BootStomp: On the Security of Bootloaders in Mobile Devices

Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi,
Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna
{*nredini, machiry, dipanjan, yanick, antoniob, edg, yans, chris, vigna*}*@cs.ucsb.edu*
University of California, Santa Barbara

Abstract

Modern mobile bootloaders play an important role in both the function and the security of the device. They help ensure the Chain of Trust (CoT), where each stage of the boot process verifies the integrity and origin of the following stage before executing it. This process, in theory, should be immune even to attackers gaining full control over the operating system, and should prevent persistent compromise of a device's CoT. However, not only do these bootloaders necessarily need to take untrusted input from an attacker in control of the OS in the process of performing their function, but also many of their verification steps can be disabled ("unlocked") to allow for development and user customization. Applying traditional analyses on bootloaders is problematic, as hardware dependencies hinder dynamic analysis, and the size, complexity, and opacity of the code involved preclude the usage of many previous techniques.

In this paper, we explore vulnerabilities in both the design and implementation of mobile bootloaders. We examine bootloaders from four popular manufacturers, and discuss the standards and design principles that they strive to achieve. We then propose BOOTSTOMP, a multi-tag taint analysis resulting from a novel combination of static analyses and dynamic symbolic execution, designed to locate problematic areas where input from an attacker in control of the OS can compromise the bootloader's execution, or its security features. Using our tool, we find *six* previously-unknown vulnerabilities (of which five have been confirmed by the respective vendors), as well as rediscover one that had been previously-reported. Some of these vulnerabilities would allow an attacker to execute arbitrary code as part of the bootloader (thus compromising the entire chain of trust), or to perform permanent denial-of-service attacks. Our tool also identified two bootloader vulnerabilities that can be leveraged by an attacker with root privileges on the OS to unlock the device and break the CoT. We conclude by proposing simple mitigation steps that can be implemented by manufacturers to safeguard the bootloader and OS from all of the discovered attacks, using already-deployed hardware features.

1 Introduction

With the critical importance of the integrity of today's mobile and embedded devices, vendors have implemented a string of inter-dependent mechanisms aimed at removing the possibility of persistent compromise from the device. Known as "Trusted Boot" [6] or "Verified Boot," [8], these mechanisms rely on the idea of a Chain of Trust (CoT) to validate each component the system loads as it begins executing code. Ideally, this procedure can verify cryptographically that each stage, from a Hardware Root of Trust through the device's file system, is both unmodified and authorized by the hardware's manufacturer. Any unverified modification of the various bootloader components, system kernel, or file system image should result in the device being rendered unusable until a valid one can be restored.

Ideally, this is an uncircumventable, rigid process, removing any possibility of compromise, even when attackers can achieve arbitrary code execution on the high-level operating system (e.g., Android or iOS). However, hardware vendors are given a great amount of discretion when implementing these bootloaders, leading to variations in both the security properties they enforce and the size of the attack surface available to an adversary.

Unfortunately, analyzing the code of bootloaders to locate vulnerabilities represents a worst-case scenario for security analysts. Bootloaders are typically closed-source [21], proprietary programs, and tend to lack typical metadata (such as program headers or debugging symbols) found in normal programs. By their very nature, bootloaders are tightly coupled with hardware, making dynamic analysis outside of the often-uncooperative target platform impractical. Manual

reverse-engineering is also very complicated, as bootloaders typically do not use system calls or well-known libraries, leaving few semantic hints for an analyst to follow.

In this paper, we first explore the security properties, implementations, and weaknesses of today's mobile device bootloaders. We begin with a discussion of the proposed standards and guidelines a secure bootloader should possess, and what, instead, is left to the discretion of manufacturers. We then showcase four real-world Android bootloader implementations on the market today.

Then, we present a static analysis approach, implemented in a tool called BOOTSTOMP, which uses a novel combination of static analysis techniques and under-constrained symbolic execution to build a multi-tag taint analysis capable of identifying bootloader vulnerabilities. Our tool highlighted 36 potentially dangerous paths, and, for 38.3% of them, we found actual vulnerabilities. In particular, we were able to identify *six* previously-unknown vulnerabilities (five of them already confirmed by the vendors), as well as rediscover one that had been previously-reported (CVE-2014-9798). Some of these vulnerabilities would allow an adversary with root privileges on the Android OS to execute arbitrary code as part of the bootloader. This compromises the entire chain of trust, enabling malicious capabilities such as access to the code and storage normally restricted to TrustZone, and to perform *permanent* denial-of-service attacks (i.e., device bricking). Our tool also identified two bootloaders that can be unlocked by an attacker with root privileges on the OS.

We finally propose a modification to existing, vulnerable bootloaders, which can quickly and easily protect them from any similar vulnerabilities due to compromise of the high-level OS. These changes leverage hardware features already present in mobile devices today and, when combined with recommendations from Google [8] and ARM [6], enforce the least-privilege principle, dramatically constraining the attack surface of bootloaders and allowing for easier verification of the few remaining attackable components.

In summary, our contributions are as follows:

- We perform a study of popular bootloaders present on mobile devices, and compare the security properties they implement with those suggested by ARM and Google.
- We develop a novel combination of program analysis techniques, including static analysis as well as symbolic execution, to detect vulnerabilities in bootloader implementations that can be triggered from the high-level OS.
- We implement our technique in a tool, called BOOTSTOMP, to evaluate modern, real-world bootloaders, and find *six* previously-unknown critical vulnerabilities (which could lead to persistent compromise of the device) as well as two unlock-bypass vulnerabilities.
- We propose mitigations against such attacks, which are trivial to retrofit into existing implementations.

In the spirit of open science, we make our analysis tool publicly available to the community[1].

2 Bootloaders in Theory

Today's mobile devices incorporate a number of security features aimed at safeguarding the confidentiality, integrity, and availability of users' devices and data. In this section, we will discuss Trusted Execution Environments, which allow for isolated execution of privileged code, and Trusted Boot, aimed at ensuring the integrity and provenance of code, both inside and outside of TEEs.

2.1 TEEs and TrustZone

A Trusted Execution Environment (TEE) is the notion of separating the execution of security-critical ("trusted") code from that of the traditional operating system ("untrusted") code. Ideally, this isolation is enforced using hardware, such that even in the event the un-trusted OS is completely compromised, the data and code in the TEE remain unaffected.

Modern ARM processors, found in almost all mobile phones sold today, implement TrustZone[1], which provides a TEE with hardware isolation enforced by the architecture. When booted, the primary CPU creates two "worlds"–known as the "secure" world and "non-secure" world, loads the un-trusted OS (such as Android) into the non-secure world, and a vendor-specific trusted OS into the secure world. The trusted OS provides various cryptographic services, guards access to privileged hardware, and, in recent implementations, can be used to verify the integrity of the un-trusted OS while it is running. The un-trusted kernel accesses these commands by issuing the Secure Monitor Call (SMC) instruction, which both triggers the world-switch operation, and submits a command the Trusted OS and its services should execute.

ARM Exception Levels (EL). In addition to being in either the secure or non-secure world, ARM processors support "Exception Levels," which define the amount of privilege to various registers and hardware features the executing code has. The 64-bit ARM architecture defines four such levels, EL0-EL3. EL0 and EL1 map directly to the traditional notion of "user-mode" and "kernel mode," and are used for running unprivileged user applications

[1] https://github.com/ucsb-seclab/bootstomp

and standard OS kernels respectively. EL2 is used for implementing hypervisors and virtualization, and EL3 implements the Secure Monitor, the most privileged code used to facilitate the world-switch between secure and non-secure. During the boot process described below, the initial stages, until the non-secure world bootloader is created, runs at EL3.

2.2 The Trusted Boot Process

In a traditional PC environment, the bootloader's job is to facilitate the location and loading of code, across various media and in various formats, by any means necessary. However, in modern devices, particularly mobile devices, this focus has shifted from merely loading code to a primary role in the security and integrity of the device. To help limit the impact of malicious code, its job is to verify both the integrity and provenance of the software that it directly executes.

As with the traditional PC boot process, where a BIOS loaded from a ROM chip would load a secondary bootloader from the hard disk, mobile bootloaders also contain a chain of such loaders. Each one must, in turn, verify the integrity of the next one, creating a Chain of Trust (CoT).

On ARM-based systems, this secured boot process is known as *Trusted Boot* and is detailed in the ARM Trusted Board Boot Requirements (TBBR) specification. While this document is only available to ARM's hardware partners, an open-source reference implementation that conforms to the standard is available [6].

While this standard, and even the reference implementation, does leave significant room for platform-specific operations, such as initialization of hardware peripherals, implementations tend to follow the same basic structure. One important aspect is the Root of Trust (RoT), which constitutes the assumptions about secure code and data that the device makes. In ARM, this is defined to be 1) the presence of a "burned-in," tamper-proof public-key from the hardware manufacturer that is used to verify subsequent stages, and 2) the very first bootloader stage being located in read-only storage.

While manufacturers are free to customize the Trusted Boot process when creating their implementations, ARM's reference implementation serves as an example of how the process should proceed. The boot process for the ARM Trusted Firmware occurs in the following steps, as illustrated in Figure 1.

1. The CPU powers on, and loads the first stage bootloader from read-only storage.
2. This first stage, known as BL1, Primary Boot Loader (PBL), or BootROM, performs any necessary initialization to locate the next stage from its storage, loads it into memory, verifies its integrity using the Root of Trust Public Key (ROTPK), and if this is successful, executes it. Since it is on space-restricted read-only media, its functionality is extremely limited.
3. BL2, also known as the Secondary Boot Loader (SBL) is responsible for creating the secure and non-secure worlds and defining the memory permissions that enforce this isolation. It then locates and loads into memory up to three third-stage bootloaders, depending on manufacturer's configuration. These run at each of the EL3, EL2, and EL1 levels, and are responsible for setting up the Secure Monitor, a hypervisor (if present), and the final-stage OS bootloader.
4. BL2 then executes BL31, the loader running at EL3, which is responsible for configuring various hardware services for the trusted and un-trusted OSes, and establishing the mechanism used to send commands between the two worlds. It then executes the BL32 loader, if present, which will eventually execute BL33.
5. BL33 is responsible for locating and verifying the non-secure OS kernel. Exactly how this is done is OS-dependent. This loader runs with the same privilege as the OS itself, at EL1.

Next, we will detail extensions to this process developed for the Android ecosystem.

2.3 Verified Boot on Android

ARM's Trusted Boot standard only specifies stages of the boot process up to the point at which the OS-specific boot loader is executed. For devices running Android, Google provides a set of guidelines for *Verified Boot* [8], which describes high-level functionality an Android bootloader should perform.

Unlike the previous stages, the Android bootloader provides more functionality than just ensuring integrity and loading code. It also allows for the user or OS to elect to boot into a special recovery partition, which deploys firmware updates and performs factory reset operations. Additionally, modern Android bootloaders also participate in enabling full-disk encryption and triggering the initialization of Android-specific TrustZone services.

Ideally, the verification of the final Android kernel to be booted would effectively extend the Chain of Trust all the way from the initial hardware-backed key to the kernel. However, users wishing to use their devices for development need to routinely run kernels not signed by the device manufacturer. Therefore, Google specifies two classes of bootloader implementations: Class A, which only run signed code, and Class B, which allow for the

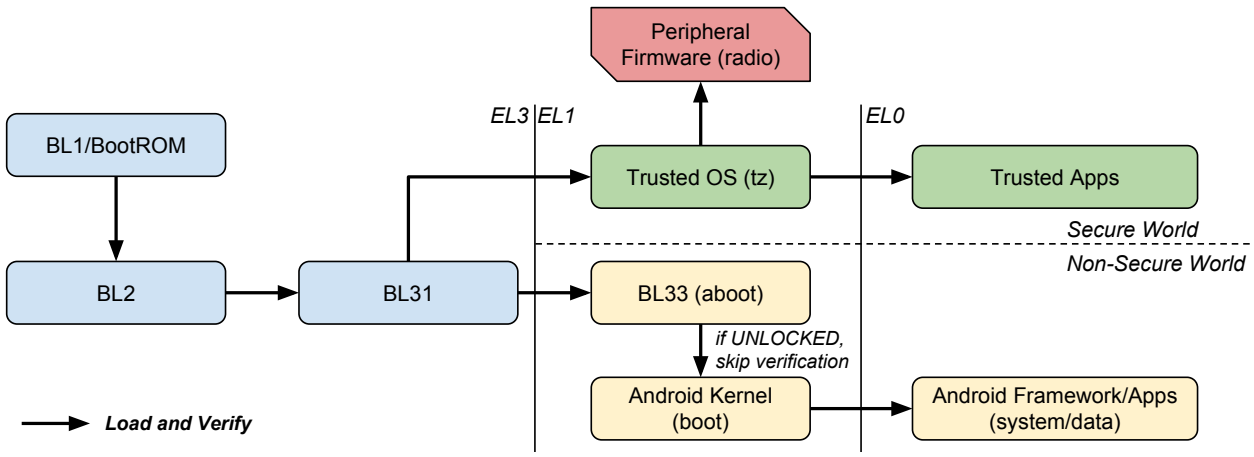

Figure 1: Overview of the Trusted/Verified Boot implementation according to the ARM and Google specifications. Between parentheses the name of the internal storage partition where the code is located in a typical implementation.

user to selectively break the Chain of Trust and run unsigned code, in a tamper-evident manner, referred to as *unlocking*. Devices will maintain a *security state* (either LOCKED or UNLOCKED) and properties of the transition between the two states must be enforced. With regard to Class B implementations, Google requires that:

- The bootloader itself must be verified with a hardware-backed key.
- If verification of the Android kernel with the OEM key (a key hard-coded by the device's manufacturer in the bootloader code) fails for any reason, a warning will be displayed to the user for at least five seconds. Then, if the bootloader is in the `LOCKED` state, the device will not boot, otherwise, if the bootloader is in the `UNLOCKED` state the Android kernel will be loaded.
- The device will only transition from the `LOCKED` state to the `UNLOCKED` state if the user first selects the "allow OEM Unlock" option from the Developer Options menu in Android's settings application, and then issues the Fastboot command `oem unlock`, or an equivalent action for devices without Fastboot.
- When the device's lock state changes for any reason, user-specific data will be rendered unreadable.

Beyond the guidelines, Android bootloaders (typically those that fall into Class B) also provide some means of rewriting partitions on internal storage over USB. Google suggests the use of the Fastboot protocol, also utilized for the locking and unlocking process, for this functionality.

3 Bootloaders in Practice

While the standards and guidelines on bootloader design in the previous section do cover many important security-related aspects, a significant amount of flexibility is given to OEMs to allow for functionality specific to their platforms. These involve both aspects of the hardware itself, but also logical issues with managing the security state of the device. Even though this flexibility makes it hard to reason about the actual security properties of bootloaders, it is difficult to envision a future for which these standards would be more precise. In fact, there are a number of technical reasons due to which the definition of these standards cannot be as comprehensive as we would hope.

One of these technical aspects is related to peripherals and additional custom hardware that is shipped with each device. While platform-specific code can be inserted at every stage in ARM's prototypical Trusted Boot implementation, no direction is given as to what code should be inserted at which points in the boot process. Additionally, initialization tasks cannot be too tightly coupled with the rest of the boot sequence, as peripheral hardware, such as modems, may incorporate code from different vendors and necessitate a modification of the initialization process. Furthermore, vendors of the final devices may not be able to alter earlier stages of the boot process to add necessary initialization code, as they may be locked to code supplied by the chip manufacturer. Finally, even aside from these issues, there are constraints on storage media. ROMs, such as those mandated for the first bootloader stage, tend to be small, and are inherently a write-once medium, precluding their use for any code that may need to be updated.

As an example, consider a mobile device with an onboard GSM or LTE modem. Depending on the hardware used, this modem could exist either as part of the System-on-a-chip (SoC) package or externally on another chip. Because the initialization of these two layouts has different requirements (e.g., initializing memory busses and

transferring code to an external modem vs. executing modem code on the same chip), this may need to happen at different phases in the boot process, where different levels of hardware access are available.

This also applies to various bootloader services, such as partition management and unlocking. Google's implementation provides the Fastboot protocol in the final-stage bootloader, but manufacturers are free to use alternative methods, as well as incorporate this functionality into other boot stages.

Where and how all of these features are implemented can have a significant security impact. If a stage in the bootloader is compromised, this could lead to the compromise of all following stages, along with any peripherals or secured storage that they manage. The impact of gaining control over a bootloader can be mitigated by using the lowest-possible Exception Level (discussed in the previous section), and performing tasks that involve taking potentially-untrusted input in later, less-privileged stages of the process. However, once again, other than the Trusted Firmware reference implementation, no guidance is given on how to manage exception levels with respect to bootloader features.

One aspect that increases the attack surface of modern bootloaders is that the code used to bootstrap additional hardware, such as modems, needs to be *updateable*, and thus needs to be stored on writable partitions. These writeable partitions, in turn, could be modified by an attacker with privileged code execution. Thus, it is critical that the content of these partitions is verified, such as by checking the validity of a cryptographic signature. This should ideally be accomplished by a previous bootloader stage, which thus needs to load, parse, and verify these partitions. This usage of data from writeable (and, as discussed previously, potentially attacker-controlled) partitions is what makes common memory corruption vulnerabilities in bootloaders very dangerous.

3.1 Bootloader Implementations

In the remainder of this section, we will explore four bootloaders from popular device manufacturers. These implementations all serve the same functions for their respective hardware platforms and aim to comply with both ARM and Google's standards, but do so in vastly different ways.

A comparison of the implementations can be found in Table 1. If an attacker can compromise the final stage bootloader, they will likely be able to also affect any functionality it contains, as well as any that it in turn loads, which in these cases, is the Android kernel and OS.

Qualcomm. The Qualcomm MSM chipset family is by far the most popular mobile chipset in devices today, representing over 60% of mobile devices [16]. While many manufacturers of MSM-based devices will customize the bootloader to fit their specific product's features, Qualcomm's "`aboot`" bootloader is still used with little modifications on many of them.

Vendor	EL	Fastboot	Modem Initialization	Peripherals Initialization
Qualcomm	EL1	✓	✗	✗
HiSilicon	EL3	✓	✓	✓
NVIDIA	EL1	✓	✗	✗
MediaTek	EL1	✓	✓	✗

Table 1: Final-stage Bootloader features, and which Exception Level they occur in

`aboot` is based on the Little Kernel (LK) open-source project, and provides the final stage non-secure OS loading functionality (equivalent to BL33 in ARM's reference implementation). In further similarity to BL33, it runs at EL1, giving it the same level of privilege as the kernel it aims to load. It conforms very closely to Google's Verified Boot guidelines, implementing the traditional set of Android-specific features, including Fastboot, recovery partition support, and unlocking. `aboot` can be used in either a Class A or Class B Verified Boot implementation, as Fastboot, and therefore unlocking can be disabled by the OEM or mobile carrier.

HiSilicon and Huawei. HiSilicon Kirin-based devices, such as those from Huawei, implement a very different bootloader architecture to the others we examined. Instead of merely being responsible for the initialization required to load Android, this loader also combines functionality usually found elsewhere in the boot process, such as initializing the radio hardware, secure OS, secure monitor, among others, giving it the equivalent roles of BL31, BL33, and BL2 in the ARM reference implementation. In fact, this bootloader is loaded directly by the ROM-based first-stage bootloader (BL1). To have the privilege necessary to perform all these tasks, HiSi's bootloader runs at EL3, and executes the Linux kernel in the boot partition at EL1 when it is finished. Along with its hardware initialization tasks, it also includes Fastboot support, by which it allows for unlocking.

MediaTek. Devices based on MediaTek chipsets, such as the Sony Xperia XA and other similar handsets, implement a bootloader similar to Qualcomm's but using a very different codebase. The Android-specific loader runs at EL1, and is also responsible for partition management and unlocking via Fastboot. Unlike Qualcomm's, this loader is also responsible for bootstrapping the modem's baseband firmware, meaning that any compromise in the bootloader could impact this critical component as well.

NVIDIA. NVIDIA's Tegra-based devices ship with a bootloader known as `hboot`. This bootloader is very similar to Qualcomm's, in that it runs at EL1, and implements only the fastboot functionality at this stage.

4 Unlocking Bootloaders

While security-focused bootloaders do significantly raise the bar for attackers wishing to persistently compromise the device, there are many cases in which "unlocking," as detailed in Section 2, has legitimate benefits. Only permitting the execution of signed code makes development of the Android OS itself problematic, as well as disallowing power-users from customizing and modifying the OS's code.

Of course, this is a very security-sensitive functionality; an attacker could unlock the bootloader and then modify the relevant partitions as a way of implementing a persistent rootkit. Google's Verified Boot standard covers the design of this important mechanism, discusses many high-level aspects of managing the device's security state (see Section 2), and even provides specifics about digital signatures to be used. However, as with the ARM specifications covering Trusted Boot, these specs must also allow for platform-specific variations in implementation, such as where or how these security mechanisms are integrated into the boot process.

Furthermore, there are many unspecified, implicit properties of Verified Boot that a valid implementation should enforce, to ensure that the device is protected from privileged code execution or unauthorized physical control. These properties include:

The device state should only transition from locked to unlocked with explicit user content. This is implicitly handled by requiring a command sent to Fastboot to unlock, as this usually requires physical access to activate, and causes a warning to be displayed to the user. Similarly, a malicious app — no matter how privileged it is — should not be able to silently unlock the bootloader.

Only the authorized owner of the device should be able to unlock the bootloader. This means that anyone in possession of a phone that is not theirs cannot simply access Fastboot or similar protocol (i.e., by rebooting the phone) and trigger an unlock. This is avoided on some devices through checking an additional flag called "OEM unlock," (or, more informally "allow unlock"). This flag is controlled by an option in the Android Settings menu, and it is only accessible if the device is booted and the user has authenticated (for instance, by inserting the correct "unlock pattern"). A proper implementation of Fastboot will honor the "OEM unlock" flag and it will refuse to unlock the bootloader if this flag is set to false.

Interestingly, there is no requirement on the storage of the device's security state. While the standard offers a suggestion about how to tie this state and its transitions to the security properties they wish to enforce, the exact storage of this information is left out, likely to account for hardware variations with respect to secured storage. Unfortunately, as we discuss in Section 5, specifics of such implementation details can negatively impact the security properties of the bootloader.

4.1 Unlocking vs Anti-Theft

Another interesting factor related to bootloaders and bootloader locking is the overall usability of a device by an attacker after it has been stolen. As mandated by laws [30] and industry standards [9], phones should implement mechanisms to prevent their usage when stolen. Google refers to this protection as Factory Reset Protection (FRP) [7], and it has been enabled in Android since version 5.0. In Google's own implementations, this means that the Android OS can restrict the usage of a phone, even after a factory-reset, unless the legitimate user authenticates.

This presents an interesting contradiction in relation to bootloader unlocking capabilities. First, since this mechanism is governed from within the OS, it could be leveraged by a malicious process with sufficient privilege. Of course, the original owner should be able to authenticate and restore the device's functionality, but this could still be used as a form of denial-of-service. Second, some manufacturers offer low-level firmware upload functionality, such as in the BL1 or BL2 stages, designed to restore the device to a working state in the event it is corrupted. This feature is in direct opposition to anti-theft functionality, as if a user can recover from any kind of corruption, this mechanism may be able to be bypassed. However, if this mechanism respects the anti-theft feature's restrictions on recovering partitions, this also means the device can be rendered useless by a sufficiently-privileged malicious process. In other words, there is an interesting tension between anti-theft and anti-bricking mechanisms: if the anti-theft is implemented correctly, an attacker could use this feature against the user to irremediably brick her device; vice versa, if an anti-bricking mechanism is available, a thief could use this mechanism to restore the device to a clean, usable state. In Section 8, we explore how this tension can be resolved.

5 Attacking Bootloaders

Regardless of implementation specifics bootloaders have many common functions that can be leveraged by an attacker. While they may appear to be very isolated from

possible exploitation, bootloaders still operate on input that can be injected by a sufficiently-privileged attacker. For example, the core task a bootloader must perform (that of booting the system) requires the bootloader to load data from non-volatile storage, figure out which system image on which partition to boot, and boot it. To enforce the Chain of Trust, this also involves parsing certificates and verifying the hash of the OS kernel, all of which involves further reading from the device's storage. In Class B implementations, the device's security state must also be consulted to determine how much verification to perform, which could be potentially stored in any number of ways, including on the device's storage as well. While bootloader authors may assume that this input is *trusted*, it can, in fact, be controlled by an attacker with sufficient access to the device in question.

In this work, we assume an attacker can control any content of the non-volatile storage of the device. This can occur in the cases that an attacker attains root privileges on the primary OS (assumed to be Android for our implementation). While hardware-enforced write protection mechanisms could limit the attacker's ability to do this, these mechanisms are not known to be in wide use today, and cannot be used on any partition the OS itself needs to routinely write to.

Given this attacker model, our goal is to automatically identify weaknesses, in deployed, real-world bootloader firmware, that can be leveraged by an attacker conforming to our attacker model to achieve a number of goals:

Code execution. Bootloaders process input, read from attacker-controlled non-volatile storage, to find, validate, and execute the next step in the boot process. What if the meta-data involved in this process is *maliciously crafted*, and the code processing it is not securely implemented? If an attacker is able to craft specified meta-data to trigger memory corruption in the bootloader code, they may achieve code execution *during the boot process*. Depending on when in the boot process this happens, it might grant the attacker control at exception levels considerably higher than what they may achieve with a root or even a kernel exploit on the device. In fact, if this is done early enough in the boot process, the attacker could gain control over Trusted Execution Environment initialization, granting them a myriad of security-critical capabilities that are unavailable otherwise.

Bricking. One aspect that is related to secure bootloaders is the possibility of "bricking" a device, i.e., the corruption of the device so that the user has no way to re-gain control of it. Bootloaders attempt to establish whether a piece of code is trusted or not: if such code is trusted, then the bootloader can proceed with their loading and execution. But what happens when the trust cannot be established? In the general case, the bootloader stops and issues a warning to the user. The user can, usually through the bootloader's recovery functionality (e.g., Fastboot) restore the device to a working state. However, if an attacker can write to the partition holding this recovery mechanism, the user has no chance to restore the device to an initial, *clean* state, and it may be rendered useless.

This aspect becomes quite important when considering that malware analysis systems are moving from using emulators to using real, physical devices. In this context, a malware sample has the capability of bricking a device, making it impossible to re-use it. This possibility constitutes a limitation for approaches that propose baremetal malware analysis, such as BareDroid [20].

One could think of having a mechanism that would offer the user the possibility of restoring a device to a clean state no matter how compromised the partitions are. However, if such mechanism were available, any anti-theft mechanism (as discussed in Section 4), could be easily circumvented.

Unsafe unlock. As discussed in Section 4, the trusted boot standard does not mandate the implementation details of storing the secure state. Devices could use an eMMC flash device with RPMB, an eFuse, or a special partition on the flash, depending on what is available. If the security state is stored on the device's flash, and a sufficiently-privileged process within Android can write to this region, the attacker might be able to unlock the bootloader, bypassing the requirement to notify the user. Moreover, depending on the implementation, the bootloader could thus be unlocked without the user's data being wiped.

In the next section, we will propose a design for an automated analysis approach to detect vulnerabilities in bootloader implementations. Unfortunately, our experiments in Section 7 show that currently deployed bootloaders are vulnerable to combinations of these issues. But hope is not lost – in Section 8, we discuss a mechanism that addresses this problematic aspect.

6 BOOTSTOMP

The goal of BOOTSTOMP is to automatically identify security vulnerabilities that are related to the (mis)use of attacker-controlled non-volatile memory, trusted by the bootloader's code. In particular, we envision using our system as an automatic system that, given a bootloader as input, outputs a number of alerts that could signal the presence of security vulnerabilities. Then, human analysts can analyze these alerts and quickly determine whether the highlighted functionality indeed constitute a security threat.

Bootloaders are quite different from regular programs, both regarding goals and execution environment, and

they are particularly challenging to analyze with existing tools. In particular, these challenges include:

- Dynamic analysis is infeasible. Because a primary responsibility of bootloaders is to initialize hardware, any concrete execution of bootloaders would require this hardware.
- Bootloaders often lack available source code, or even debugging symbols. Thus, essential tasks, including finding the entry point of the program, become much more difficult.
- Because bootloaders run before the OS, the use of syscalls and standard libraries that depend on this OS is avoided, resulting in all common functionality, including even functions such as `memcpy`, being reimplemented from scratch, thus making standard signature-based function identification schemes ineffective.

To take the first step at overcoming these issues, we developed a tool, called BOOTSTOMP, combining different static analyses as well as a dynamic symbolic execution (DSE) engine, to implement a taint analysis engine. To the best of our knowledge, we are the fist to propose a traceable offline (i.e., without requiring to run on real hardware) taint analysis completely based on dynamic symbolic execution. Other works as [24] [33] propose completely offline taint analyses on binaries. In contrast to our work, they implement static taint analyses, and are hence not based on dynamic symbolic execution.

The main problem with these types of approaches is that, though sound, they might present a high rate of false positives, which a human analyst has to filter out by manually checking them. Note that, in the context of taint analysis, a false positive result is a path which is mistakenly considered tainted. Furthermore, producing a trace (i.e., a list of basic blocks) representing a tainted path using a static taint analysis approach is not as simple as with symbolic execution.

On the other hand, our approach based on DSE, though not sound (i.e., some tainted paths might not be detected as explained in Section 7.4), presents the perk of returning a traceable output with a low false positives rate, meaning that the paths we detected as tainted are indeed tainted, as long as the initial taint is applied and propagated correctly. Note that there is a substantial difference between false positives when talking about taint analyses and when talking about vulnerability detection. Though our tool might return some false positives in terms of detected vulnerabilities, as seen in Section 7, false positives in tainted path detection are rare (we never found any in our experiments) as our tool is based on DSE. For a deeper discussion about the results obtained by BOOTSTOMP, please refer to Section 7.4.

With these considerations in mind, since the output of our analysis is supposed to be triaged by a human, we opted for a taint analysis based on DSE.

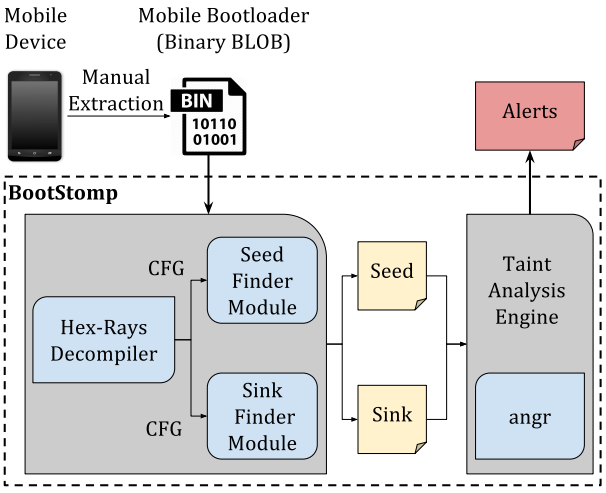

Figure 2: BOOTSTOMP's overview.

This section discusses the goal, the design features, and the implementation details of BOOTSTOMP.

6.1 Design

Our system aims to find two specific types of vulnerabilities: uses of attacker-controlled storage that result in a memory-corruption vulnerability, and uses of attacker-controlled storage that result in the unlocking of the bootloader. While these two kinds of bugs are conceptually different, we are able to find both using the same underlying analysis technique.

The core of our system is a taint analysis engine, which tracks the flow of data within a program. It searches for paths within the program in which a *seed of taint* (such as the attacker-controlled storage) is able to influence a *sink of taint* (such as a sensitive memory operation). The tool raises an alert for each of these potentially vulnerable paths. The human analyst can then process these alerts and determine whether these data flows can be exploitable.

Our system proceeds in the following steps, as shown in Figure 2:

Seed Identification. The first phase of our system involves collecting the seeds of taint. We developed an automated analysis step to find all the functions within the program that read data from any non-volatile storage, which are used as the seeds when locating memory corruption vulnerabilities. However, if the seeds have semantics that cannot be automatically identified, such as the unlocking mechanism of the bootloader, BOOTSTOMP allows for the manual specification of seeds by the analyst. This feature comes particularly in handy when source code is available, as the analyst can rely on

it to manually provide seeds of taint.

Sink Identification. We then perform an automated analysis to locate the sinks of taint, which represent code patterns that an attacker can take advantage of, such as bulk memory operations. Moreover, writes to the device's storage are also considered sinks for locating potentially attacker-controlled unlocking mechanisms.

Taint Analysis. Once the seeds of taint have been collected, we consider those functions containing the seed of taint and, starting from their entry point, perform a multi-tag taint analysis based on under-constrained symbolic execution [23] to find paths where seeds reach sinks. This creates alerts, for an analyst to review, including detailed context information, which may be helpful in determining the presence and the exploitability of the vulnerability.

In the remainder of this section, we will explore the details about each of these steps.

6.2 Seed Identification

```
1  #define SEC_X_LEN 255
2
3  void get_conf_x() {
4      //...
5      n = read_emmc("sec_x", a2, a3);
6      if (n < SEC_X_LEN) {
7          return;
8      }
9      //...
10 }
11
12 int get_user_data() {
13     // ...
14     if(!read_emmc(b1, b2, 0)) {
15         debug("EMMC_ERROR: no data read");
16         return -1;
17     }
18     // ...
19 }
```

Listing 1: By scanning every call site of `read_emmc`, BOOTSTOMP infers that the first parameter is a string, the third can assume the value zero, and the returned type is an integer.

For finding memory corruption vulnerabilities, our system supports the automatic identification of seeds of taint. We use approaches similar to those in prior work (e.g., [27]). We rely on error logging because there are many different mechanisms that may read from non-volatile memory, or different types of memory (plain flash memory vs. eMMC), and these error log strings give us semantic clues to help finding them. Our system looks for error logging functions using keywords as *mmc*, *oeminfo*, *read*, and *fail*, and avoiding keywords like *memory* and *write*.

This approach is useful for identifying functions that somehow retrieve the content from a device's storage. However, since the signature of these functions is not known, it is challenging to identify which argument of this function stores the receiving buffer. To determine the argument to be tainted, we use an approach based on type inference.

Ideally, the taint should only be applied to the seed's argument pointing to the memory location where the read data will be stored. As distinguishing pointers from integers is an undecidable problem [31], our analysis might dereference an integer in the process of applying the taint, resulting in a possible huge rate of false positive alarms. Nonetheless, during this study, we observed that, surprisingly, strings might not always be passed by reference to a function, but rather by value. During our analysis, we check every call site of the functions we retrieved using the above mentioned method and check the entity of every passed argument. If an argument is composed of only ASCII printable characters, we assume it is a string, and we consider the same argument to a be a string for every other call to the same function. When looking for the memory locations to apply the taint, we consider this information to filter out these arguments. We also do not taint arguments whose passed values are zeroes, as they might represent the NULL value.

As an example, consider Listing 1. First, BOOTSTOMP retrieves the function *read_emmc* as a possible seed function, by analyzing the error log at line 18. Then, it scans every call site of *read_emmc* and infers that the returned value is an integer (as it is compared against an integer variable), the first parameter is a string and the third parameter can assume the value zero. As *read_emmc* is a candidate seed function, it has to store the content read from a non-volatile storage in a valid buffer, pointed by a non-null pointer. Therefore, BOOTSTOMP applies the taint only to the second parameter of *read_emmc* (*a*2 and *b*2). Note that, as the receiving buffer could be returned by a seed function, if the type of the returned value cannot be inferred, the variable it is assigned to is tainted as well. Note that, when a tainted pointer is dereferenced, we taint the entire memory page it points to.

In the case of locating unlocking-related vulnerabilities, there is no bootloader-independent way of locating the unlocking function, since the implementation details significantly vary. Therefore, BOOTSTOMP also supports supplying the seeds manually: an analyst can thus perform reverse-engineering to locate which function implements the "unlock" functionality and manually indicate these to our analysis system. While this is not a straightforward process, there is a specific pattern a human analyst can rely on: Fastboot's main command handler often includes a basic command line parser that determines which functionality to execute, and the strings involved are often already enough to quickly pin-

point which function actually implements the "unlock" functionality.

6.3 Sink Identification

Our automatic sink identification strategy is designed to locate four different types of sinks:

memcpy-like functions. BOOTSTOMP locates memcpy-like functions (e.g., memcpy, strcpy) by looking for semantics that involve moving memory, unchanged, from a source to a destination. As mentioned above, there are no debugging symbols, and standard function signature-based approaches would not be effective. For this reason, we rely on a heuristic that considers the basic blocks contained within each function to locate the desired behavior. In particular, a function is considered memcpy-like if it contains a basic block that meets the following conditions: 1) Loads data from memory; 2) stores this same data into memory; 3) increments a value by one unit (one word, one byte, etc). Moreover, since it is common for bootloaders to rely on wrapper functions, we also flag functions that directly invoke one (and only one) function that contains a block satisfying the above conditions.

We note that there may be several other functions that, although satisfy these conditions as well, do not implement a memcpy-like behavior. Thus, we rely on an additional observation that memcpy and strcpy are among the most-referenced functions in a bootloader, since much of their functionality involves the manipulation of chunks of memory. We therefore sort the list of all functions in the program by their reference count, and consider the first 50 as possible candidates. We note that, empirically, we found that memcpy functions often fall within the top five most-referenced functions.

Attacker-controlled dereferences. BOOTSTOMP considers memory dereferences controlled by the attacker as sinks. In fact, if attacker-controlled data reaches a dereference, this is highly indicative of an attacker-controlled arbitrary memory operation.

Attacker-controlled loops. We consider as a sink any expression used in the guard of a loop. Naturally, any attacker able to control the number of iterations of a loop, could be able to mount a denial-of-service attack.

Writes to the device's storage. When considering unlocking vulnerabilities, we only use as sinks any write operation to the device's storage. This encodes the notion that an unlocking mechanism that stores its security state on the device's storage may be controllable by an attacker. To identify such sinks, we adopt the same keyword-based approach that we employed to identify the seeds of taint (i.e., by using relevant keywords in error logging messages).

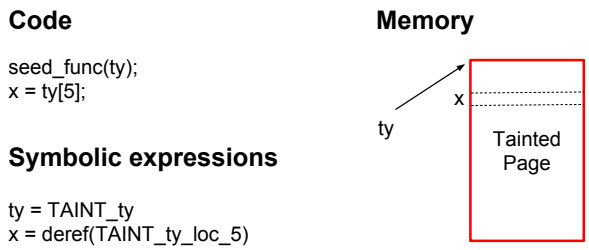

Figure 3: Taint propagation example.

6.4 Taint Tracking

While we cannot execute the bootloaders concretely, as we discussed above, we can execute them symbolically. Our interest is in the path the data takes in moving from a seed to a sink, and path-based symbolic execution lets us reason about this, while implicitly handling taint-propagation. Given a bootloader, along with the seeds and sinks identified in the previous stages, the analysis proceeds as follows:

- Locate a set of *entry points*, defined as any function that directly calls one of the identified seeds.
- Begin symbolic execution at the beginning of each entry point. Note that, before starting to symbolically execute an entry point, BOOTSTOMP tries to infer, looking for known header as ELF, where the global data is located. If it does find it, it unconstrains each and every byte in it, so to break any assumptions about the memory content before starting to analyze the entry point.
- When a path encounters a function, either step over it, or step into it, considering the code traversal rules below.
- When a path reaches a seed, the appropriate taint is applied, per the taint policy described below.
- Taint is propagated implicitly, due to the nature of symbolic execution. This includes the return values of functions handling tainted data.
- If a path reaches a sink affected by tainted data, an alert is raised.

Code traversal. To avoid state explosion, we constrain the functions that a path will traverse, using an *adaptive inter-function level*. Normally, the inter-function level specifies how many functions deep a path would traverse. However, the handling of tainted data in our analysis means that we implicitly care more about those functions which consume tainted data. Therefore, we only step into functions that consume tainted data, up to the inter-function level. For our experiments, we fixed the inter-function level at 1. More in detail, our analysis traverses the code according to the following rules:

- When no data is tainted, functions are not followed, such as at the beginning of an entry point, before the seed has been reached. Particularly, this path selection criteria allows us to have a fast yet accurate taint analysis, at the expense of possible false negative results, as some tainted paths might not be discovered due to some missed data aliases.
- Functions are not followed if their arguments are not tainted.
- Analysis terminates when all the possible paths between the entry point and its end are analyzed, or a timeout is triggered. Note that we set a timeout of ten minutes for each entry point. As we will show in Section 7.2 our results indicate that this is a very reasonable time limit.
- Unless any of the above conditions are met, we follow functions with an inter-function level of 1. In other words, the analysis will explore at least one function away from the entry point.
- We explore the body of a loop (unroll the loop) exactly once, and then assume the path exits the loop.

(Under-Constrained) Symbolic Execution. Our approach requires, by design, to start the analysis from arbitrary functions, and not necessarily from the bootloader's entrypoint, which we may not even be able to determine. This implies that the initial state may contain fewer constraints than it should have at that particular code point. For this reason, we use under-constrained symbolic execution, first proposed by Ramos et al. [23], which has been proven to reach good precision in this context.

Multi-tag taint analysis. To reach a greater precision, our system implements a multi-tag tainting approach [18]. This means that, instead of having one concept of taint, each taint seed generates tainted data that can be uniquely traced to where it was generated from. Furthermore, we create unique taint tags for each invocation of a seed in the program. This means, for example, that if a taint seed is repeatedly called, it will produce many different taint tags. This improves precision when reasoning about taint flow.

Taint propagation and taint removal. Taint is implicitly propagated using symbolic execution, as no constraint is ever dropped. This means that if a variable x depends on a tainted variable ty, the latter will appear in the symbolic expression of the former. As an example consider Figure 3. Suppose that a location of an array pointed by ty is dereferenced and assigned to x, such as $x = ty[5]$. Assuming now that ty is tainted because pointing to data read from an untrusted storage, the memory page it points to will be tainted, meaning that every memory location within that page will contain a symbolic variable in the form $TAINT\_ty\_loc\_i$. After the instruction $x = ty[5]$, the symbolic variable x will be in the form $deref(TAINT\_ty\_loc\_5)$.

On the other hand, taint is removed in two cases. Implicitly when a non-tainted variable or value is written in a tainted memory location, or when a tainted variable is constrained within non tainted values. As an example and by referring to the above tainted variable x, if a check such as $if(x < N)$, where N is non-tainted value, is present, x would get untainted.

Concretization strategy. When dealing with memory writes in symbolic locations, target address needs to be concretized. Unlike existing work [5], our analysis opts to concretize values with a bias toward smaller values in the possible range (instead of being biased toward higher values). This means that, when a symbolic variable could be concretized to more than one value, lower values are preferred. In previous work, higher values were chosen to help find cases where memory accesses off the end of an allocated memory region would result in vulnerabilities. However, these values may not satisfy conditional statements in the program that expect the value to be "reasonable," (such as in the case of values used to index items in a vector) and concretizing to lower values allows paths to proceed deeper into the program. In other words, we opt for this strategy to maximize the number of paths explored. Also, when BOOTSTOMP has to concretize some expressions, it tries to concretize different unconstrained variables to different (low) values. This strategy aims to keep the false positive rate as low as possible. For a deeper discussion about how false negatives and positive might arise, please refer to Section 7.4.

Finally, our analysis heavily relies on angr [28] (taint engine) and IDA Pro [11] (sink and seed finding).

7 Evaluation

This section discusses the evaluation of BOOTSTOMP on bootloaders from commercial mobile devices. In particular, for each of them, we run the analysis tool to locate the two classes of vulnerabilities discussed in Section 6. As a first experiment, we use the tool to automatically discover potential paths from attacker-controllable data (i.e., the flash memory) to points in the code that could cause memory corruption vulnerabilities. As a second experiment, we use the tool to discover potential vulnerabilities in how the lock/unlock mechanism is implemented. We ran all of our experiments on a 12-Core Intel machine with 126GB RAM and running Ubuntu Linux 16.04.

We first discuss the dataset of bootloaders we used, an analysis of the results, and an in-depth discussion of several use cases.

7.1 Dataset

For this work, we considered five different bootloaders. These devices represent three different chipset families: Huawei P8 ALE-L23 (Huawei / HiSilicon chipset), Sony Xperia XA (MediaTek chipset), and Nexus 9 (NVIDIA Tegra chipset). We also considered two versions of the LK-based bootloader, developed by Qualcomm. In particular, we considered an old version of the Qualcomm's LK bootloader (which is known to contain a security vulnerability, CVE-2014-9798 [19]) and its latest available version (according to the official git repository [22]).

7.2 Finding Memory Corruption

We used BOOTSTOMP to analyze the five bootloaders in our dataset to discover memory corruption vulnerabilities. These vulnerabilities could result in arbitrary code execution or denial-of-service attacks. Table 2 summarizes our findings. In particular, the table shows the number of seeds, sinks, and entry points identified in each bootloader. The table also shows the number of alerts raised for each bootloader. Out of a total of 36, for 12 of them, the tool identified a potential path from a source to memcpy-like sink, leading to the potential of a buffer overflow. The tool raised 5 alerts about the possibility of a tainted variable being dereferenced, which could in turn constitute a memory corruption bug. Finally, for 19, the tool identified that tainted data could reach the conditional for a loop, potentially leading to denial-of-service attacks. We then manually investigated all the alerts to determine whether the tool uncovered security vulnerabilities. Our manual investigation revealed a total of seven security vulnerabilities, *six* of which previously-unknown (five are already confirmed by the respective vendors), while the remaining one being the previously-known CVE-2014-9798 affecting an old version of Qualcomm's LK-based bootloader. Note that, as BOOTSTOMP provides the set of basic blocks composing the tainted trace together with the involved seed of taint and sink, manual inspection becomes easy and fast even for not-so-experienced analysts. We also note that, due to bugs in angr related to the analysis of ARM's THUMB-mode instructions, the MediaTek bootloader was unable to be processed correctly.

These results illustrate some interesting points about the scalability and feasibility of BOOTSTOMP. First, we note that each entry point's run elapsed on average less than five minutes (Duration per EP column), discovering a total of seven bugs. We ran the same set of experiments using a time limit of 40 minutes. Nonetheless, we noticed that no additional alerts were generated. These two results led us to believe that a timeout of ten minutes (i.e., twice as the average analysis run) was reasonable. Second, we noted a peak in the memory consumption while testing our tool against LK bootloaders. After investigating, we found out that LK was the only bootloader in the dataset having a well known header (ELF), which allowed us to unconstrain all the bytes belonging to the *.data* and *.bss* segments, as stated in Section 6. Third, we note that the overall number of alerts raised is very low, in the range that a human analyst, even operating without debugging symbols or other useful reverse-engineering information, could reasonably analyze them. Finally, as we show in the table, more than one alert triggered due to the same underlying vulnerability; the occurrence of multiple alerts for the same functionality was a strong indicator to the analyst of a problem. This can occur when more than one seed fall within the same path generating a unique bug, for instance, when more than one tainted argument is present in a memcpy-like function call.

With this in mind, and by looking at the table, one can see that around 38.3% of the tainted paths represent indeed real vulnerabilities. Note also that in the context of tainted paths, none of the reported alerts were false positives (i.e., not tainted paths), though false positives are theoretically possible, as explained in Section 7.4.

Our tool uncovered five new vulnerabilities in the Huawei Android bootloader. First, an arbitrary memory write or denial of service can occur when parsing Linux Kernel's device tree (DTB) stored in the boot partition. Second, a heap buffer overflow can occur when reading the root-writable `oem_info` partition, due to not checking the `num_records` field. Additionally, a user with root privileges can write to the `nve` and `oem_info` partitions, from which both configuration data and memory access permissions governing the phone's peripherals (e.g., modem) are read. The remaining two vulnerabilities will be described in detail below.

Unfortunately, due to the architecture of the Huawei bootloader, as detailed in Section 3.1, the impact of these vulnerabilities on the security of the entire device is quite severe. Because this bootloader runs at EL3, and is responsible for the initialization of virtually all device components, including the modem's baseband firmware and Trusted OS, this vulnerability would not only allow one to break the chain of trust, but it would also constitute a means to establish persistence within the device that is not easily detectable by the user, or available to any other kind of attack. Huawei confirmed these vulnerabilities.

BOOTSTOMP also discovered a vulnerability in NVIDIA's `hboot`. `hboot` operates at EL1, meaning that it has equivalent privilege on the hardware as the Linux kernel, although it exists earlier in the Chain of Trust, and therefore its compromise can lead to an attacker gaining persistence. We have reported the vulnerability to NVIDIA, and we are working with them on a fix.

Bootloader	Seeds	Sinks	Entry Points	Total Alerts			Bug-Related Alerts			Bugs	Timeout	Total Duration	Duration per EP	Memory
				loop	deref	memcpy	loops	deref	memcpy					
Qualcomm (Latest)	2	1	3	1	1	2	0	0	0	**0**	1	12:49	04:16	512
Qualcomm (Old)	3	1	5	3	0	5	0	0	4	**1**	0	10:14	02:03	478
NVIDIA	6	1	12	7	0	0	1	0	0	**1**	0	24:39	02:03	248
HiSilicon	20	4	27	8	4	5	8	4	3	**5**	1	21:28	00:48	275
MediaTek	2	2	2	-	-	-	-	-	-	**-**	-	00:08	00:04	272
Total	33	9	49	19	5	12	9	4	7	**7**	2	69:18	09:14	1785

Table 2: Alerts raised and bugs found by BOOTSTOMP's taint analysis. Time is reported in `MM:SS` format, memory in MB.

Finally, we rediscovered a previous vulnerability reported against Qualcomm's aboot, CVE-2014-9798. These vulnerabilities allowed an attacker to perform denial-of-service attack. However, this vulnerability has been patched, and our analysis of the current version of aboot did not yield any alerts.

Case study: Huawei memory corruption vulnerability. BOOTSTOMP raised multiple alerts concerning a function, whose original name we believe to be `read_oem()`. In particular, the tool highlighted how this function reads content from the flash and writes the content to a buffer. A manual investigation revealed how this function is vulnerable to memory corruption. In particular, the function reads a monolithic record-based datastructure stored in a partition on the device storage known as `oem_info`. This partition contains a number of *records*, each of which can span across multiple *blocks*. Each block is 0x4000 bytes, of which the first 512 bytes constitute a header. This header contains, among others, the four following fields: `record_id`, which indicates the type of record; `record_len`, which indicates the total length of the record; `record_num`, which indicates the number of blocks that constitute this record; `record_index`, which is a 1-based index.

The vulnerability lies in the following: the function will first scan the partition for blocks with a matching `record_id`. Now, consider a block whose record_num is 2 and whose record_index is 1. The fact that record_num is 2 indicates that this record spans across two different blocks. At this point, the `read_oem` function assumes that the length of the current block is the maximum, i.e., 0x4000, and it will thus copy all these bytes into the destination array, *completely ignoring the `len` value passed as argument*. Thus, since the `oem_info` partition can be controlled by an attacker, an attacker can create a specially crafted record so that a buffer overflow is triggered. Unfortunately, this bootloader uses this partition to store essential information that is accessed at the very beginning of every boot, such as the bootloader's logo. Thus, an attacker would be able to fully compromise the bootloader, fastboot, and the chain of trust. As a result, it would thus be possible for an attacker to install a persistent rootkit.

Case study: Huawei arbitrary memory write. The second case study we present is related to an arbitrary memory write vulnerability that our tool identified in Huawei's bootloader. In particular, the tool raised a warning related to the `read_from_partition` function. Specifically, the tool pinpointed the following function invocation `read_from_partition("boot", hdr->kernel_addr)`, and, more precisely, the tool highlighted that the structure `hdr` can be attacker-controllable. Manual investigation revealed that not only `hdr` (and its field, including `kernel_addr`) are fully controllable by an attacker, but that the function actually reads the content from a partition specified as input ("boot", in this case), and it copies its content to the address specified by `hdr->kernel_addr`. Since this destination address is attacker-controllable, an attacker could rely on this function to write arbitrary memory (by modifying the content of the "boot" partition) to an arbitrary address, which the attacker can point to the bootloader itself. We note that this vulnerability is only exploitable when the bootloader is unlocked, but, nonetheless, it is a vulnerability that allows an attacker to run arbitrary code as the bootloader itself (and not just as part of non-secure OS). Moreover, the next section provides evidence that, at least for this specific case, it is easy for an attacker to unlock the bootloader.

7.3 Analyzing (In)Secure State Storage

As a second use case for our tool, we use it to analyze the same five bootloaders we previously consider to determine how their security state (i.e., their lock/unlock state) is stored. In particular, as we discussed in Section 4, if the bootloader merely stores the security state on one of the flash partitions, then an attacker may be able to change the content of this partition, unlock the phone without the user's consent, and thus violate one of Google's core Verified Boot principles.

To run this experiment, we begin with the manually-identified unlocking functionality, as described in Section 6.2, and locate paths that reach automatically-identified writes to the device's storage. This means that each bootloader has one entry point. Table 3 shows the overall results of this experiment, including the number

Bootloader	Sinks	Potentially vulnerable?	Timeout	Duration	Remarks
Qualcomm (Latest)	6	✓	✗	01:00	Detected write on flash and mmc
Qualcomm (Old)	4	✓	✗	00:40	Detected write on flash and mmc
NVIDIA	9	✗	✗	02:21	Memory mapped IO
HiSilicon	17	✓	✓	10:00	Write `oeminfo`
MediaTek	1	✗	✓	10:00	Memory mapped IO

Table 3: Alerts raised by BOOTSTOMP on potentially vulnerable `write` operation inside unlock routines. Time is reported in `MM:SS` format.

of possible write operations to the device's storage that occurred within the unlocking functionality. Our system was easily able to locate paths in Qualcomm's bootloader (both the old and the newest version) and Huawei's bootloader where the security state was written to the device's non-volatile storage. Upon manual investigation, we discovered that Qualcomm's simply stores the bit '1' or '0' for whether the device is locked. Huawei's stores a static hash, but can still be recovered and replayed (see case study at the end of this section). In both cases, writing the needed value to the flash will unlock the bootloader, potentially bypassing the mandatory factory reset, if additional steps are not taken to enforce it, such as those mentioned in Section 8. Our tool did not identify any path to non-volatile storage for the NVIDIA's or MediaTek's bootloaders. Upon manual investigation, we discovered that these two bootloaders both make use of memory-mapped I/O to write the value, which could map to anything from the flash to special tamper-resistant hardware. Thus, we cannot exclude the presence of vulnerabilities.

Case Study: Huawei bootloader unlock. Our tool identified a path from a function, which we believe to be called `oem_unlock`, to a "write" sink. Upon manual investigation, we were able to determine the presence of a vulnerability in the implementation of this functionality, as shown in Figure 4. In a normal scenario, the user needs to provide to the bootloader a device-specific `unlock_code`. Such code can be obtained by a user through Huawei's website, by providing the hardware identifiers of the device. The problem lies in the fact that the "correct" MD5 of the `unlock_code`, `<target_value>`, is stored in a partition of the device's storage. Thus, even if it not possible to determine the correct `unlock_code` starting from its hash, an attacker could just *reuse* the *correct* MD5, compute the expected `unlock_state`, and store it to the `oem_info` partition, thus entirely bypassing the user's involvement.

```
1 x = md5sum(unlock_code);
2 if (x == ``<target_value>'') {
3   unlock_state = custom_hash(x);
4   write(oem_info,unlock_state);
5 }
```

Figure 4: Implementation of the (vulnerable) unlock functionality in Huawei's bootloader.

7.4 Discussion

As stated in Section 6, and as demonstrated by the results in this section, our tool might present some false negatives as well as false positives. In this section we consider the results achieved by our taint analysis engine, and we discuss how false positive and false negatives might arise.

As symbolic execution suffers from the path explosion problem, generally speaking, not all the possible paths between two program points can be explored in a finite amount of time. This might cause some tainted paths to be missed, causing some vulnerabilities to be missed. False negatives might be present also because BOOTSTOMP does not follow function calls when no taint is applied. This approach is very useful, since it makes our tool faster as less code has to be analyzed, but it might miss some correlation between pointers. In fact, if a future tainted variable is aliased, within a skipped function to a variable whose scope falls within the current function, and this variable later happens to reach a sink, it will not be reported.

Furthermore, since BOOTSTOMP relies on a maximum fixed inter-function level, it might not follow all the function calls it encounters, possibly resulting in some tainted variables not to be untainted as well as some pointer aliases not being tainted. This problem might create both false positives and false negatives.

Additionally, false positives could possibly arise from the fact that not all the reported tainted paths lead to actual vulnerabilities. In fact, when the initial taint is applied, our tool tries to understand which parameter represents the variable(s) that will point to the read data, as explained in Section 6. If the taint is not applied correctly, this will result in false positive results. Note however, that our tool would taint every parameter that our type inference heuristic does not exclude. Therefore, false negatives are not possible in this case.

Our concretization strategy could possibly introduce both false positives and false negatives. Given two unconstrained pointers, intuitively it is unlikely that they

will point to the same memory location. Therefore, the most natural choice is to concretize them (if necessary) to two different values. Assuming that these two pointers are indeed aliases, if one of them is tainted and the other reaches a sink, no alarm will be raised causing then a false negative. On the other hand if both of them are tainted, but the former becomes untainted and the latter reaches a sink, an alarm would be raised causing then a false positive. According to our observations these cases are very rare though, as we never encountered two unconstrained pointers that happened to be aliases.

Finally, it is worth noting that while we found some tainted paths that were not leading to actual vulnerabilities, our tool never detected a tainted path which was supposed to be untainted.

8 Mitigations

In this section, we will explore ways of mitigating the vulnerabilities discovered in the previous section. With the increasing complexity of today's devices, it may be difficult to completely ensure the correctness of bootloaders, but taking some simple steps can dramatically decrease the attack surface.

As we have discussed throughout the previous sections, the goal of Trusted Boot and Verified Boot is to prevent malicious software from persistently compromising the integrity of the operating system and firmware. The attacks we discovered all rely on the attacker's ability to write to a partition on the non-volatile memory, which the bootloader must also read. We can use hardware features present in most modern devices to remove this ability.

Binding the Security State. Google's implementations of Verified Boot bind the security state of the device (including the lock/unlock bit) to the generation of keys used to encrypt and decrypt user data, as described in Section 2.3. While not specifically requiring any particular storage of the security state, this does ensure that if the security state is changed, the user's data is not usable by the attacker, and the system will not boot without first performing a factory reset. This, along with the cryptographic verification mandated by Verified Boot, achieves the goals Google sets, but does not completely shield the bootloader from arbitrary attacker-controlled input while verifying partitions or checking the security state.

Protect all partitions the bootloader accesses. Most modern mobile devices utilize non-volatile storage meeting the eMMC specification. This specifies the set of commands the OS uses to read and write data, manage partitions, and also includes hardware-enforced security features. Since version 4.4, released in 2009 (a non-public standard, summarized in [17]), eMMC has supported *Power-on Write-Lock*, which allows individual partitions to be selectively write-protected, and can only be disabled when the device is rebooted. The standard goes as far as to specify that this must also be coupled with binding the reset pin for the eMMC device to the main CPU's reset pin, so that intrusive hardware attacks cannot be performed on the eMMC storage alone.

While we are not able to verify directly whether any handsets on the market today makes use of this feature, we note that none of the devices whose bootloaders we examined currently protect the partitions involved in our attacks in this manner. Furthermore, we note that many devices today make use of other features from the same standard, including Replay-protected Memory Blocks (RPMB) [17] to provide a secure storage accessible from Secure-World code.

eMMC Power-on Write-protect can be used to prevent any partition the bootloader must read from being in control of an attacker with root privileges. Before executing the kernel contained in the boot partition, the final stage bootloader should enable write protection for every partition which the bootloader must use to boot the device. In Android, the `system` and `boot` partitions contain entirely read-only data (excluding during OS updates), which the bootloader must read for verification, and therefore can be trivially protected in this way. To close any loopholes regarding unlocking the bootloader, the partition holding device's security state should also be write-protected. The `misc` partition used by Qualcomm devices, for example is also used to store data written by the OS, so the creation of an additional partition to hold the security state can alleviate this problem.

This does not impede any functionality of the device, or to our knowledge, cause any impact to the user whatsoever. Of course, this cannot be used to protect partitions the OS must write to. While the OS does need to write to `system` and `boot` to perform routine software updates, this too can be handled, with only small changes. If an update is available, the bootloader should simply not enable write-protection when booting, and perform the update. This increases only marginally the attack surface, adding only the update-handling code in the bootloader.

It should be noted that this method cannot protect the status of the "Allow OEM Unlock" option in the Android Settings menu, which by its very design must be writable by the OS. This means that a privileged process can change this setting, but unlocking the bootloader still requires physical control of the device as well.

Alternative: Security State in RPMB. eMMC Power-on Write Lock can be used to protect any partition which is not written to by the OS. If, for whatever reason, this is not possible, this could also be stored in the Replay-protected Memory Block (RPMB) portion of the eMMC

module.

We can enforce the property that the OS cannot tamper with the security state by having the Trusted OS, residing in the secure world, track whether the OS has booted, and only allow a change in the security state if the bootloader is running. Using RPMB allows us to enforce that only TrustZone can alter this state, as it holds the key needed to write the data successfully.

When the device boots to the final stage bootloader, it will signal to TrustZone, allowing modifications to the security state via an additional command. Once the bootloader is ready to boot the Android OS, it signals again to TrustZone, which disallows all writes to the device until it reboots.

While this requires minor modifications to the Trusted OS and final-stage bootloader, it does not require a change in the write-protection status or partition layout.

9 Related Work

Trusted Boot Implementations and Vulnerabilities Methods that utilize the bootloader to bootstrap a trusted environment have been studied extensively in the past. Recent Intel-based PC systems utilize UEFI Secure Boot, a similar mechanism for providing verification of operating system components at boot-time. This too has been prone to vulnerabilities.

Specifically, Wojtczuk et al., studied how unprivileged code can exploit vulnerabilities and design flaws to tamper with the SPI-flash content (containing the code that is first executed when the CPU starts), completely breaking the chain-of-trust [34] in Intel systems. Kallenberg et al., achieved a similar goal by exploiting the update mechanisms exposed by UEFI code [14]. Researchers have also shown how the chain-of-trust can be broken on the Mac platform, using maliciously crafted Thunderbolt devices [13, 12]. Other research focused on the way in which Windows bootloader, built on top of UEFI, works and how it can be exploited [4, 25]. Bazhaniuk et al., provided a comprehensive study of the different types of vulnerabilities found in UEFI firmware and propose some mitigations [2], whereas Rutkowska presented an overview of the technologies available in Intel processors, which can be used to enforce a trusted boot process [26].

All these works show how the complexity of these systems, in which different components developed by different entities have to collaborate, and the different, sometimes conflicting, goals they try to achieve has lead to both "classic" vulnerabilities (such as memory corruption), but also to hard-to-fix design issues. Our work shows how this is true also in the mobile world.

While all of the previously mentioned works rely entirely on manual analysis, Intel has recently explored auditing its own platform using symbolic execution [3]. This is similar in approach to our work, but it has a different goal. In particular they focus on detecting a very specific problem in the UEFI-compliant implementation of BIOS (out of bound memory accesses). Instead, we focus on vulnerabilities explicitly triggerable by an attacker inside the bootloader code of ARM mobile device, considering both memory corruption as well as additional logic flaws related to unlocking.

A recent work, BareDroid [20], proposes and implements modifications to the Android boot process to build a large-scale bare-metal analysis system on Android devices. Although with a different goal, in this work, authors introduce some aspects related to ours, such as difficulties in establishing a chain of trust in Android devices and how malware could permanently brick a device. We expand and integrate their findings, comparing different implementations and devices.

Automatic Vulnerability Discovery Our approach, as outlined in Section 6, attempts to automatically locate vulnerabilities statically. Other approaches include fully-dynamic analysis, such as coverage-based fuzzing [36], or hybrid systems, such as Driller [10] and Dowser [29], which switch between the static and dynamic analysis to overcome the limitations of both. Unfortunately, we could not use any approach leveraging concrete dynamic execution, as it is currently impossible to overcome the tight coupling of bootloaders and the hardware they run on. Previous work has looked into hardware-in-the-loop approaches [35, 15] to address this issue, by passing events directed at hardware peripherals to a real hardware device tethered to the analysis system. Unfortunately, none of this work can be adapted to our platform, as the hardware under analysis lacks the necessary prerequisites (e.g., a JTAG interface or a completely unlocked primary bootloader) that would be needed.

Many previous works have also proposed statically locating memory corruption vulnerabilities, including Mayhem [5] and IntScope [32], focusing on user-land programs. These approaches are not directly applicable to our goals, since in our work we are not focusing solely on memory corruption and our analysis requires an ad-hoc modeling and identification sources and sinks. FirmAlice [27] proposes a technique for locating authentication bypass vulnerabilities in firmware. The vulnerabilities we wish to locate stem from the presence and specific uses of "user input" (in this case, data from the non-volatile storage), whereas FirmAlice can detect its absence, en route to a pre-defined program state.

10 Conclusion

We presented an analysis of modern mobile device bootloaders, and showed that current standards and guidelines are insufficient to guide developers toward creating secure solutions. To study the impact of these design decisions, we implemented a static analysis approach able to find locations where bootloaders accept input from an adversary able to compromise the primary operating system, such as parsing data from partitions on the device's non-volatile storage. We evaluated our approach on bootloaders from four major device manufacturers, and discovered six previously-unknown memory corruption or denial of service vulnerabilities, as well as two unlock-bypass vulnerabilities. We also proposed mitigation strategies able to both limit the attack surface of the bootloader and enforce various desirable properties aimed at safeguarding the security and privacy of users.

Acknowledgements

We would like to thank our reviewers for their valuable comments and input to improve our paper. We would also like to thank Terry O. Robinson for several insightful discussions. This material is based on research sponsored by the Office of Naval Research under grant numbers N00014-15-1-2948 and N00014-17-1-2011, the DARPA under agreement number N66001-13-2-4039 and the NSF under Award number CNS-1408632. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, or the U.S. Government. This work is also sponsored by a gift from Google's Anti-Abuse group.

References

[1] ARM. ARM TrustZone. `http://www.arm.com/products/processors/technologies/trustzone/index.php`, 2015.

[2] BAZHANIUK, O., BULYGIN, Y., FURTAK, A., GOROBETS, M., LOUCAIDES, J., MATROSOV, A., AND SHKATOV, M. Attacking and Defending BIOS in 2015. In *REcon* (2015).

[3] BAZHANIUK, O., LOUCAIDES, J., ROSENBAUM, L., TUTTLE, M. R., AND ZIMMER, V. Symbolic execution for bios security. In *Proceedings of the 2015 USENIX Conference on Offensive Technologies* (Washington, DC, USA, 2015), WOOT'15.

[4] BULYGIN, Y., FURTAK, A., AND BAZHANIUK, O. A tale of one software bypass of Windows 8 Secure Boot. *Black Hat USA* (2013).

[5] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (San Jose, CA, USA, 2012), SP '12.

[6] GITHUB. ARM Trusted Firmware. `https://github.com/ARM-software/arm-trusted-firmware`, 2017.

[7] GOOGLE. `https://support.google.com/nexus/answer/6172890?hl=en`, 2016.

[8] GOOGLE. Verifying Boot. `https://source.android.com/security/verifiedboot/verified-boot.html`, 2017.

[9] GSMA. Anti-theft Device Feature Requirements, 2016.

[10] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 2013 USENIX Conference on Security* (Washington, DC, USA, 2013), SEC'13.

[11] HEX-RAYS. IDA Pro. `https://www.hex-rays.com/products/ida/index.shtml`, 2017.

[12] HUDSON, T., KOVAH, X., AND KALLENBERG, C. ThunderStrike 2: Sith Strike. *Black Hat USA* (2015).

[13] HUDSON, T., AND RUDOLPH, L. Thunderstrike: Efi firmware bootkits for apple macbooks. In *Proceedings of the 2015 ACM International Systems and Storage Conference* (New York, NY, USA, 2015), SYSTOR '15.

[14] KALLENBERG, C., KOVAH, X., BUTTERWORTH, J., AND CORNWELL, S. Extreme privilege escalation on Windows 8/UEFI systems. *BlackHat, Las Vegas, USA* (2014).

[15] KOSCHER, K., KOHNO, T., AND MOLNAR, D. SURROGATES: enabling near-real-time dynamic analyses of embedded systems. In *Proceedings of the 2015 USENIX Conference on Offensive Technologies* (Washington, D.C., 2015), WOOT'15.

[16] LADY, K. Sixty Percent of Enterprise Android Phones Affected by Critical QSEE Vulnerability. `https://duo.com/blog/sixty-percent-of-enterprise-android-phones-affected-by-critical-qsee-vulnerability`, 2016.

[17] MICRON TECHNOLOGIES. eMMC Security Features, 2016.

[18] MING, J., WU, D., XIAO, G., WANG, J., AND LIU, P. Taintpipe: Pipelined symbolic taint analysis. In *Proceedings of the 2015 USENIX Conference on Security Symposium* (Washington, DC, USA, 2015), SEC'15.

[19] MITRE. LK bootloader security vulnerability, CVE-2014-9798. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9798`.

[20] MUTTI, S., FRATANTONIO, Y., BIANCHI, A., INVERNIZZI, L., CORBETTA, J., KIRAT, D., KRUEGEL, C., AND VIGNA, G. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 2015 Annual Computer Security Applications Conference* (New York, NY, USA, 2015), ACSAC 2015.

[21] OUTLER, A. Have you paid your linux kernel source license fee? `https://www.xda-developers.com/have-you-paid-your-linux-kernel-source-license-fee/`, March 2013.

[22] QUALCOMM. (L)ittle (K)ernel based Android bootloader. `https://www.codeaurora.org/blogs/little-kernel-based-android-bootloader`.

[23] RAMOS, D. A., AND ENGLER, D. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 2015 USENIX Conference on Security Symposium* (Washington, DC, USA, 2015), SEC'15.

[24] RAWAT, S., MOUNIER, L., AND POTET, M.-L. Static taint-analysis on binary executables, 2011.

[25] ROL. ring of lightning. `https://rol.im/securegoldenkeyboot/`, 2016.

[26] RUTKOWSKA, J. Intel x86 considered harmful, 2015.

[27] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium* (San Diego, CA, USA, 2015), NDSS 2015.

[28] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy* (San Jose, CA, USA, 2016), SP '16.

[29] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium* (San Diego, CA, USA, 2016), NDSS 2016.

[30] VAAS, LISA. Smartphone anti-theft kill switch law goes into effect in California, 2015.

[31] WANG, S., WANG, P., AND WU, D. Reassembleable disassembling. In *Proceedings of the 2015 USENIX Conference on Security Symposium* (Washington, DC, USA, 2015), SEC'15.

[32] WANG, T., WEI, T., LIN, Z., AND ZOU, W. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the 2009 Network and Distributed System Security Symposium* (San Diego, CA, USA, 2009), NDSS 2009.

[33] WANG, X., JHI, Y.-C., ZHU, S., AND LIU, P. Still: Exploit code detection via static taint and initialization analyses. In *Proceedings of the 2008 Annual Computer Security Applications Conference* (Anaheim, CA, USA, 2008), ACSAC '08.

[34] WOJTCZUK, R., AND KALLENBERG, C. Attacking UEFI boot script. In *31st Chaos Communication Congress (31C3)* (2014).

[35] ZADDACH, J., BRUNO, L., FRANCILLON, A., AND BALZAROTTI, D. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the 2014 Network and Distributed System Security Symposium* (San Diego, CA, USA, 2014), NDSS 2014.

[36] ZALEWSKI, M. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, 2007.