SANDIA REPORT
SAND2022-13571
Printed October 2022

Sandia National Laboratories

Emulating the Android Boot Process

Alex R Bertels, Robert E Bell, Brandon K Eames

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico
87185 and Livermore,
California 94550

ABSTRACT

Critical vulnerabilities continue to be discovered in the boot process of Android smartphones used around the world. The entire device's security is compromised if boot security is compromised, so any weakness presents undue risk to users. Vulnerabilities persist, in part, because independent security analysts lack access and appropriate tools. In response to this gap, we implemented a procedure for emulating the early phase of the Android boot process.

This work demonstrated feasibility and utility of emulation in this space. By using HALucinator, we derived execution context and data flow, as well as incorporated peripheral hardware behavior. While smartphones with shared processors have substantial code overlap regardless of vendor, generational changes can have a significant impact. By applying our approach to older and modern devices, we learned interesting characteristics about the system.

Such capabilities introduce new levels of introspection and operation understanding not previously available to mobile researchers.

ACKNOWLEDGEMENTS

The following individuals provided invaluable contributions to this effort:

- During the project's early stage, Logan Carpenter was critical to establishing the emulation environment specific to the target devices. This included working on framework support for the device instruction set, collecting memory layout information, implementing initial execution point handlers, and interfacing with a development board.
- Samuel Short was an important team member during his internship in which he assisted with the snapshot capability, progressed the operating system bootloader emulation, and investigated the transition to the trusted execution environment.
- The foundational framework, HALucinator, required some minimal modifications to support our use case. Abraham Clements assisted in supporting this project's target firmware by exposing some internal framework features such as allowing specific image loading and passing parameters to emulated hardware component drivers.
- Christopher Valicka served as the original manager over the project. His guidance was invaluable for developing the early strategies and determining directions in unexpected obstacles.

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

CONTENTS

LIST OF FIGURES

This page left blank

ACRONYMS AND TERMS

Acronym/Term	Definition
AOSP	Android Open Source Project
CVE	Common Vulnerabilities and Exposures
DTS	Device Tree Source
ELF	Executable and Linkable Format
GUID	Globally Unique IDentifier
HAL	Hardware Abstraction Layer
OS	Operating System
PBL	Primary Boot Loader
QEMU	Quick EMUlator
REE	Rich Execution Environment
SBL	Secondary Boot Loader
SMC	Secure Monitor Call
TEE	Trusted Execution Environment
UART	Universal Asynchronous Receiver-Transmitter
UEFI	Unified Extensible Firmware Interface
UFS	Universal Flash Storage
XBL	eXtensible Boot Loader

This page left blank

1. INTRODUCTION

Consider CVE-2021-35134, a vulnerability located in the boot stage of the most modern of Qualcomm processors [1]. A seemingly insignificant, but also incorrect, verification step posed a critical risk to devices trusted everyday by users around the world. Rather than through the common practice of publicly promoted bug bounty programs, this vulnerability was discovered by an internal review. Other recent boot time vulnerabilities associated with just two vendors, Qualcomm and Samsung, include: CVE-2020-11284, CVE-2020-11127, CVE-2021-39647, CVE-2021-25481, CVE-2020-12746, CVE-2020-10850 [2].

Bug detection amongst Android Operating System (OS) based smartphone processors relies heavily on internal exploration of chipset vendors without the security validation of independent security researchers, but broader external review opportunities increase the rate of vulnerability discovery and reduce exposure time of production security flaws. Unfortunately, external analysts face challenges with performing this validation due to lack of access and lack of appropriate tools. Android's sizeable market has a diverse set of vendors, models, and chipsets, each with unique variations on the boot process; such diversity of firmware elements compounds the difficulty of crafting portable, broadly applicable, cross-platform tools that support security analyses of boot time software.

In response to this technological gap, we implemented a procedure for emulating parts of the Android boot process for two devices. Our procedure offers a crucial first step toward understanding this fundamental aspect of Android device security. If boot security is compromised, then the entire device's security is compromised, so the boot process is the root of all trust.

1.1. Background

From the perspective of a smartphone end-user, the boot stage may seem like a moderately short inconvenience. However, the comprised phases are quite complex – a rigorous secure boot process is difficult and involves many phases. Intricacy is also demanded because the chip vendors do not own the full system ecosystem. Peripheral components integrated into chips are provided by third parties, and their requisite device driver software must be incorporated and provisioned during the boot process – while avoiding boot process compromise. Furthermore, there is a balance between chipset vendors, smartphone manufacturers, and operating system developers; their interactions increase the complexity.

Understanding the internal system state assumptions before each boot phase executes is essential for emulation. These assumptions include the state of the data or code in memory that has been previously initialized, the permissions assigned to address ranges in memory, the physical hardware components available for use by the firmware, the verification steps required before transitioning to the next stage, and the capabilities of the execution environment. In absence of documentation, collecting this information is not trivial.

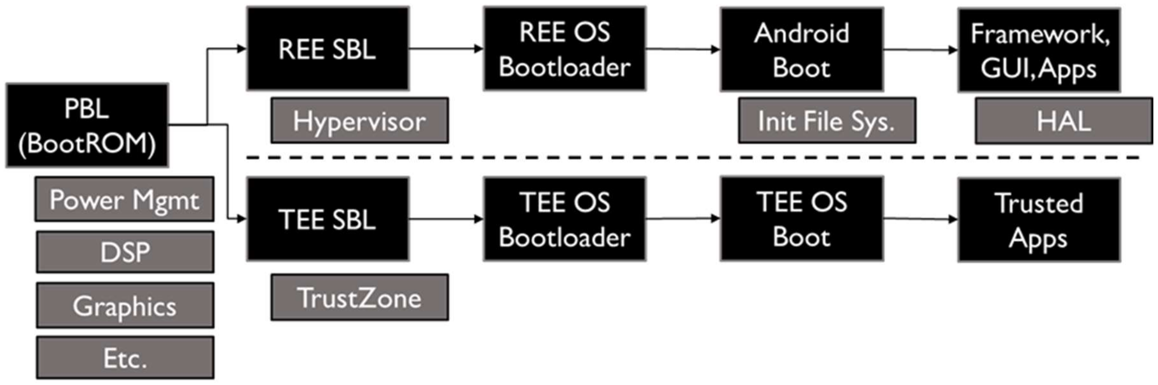

Figure 1. The figure illustrates the high-level stages of the boot sequence for Android devices. Starting with the primary bootloader (PBL), each step must initialize and verify the next step from the secondary bootloaders (SBLs) up through the applications for both the Rich Execution Environment (REE) and the Trusted Execution Environment (TEE).

While the boot sequence of each device varies with chipset and phone vendor implementations – as well as generational changes – there are several similar components across devices. An approximate outline of a general Android boot sequence is illustrated in Figure 1. The foundation of the boot process is provided by the chipset, or the processor [3] [4] [5]. The processor establishes the environment data and interfaces with internal peripheral hardware components. The first phase is the primary bootloader (PBL), which is fetched from the BootROM. The PBL is a relatively small, unmodifiable piece of code responsible for initiating the boot of other components and starting the chain-of-trust in the sequence. As components are booted, the initiating component must verify that the new component passes some security check (e.g., digital signature, hash comparison). The PBL initializes hardware responsible for power management, digital signal processing (DSP), and graphics processing, and then the PBL verifies and launches the secondary bootloaders (SBLs). The boot process also splits between the normal world and the secure world. The secure world contains the ARM TrustZone Trusted Execution Environment (TEE). The TEE operates in parallel with the Rich Execution Environment (REE), but it is isolated by the underlying hardware. On the REE side, the SBL establishes the normal world components, such as a hypervisor, before moving on to the operating system (OS) loader. Several vendors follow similar specifications for loading the OS kernel and peripheral images (e.g., Bluetooth, WLAN, cellular modem). The file system is initialized, and, subsequently, the user interface and applications are launched. Each smartphone vendor makes unique modifications to the Android OS and supplies required libraries and applications.

1.2. Related Work

Multiple disjoint efforts in emulating aspects of Android smartphone boot processes exist.

In several blog posts, researchers demonstrated their capabilities. Roee Hay and Noam Hadad detailed the steps to extract and analyze the PBL for several older Qualcomm chipsets [6]. Frederic Basse described how to emulate the PBL of a Samsung Exynos 4210 process extracted from a Samsung Galaxy S2 [7]. Alexander Tarasikov emulated most of the TEE bootloader and TEE OS for a Samsung Galaxy S10 [8]. Additionally, Aris Thallas emulated the hypervisor for a Samsung Galaxy S8+ [9]. The demand for such efforts for iOS is evident in the fact that an iPhone emulator was implemented by Corellium [10]. While Corellium has not publicly released their iOS emulator, an attempt at replicating a small portion of that work was documented by Zhuowei Zhang [11].

Academic sources also provide some insight and tools. For example, BootStomp is a static analysis and symbolic execution tool for searching the early boot process for potential vulnerabilities [12].

Another group of researchers from Samsung and the University of California, Berkeley, investigated the ability to emulate the TrustZone OSs of multiple vendors in a project called PARTEMU [13]. Given the complexity of executing the SBL prior to transferring control to the TrustZone components, the PARTEMU team opted to skip that phase and address the missing data elements as needed. This leaves a critical link in the boot chain of trust without a means for effective security assessment.

Even with all these emulation efforts, the current best approach for independent researchers to assess the security of the SBL is to employ static analysis techniques. By importing the firmware into a common disassembler or decompiler, an analyst can see the functionality built into the SBL. However, debug strings are limited, and context and data flow are difficult to track outside of actual execution. Being able to step through code and monitor state information offers an insight unavailable through static methods, especially given the size and complexity of the firmware. The SBL is dependent on external configuration and peripheral hardware components as well as other executable binaries that must first be extracted or loaded from storage. Dynamic analysis through emulation will by no means replace static analysis, but it presents an opportunity to complement other approaches.

2. INITIAL PROOF-OF-CONCEPT

2.1. Device Selection

The aim of this project was to provide a capability to evaluate the security properties of boot processes. Before we can produce a portable emulation-based capability, we must prove the idea through proof-of-concept implementations. Diversity of the Android ecosystem presents numerous devices to select from. There are several smartphone and chipset vendors, along with generational changes of hardware, firmware, and Android versions. We needed to choose a single device to emulate initially. To reduce development time, initial constraints were that the associated firmware execute with an architecture that has documented support by an emulation framework and that the device represent vendors with significant market coverage. While QEMU has implemented features for ARMv8.1+ instruction sets, QEMU specifically states that available Cortex-A architectures include 53, 57, and 72; our selection was limited to devices containing a processor with those specifications. Qualcomm Snapdragon processors appear in various models of Android-based smartphones manufactured by Samsung, Huawei, Xiaomi, Vivo, OPPO, Google, Motorola, and many others. To ensure a generalizable process, we wanted a device operating on a Qualcomm chipset. Additionally, the device needed to be demonstrative of modern devices by including characteristics such as UEFI and UFS. These constraints drove our choice of the first device we emulated.

2.2. Resource Acquisition

Often smartphone firmware is packaged up for the purposes of providing offline updates to devices. These packages for various vendors can be found through unofficial and official hosts. As an example, Google hosts factory firmware packages for Nexus and Pixel devices [14]. We used similarly packaged firmware for this emulation.

As for useful supplementary materials, Google releases kernel source [15] that includes information about peripheral device drivers and device tree layouts for different models, and Samsung Open-Source services release code pertaining to the Samsung Galaxy models that employ Qualcomm Snapdragon processors [16].

We additionally turned to public forums for information on this phone and firmware.

2.3. Environment Setup

Our next phase was to begin configuring the emulation environment for the needed memory layout. In the interest of quick adaptation to the target, Sandia's HALucinator [17] – a firmware rehosting framework – was adopted. In collaboration with HALucinator developers, we added support for 64-bit ARM instructions and exposed internal features necessary for our use case. HALucinator wraps avatar2 and, consequently, the QEMU engine, but adds features for easy configuration, debugging, and execution modification. While not always necessary for dynamic analysis of systems, debugging allows access to information beyond run logs. Users can step through the software and pause to examine memory when needed. Also, users can alter the current state of memory, potentially affecting the path taken through the firmware. With this, someone could understand what the software did and also what it might do. When trying to make sense of the Android boot sequence, being able to trigger different modes and to engage security mechanisms is important.

Remaining hardware-independent and only basing emulation on update firmware presents some limitations, forcing us to focus on modifiable firmware and data. The PBL is not included in firmware update packages because it is implemented as immutable instructions in the BootROM of the processor. This means that the emulation would start with the SBL or (in Qualcomm terms) the eXtensible Boot Loader (XBL).

Before execution can begin, firmware associated memory regions must be properly initialized. As with many firmware image files at this early phase, the XBL is stored as an Executable and Linkable Format (ELF) file. A readily available tool (i.e., readelf), was used to determine the permissions of the required XBL memory regions. As the team progressed through the emulation, we were able to determine and configure HALucinator with the necessary memory regions and permissions (i.e., read, write, execute). Embedded within the XBL image is a character string that is passed to a later boot stage called UEFI. This string is useful in expanding one's knowledge of the device data layout. The XBL will parse PBL-provided data and confirm configuration of hardware peripherals, each of which have their own dedicated address ranges in memory. By searching open-source files, we identified Device Tree Source (DTS) files that detailed the respective locations of hardware component interfaces.

2.4. Emulation

For our first proof of concept device emulation, we spent much of the time trying to understand the operations of the phone XBL. This required both static reverse engineering and dumping state information from the emulation. Our process primarily iterated through the following steps:

1. Run the emulation until a problem occurs (e.g., crash, error, infinite loop, etc.)
2. Locate the appropriate XBL execution points in the static binary analysis tool
3. Add HALucinator handlers to break execution at points-of-interest and dump the state
4. Continue steps 2 and 3 until the problem is identified
5. Use handlers to modify state information to meet expectations
6. Record properties of the functionality for future reference and understanding
7. Repeat steps 1-6

Often when the execution gets caught in an infinite loop, there is no immediate indication of this. The instruction trace log can quickly become quite large, so we spun off a process to monitor the log to detect loops and automatically shut down the emulation. With lengthy log files, following the instructions back to the source of an error can take time, so we also wrote scripts to read instructions from the trace log and identify scope changes, reducing the time to find approximate error locations. To improve emulation runtime, we increased memory write sizes and employed QEMU techniques to store and load state snapshots.

The XBL starts by initializing its own read-write data region with configuration information provided by the PBL. The XBL is aware of the memory layout of the system, but it needs to be told about environment details such as the storage (e.g., UFS, eMMC), whether secure boot is possible, and whether current execution is on a development board. In our case, we had to locate the storage field and indicate that UFS storage is available. Also, during initialization the XBL will check the state of various peripheral hardware components. Each component has a series of registers that the XBL would either write to or read from. When a read register value did not meet required values, the emulation execution would either enter an error state or get caught in an infinite loop waiting for the external hardware to change a status register. To force progress, we would simply add a

HALucinator handler to the instruction address checking the hardware device and alter the incorrect value.

Once the XBL is sufficiently initialized, the next image files needed to be loaded from storage. Each image has a data structure describing how this image should be loaded. The image has a reference to the respective Globally Unique Identifier (GUID). These GUIDs are used to request the image from the connected storage device. The image data structure also has references to a list of functions that will be executed before the image is loaded. In some cases, this list of functions will explicitly load the image or other support images from storage. This is because the default process is to read an image from storage and load the image as an ELF file. However, not all images fit the standard, and some need a different loading process. (The image data structure also has a list of functions to be executed after the image is loaded.) The image data structure contains status fields and a reference to the image description character string. As each section of an ELF image is loaded, the address range is compared against a growing list of other ranges to ensure no other regions are being clobbered by the new section. This check occasionally failed and had to be forced.

After an image is loaded from storage in this phone, the XBL will make a couple of secure monitor function calls (SMCs) back to the PBL, including authenticating the image and performing a hash comparison. Without the PBL and the relevant authentication data, we wrote an interrupt handler to catch these SMCs and respond as if the checks had passed. We did have to skip the execution of some images. For instance, power management was skipped because no power source exists in the emulator. Also, code related to serial devices was skipped. Instead, all outgoing log information was collected by a HALucinator handler attached to the logging function start address and was printed to terminal console. After all the images were loaded, the XBL would decompress the REE OS bootloader – in this case the Unified Extensible Firmware Interface (UEFI) image – from its own memory regions.

We were able to execute much of the XBL code from our first device, with some exceptions. Stages that expected feedback from external sources (e.g., clock, DDR) had to be specially emulated or skipped. The largest contributor to the necessary modifications was the communication with the UFS storage. Using HALucinator handlers, we initially produced a work-around for loading executable image files from the host filesystem. The largest downside is that this solution would be unusable at later stages such as in UEFI or in initializing the Android filesystem. These phases normally interact with storage devices at established memory regions, but the handlers are tied to XBL-specific instruction addresses. Relying on specific address-based handlers would also be a recurring problem in supporting additional devices.

Later, by studying Qualcomm's interface and reviewing the UFS specification documents [18], we implemented a proof-of-concept UFS driver for QEMU.[1] This allows us to load the executable images into virtual drives and place the interface at the appropriate address on the memory map so that the XBL can read data from the virtual drives. Ideally, this interface will be utilized by later stages of the boot process and the driver could be ported with minimal modification to other devices.

We also implemented a Universal Asynchronous Receiver-Transmitter (UART) QEMU driver. Rather than locating the logging functions and implementing a HALucinator handler at that instruction address to read and print log information, we could simply attach the driver to the UART interface and store the output to a file.

[1] QEMU does not currently supply a peripheral driver for UFS storage.

When the XBL came to its end, control would be handed over to the next image. For some reason, the data structure informing execution of which address to execute next was misconfigured. This configuration data had to be forced by using a HALucinator handler to assign that next instruction address.

Rather than transitioning to the TrustZone image or the hypervisor, we opted to transition to the UEFI image so that we could better understand how completely the XBL set up the environment required by the dependent boot phases. By emulating some of the UEFI phase, we learned that our initial XBL emulation attempt misconfigured the system and required changes. Some were simply altering a few HALucinator handler forced values. By working through the UEFI Driver eXecution Environments (DXEs), we came to realize that logging and peripheral hardware components that we had implemented handlers for would need to be reimplemented with the new functions in the later phase. With lessons learned and a much greater understanding of the boot process, we knew that we needed to back track to correct shortcomings in our approach before moving forward into new boot phases.

3. DEVICE-AGNOSTIC APPROACH

This was a pivotal moment in the project to decide which direction to move. With limited time, only one route could be taken. We could try to acquire a PBL for the first device, start the first device or a related second device from scratch to correct our approach, or select a more modern Android smartphone to bring our emulation closer to future needs. Ultimately, choosing to confirm emulation for newer devices seemed to provide the most long-term benefits.

3.1. Device Selection

For our second device, we wanted to be able to reuse much of our prior work but also consider a widely used modern smartphone. We chose a phone using the Qualcomm Snapdragon 888 processor.

3.2. Resource Acquisition

Again, we used a public firmware update package and related open source files for this device.

During this second effort, we also used a physical version of the phone for validation. The firmware version was newer on the physical device but was similar enough for our purposes.

3.3. Environment Setup

In a similar fashion to establishing a base line memory layout for our first device, we used a combination of the following to start the memory layout for our boot process emulator for our second device:

- A UEFI platform configuration file that can be found included in an XBL image
- A "readelf" section analysis of each image file in the firmware bootloader directory
- A few known peripheral hardware components found in the DTS files

Before we started the emulation, we were prepared to establish the known QEMU peripheral hardware driver interfaces. Given that we had already implemented an older version of the UFS driver, that was the first to be added. We needed to use the relevant DTS file to determine the address ranges for the interfaces. Figure 2 illustrates a Samsung Open Source example DTS file – listing registers of UFS starting at address 0x01d84000 with all other registers following. This example UFS driver is compatible with "qcom,ufshc". To better understand how a phone would communicate through this interface, we can visit the compatible source code in the "Kernel/drivers/scsi/ufs/" directory. Similarly, the "ufsphy_mem" source code can be found at "Kernel/drivers/phy/qualcomm/". The register offsets for that phone are also given throughout the various header files in those directories.

In this case, the team's prior experience was limited. Rather than constructing the QEMU driver from a blank file, we decided to fork and modify another Small Computer System Interface (SCSI)-related QEMU driver and to rely on UFS specifications. The extent that the Android kernel interacts

```
        ufsphy_mem: ufsphy_mem@1d87000 {
                reg = <0x1d87000 0xe10>;
                reg-names = "phy_mem";
…
        };

        ufshc_mem: ufshc@1d84000 {
                compatible = "qcom,ufshc";
                reg = <0x1d84000 0x3000>,
                      <0x1d88000 0x8000>,
                      <0x1d90000 0x9000>;
                reg-names = "ufs_mem", "ufs_ice", "ufs_ice_hwkm";
…
        };
```

Figure 2. UFS hardware interfaces from Kernel/arch/arm64/boot/dts/vendor/qcom/lahaina.dtsi included in the Samsung Open Source package

with storage does not necessarily match that of the XBL image, so we only implemented the registers and minimal functionality required by the XBL.

With the driver in place, we needed disks containing the firmware images for XBL to load. Using the qemu-img tool, we were able to create multiple disks. But, to understand the partition layouts on those disks, we needed to look at a physical smartphone. The file "/proc/partitions" shows the partitions sizes for sda, sdb, sdc, sdd, and sde. To get the names for each partition, we pulled those by matching the symbolic links in "/dev/block/by-name/". Next, we had to find the GUIDs for the partitions used by XBL. There are some common GUIDs used by Android, such as that of the CDT partition: A19F205F-CCD8-4B6D-8F1E-2D9BC24CFFB1. XBL stores a list of these internally in a mixed-endian format. Which for CDT would appear as: "`5f 20 9f a1 d8 cc 6d 4b 8f 1e 2d 9b c2 4c ff b1`". One can discover the less common GUIDs by using a static analysis tool and searching for the CDT GUID value in memory and noting the surrounding byte values. Each GUID is referenced by a data structure that also contains a reference to an image descriptor string. Once the GUIDs were matched to partitions, we used "gdisk" to update the disk alignment, modify partition names and GUIDS, and write the decompressed image files to the appropriate partitions.

We were able to add some additional QEMU device drivers to eliminate the need for some HALucinator handlers. Unfortunately, the UART device driver had changed between the first device and the second. To address this, we wrote three additional minimal UART drivers.

3.4. Emulation

We mostly replicated our previous approach of repeatedly finding errors and addressing them, but, this time, we tackled missing hardware by first trying to emulate the peripheral hardware component interactions by implementing compatible QEMU drivers. Only when the drivers were overly complicated would a HALucinator handler be employed to address the problem. Ideally, to prepare the emulation for later boot phases, most of the drivers mentioned by the DTS would need to be included.

Early in the emulation, we discovered that with the same configuration and firmware images, the execution trace and errors across team members were different. By upgrading HALucinator's

QEMU core from version 4 to 6, the problem no longer occurred. The source of the non-deterministic behavior remains unknown.

We were able to keep some of the features from the initial proof-of-concept emulator such as the SMC interrupt handler, loop detection, trace analysis, and snapshots. Unfortunately, there were some unexpected changes in moving to a newer firmware version. When working with the first device, there was no memory enforcement, so we could establish memory regions as needed. However, with the second device, establishing and accessing arbitrary memory regions was not possible. This became a problem given that, for us to create a region dedicated to the PBL configuration data without overwriting other data, we needed to be able to incorporate new memory regions. We were able to add a feature to edit the page tables and flush the Translation Lookaside Buffer to fix the problem.

After setting the location of our PBL configuration data, we had to set that this device was using "UFS" for storage. We also ran into the problem of an internally managed "whitelist" of allowed memory ranges. Each list was specific to a loaded image, but some were corrupted by logging data. Without knowing the cause of the corruption, we opted to add a HALucinator handler to ignore this check. Due to limited time and unnecessary functionality, we also skipped sleep features, power management, Universal Serial Bus (USB), and DDR training.

In most cases where the emulation environment was improperly configured, the firmware would enter an error state and dump some execution related information. After the error information was logged, the firmware would then proceed to try to enter an emergency download mode. In the interest of progressing the emulation, we did not investigate whether this mode had included firmware or if that was another missing software component.

Our emulated UART devices write out log information to a file rather than the same stream as either the HALucinator or QEMU output. Figure 3 illustrates the format of the early output for our second device. This logging data was validated against the log from a physical device.

```
Format: Log Type - Time(microsec) - Message - Optional Info
Log Type: B - Since Boot(Power On Reset),  D - Delta,  S - Statistic
...
S - Boot Interface: UFS
S - Secure Boot: On
...
S - Core 0 Frequency, 1459 MHz
S - PBL Patch Ver: 0
D -         0 - pbl_apps_init_timestamp
D -         0 - bootable_media_detect_timestamp
D -         0 - bl_elf_metadata_loading_timestamp
D -         0 - bl_hash_seg_auth_timestamp
D -         0 - bl_elf_loadable_segment_loading_timestamp
D -         0 - bl_elf_segs_hash_verify_timestamp
D -         0 - bl_sec_hash_seg_auth_timestamp
D -         0 - bl_sec_segs_hash_verify_timestamp
D -         0 - pbl_populate_shared_data_and_exit_timestamp
S -         0 - PBL, End
B -         0 - SBL1, Start
...
B -         0 - usb: usb2_rcal
B -         0 - usb: platform
B -         0 - usb: usb_shared_hs_phy_init: hs phy cfg size , 0xc
D -         0 - sbl1_hw_init
B -      4544 - UFS INQUIRY ID: QEMU    QEMU HARDDISK   2.5+
B -      4544 - UFS Boot LUN: 0
B -      6557 - UFS MD   : CE0321
B -      6557 - UFS size : 1GB
B -      6557 - UFS spec : 0310
D -      6557 - boot_media_init
D -         0 - smss_load_cancel
B -      6557 - SMSS -  Image Load, Start
D -         0 - SMSS -  Image Loaded, Delta - (0 Bytes)
D -         0 - Auth Metadata
D -         0 - sbl1_xblconfig_init
B -      6557 - XBL Config -  Image Load, Start
D -         0 - shrm_load_cancel
B -      6557 - SHRM -  Image Load, Start
D -         0 - Auth Metadata
D -         0 - Segments hash check
D -         0 - SHRM -  Image Loaded, Delta - (40096 Bytes)
…
```

Figure 3. XBL logging output collected from UART device.

4. DISCUSSION

Comparing the firmware structure of different processor vendors is important for determining the investment needed to support other devices. PC processors have made the shift to supporting the UEFI specifications, but smartphone processors lag. UEFI is a valuable first step toward removing the boot dependence on a particular OS. In this way, Qualcomm Snapdragon processors technically do not require Android OS as the next stage after the UEFI bootloader. Similarly, generations of MediaTek smartphone processors incorporate U-Boot, an OS bootloader that meets a subset of the UEFI specification. Samsung Exynos processors, however, combine many of the boot phases into a single image called S-BOOT. The SBL, OS bootloader, and Android kernel launch all occur within the span of S-BOOT's execution. This process largely falls in line with the source code provided by ARM Trusted Firmware-A [19]. Google's Tensor processor [20], which is based on Eyxnos processors, breaks up S-BOOT into the separate stages, but – like Exynos – relies on Little Kernel rather than UEFI.

While some vendors differ, if the DTS files of the image can be obtained and appropriate open-source files are available, then QEMU peripheral component drivers could be developed and added at the correct address ranges. We have considered some alternatives to implementing each specific component. One attempted approach was to use QEMU TCG plugins to try to determine when a loop has been encountered while waiting for a hardware status update – and then use the current trace information to modify register values to meet exit conditions. Unfortunately, TCG plugins cannot modify memory or register values without modifying QEMU's framework. Making such alterations comes with risk of damaging the integrity of the execution. Given this, we have also considered using TCG plugins to collect ongoing state information and, upon entering a loop, inform another process to snapshot and kill the current emulation, then provide information to generate a new device. This information could include addresses accessed and conditions checked around those values. Once a new driver is added, then the execution could pick-up from the saved snapshot or an earlier point in execution if necessary. We have also considered using HALucinator Peripheral objects, but that can be slow if the hardware is accessed frequently.

5. CONCLUSION

This work has demonstrated the feasibility and utility of emulation as a form of dynamically analyzing the early boot process of Android-based devices. By using features provided by HALucinator and QEMU, we were able to not only derive execution context from limited logging strings and data flow, but also incorporate behavior of selected peripheral hardware components such as storage. Both the acquired context and learned hardware behavior is invaluable in supporting the emulation and program understanding of additional devices. We should note that while there can be significant overlap of code from different smartphone vendors that employ the same processor, the generational changes with modern devices can have a significant impact in functionality and data structures. In some cases, certain loaded image types were removed, and new images were added. By applying our approach first on an older device and then on a modern device, we were able to use the tool to learn some interesting things about the system.

At boot time, image and data files are accessed by GUID. This type of access does not consider what the files are named, or even what drive they are stored on. If there is a GUID match on any mapped partition in the drive, that is the content that gets loaded. As a result, for us to match GUIDs with actual firmware image files, we had to either look for strings in the debug output or find a header file online which makes the mapping. If two files have the same GUID, the system does not crash, but merely loads the first file it comes to – this was verified with experimentation.

Though the ARM architecture in our first device allowed for memory protection, memory protection was not implemented at boot time. On the other hand, the second device enables protections very early in its boot process. More surprising is the fact that QEMU gdb debug honors the emulated phone's memory protections. In other words, if a piece of memory is marked inaccessible in the emulated phone's page tables, it cannot be accessed by our handlers. This implies that gdb walks the page tables to access memory, which makes sense as it needs to obtain the physical address to read. If this is the case, it is more surprising that we were able to access anywhere in memory on the first device. That would imply that either that device is operating with page tables turned off, or that the early boot of the device uses very rudimentary page tables with everything mapped in as a one-to-one mapping.

The most unexpected behavior we learned about Android boot is that the system management calls (SMCs) at boot time appear to be for integrity checking purposes only. The SMC environment is setup by the PBL, which we did not have access to. As an early implementation, we hooked all SMC calls and just returned 0 (the universal OK), thinking that for some of the calls actual work would need to be performed and we would have to reverse engineer those situations when we got to them. That never happened for either phone. In other words, returning 0 for all SMCs was sufficient emulation for either boot. This strongly points to the SMCs being for pass/fail checks only, and not for actual work or calculation. We were able to verify that integrity checking does not exist in the XBL itself (supporting that image authentication is in the SMCs). We modified a couple of the many ELF files that get loaded at boot, with no effect on the simulation. To prove the files were getting loaded, we then modified the files in a way that, if loaded, they would crash the system – and they did.

Project software is available in an internal git repository and relevant documentation is stored in a collaborative drive. The emulation platform will be used and supported for at least the next 2 years.

This research is far from complete. Other processor vendors must be investigated to ensure a truly device agnostic approach. Also, adopting an approach to automatically handle responses for unknown peripheral components would greatly speed up the process. Attempting this from static analysis prior to execution is made difficult by the loading and decompressing of other firmware images. However, we discussed possible ways to tackle this issue. In combination with prior work on emulating other boot phases, a more comprehensive picture can be painted and enable an unmatched capability in mobile security analysis.

REFERENCES

[1] Qualcomm Technologies Inc., "June 2022 Security Bulletin," 06 June 2022. [Online]. Available: https://docs.qualcomm.com/product/publicresources/securitybulletin/june-2022-bulletin.html.

[2] The MITRE Corporation, "CVE," [Online]. Available: https://www.cve.org/. [Accessed 8 August 2022].

[3] J. Levin, Android Internals: A Confectioner's Cookbook, vol. Volume I: The Power User's View, Cambridge, MA: Technologeeks.com, 2015.

[4] R. P. Nakamoto, "Secure Boot and Image Authentication," Qualcomm Technologies Inc., San Diego, 2016.

[5] J. Snyder, "Samsung Trusted Boot and TrustZone integrity management explained," Samsung, 4 September 2019. [Online]. Available: https://insights.samsung.com/2019/09/04/samsung-trusted-boot-and-trustzone-integrity-management-explained/.

[6] R. Hay and N. Hadad, "Exploiting Qualcomm EDL Programmers," Aleph Research, 22 January 2018. [Online]. Available: https://alephsecurity.com/2018/01/22/qualcomm-edl-1/.

[7] F. Basse, "Emulating Exynos 4210 BootROM in QEMU," 7 March 2018. [Online]. Available: https://fredericb.info/2018/03/emulating-exynos-4210-bootrom-in-qemu.html.

[8] A. Tarasikov, "Reverse-engineering Samsung Exynos 9820 bootloader and TZ," 29 May 2019. [Online]. Available: http://allsoftwaresucks.blogspot.com/2019/05/reverse-engineering-samsung-exynos-9820.html.

[9] A. Thallas, "Hypervisor Necromancy; Reanimating Kernel Protectors," Phrack, 14 February 2020. [Online]. Available: http://www.phrack.org/papers/emulating_hypervisors_samsung_rkp.html.

[10] T. Brewster, "This Super Stealth Startup Has Built An Apple Hacker's Paradise," Forbes, 15 February 2018. [Online]. Available: https://www.forbes.com/sites/thomasbrewster/2018/02/15/corellium-virtual-apple-iphones-for-hacking/#5711822c4a3b.

[11] Z. Zhang, "Almost booting an iOS kernel in QEMU," 15 July 2018. [Online]. Available: https://worthdoingbadly.com/xnuqemu/.

[12] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel and G. Vigna, "BootStomp: On the Security of Bootloaders in Mobile Devices," in *Proceedings of the 26th USENIX Security Symposium*, Vancouver, 2017.

[13] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen and M. Grace, "PartEmu: Enabling Dynamic Analysis of Real-World," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[14] Google, "Factory Images for Nexus and Pixel Devices," [Online]. Available: https://developers.google.com/android/images. [Accessed October 2020].

[15] Google, "Google Git," [Online]. Available: https://android.googlesource.com/kernel/common/. [Accessed 8 August 2022].

[16] Samsung, "Samsung Open Source," [Online]. Available: https://opensource.samsung.com/main. [Accessed 8 August 2022].

[17] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi and M. Payer, "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[18] JEDEC, "Universal Flash Storage (UFS) Version 3.1," JEDEC Solid State Technology Association, Arlington, 2020.

[19] ARM Limited and Contributors, "Firmware Design," [Online]. Available: https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/design/firmware-design.rst. [Accessed 8 August 2022].

[20] Google, "Binary Transparency for Pixel Factory Images," [Online]. Available: https://developers.google.com/android/binary_transparency/pixel. [Accessed 8 August 2022].

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	1911	sanddocs@sandia.gov

This page left blank

This page left blank