– Independent Work Report Fall, 2016 –

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

David Gilhooley

Advisor: Sharad Malik

Abstract:

Formal verification is an important tool for security verification because of its ability to do comprehensive corner case testing. It does this by providing either a guarantee that a specific property holds, or a counterexample that serves as a violation of the property. While formal verification has been increasingly popular in the checking of both software programs and hardware modules, checking both in a unified fashion is still under-developed. As computer architecture moves increasingly to specialized hardware for accelerators, security, and power saving purposes, this unified formal verification will become more more important. Real world systems like Chrome OS are making security claims about firmware that require verification across the hardware-firmware boundary. This paper begins to investigate the tools available for this verification on a large scale platform that is widely used.

Contents

1 Introduction

The importance of computing in modern society can not be overstated. The increased responsibilities of computers means that security problems can have large impacts on both the personal and societal level. 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.

Verification against vulnerabilities can be done through simulation or Unit Testing, where inputs are fed into the program and compared with the observed outputs. However, as the number of inputs grows, the possibilities rise exponentially and corner cases become increasingly difficult to check. If it computationally infeasible to check every possibility, then we cannot be sure that a platform is entirely bug-free.

An alternative to simulation, formal verification provides either guarantee that a property holds, or an example of inputs that violate the property. These guarantees are helpful for all programs but especially security programs where bugs can release personal information or allow malicious behavior. As platforms become more complicated, security protocols become more difficult to check and formal verification is increasingly important.

1.1 Background and Context

Formal verification of hardware is a well researched area and as such there are several approaches that have been used by industry and academics for formal verification. Formal verification can be defined as the application of mathematical methods to the specification and verification of systems [1]. Combinational circuits can be described completely through a propositional logic formula, and this formula can be verified using a Boolean Satisfiability (SAT) solver. Solving SAT problems consists of finding a counterexample, or a subset of of clauses from the Conjunctive Normal Form (CNF) of the formula such that the subset is not satisfiable [2]. The CNF of a formula is the combined AND of a set of clauses where each individual clause is the combined OR of a set of smaller clauses. SAT solvers are useful for verifying hardware, but they can be extended to verifying properties of any system that can be described through temporal logic.

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), which essentially replaces Boolean logic elements with predicates such as inequalities or uninterpreted functions. Automated tools exist to transform software programs with assertions into an SMT solver for formal verification [3]. Another historically common tool to verify properties is Computation Tree Logic (CTL). CTL adds temporal distinctions over propositional logic and this allows one to easily describe how a system changes over time. Through the rest of this paper, software and hardware properties relevant to security will be expressed in CTL. Once the properties are expressed formally, they will be added to the software programs through assertions and checked using SAT or SMT solvers.

1.2 Problem Statement

The trend in modern Computer Architecture is to offload computation to specialized hardware for either speed or power savings [4]. As this pattern increases the job of verifying functionality becomes more difficult. Firmware and hardware can be verified separately but this opens the possibilities of implementation mismatches and missed vulnerabilities. Verifying firmware and hardware together is necessary but it does not scale well. Steps are being taken to create viable tools for abstraction of hardware that help with the scalability issue of formal verification. Instruction Level Abstraction (ILA) is one such technique that abstracts hardware to functional states that are visible to firmware [5]. This technique claims to allow firmware and hardware to be verified simultaneously and at scale using formal methods. In order to show the full benefits of formal verification on SoCs, the hardware and software of the SoC needs to be available and conducive to testing. In the past, small SoCs have been created for the purpose of testing verification. 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 common security protocols such as Secure Boot for small, Open Source SoC designs [6]. While example SoCs are useful for the first steps in testing verification methods, these methods will be most useful if they can be applied to large designs that already exist and are in use.

The contributions that this thesis makes are the research into the open source firmware project Verified Boot created by Google; the expansion of the formal verification project into an open source software platform; and the evaluation of bounded

model checking on verifying security properties of both hardware and software.

2 System Design

Verification across the Hardware Firmware boundary requires a system that utilizes both Hardware and Firmware in equal parts to achieve a single goal. A secure bootloader is a program that ensures only encrypted and authenticated software is loaded onto a platform [8]. In order for a bootloader to be secure it must run before any other code is executed on the platform. Because of this, the bootloader is responsible for doing all system initialization, including hardware initialization of modules. The bootloading process must also complete before any of the other platform code is run, meaning that program speed is important. The cryptographic functions of hashing and signing the authenticated software are computationally expensive and are therefore offloaded to hardware accelerators. All of these factors make secure boot ideal for formal verification techniques.

Verified Boot is an implementation of secure boot that has been designed by Google to run on their Chromebook laptops. Verified Boot is the bootloader to the Chrome Operating System and verifies that all system code has been signed by Google. Google has released both Verified Boot and Chrome OS as Open Source software, making this a viable candidate for formal verification. While the specific hardware in each laptop is still closed source, Google has documents for extending their Verified Boot program and outlines the minimum number of hardware modules required for security.

The hardware modules can be broken into two groups, secure storage and hardware accelerator. Under secure storage there is an Flash memory module that provides non-volatile storage, and a Trusted Platform Module (TPM) that provides non-volatile storage that can be locked until the next boot. Under hardware accelerators there are the cryptographic functions implementing Secure Hash Algorithm (SHA) and RSA encryption. The CPU accesses the hardware modules through different protocols as seen in Figure. These interaction methods can be abstracted to Memory Mapped I/O without losing security information.

2.1 Hardware

2.1.1 Flash Memory

Each Chrome book comes with 8 Mbs of non-volatile Flash memory [9]. The memory is organized according to Figure 1. 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. The important feature of the Flash Memory is it's hardware protected Read-Only (RO) section of memory. The RO section is enforced by the placement of a physical screw known as the Write Protect (WP) Screw. If the WP Screw remains in the laptop, all writes to the RO section will fail. If the screw is removed, all of Google's security claims can be violated. Google is aware of this and considers the warranty of the laptop to be voided if the screw is removed.

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. In this way the RO Flash acts as the "Root of Trust" for the system. In security terms, the Verified Boot process should be secure as long as the Root of Trust holds. Inversely, if someone removes the WP screw, none of the Verified Boot security properties will hold.

The organization of the Flash Memory can be seen in Figure 1. This map of Flash memory is referred to as the "fmap" and it is stored in Flash as can be seen in the figure. The gray sections of the figure are the Read Only sections and provide the Root of Trust of the boot system as discussed previously.

2.1.2 TPM

The Trusted Platform Module[10] or TPM is a generic name for any secure crypto-processor that conforms to the standards written by the Trusted Computing Group (TCG) [11]. Although Google has released Chromebooks with both 1.2 and 2.0 TPMs, for the scope of this project we will be looking at the TPM 1.2 specification.

The TPM allows for secure access of non-volatile store, with permissions that can be permanently set until the platform is powered off. The TPM can also create and store RSA keys as well as perform RSA encryption and decryption. The TPM is communicated with by sending binary commands and data as outlined by the TCG[10].

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 1: The Flash Map for Chrome OS's SPI Flash. The gray sections have been marked as Read Only [9]

2.1.3 SHA

2.1.4 RSA

3 Verified Boot

Verified Boot (Vboot) is a cryptographic boot process that verifies the rest of the system code before running it, claiming to only run code that has been verified as signed by Google and untampered. Vboot verifies both the system's firmware and operating system. It is important for security reasons that Vboot itself is unmodified and cannot be removed by a hacker wishing to replace the system's firmware or OS undetected. For this reason the Vboot code is stored and executed out of the Read Only Section of Flash which is designed to be unmodifiable without tampering with the system's hardware. The Vboot process verifies a firmware image and kernel image in a two step process. For Vboot to fulfill its security promises, it must only load and run images if the image was signed by Google and unmodified. Images that have been calculated to be unsecure will not be loaded and the system will restart into a Recovery Mode. Google has implemented different modes that Vboot can boot with and these modes alter the program flow and security guarantees. In some modes, Verified Boot can even be removed from the system. While seemingly counterintuitive to security, these modes have been outlined with their own security claims and properties. I will be describing these modes in more detail in Section 3.3.

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" [13]. Vboot protects against vectors of attack that are relatively quick to exploit and it will realize the attack and nullify it on the next boot cycle. An example of an attack that would be caught by Vboot would be the installation of a malicious kernel driver to act as a keylogger, or replacing the kernel with an older version with known vulnerabilities.

There are many attacks that Vboot is not able to recognize or protect against. For example, Vboot does not make any promises to the safety of the system once the kernel is running and the user has control. Any attacks to the userspace of the Chromebook (for example the web-browser) would not affect the firmware or kernel image and would remain undetected. However, userspace programs by definition have lower priorities and this would reduce the severity and influence of the attack. Any attack to the kernel during runtime would remain until the system was powered down,

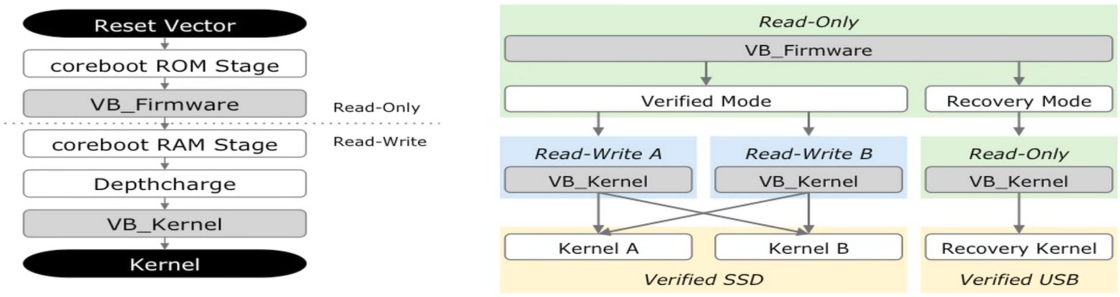

Figure 2: Verified Boot is separated into hierarchical stages for clarity and security. This also shows the separate recovery path in case of a firmware issue.

which in certain situations could provide plenty of time for an attack to steal valuable data.

3.2 Verified Boot Stages

The boot process of a laptop is very complex and will include very different code across different laptops with different hardware. In order to divide the code modularly and have Verified Boot be extensible for current and future platforms, Google has split the process into three main sections as seen in Figure 2. These sections are referred to as Read-Only Firmware (RO FW), Read-Write Firmware (RW FW), and Kernel. These sections are run in sequence with each section verifying the section that comes later. In this way the root of trust starts from the Read-Only portion and builds until the system has booted completely.

The RO Firmware runs first and as such is the root of trust for the entire platform. It is responsible for verifying the RW Firmware image and it contains all of the code and hardware drivers it needs to accomplish this task. It verifies the RW Firmware Image signature using Google's main public key. This public key is packaged with the RO Firmware and is also read only so it is guaranteed to be secure. The RO Firmware is also responsible for handing the main control of Vboot, and this includes the transitions between Developer Mode, Safe Mode, and Recovery Mode. These transitions will be described in more detail in Section 3.3.

The RO Firmware passes control to the RW Firmware once it can confirm that it has been signed by Google and is unmodified. The RW Firmware contains the drivers for the rest of the hardware on the platform. It initializes the entire platform in preparation for booting and then it attempts to verify the kernel image. The RW

Firmware verifies the kernel using one of Google's public keys that is stored within the RW Firmware Image. This public key had previously been verified as secure by the RO Firmware. The RW Firmware Image is verifying the Kernel image off of its partition in the main hard drive.

3.3 Boot Modes

Vboot has three major boot states that influence the behavior of the boot. The first state is the Safe state which goes through the full process of Verified Boot normally. The second state is the Recovery state which allows for a broken machine to format itself to its original state. The third state is the Developer state and in this state the RSA signature is not verified. The Developer state exists so that hobbyists can write their own firmware boot code and run operating systems other than Chrome OS.

Developer mode poses interesting security questions to Verified boot because it essentially disables the security guarantees and allows the system to move into an insecure state [20]. There are various security requirements around the Developer state transition. First, a physical presence is required to fully complete the developer mode transition. This means that a person must sit at the computer once it has been rebooted and press a certain key combination (Control + D) when the developer mode screen appears. 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. Once Developer Mode is enabled, the system can no longer claim guarantees about the boot process or its own secure storage. In order to prevent an attacker from enabling Developer Mode so that they could read 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 to 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. The assumptions that Google places on secure storage require that it can only be written to and read from by Google. The platform will not be able to recognize if

secure storage is in an insecure or malicious state and so wiping it is the easiest way to ensure that the platform has full control. On the transition from Developer to Safe Mode, the system has to go through 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 in 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 is for a Recovery Image stored on external memory, either on an SD card or a USB drive. The path for recovery uses a different RSA key for modularity and this recovery key is also stored in Read Only Flash in order to be unmodifiable. Once the recovery image is verified to be secure and unmodified, the image is loaded off of the external storage and it is used to replace the images currently stored on the device. Recovery mode is also responsible for wiping secure memory, including 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. From a user perspective, Recovery is an easy way to wipe the device and restore it to Factory Settings. Recovery is also the only way to roll the device back to an early image, and this rollback is sometimes necessary if there are known problems with a newly uploaded image.

3.4 Data Structures

To start understanding the Vboot verification process, it is necessary to talk about the data structures that are used throughout. These data structures are populated by the Firmware or Kernel Image that is attempting to be verified. Through this section I will start at the highest hierarchical level of data structure then explain the structures that are contained within.

3.4.1 Firmware/Kernel Image

The actual image is not a data structure but a chunk of data that is stored contiguously in non-volatile memory. The image structure, as seen in Figure 3, consists of three parts: a key block, a preamble, and the main body of the image. The key block is verified first, and it is verified by the main RSA key stored in Read Only memory. Once the key block has been shown to be safe, the RSA key located within will be used to verify the preamble. The preamble contains the hash of the image's body. The image's body contains the code that is going to be run next and it is checked against the hash

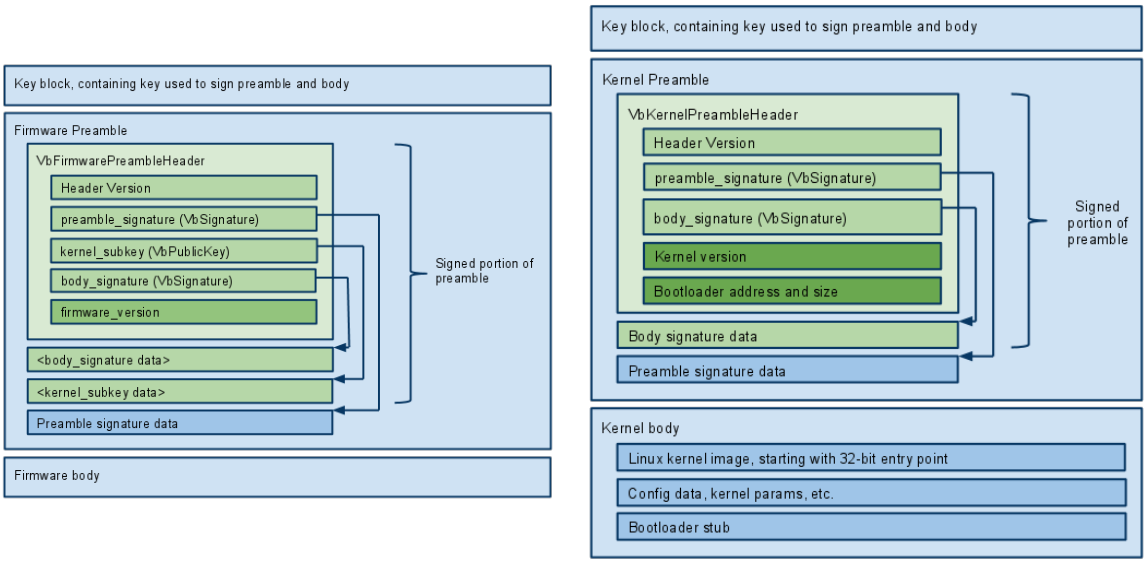

Figure 3: Layout of the Firmware and Kernel Images [21]

stored in the preamble. If the preamble is for the Firmware image then it will contain the RSA key used to verify the key block of the Kernel image. If the preamble is for the Kernel image then it will contain information about the location of the Kernel's bootloader, like where it exists in the Kernel image and how it should be loaded into RAM.

3.4.2 Key Blocks

The Key block is the first part of the image that is validated and it is used to validate the rest of the image. The Key block is the data structure that allows a hierarchy of RSA keys to be used during Vboot. Figure 4 shows the structure of the keyblock. The key block flags that are mentioned 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 keyblock 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.

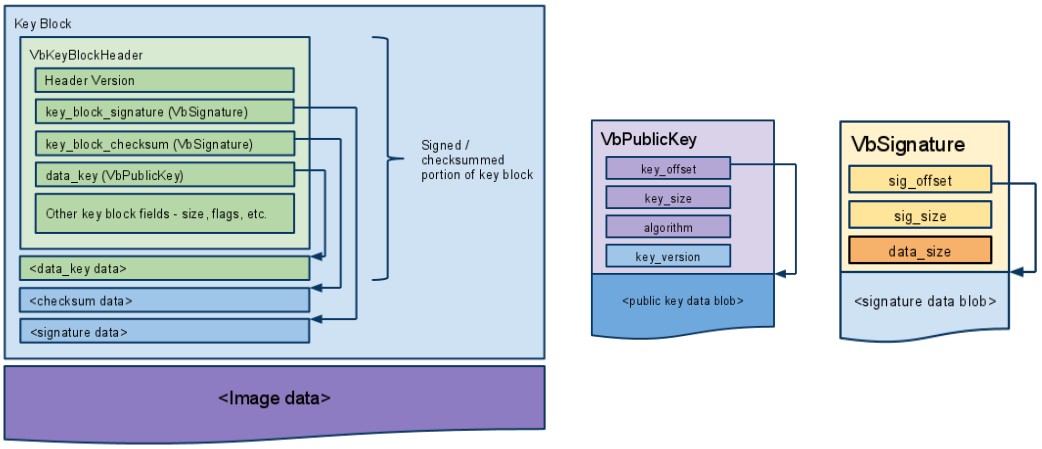

Figure 4: The Keyblock data structure and Metadata for Keys and Signatures

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 various boot screens.

3.5 Code Organization

Like almost all 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 [12]. Google has built a tool called "repo" that is used on top of git [14]. 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, depthcharge, are the repositories used for the firmware boot process. The flow of Verfied Boot through the repos can be seen in Figure 5 and the purpose of each repo is described below.

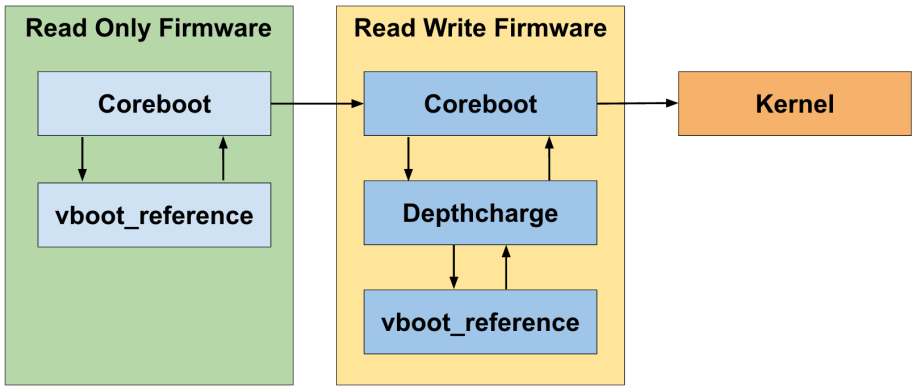

Figure 5: ChromeOS's boot flow goes through Coreboot, Depthcharge, and the Vboot Library twice for Firmware and Kernel verification

3.5.1 Coreboot

Coreboot is a fully Open-Sourced alternative to traditional BIOS implementations. [15] 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" [16]. This payload is 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.

3.5.2 DepthCharge

Depthcharge contains the minimum number of drivers necessary for the virtual boot to work successfully [18]. The important drivers include the TPM, I2C, the EC, SPI and SPI Flash, and the display [17]. Once the drivers are initialized successfully, Depthcharge creates the necessary structures for Vboot and then passes control into the Vboot library.

Depthcharge is Google's repo that holds platform dependent code. The different branches in the repository hold different drivers as they are needed for each of the

Chromebook's hardware.

3.5.3 Vboot_reference

The vboot_reference repo contains all of the control and algorithms for the vboot process [19]. 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.

4 Read-Only Firmware

The Read-Only Firmware's purpose is to provide an initial configuration of the board's hardware and to verify the block of code that has been provided as the Read-Write Firmware. The RO Firmware requires the SHA and RSA accelerators to do cryptographic functions, and the TPM and Flash for secure storage. If Developer Mode or Recovery Mode are enabled then the Keyboard driver is required for user input and the Display driver is required for updating the user with the boot status. If Recovery Mode is enabled then the USB and SD drivers are required so that the recovery image can be read off of external storage. Implementations of all of these drivers are packaged with the RO Firmware so that the Vboot library can succeed.

The first thing that RO Firmware is responsible for is loading in all of the Flash memory to RAM so that it is more easily accessible. After Flash memory is in RAM, data structures like the GBB and the Firmware Image are populated by reading the flash map, which is always stored at a set location (see Figure 1). Once the data structures are populated and the hardware drivers are loaded, the RO Firmware moves into the Vboot library to verify the firmware image.

The RO Vboot is responsible for checking the validity of the firmware image validity and either executing the code within the firmware image or failing appropriately. If the firmware image is insecure or the verification process fails for other reasons, it is likely that the system will be rebooted into recovery mode. RO Vboot is responsible for setting a 32 bit flag known as a "recovery reason" for failure. This reason influences the Recovery process and is stored in the read write portion of Flash so that it is consistent across boots.

4.1 TPM Properties Used

The RO Firmware is responsible for controlling the TPM setup and ensuring that its properties hold. The TPM is responsible for attesting the boot flags and protecting against rollback attacks. Setting the TPM up requires that the TPM is lead through its self-check, and that the data zones required by Chrome exist within the TPM's state.

Chrome disables the physical protection in the RO Firmware. Physical presence is something that can be set high or low on a TPM either through software commands or a hardware wire. If Physical Presence is set high, then the ownership of the TPM can be changed and the re-provisioning operations become available. For this reason, Chrome completely disables this option by permanently locking the TPM to a low Physical Presence.

The other thing that the TPM protects from is Rollback Attacks. In this situation a rollback attack is when older software with known vulnerabilities replaces newer, protected software in a malicious "upgrade". This attack works against a naive implementation because the attacker is relying on an older version of ChromeOS that is available, and unmodified. Because the attacker is using Google's software, the software is signed by Google's private key and will be accepted by the VBoot algorithm.

4.2 RO Firmware Properties

The properties that we will be checking can be separated into 4 categories: array accesses, program flow, hyperproperties, and correctness. The array access property restricts the writeable addresses in memory to valid array boundaries. Program flow properties refer to the execution path and function call stack throughout the program. Correctness properties refer to facts about the code such that, if things were designed successfully, certain states shouldn't be reached. Together these categories of properties should encompass each of claims that the system makes about security.

These properties will be specified in first order logic with the addition of a temporal logic like CTL. The boolean connectives in first logic logic are $\neg$ for NOT, $\wedge$ for AND, $\vee$ for OR, and $p -> q$ for p implies q. CTL describes how first order logic changes through time. In CTL Xp indicates that p is true in the next state; Fp indicates that p is true eventually; Gp indicates that p is always true.

4.2.1 Memory Access Locations

Memory accesses in C are very important because the language 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. Buffer overruns allow a program to write or read to arbitrary memory. In this program, 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 purposes of Vboot and for this reason it is important to prevent the accessing of arrays past the array boundaries. We can represent correct array accesses through formal specification using the formula below.

$$G(a \rightarrow (i > 0 \wedge i < size(ptr))) \tag{1}$$

In this equation a is any array access, ptr is the start of the array and i is the access offset. This equation 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. Luckily, this problem is common enough that CBMC provides checking by default. However, This equation on its own is not strong enough to provide array access security for verified boot. This is because Vboot uses pointer manipulation throughout in order to access structures and data within the Firmware Image. Verified boot reads the image in as one contiguous blob and then uses size and offset variables to read more information. CBMC is not able to detect that this pointer manipulation is an out of bounds array access so this must be manually checked through the use of assertions each time something within the image is accessed. The pointer manipulation can be expressed formally through the following formula.

$$G(s->offset(struct) + size(struct) < size(img) \wedge offset(struct) > 0) \tag{2}$$

In this equation s is each time a structure is created from a pointer dereference, *struct* is the structure, and *img* is the image that it is being pointed to. The equation 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 or before the beginning of the image.

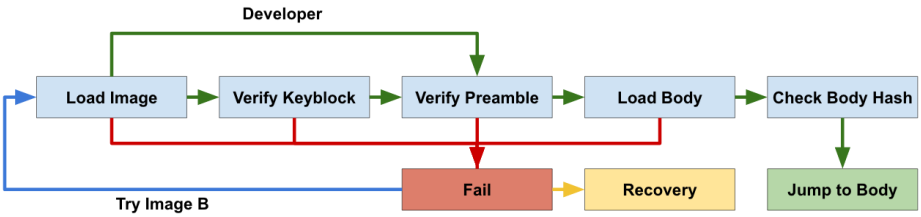

Figure 6: The logical flow for verifying the firmware image

Together these 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.

4.2.2 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 6. When verifying the flow of Vboot it is necessary to note that there are two firmware 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.

Let s_i correspond to the ith stage from Figure 6 and s_{ai} or s_{bi} correspond to that stage for Image A or Image B respectively.

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

$$G(\bigvee_{i=0}^{4} s_{ai} \vee s_{bi}) \tag{3}$$

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

$$G(s_{ai} \rightarrow s_{ai} \vee s_{ai+1} \vee s_{b0}) \tag{4}$$

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

$$G(s_{bi} \rightarrow s_{bi} \vee s_{bi+1}) \tag{5}$$

The final equation refines the transition for state s_0 for Image A and B. In Figure 6 we can see that state s_0 can transition to either s_1 but it also has a special transition to s_2 that is only valid if Developer Mode is enabled.

$$G(s_0 \rightarrow (s_0 \vee s_1 \vee (M_D \wedge s_2))) \tag{6}$$

All together these equations fully describe the program flow graph in Figure 6. 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.

4.2.3 Correctness properties

Correctness properties refer to the code itself and help to determine if it has been written correctly. For Verified Boot, correctness properties are focusing on the correct attestation of the image and attempting to guarantee that an incorrect image cannot be marked as correct and loaded for execution. For example, the following property claims 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. It is important that failure and passing is tracked on an image basis because it is possible for Image A to have a bad signature but for Vboot to verify and load a correct Image B.

$$\neg M_d \wedge \neg \text{verifySignature(img, rootKey)} \rightarrow XF(\text{img.pass} = F) \tag{7}$$

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

$$\text{hash(img.data)} \neq \text{img.hash} \rightarrow XF(\text{img.pass} = F) \tag{8}$$

The next property checks against the rollback attack. Rollback claims 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.

$$\neg M_r \wedge (\text{img.version} < \text{prevVersion}) \rightarrow XF(\text{img.pass} = F) \tag{9}$$

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

$$\neg \text{imgA.pass} \wedge \neg \text{imgB.pass} \rightarrow XF(\text{pass} = F) \tag{10}$$

In the equations 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 RW firmware version and prevVersion is the last seen version that is stored in the TPM. These properties will be proved contrapositively, meaning that when verified boot passes we will check that the antecedent is also false.

4.3 CBMC Results

CBMC was able to be successfully run on portions of the verified boot library and provide information on the satisfiability of various assertions. In order to get CBMC running, certain limitations had to be observed. The SAT generator of CBMC unrolls loops before converting the code into a model that can be checked. This loop unroller is not able to unroll loops where the upperbound is data dependant, so these loops must either be avoided, or a limitation can be placed on the unrolling through the commandline. If the manually unrolling limitation is too small, then CBMC results may be incorrect, while if the limit is too large the state space may be too large for CMBC to finish in a reasonable time. Luckily there was only a single loop of this nature, which existed in the method "Memcpy". Using the C implementation of Vboot I was able to determine that Memcpy needed no more than 1050 loop unrollings for the data driven portion of Vboot.

Running CBMC on the Read-Only section of Vboot with a full Firmware Image is not possible as the program will consume up to 4GB of RAM on my local machine

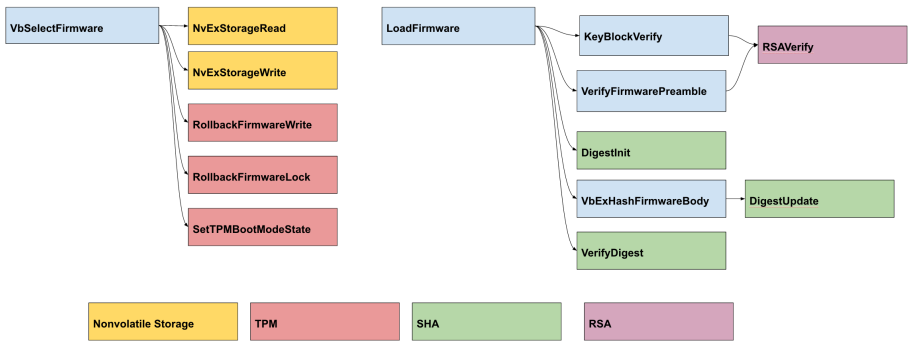

Figure 7: The call graph of functions for Read Only Firmware

and then be cancelled. In order to check properties, abstractions needed to be found such that parts of the Vboot flow could be checked at a single time. The key insight that was used for this project is that individual functions are essentially self contained and can be abstracted away based on the properties that one desires to check. Each function throughout Vboot returns an error code if it is not successful. When a function is abstracted away, it is programmed to return a non-deterministic value (this is done easily through CBMC's nondet_int() function). 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 super-set 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.

The full graph of functions can be seen in Figure 7. The blue functions are fully software functions that are implemented in the Vboot Library. The yellow functions connect to the Flash firmware on a real platform, but have been abstracted to return either non-deterministic data for Read-Write information, or hardcoded data for Read-Only information. The red functions connect to the TPM on a real platform, but have been abstracted to return static variables are non-deterministic at the program's start. The purple functions connect to either an RSA accelerator or a software RSA library. Instead of going through the full RSA algorithm, the function has been abstracted to a simple non-deterministic return of true or false. The green functions connect to

either an SHA accelerator or a software SHA library. The SHA algorithm has been abstracted in the same manner as RSA to return simply true or false at random.

For each of the next sections I will explain the verification process for individual functions. This explanation will include the function abstractions that were necessary, and a description of each of the tests run. There will be a table describing the results of each run of CBMC. CMBC uses an implementation of a SAT solver known as the MiniSat solver[22]. This solver outputs unSAT if the assertions hold and SAT if they do not.

4.3.1 LoadFirmware

The LoadFirmware function is where the main work is done in the Read-Only Vboot process. LoadFirmware is responsible for calling functions that Keyblock, Preamble, and Image verification. When LoadFirmware returns, it will have set the pass or fail statuses of both Firmware Images.

All CBMC test results can be seen in Table 1. The Google unit tests involve adding a wrapper around Google's assertions within Vboot such that they hook into CBMC assertions. This test shows that CMBC can be easily adapted into an existing framework and prove that Google specific assertions will not be thrown under any input. The Rollback test checks that under no input conditions will LoadFirmware choose a firmware image with a lower version number (see Property 9). In order to check that Rollback was working correctly, Rollback/rec tests that for rollback conditions without realizing that Rollback is allowed under recovery. The Rollback/Dev test correctly finds the counter-example where rollback is allowed and we can see that the recovery flag is enabled. Hash Failure tests Property 8, or that it is impossible to verify an image as correct if the VbExHashFirmwareBody function returns an error. RSA Failure tests Property 7, or that is impossible to verify an image as correct if the RSA key is formatted incorrectly or one of the Verify functions returns an error. Array accesses check for all array bounds errors.

We can see from these tests that CBMC is working successfully and in a reasonable time. The Google unit test framework is can be put to use with the CBMC framework using minor modifications, and the formal verification can be done in a very reasonable time.

Table 1: CBMC tests on the LoadFirmware function

test name	steps	VCCs	vars	clauses	time (s)	sat/unsat
Google unit_tests	6460	34	1095643	24391723	67.02	unsat
rollback	4125	1	135239	1026823	4.79	unsat
rollback/rec	4123	1	135276	1026983	4.68	sat
Hash Failure	4182	2	137431	1043724	4.45	unsat
RSA Failure	4180	2	137825	1043921	4.45	unsat
Array Accesses	4130	22	102660	640715	4.51	unsat

Table 2: CBMC tests on function VbSelectFirmware

test name	steps	VCCs	vars	clauses	time (s)	sat/unsat
TPM locking	6	5269	13513	16784	0.07	unsat
LoadFirmware	2	5264	13404	16209	0.07	unsat
Array Accesses	5257	0	0	0	unsat	

4.4 VbSelectFirmware

The VbSelectFirmware function is the wrapper around LoadFirmware. It is responsible for loading flags in from secure storage and for accessing the TPM. VbSelectFirmware is also responsible for locking the TPM's firmware version before control passes to the firmware image and updating the version if a newer image has been loaded. This function's final responsibility is to load TPM's PCR0 with the hash of the boot flags.

The TPM test for for this function is a series of correction properties. These assertions claim that the TPM must always be locked, and that VbSelectFirmware will not be successful if any of the TPM functions return an error.

The LoadFirmware test for this function simply checks that VbSelectFirmware will not not be successful if LoadFirmware returns an error. This is necessary for the properties checked in LoadFirmware to propagate up to the entire boot flow, and it is what allows for our function abstractions.

5 Conclusion

Throughout this paper, we investigated the importance and 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 firmware boundary. Verified Boot's properties were expressed formally using CTL and many were verified against the C implementation using the model checker CMBC. Model checking abstractions were discussed, such as unimplemented functions, which helped make the model checking possible in a short time. Additionally, model checking was shown to be a good replacement for unit testing which is in place throughout many C systems.

5.1 Future Work

Throughout the next semester I hope to refine my security properties and gather more CBMC data now that I have the model checking framework running. At the moment all hardware functions are abstracted as C programs, but this may provide different implementation mismatches between the simulation and a real platform. I hope to implement previous work which is able to take hardware specifications and build an accurate implementation for model checking[5]. In particular, a model of the TPM is necessary to fully verify that it can act as secure storage. As TPMs are widely used in systems for various security protocols, an ILA model may be useful beyond the scope of this project.

[1] Christoph Kern, Mark R. Greenstreet, Formal verification in hardware design: a survey, ACM Transactions on Design Automation of Electronic Systems (TODAES), v.4 n.2, p.123-193, April 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] de Moura L., Bjrner N. (2008) Z3: An Efficient SMT Solver. In: Ramakrishnan C.R., Rehof J. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008. Lecture Notes in Computer Science, vol 4963. Springer, Berlin, Heidelberg

[4] R. Weber, A. Gothandaraman, R. J. Hinde and G. D. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study," in IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 1, pp. 58-68, Jan. 2011.

[5] S. Malik and S. Subramanyan, "Specification and Modeling for Systems-on-Chip Security Verification"

[6] E. Chou and S. Malik, "A Secure Bootloader for Demonstrating Formal Verification of Hardware-Firmware Interactions on SoCs"

[7] Lee, Ruby B. "Security basics for computer architects." Synthesis Lectures on Computer Architecture 8.4 (2013): 1-111.

[8] Cammack, William E., and Jason Douglas Kridner. "Secure bootloader for securing digital devices." U.S. Patent No. 7,237,121. 26 Jun. 2007.

[9] "Chrome OS Firmware Overview" Google Inc. Accessed November 1, 2016. https://docs.google.com/presentation/d/1h-nsDGlQmYI21dr95nYgLmyCYDgBIpJWSt9b7AqTZaw/pub?start=false&loop=false&delayms=3000&slide=id.g2ba203e62_048

[10] "Trusted Platform Module" Trusted Computing Group.https://trustedcomputinggroup.org//work-groups/trusted-platform-module/

[11] Trusted Computing Group. http://www.trustedcomputinggroup.org.

[12] "Git. Fast version Control" Accessed December 10, 2016. https://git-scm.com/

[13] "Firmware Boot and Recovery" Google Inc. Accessed November 1, 2016. http://www.chromium.org/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery

[14] "Developing with Git and Repo" Google Inc. Accessed January 2, 2017. http://source.android.com/source/developing.html

[15] "Coreboot" Accessed November 3, 2016. https://www.coreboot.org/

[16] "Acquiring, Building, and Configuring the Payload Compatible with the Coreboot Reference Bootloader" Intel. December 2015 http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/coreboot-reference-bootloader-white-paper.pdf

[17] "Depthcharge: The ChromeOS bootloader" Google Inc. Accessed November 3, 2016. https://a77db9aa-a-7b23c8ea-s-sites.googlegroups.com/a/chromium.org/dev/chromium-os/2014-firmware-summit/ChromeOS%20firmware%20summit%20-%20Depthcharge.pdf?attachauth=ANoY7crD3ii0STwDVZHwBB0OCdcPXkiczTeo9TaIDJ4pntDcLPqwiA_zHy98OVs_SgenmBV5c-1HcuyT50zByhY0Y5qngp9ju8ziU74J6OMh3xPpPsqpBVLlDsCu57BSzIKvB4XZ9MLlDAZXMO8NaIRRwTIsaj1iPRrntmN_OtJtJRI7KmDd5zMeopwIbupHKCaUW24wQF_4KudBJ7oLp6s01cjknwyW-wY9op-M8Wg%3D&attredirects=1

[18] "Depthcharge Codebase" Google Inc.
https://chromium.googlesource.com/chromiumos/platform/depthcharge/

[19] "Verified Boot CodeBase" Google Inc.
https://chromium.googlesource.com/chromiumos/platform/vboot_reference/

[20] "Chrome OS Developer Mode" Google Inc.
http://www.chromium.org/chromium-os/chromiumos-design-docs/developer-mode

[21] "Verified Boot Data Structures" Google Inc. http://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot-data-structures

[22] Sorensson, Niklas, and Niklas Een. "Minisat v1. 13-a sat solver with conflict-clause minimization." SAT 2005 (2005): 53.

[23] D. Gilhooley. https://bitbucket.org/david_gilhooley/personal-vboot, 2017

6 Appendix

The code for this report is based off of Google's Verified Boot reference repository[19] and the modifications can be accessed in a cloned and edited repository[23]. Changes were restricted to the created folder `tests/cbmc` within the repository. In order to minimize changes to the existing codebase, if functions were edited elsewhere, these changes would only take effect if the C macro "`CBMC`" is defined.

6.1 CBMC Commands

Python scripts have been created to generate the CBMC command for different C files. These scripts are based off of a simple base script with the following editable arrays.

- `includedDirs` — list of all directories with related C header files
- `includedFiles` — list of all C file with necessary functions
- `extras` — list of CMBC specific commands including array bounds checking, C Macro definitions, and loop unwinding limits

6.2 CBMC Output

The most helpful function of CBMC is its ability to provide a trace of variables that provides a counterexample to a given assertion. However, the size of the codebase and the number of possible set variables makes it difficult to manually examine the counterexample to see what exactly is causing the assertion to be proved incorrect. For this reason I have created the `parse_output.py` file which takes in CBMC output and displays the non-deterministic variables in a formatted manner. The most helpful aspect of this program is that it is able to parse the binary flags and reveal in human readable format what is enabled or disabled.