
73      m.set_next("fifo_out_amt", ila.choice("fifo_out_amt_choice",
74        [fifo_out_amt, fifo_out_amt-1, ZERO, THIRTY]))
75
76      # The fifo out memory
77      # another 256 8 bit registers
78      fifo_outdata = m.mem("fifo_outdata", 8, 8)
79      m.set_next("fifo_outdata",
80        fifo_outdata)
81
82      # dataout
83      # this is what is returned by a read or write
84      dataout = m.reg("dataout", 8)
85      m.set_next("dataout", ila.choice("dataout_choice",
86        [ZERO, fifo_outdata[fifo_out_amt], fifo_sts, fifo_def.FIFO_MAX_AMT - fifo_in_amt, fifo_out_amt
          ]))
87
88      # ------------------------------------------------------------
89      # Decode Logic
90      # ------------------------------------------------------------
91      # General Information
92      addresses = [fifo_def.STS_ADDR, fifo_def.FIFO_ADDR, fifo_def.BURST_ADDR]
93      commandData = [fifo_def.STS_COMMAND_READY, fifo_def.STS_GO]
94
95      # Commands start and end
96      cmds = [(cmdaddr == fifo_def.STS_ADDR)
97        & (cmd == fifo_def.WR)
98        & (cmddata == d)
99        & (fifo_out_amt == a)
100       for d in commandData for a in range(2)]
101
102     # actual commands
103     pcr_extend = [(cmdaddr == fifo_def.STS_ADDR)
104       & (cmd == fifo_def.WR)
105       & (cmddata == fifo_def.STS_GO)
106       & (fifo_state  == fifo_def.FIFO_ACCEPTING)
107       & (fifo_in_amt  == fifo_in_cmdsize)
108       & (ila.load(fifo_indata, m.const(0x6, 8)) == 0)
109       & (ila.load(fifo_indata, m.const(0x7, 8)) == 0)
110       & (ila.load(fifo_indata, m.const(0x8, 8)) == 0)
111       & (ila.load(fifo_indata, m.const(0x9, 8)) == 0x14)
112       ]
113
114     # General Reading and Writing in every state + Address
115     general = [(cmdaddr == a)
116       & (cmd == c)
117       & (fifo_state == s)
118       for a in addresses for c in [0,1,2] for s in range(5)]
```

```
119     m.decode_exprs = general + cmds + pcr_extend
120
121     # ------------------------------------------------------------
122     # Synthesize
123     # ------------------------------------------------------------
124     f = fifo()
125     sim = lambda s: f.simulate(s)
126     for var in f.all_state:
127       synth(m, var, sim)
128     m.generateSim('tpm_sim.cpp')
```

Appendix B

Images and Charts

B.1 Images

BASE	SIZE	SECTION	DESCRIPTION
0x000000	0x200000	SI_ALL	Descriptor + ME
0x200000	0x0f0000	RW_SECTION_A	Read-Write Firmware A
0x2f0000	0x0f0000	RW_SECTION_B	Read-Write Firmware B
0x3e0000	0x010000	RW_MRC_CACHE	Memory Training Cache
0x3f0000	0x004000	RW_ELOG	Event Log
0x3f4000	0x004000	RW_SHARED	Shared Data
0x3f8000	0x002000	RW_VPD	Read-Write VPD
0x400000	0x200000	RW_LEGACY	Legacy Firmware
0x600000	0x004000	RO_VPD	Read-Only VPD
0x610000	0x000800	FMAP	Flash Map
0x610800	0x000040	RO_FRID	RO Firmware ID
0x611000	0x0ef000	GBB	Google Binary Block
0x700000	0x100000	BOOT_STUB	Read-Only Firmware

Figure B.1: The Flash Map for Chrome OS's SPI Flash. The gray sections have been marked as Read Only [14]

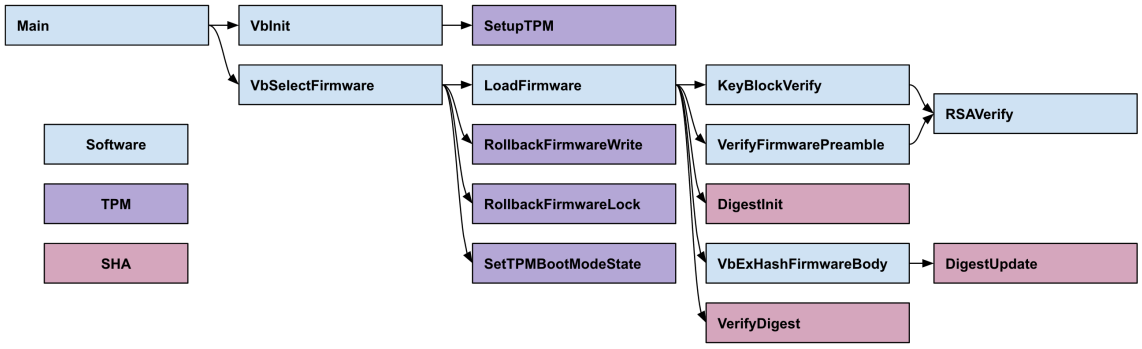

Figure B.2: The full function call graph for Verified Boot

B.2 Complete Verification Results

- TPM locking – checks that it is not possible for VbSelectFirmware to return success if it is unable to lock the TPM
- LoadFirmware – checks that it is not possible for control to pass to an Image if it is not selected by LoadFirmware
- Array accesses – check for all possibilities of array bounds errors.
- Google Assertions – this test replaces Google's assertions with CBMC assertions and checks that none of them are violated under any inputs
- Rollback test – checks that under no input conditions will LoadFirmware choose an Image with a lower version number (see Property 4.9).
- Rollback/rec – this check is supposed to find a counter-example. It checks for rollback conditions without realizing that rollback is allowed under recovery.
- Hash Failure – checks that it is impossible to verify an image as correct if hashing the Image Body returns an error (see Property 4.8).
- RSA Failure – checks that is impossible to verify an image as correct if verifying either the Keyblock or Preamble returns an error (see Property 4.7).
- Array accesses – check for all possibilities of array bounds errors.
- Error Handling – the TPM returns success only if the TPM command completes successfully
- SetupTPM – the TPM is initialized successfully with well-defined states
- RollbackFirmwareWrite – the RollbackFirmwareWrite function sends the write value to the TPM unmodified and no TPM assertions are broken
- RollbackFirmwareLock – the RollbackFirmwareLock function will always lock the correct section and no TPM assertions are broken
- SetTPMBootModeState – the SetTPMBootModeState function sends the PCR value to the TPM unmodified and no TPM assertions are broken
- Liveness – Checks that TpmSendReceive will not enter a state where it polls on an unchanging TPM register

- Error Handling – Checks that TpmSendReceive will only return success if the TPM has performed a Send and Receive action successfully
- Data Receive – Checks that TpmSendReceive will not return if the TPM still has data to send

Table B.1: All Verification results

test name	steps	VCCs	time (s)	memory (Mb)	sat/unsat
SelectFirmware					
TPM locking	5269	6	2	164	unsat
LoadFirmware	5264	2	3	164	unsat
Array Accesses	5257	10	3	165	unsat
LoadFirmware					
Google Assertions	6460	34	123	2312	sat
Rollback	4125	1	4	259	unsat
Rollback/rec	4123	1	6	394	sat
Hash Failure	4182	2	5	258	unsat
RSA Failure	4180	2	5	258	unsat
Array Accesses	4130	57	126	2487	unsat
LoadFirmware (malloc)					
Google Assertions	6460	34	3799	60496	unsat
Rollback, Hash, RSA	2858	6	3733	59874	unsat
TLCL					
Error Handling	34639	10	11	1579	unsat
SetupTPM	613418	140	1674	6960	unsat
RollbackFirmwareWrite	478639	108	952	5447	unsat
RollbackFirmwareLock	17450	4	7	369	unsat
SetTPMBootModeState	34621	8	11	397	unsat
TPM					
Liveness	20369	2217	359	56434	unsat
Data Send	21703	259	34	3991	unsat
Data Receive	20231	2	391	61040	unsat

Bibliography

[1] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 2, pp. 123–193, 1999.

[2] L. Zhang and S. Malik, "Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, p. 10880, IEEE Computer Society, 2003.

[3] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories.," *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[4] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 389–391, Springer, 2014.

[5] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, "Comparing hardware accelerators in scientific applications: A case study," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 58–68, 2011.

[6] E. Chou and S. Malik, "A secure bootloader for demonstrating formal verification of hardware-firmware interactions on socs," 2015. Senior Thesis E.E Princeton University.

[7] G. Bai, J. Hao, J. Wu, Y. Liu, Z. Liang, and A. Martin, "Trustfound: Towards a formal foundation for model checking trusted computing platforms," in *International Symposium on Formal Methods*, pp. 110–126, Springer, 2014.

[8] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor, "Security of soc firmware load protocols," in *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*, pp. 70–75, IEEE, 2014.

[9] S. Malik and P. Subramanyan, "Invited: specification and modeling for systems-on-chip security verification," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[10] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, "Template-based synthesis of instruction-level abstractions for soc verification," in *Formal Methods in Computer-Aided Design (FMCAD), 2015*, pp. 160–167, IEEE, 2015.

[11] G. Inc., "Verified boot." Available: https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot. [Accessed 15-April-2016].

[12] W. E. Cammack and J. D. Kridner, "Secure bootloader for securing digital devices," June 26 2007. US Patent 7,237,121.

[13] A. Regenscheid, "Roots of trust in mobile devices," *ISPAB, Feb*, 2012.

[14] G. Inc., "Chrome os firmware overview." Available: https://docs.google.com/presentation/d/1h-nsDGlQmYI21dr95nYgLmyCYDgBIpJWSt9b7AqTZaw. [Accessed 1-November-2016].

[15] T. C. Group, "Trusted platform module." Available: https://trustedcomputinggroup.org//work-groups/trusted-platform-module/. [Accessed 10-January-2016].

[16] T. C. Group, "Homepage." Available: http://www.trustedcomputinggroup.org. [Accessed 10-January-2016].

[17] T. C. Group, "Tpm main: Part 1 design principles." Available: https://trustedcomputinggroup.org/wp-content/uploads/Main-Part1-Rev94.pdf. [Accessed 2-February-2017].

[18] T. C. Group, "Tpm main: Part 2 tpm structures." Available: Group.https://trustedcomputinggroup.org/wp-content/uploads/TPM-main-1.2-Rev94-part-2.pdf. [Accessed 2-February-2017].

[19] T. C. Group, "Tpm main: Part 3 commands." Available: https://trustedcomputinggroup.org/wp-content/uploads/TPM-main-1.2-Rev94-part-3.pdf. [Accessed 2-February-2017].

[20] R. Ng, "Tpm fundamentals." Available: http://www.cs.unh.edu/~it666/reading_list/Hardware/tpm_fundamentals.pdf. [Accessed 10-March-2017].

[21] QEMU, "Qemu: The fast processor emulator." Available: http://www.qemu-project.org/. [Accessed 10-February-2017].

[22] "Multiboot specification." Available: https://www.gnu.org/software/grub/manual/multiboot/multiboot.html. [Accessed 5-January-2017].

[23] G. Inc., "Firmware boot and recovery." Available: http://www.chromium.org/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery. [Accessed 1-November-2016].

[24] G. Inc., "Chrome os developer mode." Available: http://www.chromium.org/chromium-os/chromiumos-design-docs/developer-mode. [Accessed 3-November-2016].

[25] G. Inc., "Verified boot data structures." Available: http://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot-data-structures. [Accessed 3-November-2016].

[26] Git, "git: fast version control." Available: https://git-scm.com/. [Accessed 10-December-2016].

[27] G. Inc., "Developing with git and repo." Available: http://source.android.com/source/developing.html. [Accessed 2-January-2017].

[28] G. Inc., "Coreboot." Available: https://www.coreboot.org/. [Accessed 3-November-2016].

[29] Intel, "Acquiring, building, and configuring the payload compatible with the coreboot reference bootloader." Available: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/coreboot-reference-bootloader-white-paper.pdf. [Accessed 11-December-2016].

[30] G. Inc., "Verified boot codebase." Available: https://chromium.googlesource.com/chromiumos/platform/vboot_reference/. [Accessed 1-October-2016].

[31] S. Berger, "Swtpm." Available: https://github.com/stefanberger/swtpm. [Accessed 10-February-2017].

[32] S. Berger, "Libtpms." Available: https://github.com/stefanberger/libtpms. [Accessed 10-February-2017].

[33] S. Berger, "Qemu-tpm." Available: https://github.com/stefanberger/qemu-tpm. [Accessed 10-February-2017].

[34] D. Gilhooley, "Cbmc-vboot repository." Available: https://github.com/gilhooleyd/CBMC-Vboot. [Accessed 6-March-2017].

[35] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar, "Efficient sat-based bounded model checking for software verification," *Theoretical Computer Science*, vol. 404, no. 3, pp. 256–274, 2008.

[36] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, no. 53, 2005.

[37] S. Bratus, N. DCunha, E. Sparks, and S. W. Smith, "Toctou, traps, and trusted computing," in *International Conference on Trusted Computing*, pp. 14–32, Springer, 2008.

[38] D. Cooper, W. Polk, A. Regenscheid, and M. Souppaya, "Bios protection guidelines," *NIST Special Publication*, vol. 800, p. 147, 2011.

[39] A. Inc., "At97sc3201 the atmel trusted platform module." Available: `http://www.atmel.com/images/doc5128.pdf`. [Accessed 10-April-2017].

[40] B. Inc., "Bcm5752 – single-chip device for desktop lan-on-motherboard applications." Available: `http://www.recomb-omsk.ru/published/SC/html/scripts/doc/BCM5752-PB02-R.pdf`. [Accessed 10-April-2017].

[41] I. Inc., "Technologies ag. product brief tpm 1.2 hardware.." Available: `http://www.infineon.com/tpm`,. [Accessed 10-April-2017].

[42] A.-R. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, and M. Winandy, "Tcg inside?: a note on tpm specification compliance," in *Proceedings of the first ACM workshop on Scalable trusted computing*, pp. 47–56, ACM, 2006.

[43] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel, "A formal analysis of authentication in the tpm," in *International Workshop on Formal Aspects in Security and Trust*, pp. 111–125, Springer, 2010.

[44] S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga, "Security evaluation of scenarios based on the tcgs tpm specification," in *European Symposium on Research in Computer Security*, pp. 438–453, Springer, 2007.