Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Emmutaler: Fuzzing the iOS Boot Loader

Bachelor Thesis

Author: Leonardo Galli

Tutor: Finn De Ridder

Supervisor: Prof. Dr. Kaveh Razavi

March 2021 to September 2021

Abstract

With the rise of smartphones and their importance in everyday life, they have also become an increasingly larger target of cyberattacks. iPhones in particular claim to be very secure and also privacy focused. This starts at the secure boot loader, which forms the beginning of a strong chain of trust. It stands to reason, that breaking this chain of trust would be of particular interest to adversaries.

In this thesis, we present emmutaler, a set of tools to enable fuzzing of the iPhone boot loader. After intensive static analysis of the boot loader, emmutaler allows us to create a normal Linux ELF containing it. This allows us to fuzz specific parts efficiently, using conventional Linux fuzzers. We present the design and implementation of emmutaler, as well as multiple fuzzing experiments ran using it, such as fuzzing the USB and firmware parsing stack. Furthermore, we introduce FETA, a novel, thread-safe allocator tailored towards fuzzing, that enables the discovery of heap vulnerabilities hidden by other allocators. In particular, we managed to find “checkm8” [3] - a previously known UAF vulnerability - only using FETA and hence showed that it is still present on A13 chips. Lastly, we discuss some innovative mitigations in use by iPhone boot loaders, that we found during the development of emmutaler.

Acknowledgements

I would like thank my Tutor, Finn de Ridder, for providing continued guidance, support and always accommodating to my weird timezones. Furthermore, I would like to thank my supervisor, Prof. Kaveh Razavi, for not only letting me write this thesis, but always showing interest and having great suggestions. I would also like to thank Patrick Jattke and the rest of the COMSEC group for letting me use their servers for experiments and putting up with me crashing them.

Special thanks go to my friends and roommates who are always there and supportive, even in uncertain times. In no particular order, I thank Lucas for proof reading every last email, proposal and this thesis, and supporting everything I do. I thank Matteo, for helping me rubber duck QEMU bugs and general kernel weirdness. I thank Luca, for letting me rant about anything and helping me with the initial proposal, as well as a bunch of smaller things. I thank Noah, for letting me bounce of ideas regarding heap and USB implementations and also providing his valuable input on a multitude of findings. I thank Fabian, for coming up with the name “emmutaler”, and ParmeSAN [30] for the inspiration.

Finally, I want to mention my CTF teams, “flagbot” and “organizers”, harbouring an incredible amount of talented people and making my dream of going to DEF CON CTF a reality. While I spent many weekends playing CTF, I met many interesting people, which would offer their occasional assistance. Of particular interest is Robin, who would always have a challenge ready or five and I could always ask about any weird cryptography questions I had.

Contents

Chapter 1

Introduction

With the ever growing cases of ransomware and other state and non-state sponsored cyberattacks, security is becoming increasingly important in every context. Furthermore, with the advent of smartphones and social media, an ever increasing part of our lives is not only spent but also kept in digital form. It stands to reason, that the security of smartphones is of highest importance, not only to people hiding from three-letter agencies, but also the wider public. As seen with the likes of "Superfish" [27], all it takes is a careless vendor and suddenly even so called "script kiddies" can access your personal data. Of particular interest are iPhones, which are marketed as being designed with a focus on privacy and security. Or in Apple's own words, "Apple continues to push the boundaries of what's possible in security and privacy." [13].

At the core of every modern smartphone is its boot loader. Once compromised, any security guarantees are usually lost and an attacker can attain complete control of the device. As such, the iPhone boot loaders seems like a prime target to pick apart and search for vulnerabilities, especially since these have been found here in the past [3, 14]. With the recent resurgence of interest in fuzzing and applying it to more and more fields, it is natural to want to apply fuzzing to the iPhone boot loader as well.

In this paper, we hence present the design and implementation of "emmutaler"[1], a set of tools, patches and much more to enable fuzzing of the iPhone boot loader, using a state of the art coverage-guided fuzzer. We show that using emmutaler, we can fuzz certain parts very efficiently and achieve great coverage in the targeted areas of the iPhone boot loader and even find previously discovered vulnerabilities quickly. Finally, we present interesting discoveries about the security architecture of the boot loader, made during the development of emmutaler.

1.1 Motivation

As described above, the recent surge of interest in smartphone security gives a great reason to pick apart any software found in iPhones. To our knowledge, this thesis is the first to publicly apply state of the art fuzzing to the iPhone boot loader. There are two main resons why the boot loader is a good target for fuzzing. Firstly, there is a lot of code related to parsing - which tends to be especially prone to bugs - of various complicated formats, such as USB messages or ASN.1. Secondly, from the point of an adversary, the boot loader forms the root of a strong chain of trust. Compromising it can prove to be very valuable.

[1] A combination of "emulation" and "Emmentaler", a famous Swiss cheese.

1.2 Challenges

When trying to fuzz the boot loader, we face three major challenges. The boot loader is a statically linked binary blob without any symbol or section information. This stands in stark contrast to normal fuzzing operations, where one has access to source code. Furthermore, it is designed specifically to run on Apple processors, so it often relies on implementation defined behaviour. Since the ARM standard is extensible, it even has custom Apple specific registers and instructions. Finally, it expects to run on bare metal, including full access to RAM and multiple devices. Hence, we cannot use conventional fuzzing tools, as those are made with user space binaries in mind.

1.3 Solution

Our solution - emmutaler - is a set of tools that help in creating a normal Linux ELF binary that can call certain parts of the boot loader. One important tool, is a binary patcher allowing us to handle any Apple specifics or missing devices. Additionally, thanks to a significant reverse engineering effort, most symbols can be recovered and many parts can be used by the Linux ELF. All in all, this allows us to use state of the art fuzzers without any modifications on the fuzzers themselves.

1.4 Overview

Chapter 2 gives the background information needed to understand the next chapters. Before Chapter 4 presents the design of emmutaler, Chapter 3 introduces the assumed threat model. Next, Chapter 5 describes the implementation of emmutaler in detail, along with the specifics of our fuzzing setup. Chapter 6 presents the fuzzing results obtained with emmutaler and Chapter 7 discusses shortcomings of our solution as well as potential future research. Finally, Chapter 8 puts this thesis into context of related works.

Chapter 2

Background

Almost all devices from Apple share the same common boot loader and even most of their OS[1] [16]. Their latest ARM Macs even have the same boot loader as the latest iPhones [18]. Therefore, anything discussed henceforth about iPhones / iOS should apply to most of their devices, with some differences.

For a better understanding, we provide some background on Apple specific terminology, the complete iOS boot sequence, the boot image format used by it and finally a recently found boot loader vulnerability. Additionally, there will be a brief overview of the most important aspects of fuzzing used in this thesis.

2.1 iOS Boot Sequence

The boot sequence on Apple devices consists of not one single stage boot loader, but one comprised of multiple stages. Although the different stages have different names and purposes, they stem from the same code base. Figure 2.1 shows how the boot sequence works, starting at device power on and including events such as recovery or upgrading. Together the different stages form a secure boot chain. Every stage loads, then verifies and finally executes - if verification passes - the next one. A more detailed description of the verification follows in Section 2.3.

The purposes of the different stages and related processes are as follows:

- **SecureROM** is burnt into the device read only memory (ROM)[2] and hence cannot be updated or changed. It is the initial boot loader and has as little functionality needed as possible. To establish the secure boot chain, it contains the hardcoded Apple root certificate.
- **Device Firmware Upgrade (DFU)** is a part of SecureROM. It is a USB protocol to load iBSS in case of failure - for example an invalid iBoot image - or on purpose by holding specific buttons[3].
- **iBoot** is the main part of the boot loader. It contains the most functionality, including loading of the kernelcache, starting the recovery or update process, and even diagnostics.
- **iBoot Single Stage (iBSS)** is a stripped down version of iBEC and responsible for loading it.

[1] Even though their marketing department might disagree.

[2] Where the name comes from.

[3] For example, on an iPhone X you have to hold both the power and volume down buttons for 5 seconds, then release the power button and keep holding volume down for another 5 seconds.

- **iBoot Epoch Change (iBEC)** is responsible for carrying out the upgrade and recovery processes.
- **Kernelcache** is the Darwin kernel and all loaded kernel extensions merged into a single file. Once control is transferred to the kernelcache, the OS finally boots, just like on any other device.

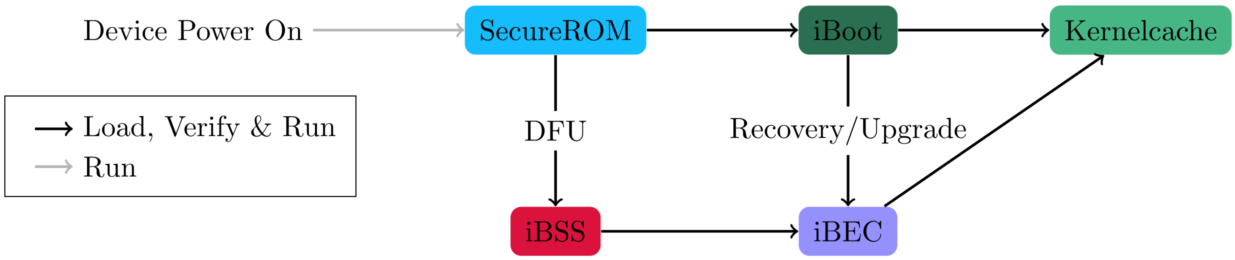


Figure 2.1: The boot sequence of an iPhone, adapted from [16].

2.2 Apple Specific Terminology

Throughout this thesis, we will often use Apple specific terminology and abbreviations. The most important ones are collected in this section.

The Application Processor (AP) refers to the main processor in a device, where for example the actual applications are executed. In addition, most devices have a so called Secure Enclave Processor (SEP), a co-processor, responsible for encrypting user data amongst other security relevant tasks. Since SecureROM is read only, it is burnt into the AP during manufacturing. Therefore, the corresponding AP marketing name will often replace the lengthy SecureROM version information. See Table 2.2 for the correspondence between AP names, iPhones[4] and SecureROM versions.

iPhone	AP Marketing / Internal Name	SecureROM Version[5]	ISA
iPhone 7	A10 / T8010	`iBoot-2696.0.0.1.33`	ARMv8.1-A
iPhone X	A11 / T8015	`iBoot-3332.0.0.1.23`	ARMv8.2-A
iPhone XS	A12 / T8020	`iBoot-3865.0.0.4.7`	ARMv8.3-A
iPhone 11	A13 / T8030	`iBoot-4479.0.0.100.4`	ARMv8.4-A
iPhone 12	A14 / T8101	`iBoot-5281.0.0.100.22`	ARMv8.5-A

Table 2.2: AP marketing names, their corresponding iPhone and latest SecureROM version of each AP we have.

Other commonly used terms in relation to APs are the so called "fuses". Like any modern chip, Apple's processors have a multitude of these, used to store configuration parameters. They are memory mapped to specific locations and accessed by the AP through these locations. Some important configuration stored in fuses includes *security mode* and *production status*. The security

[4] Of course SecureROM and APs are also present in other devices, but this thesis focuses mainly on iPhones.

[5] Even though the version string contains iBoot, this is the actual version string found inside the SecureROM image.

mode of a device indicates whether it can load only code signed by Apple or any code. Insecure devices are almost exclusively used by Apple employees[6]. Any device sold in stores will have its production status set to true and is secure.

2.3 IMG4 Format

Like any application binary, every single one of the previously mentioned stages consists of both code and data. Except for the kernelcache, code and data of a stage are found together in a single binary blob, called an image. When loaded, the execution begins at the start of the binary and the binary itself then relocates the data part to another part of memory. Contrary to many modern operating systems, there is no complicated metadata, such as relocations, inside the binary. In contrast, the kernelcache is a normal Mach-O file featuring such metadata.

Except for SecureROM however, the image of a stage is not stored by itself, but rather contained in a so called IMG4 file. This file contains any additional metadata, needed for the verification process of a stage. The IMG4 format is not only used for the boot loader binaries, but also auxiliary files, such as a low battery image. An IMG4 file is just an ASN.1 value[7] encoded as DER and there is a detailed description[8] of it in Appendix B.2.

In short, the format consists of four important ASN.1 values:

- **IMG4** is the main value and wraps the other parts, unless they are in separate files.
- **IMG4 Payload (IM4P)** wraps the actual payload - for example the low battery image or iBoot binary - and contains related information. This might include, information about the compression or encryption parameters[9].
- **IMG4 Manifest (IM4M)** contains all the information necessary to verify the validity of the payload.
- **IMG4 Restore (IM4R)** contains additional information necessary to verify a restore process. It is only needed when performing a restore and hence is optional[10].

Usually, IM4P, IM4M and IM4R are shipped separately to the end user, where Apple's software update tools combine them into an IMG4 file and flash them to the device. The IM4M further contains important ASN.1 values:

- **Manifest Body (MANB)** which contains the **Manifest Properties (MANP)** and the digest of one or more IM4P items[11].
- A list of X.509 certificates, forming a certificate chain. In particular, the root of the chain is burnt into the boot loader and not included.
- The RSA signature of MANB, signed by the public key of the leaf certificate of the certificate chain.

[6] "Lost" insecure devices can fetch quite the price on online platforms.

[7] The exact details of what an ASN.1 value is, is described right afterwards.

[8] We figured out the specification by reversing SecureROM, hence why some fields might be unknown or missing.

[9] iBoot images are usually encrypted, but later stages - such as the kernelcache - are not anymore [6].

[10] IM4R is not used in this thesis and hence not detailed further.

[11] Although a single IMG4 file can contain only one IM4P, the manifest might be separate from the payload and valid for multiple of them.

The manifest properties specify properties to restrict the usage of an image. In addition to a valid signature, the environment must also match all the provided properties for the image to be considered valid. The environment is a collection of properties, most of them retrieved from fuses in the device directly, but some might be overridden by software. The most important properties are:

- **Exclusive Chip Identification (ECID)** is a unique ID for ever SoC. Hence, IMG4 files are usually personalized for a single unique device.
- **Board (BORD) and Chip ID (CHIP)** are nice numbers identifying the board and chip of the device. For example, an A13 chip will have a CHIP value of `0x8030`.
- **Production Status and Security Mode** are defined as previously explained.

Verification of an IMG4 file hence proceeds as follows:

1. Verify the X.509 certificate chain found in IM4M.
2. Verify the RSA signature found in IM4M with the digest of MANB, using the leaf certificate of the certificate chain.
3. Verify the properties from MANP match the equivalent properties in the environment.
4. Verify the digest found in MANB matches the digest of IM4P.

2.3.1 ASN.1 and DER

Abstract Syntax Notation One (ASN.1) [26] is an interface description language used for describing abstract data types enabling encoding and decoding in a cross platform manner. It is most well known for its use in specifying the format of X.509 certificates. An example of an ASN.1 description can be seen in Code 2.1.

```
Keybag ::= SEQUENCE {
    type INTEGER,
    iv OCTET STRING,
    key OCTET STRING
}
```

Code 2.1: Example specification of the keybag structure present in the IM4P structure.

Apple uses the so called Distinguished Encoding Rules for serializing ASN.1. Distinguished Encoding Rules (DER) is a type-length-value (TLV) encoding, that unambiguously encodes an ASN.1 value, while minimizing the amount of bytes needed. For every ASN.1 value, DER first encodes the type, then the length and finally the data. To illustrate why it might be difficult for a fuzzer to mutate ASN.1, we give a brief description of the encoding process:

- The type - or rather its tag number - is encoded using a variable length encoding scheme. Any tag smaller or equal than 30 is encoded as a single byte. Tags larger than 30, use at least one extra byte, or as many as are necessary. Subsequent bytes have their highest bit set to 1, until the last byte, which has it set to 0.
- The length is also encoded using a variable length scheme, differing from the one used for the tag. If the length is smaller or equal than 127, it is encoded in a single byte. Otherwise the first byte specifies the number of following bytes used to encode the length.

- Finally, the actual contents are encoded in a number of different ways depending on the type of the value. For example, a sequence is just a sequence of the encoded bytes of its content values.

A commonly used ASN.1 type - especially in the context of X.509 certificates - is the so called Object Identifier (OID). According to the ITU, "an OID is an extensively used identification mechanism jointly developed by ITU-T and ISO/IEC for naming any type of object, concept or 'thing' with a globally unambiguous name which requires a persistent name (long life-time)" [20]. All OIDs form a hierarchy together, the so called "OID tree". It is mostly used to uniquely name attributes of X.509 certificates. An example OID is 1.2.840.113549.1.1.12, identifying the use of SHA-384 with RSA encryption. It is consists of the following parents in the OID tree:

- 1.2.840.113549.1.1: PKCS-1
- 1.2.840.113549.1: PKCS
- 1.2.840.113549: RSADSI
- 1.2.840: USA
- 1.2: ISO member body
- 1: ISO assigned OIDs

2.4 checkm8

Contrary to what the name SecureROM implies, there have been publicly disclosed vulnerabilities in the past [14, 3]. "checkm8" is the most recent example and will be of particular interest in this thesis. At its core, "checkm8" exploits a use-after-free vulnerability, in the DFU protocol handling. A use-after-free (UAF) occurs, when a pointer to a buffer on the heap is used, after said pointer has been freed. This usually leads to heap corruption and - if exploited correctly - to arbitrary code execution.

Alongside the UAF vulnerability, "checkm8" first performs a complicated "heap feng shui", to be able to achieve code execution. "Heap feng shui" is the name of the process of carefully manipulating the heap, allowing for exploitation, for example achieving code execution or write what where [25]. With most heap implementations, this is necessary, for corrupting the heap in an exploitable manner. Heap feng shui, usually consists of allocating very specific sizes in a very specific order as well as then freeing specific chunks. "checkm8" achieves this by forcing SecureROM to send responses of specific sizes[12] and ensuring that some of them are never transmitted correctly[13].

The full exploit chain is confirmed to work on all processors from A5 to A11. However, the core vulnerability - the UAF - was supposedly only fixed on the A14 chip. There are conflicting reports on the reason the exploit chain does not work on A12 and up. Nonetheless, all of them suggest that underlying changes to the heap make it impossible without a secondary vulnerability to exploit.

[12] The response objects are allocated on the heap.

[13] And hence never freed.

2.5 Modern Security Measures on ARM

Although Apple's marketing team shrouds their processors in mystery and never mentions ARM in any official documents, their processors are based on the ARM instruction set. One important aspect of newer ARM processors, is Pointer Authentication Codes (PACs). Introduced with ARMv8.3-A, PAC allows important pointers, such as the return address, to be protected with cryptographic primitives from overwriting [17]. In simple terms, there are instructions to cryptographically sign a pointer with a key and associated context. Other instructions can then authenticate the signature - to ensure the pointer was not maliciously overwritten - and proceed[14]. There are five different keys for different purposes and different levels of isolation: `IA`, `IB`, `DA`, `DB` and `GA`. Usually, each `*A` key is shared amongst all processes of a system, whereas each `*B` key is unique per process[15]. `I*` keys and `D*` keys are used for signing instruction and data pointers, respectively. The `GA` key is used for general purpose signing.

2.6 Fuzzing

In its most basic form, fuzzing is the process of providing software with lots of input data - often random or unintended - and monitoring it for unexpected behaviour - usually crashes and memory corruptions or leaks. What started out as blindly hammering your keyboard to get software to crash, is now focused on so called coverage guided fuzzing. Using coverage information, fuzzers can make educated decisions of how to mutate input data to cover more of the software. The idea is, that if every part of a program is covered, any bugs should be found relatively quickly. A modern and widely used coverage guided fuzzer is AFL++ [8], which will also be used throughout this thesis.

Additionally, modern fuzzers use snapshotting and other techniques to improve the speed at which fuzzing can be carried out [29]. In particular, instead of starting the target anew for every new test case, the fuzzer snapshots the state of the target (memory pages, file descriptors and more) before entering the target function and restores the snapshot every time the target function returns. It can then enter the function again, while changing the parameters to point to the new test case. To gain even more performance, [29] implements the snapshotting inside the kernel using copy-on-write (COW), to only restore changed memory pages. Using this and many more tricks, [29] is able to attain speeds starting from around 1500 executions per second to almost 10'000 executions per second on a single core.

Unfortunately, most fuzzers, including AFL++, are geared towards working with source code and directly on parsing libraries, two points increasing the difficulty of this project, as neither are available for us.

[14] There are also instructions for authenticating the return pointer and immediately returning, to make for more compact code.

[15] This is for example not the case in SecureROM, the B keys are also shared amongst all tasks.

Chapter 3

Threat Model

As seen in Section 2.1, iPhones build a secure chain of trust, starting at SecureROM, allowing only code signed by Apple to run. Normally, attackers try to defeat this chain from the end, by having a complicated exploit chain to get kernel code execution[1].

On the other hand, an attacker can also focus their efforts on the other end of the chain, SecureROM. If an exploit were to be found, it can be very powerful, since it allows one to run any code and load even custom kernels with relative ease[2]. Furthermore, it can lead to a larger attack surface for SEP, allowing an attacker to potentially access a users encrypted data [28]. Lastly, code execution in SecureROM allows an attacker to dump SecureROM itself as well as important encryption keys[3], helping them find further vulnerabilities [23].

Knowing the implications of finding a vulnerability in SecureROM, it makes sense to fuzz it searching for such. Even though SecureROM is fairly limited in scope at first glance, there is still a large surface that could be good targets for an attacker. To see which parts of SecureROM are threatened by an attacker - and hence useful and interesting to fuzz - one has to consider the possible threat models. We identified two major threat models, depending on what kind of access an attacker might have to the device.

3.1 Physical Access

Also called the “evil maid attack”, we assume an attacker to have physical access to the phone, which might still be locked or even turned off. The only convenient interface the attacker has to SecureROM here, is the USB DFU protocol, after putting the device into DFU mode. This has also been where historically vulnerabilities were found, since USB message parsing and handling can be very complex and involves a lot of state. Additionally, the DFU protocol exposes another surface to the attacker, IMG4 parsing and verification. Via DFU, any file can be uploaded to SecureROM’s memory and ran through the IMG4 verification process. ASN.1 and DER are notorious for their complexity, to the point where many vulnerabilities in the past were caused by an ASN.1 parser. Since IMG4 verification involves parsing of the underlying ASN.1, this provides an excellent attack surface.

[1] Usually, such an exploit chain needs multiple vulnerabilities to work. First the user space sandboxing needs to be defeated. Then one needs to get kernel code execution. However, one is then still limited by the fact that the kernel is protected from changes by a complex system called APRR [24].

[2] For example, the attacker could overwrite the trusted root certificate with their own, by just having an arbitrary write in SecureROM.

[3] iBoot images are still encrypted with keys protected by SEP.

Contrary to normal operation - where Apple has limited the USB connectivity to unlocked devices only - the DFU protocol must always be accessible. Most defences depend on strengthening the security of SecureROM itself with a more secure heap implementation, pointer authentication or even running in user space[4].

3.2 Root on Device

Once an attacker (remote or with physical access) has gained root on the device, they are also limited in their options to interface with SecureROM. There is one main way to interface with SecureROM from the kernel, the storage medium for subsequent stages of the boot loader. Again, an attacker might want to break the IMG4 verification process by modifying the IMG4 file of the next stage. On older iPhones - the ones using APs before A11 - an additional surface was present, namely the parsing of NVME namespaces to find the next stage. On newer iPhones, this surface was removed and the next stage is loaded directly from SPI NAND without any complicated parsing.

Like in the previous threat model, most defences depend on strengthened security of SecureROM itself.

3.3 Conclusions for Fuzzing

Figure 3.1 shows a summary of the identified attack surfaces found in both threat models, the DFU handling (`getDFUImage`) and the IMG4 verification (`image_load`)[5]. An IMG4 file is either loaded from disk or - after it has been received via the DFU protocol - from memory. One can see, that an attacker with physical access has a larger attack surface, since they can attack both the DFU handling and IMG4 parsing. On the other hand, a vulnerability inside the IMG4 verification is useful to both attacker scenarios. Therefore, we decided to fuzz both, DFU handling and IMG4 verification, indicated by *. Henceforth we are considering the physical access threat model.

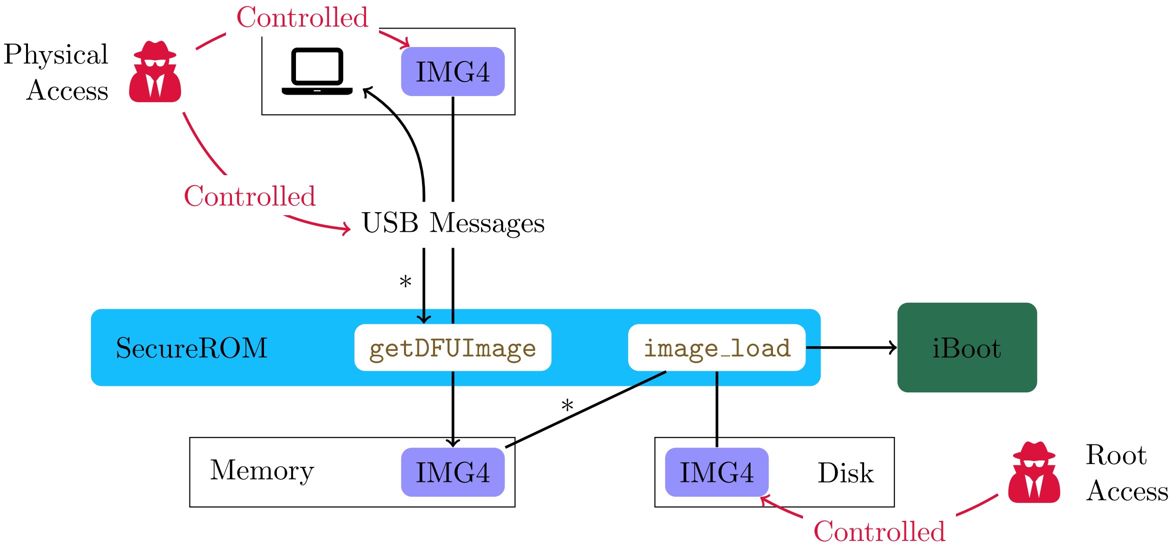


Figure 3.1: Summary of the two threat models and their attack surfaces.

[4] These mitigations are detailed in Sections 6.1.1 to 6.1.3.

[5] The significance of these functions will be explained in detail in Section 4.4.

Chapter 4

Design

We now go over the design goals of emmutaler, the major challenges present and their solutions. Finally, we present the design for the final fuzzing process alongside some terminology related to fuzzing.

4.1 Goal(s)

As revealed by the title, the main goal of the thesis is to fuzz the iOS boot loader. More specifically, the goal is to fuzz IMG4 parsing and verification, and the USB DFU protocol inside SecureROM. To this end, there are multiple related goals.

The solution for fuzzing SecureROM should be as general as possible. Ideally, one can fuzz many different versions of SecureROM without much additional effort. Since SecureROM cannot be updated, even older versions are still of interest to many people with older devices. Additionally, fuzzing different parts - such as USB or IMG4 - should require little additional effort. It might be interesting to fuzz more parts in the future.

Furthermore, there should be rich debugging capabilities to not only help with any found crashes, but also to simplify triaging unintended or wrong behaviour. This means, the resulting solution should also allow emulating or running SecureROM outside of any complicated fuzzing harness. Trying to discern between "real" crashes and crashes caused by our own implementation is a lot more difficult, if it requires a complicated fuzzing harness to be attached.

An explicit non-goal of this thesis is finding any novel vulnerabilities. While this would be nice to have, SecureROM has always been quite secure and unofficial sources have hinted at Apple increasingly using fuzzing. It would therefore be quite surprising to find vulnerabilities not already uncovered by Apple themselves[1].

4.2 Challenges

Fuzzing SecureROM presents quite a set of additional challenges, not present when trying to fuzz "common" targets. Instead of having access to the source code, we only have a compiled binary to work with. Furthermore, the binary is inherently designed for Apple processors and is expecting to run on bare metal, with access to various special peripherals present on Apple devices.

[1] Or additionally, any of the many companies backed by three-letter agencies.

4.2.1 Binary Blob

While source code for all of SecureROM and iBoot has previously been leaked, it is quite out of date and is missing some necessary parts to allow compilation. In particular, all of the IMG4 methods are missing and hence makes fuzzing IMG4 from source impossible. Therefore, we have only the compiled binary of SecureROM to work with. While modern fuzzers - such as AFL++ [8] - allow fuzzing of binaries without source, there are more challenges awaiting due to not having access to the source code.

For one, the input format for IMG4 has to be reverse engineered, since any existing documentation is either dated or quite confusing and incomplete. Additionally, many parts of SecureROM - for example setting up page tables or initializing timers - are unrelated to our goal and hence should be avoided. This in turn requires that the exact methods for IMG4 and USB handling are determined.

The next two challenges are also made harder without source code access.

4.2.2 Designed for Apple Processors

Not only does ARM give vendors the ability to have vendor specific registers and instructions, some instructions have implementation defined behaviour. Furthermore, Apple processors often implement the latest ARM specifications, whereas other manufactures are quite behind. Therefore, one cannot simply run SecureROM on easy to access ARM devices, such as a Raspberry Pi. While one could try running it on the recently released ARM Macs, there is no fully working version of Linux as of writing this thesis[2].

4.2.3 Bare Metal or Bust

Like any modern processor, ARM provides different exception levels (sometimes called rings in X86 or privilege levels), shown in Table 4.2 [15]. The current exception level indicates what privileges the current execution context has. There are two types of privileges, memory privileges and register access. Memory privileges indicate whether the MMU will use the unprivileged or privileged permissions when checking the rights of the current execution context with regards to a page[3]. For register access, any system registers have a suffix indicating the minimum required exception level to read or write them. Finally, a change in exception level can only occur by taking or returning from an exception.

Exception levels are important for our design, as SecureROM not only expects to have full access to the physical RAM and any system registers, newer devices also run at EL3 at points, which cannot be virtualized[4]. Additionally, SecureROM expects various peripherals, such as the SEP coprocessor or a USB interface, that can be communicated with. Such peripherals are hard to emulate properly and require a significant reverse engineering effort.

4.3 Solutions

Our solution to the aforementioned challenges and goals consists of multiple parts. The fuzzing is mostly done through AFL++ [8] and its design is discussed in Section 4.4. A major part of it,

[2] Most fuzzing tools are geared towards Linux or Windows.

[3] In other words, every memory page has two sets of permissions associated with it. One set for EL0 and the other set for EL1 and above.

[4] Often accesses to registers and other operations by a lower level, can be trapped by a higher one. Since EL3 is the highest level, it is not possible to trap accesses at EL3.

Shorthand	Memory Privilege	Register Access	Typical Use Case
EL0	unprivileged	`*_EL0`	application
EL1	privileged	`*_EL(0|1)`	operating system
EL2	privileged	`*_EL(0|1|2)`	hypervisor
EL3	privileged	`*_EL(0|1|2|3)`	firmware or security code

Table 4.2: ARMv8-A exception levels and their typical usage.

is static analysis of SecureROM. Thanks to static analysis, we can then do binary patching and finally create a Linux ELF based on the patched binary. A much more detailed explanation of our implementation of these solutions is presented in Chapter 5.

4.3.1 Static and Dynamic Analysis

Static - and in lesser parts dynamic - analysis forms the major basis of our solution. Using a good static analysis tool - in our case IDA - we can recover function names and type information, by reverse engineering SecureROM. With custom tooling, the symbols can be exported and called from outside SecureROM, meaning irrelevant parts can be ignored.

Furthermore, with static and dynamic analysis, Apple specific registers can be found and either figure out their meaning or how to work around them. It also helps identifying external peripherals or other memory mapped registers. We can then understand their purpose and how they could be emulated or even replaced.

4.3.2 Binary Patching

Another important part is a well working binary patcher. We use it to patch Apple specific registers, non-user mode instructions or otherwise instructions that cannot be emulated easily. Additionally, it allows us to selectively hook functions - for example replacing the heap implementation, see Section 5.8 - and even override single instructions with a custom implementation.

4.3.3 Linux ELF

Bringing it all together, we create a “normal” Linux ELF as our fuzzing target. By using a conventional Linux ELF, we can use existing fuzzing tools without any modifications, given that they support binary only fuzzing already. Furthermore, it also facilitates debugging, since any existing debugger will work fine.

To be able to build a conventional Linux ELF, we use the following high level process which could also be applicable to other platforms. First, the SecureROM binary is mapped at the memory location it expects to be at. Then, using normal C, we can call any SecureROM function, whose symbol was exported from IDA. In a sense, we treat SecureROM as any other library, whose functions we can call normally from our C code. Finally, we also patch or remove any irrelevant functions.

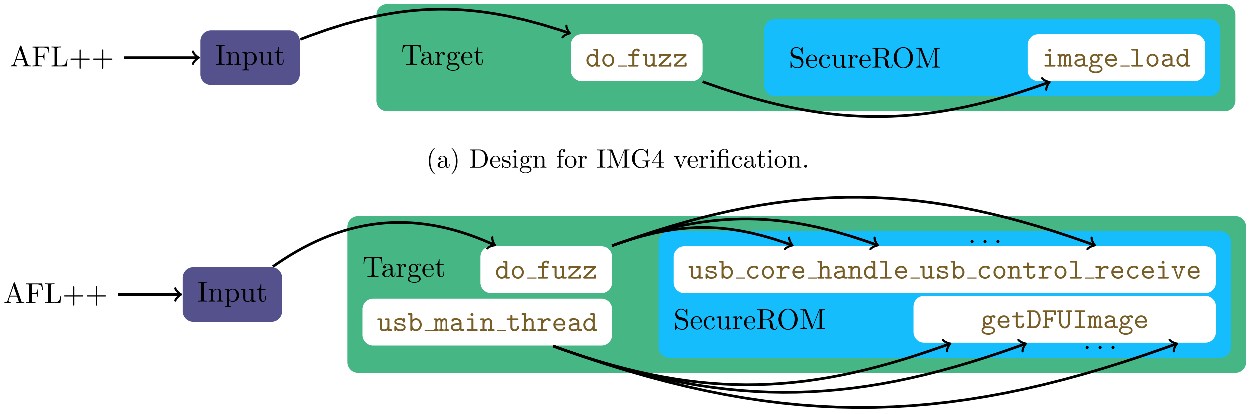
4.4 Fuzzing Design

For fuzzing we use AFL++ [8]. To that end, we give AFL++ a target - the Linux ELF - and some example inputs also called *seeds*. AFL++ then continuously runs the target, providing it with an input file, and creates new inputs via mutation. Alternatively, AFL++'s *persistent mode* calls a function in the target directly, that accepts a buffer and length parameter. The buffer and length passed in from AFL++ correspond to the contents and size of the input file respectively. We will mostly use persistent mode, as it claims to provide significant speed benefits.

The high level design for the fuzzing process is very similar between IMG4 verification and DFU handling. Figure 3.1 illustrates the important concepts from SecureROM as well and will be referenced henceforth. First, we identify the functions of SecureROM relevant for both IMG4 verification and DFU handling, which are already present in the figure. Next, we create separate target binaries for fuzzing both parts, as mentioned before. In essence, this constitutes to replacing the arrows marked with * in Figure 3.1 with AFL++ generating inputs and our binary passing those along. In particular, using SecureROM as a library, we create a simple Linux ELF that initializes some important state, accepts a single input and then passes it to the relevant functions of SecureROM.

In the case of IMG4 fuzzing, this is simple to do and we can directly pass the input to `image_load`. For DFU fuzzing, this is a lot more complicated. We have to reverse engineer major parts of the USB stack to see which functions are responsible for handling messages. We will then provide our own USB driver, that is tailored towards fuzzing. Additionally, the fuzzing input is split up into messages - details on this follow in the next chapter - which are passed to the message handling function `usb_core_handle_usb_control_receive`. Since DFU handling is inherently multithreaded, the created binary is careful to also replicate this behaviour with two threads. One passes along the messages, while the other continuously calls `getDFUImage`. Figure 4.1 presents a schematic view of both of these designs.

Once we have our binary, we call the process of running it with a single input under the fuzzer a *fuzzing run*. A single instance of a fuzzer, continuously executing fuzzing runs, is called a *fuzzing instance*. Usually different fuzzing instances have different parameters, that allow for different mutations or coverage information. In our case, a fuzzing instance also corresponds to one process with children and occupies one CPU core. A *fuzzing cluster* is a collection of fuzzing instances working together, that is they exchange interesting input files and all work on the same target binary. In our experiments, the only differences between fuzzing clusters is which binary they target - for example without certain bounds checking enabled - or what mode of the fuzzer they use. The fuzzing instances usually have the same parameters and everything else is also kept as consistent as possible.



(a) Design for IMG4 verification.

(b) Design for DFU handling.

Figure 4.1: Schematic view of our fuzzing design.

Chapter 5

Implementation

After having introduced the high level design of emmutaler, we now go over its implementation in detail, describe any additional tools and discuss ways to improve fuzzing speed. In particular, we first present our approach to static analysis, followed by a detailed description of how we use SecureROM as a library, including our binary patcher and how we export symbols. Then, we discuss the fuzzing process in detail, including input generation, AFL++ parameters, improving speed and our custom USB driver. Finally, we provide the implementation of our own allocator tailored towards fuzzing and detail how we collect coverage information for analysis.

5.1 Static Analysis

Continuous static analysis was performed during the implementation phase. This included, figuring out how input formats worked, how USB messages are handled, where important configuration values are stored and much more. Although leaked source code from 2016[1] helped quite a bit with static analysis, it also was outdated in key parts and firmly lacking in documentation[2].

The first hurdle in beginning the static analysis, was obtaining genuine SecureROM binaries. Fortunately, someone already collected a huge sample of them [12]. At the time of starting static analysis, the most recent SecureROM available was for the A13 chip. This is the SecureROM version most of the static analysis was based on. Recently, the collection was updated to include the latest SecureROM from the A14 chip, which has some additional interesting mitigations. Those are covered in Sections 6.1.1 to 6.1.3.

5.1.1 Memory Map

The first important step in not only reverse engineering SecureROM, but also loading it into an ELF was understanding the memory layout of SecureROM. Thanks to the source code and reversing the initial relocation loop, we identified a small part[3] storing some metadata and important memory address ranges. The full information present can be seen in Appendix A.1, but the most important parts are the locations of the `__text`, `__data` and `__bss` section. Using this and some more static analysis, we were able to have a complete memory map of SecureROM, an example of which can be seen in Appendix A.2.

[1] These days it is not so difficult to find online and - even though it should be legal to link it in an academic paper - we do not want to test Apple's lawyers.

[2] The description of the `docs` folder is aptly just "Optimisim.", including the typo.

[3] Starting at `0x200` from the base of SecureROM.

While the memory map is important for building the ELF binary, it is also useful for reverse engineering the rest of SecureROM. For example, SecureROM assumes that the data section is mapped at a specific address and hence references to those addresses are made. We wrote a custom loader for IDA, that places the different sections at the correct places in memory, so that references are resolved correctly.

5.1.2 Helper Scripts

In addition to writing a custom loader for IDA, we also created a few helper scripts, that automated certain - otherwise tedious - parts of static analysis. The more involved and interesting helper scripts are described in the following sections. In particular, these where involved in reverse engineering the signature verification and the USB stack. To be able to generate our own inputs for the fuzzer, we need to understand the input format and signature verification. Furthermore, - as seen in Section 5.7 - a deep understanding of the USB stack is needed to allow USB fuzzing.

5.1.2.1 Annotating OIDs

As previously explained, OIDs are very prevalent in any kind of signature parsing context, that uses X.509 certificates. To help figuring out which signature algorithm is used and how the signature verification works, a script was created to automatically find and annotate them. It is very simple, but makes reversing X.509 algorithms a lot easier.

Luckily, an OID is just stored as the DER encoding of the OID. For example, the OID 1.2.840.113549.1.1.1 becomes the sequence `0x2a`, `0x86`, `0x48`, `0x86`, `0xf7`, `0x0d`, `0x01`, `0x01`, `0x01`. Furthermore, the raw byte sequence of an OID, is referenced by an encapsulating structure. The script uses this to verify that the found byte sequence is really an OID and not just happens to look like one. The basic algorithm is shown in Code 5.1.

```
for oid in OIDS:
    addr = 0
    while addr != BADADDR:
        # Find the next address following addr that contains the byte sequence of oid
        addr = find_oid_bytes(oid, addr)
        # Find any data items referencing addr
        dref = get_data_references(addr)
        # The length field must be at dref + 8
        len = get_qword(dref + 8)
        if dref != BADADDR and len == oid_length(oid):
            # Annotate oid found at addr, i.e. sets name and type information
            annotate_oid(oid, addr)
```

Code 5.1: Simplified version of annotating all OIDs.

5.1.2.2 Annotating Management Interface Base (MIB) Accesses

Normally, a MIB refers to a database for managing entities in a telecommunication network. However, BSD (and hence likely Apple) uses it to store tuneable parameters, for example the number of ticks in a microsecond.

Throughout SecureROM an MIB - or at least Apple's version of one - is used for retrieving configuration data, such as the size of buffers or the size of a cacheline. Sometimes it was necessary to know what exactly an MIB entry contains or should contain. Unfortunately, annotating MIBs was not as easy as OIDs. Contrary to OIDs which were just static data items referenced by their address[4], MIBs were read out via a function call, whose argument is a constant enum. We wrote a script to first load all MIBs, including their type, the constant and the value, then comment any usage of the function with the resolved values and type information. See Code 5.2 for an example result.

```
unsigned int ticks_per_usec()
{
  unsigned int result; // x0
  unsigned int value; // [xsp+Ch] [xbp-14h] BYREF

  result = (unsigned int)cached_ticks_per_usec;
  if ( cached_ticks_per_usec )
    return result;
  // Value: 0x16e3600, Type: Int32 (Nix)
  mib_get_size(0x900u, kOIDTypeUInt32, &value);
  if ( !value || (result = value / 1000000, value % 1000000) )
    panic(&byte_10001EBC0, &byte_10001EBC0);
  cached_ticks_per_usec = value / 1000000;
  return result;
}
```

Code 5.2: Example of an annotated MIB call.

5.2 Building the Linux ELF

As outlined in Section 4.3.3, creating a Linux ELF binary can solve many of the presented problems. Additionally, - as seen in later sections - QEMU in user mode (and hence AFL++)[5] only works on Linux. Otherwise, one could have chosen a different operating system to target.

The detailed implementation is quite complicated, but some important aspects are described here. For an overview, using custom tooling, we can combine normal C code and the raw SecureROM binary, to create a Linux ELF for fuzzing. The C part provides a kind of wrapper around SecureROM, calling the functions deemed interesting to fuzz. In this sense, SecureROM is treated kind of like a library, that just takes some more effort to be linked into our - otherwise - normal C application.

5.2.1 Creation of ELF Sections

The first step in being able to treat SecureROM as just another library, was the ability to map sections of SecureROM into the final ELF. To this end, an assembler file `rom.S` is created. In there, normal GNU Assembler (GAS) syntax can be used to create individual sections. Additionally, using special syntax, the SecureROM binary can directly be included in these sections. An example of the text section can be seen in Code 5.3a. In contrast, sections that did not contain anything from the binary, but rather should be zero initialized, can be created using the `.fill` directive, as seen in Code 5.3b.

Finally, to make `ld` place the sections at the correct addresses - for example the text section at `0x100000000` - one just has to specify `-Wl,--section-start=.rom.text=0x100000000` on `gcc`'s command line.

[4] and hence once renamed showed automatically in the decompiler output

[5] See Section 5.4.

```
/*
    The text section,
    extracted from the secure rom image.
*/
.section .rom.text, "ax"
.incbin "../rom", 0, 0x25390
```

(a) Example of the text section.

```
/*
    Section used for storing the boot image.
    Only really used for fuzzing at the moment.
*/
.section .rom.img, "awx"
.global __start_rom_img
__start_rom_img: .fill 0x10000
```

(b) Example of a zero-filled section.

Code 5.3: Two different sections in assembly.

```
// Convert PAC branch to normal branch
r.PatchInstruction("blraaz ").Patch(r.PatchTmpl("blr {{(index .Args 0)}}"))
// Force bzero to never use dca
symb.rom__bzero.PatchOffset(0x18).Patch(r.PatchASM("cmp x2, #0x40000"))
// Override USB driver with custom one
symb.rom_synopsys_otg_controller_init.PatchOffset(0).Patch(
        r.PatchFunctionNoLink("emmutaler_controller_init")
)
// Handle exception vector
r.PatchInstruction("msr s3_0_c12_c0_0,").Patch(
        r.PatchFunctionTmpl("vbar_el1_handler", "{{(index .Args 1)}}")
)
// Patch in custom root certificate
certPath := filepath.Join(filepath.Dir(r.inputPath), "..", "certs", "root_ca.der")
certData, _ := os.ReadFile(certPath)
r.RawPatch(symb.rom_root_ca.Start, len(certData), fmt.Sprintf(`.incbin "%s"`, certPath))
```

Code 5.4: Example of patching, showcasing the different possibilities.

5.2.2 Patching the Binary

One important aspect of the build tooling, is the binary patching mechanism. We needed to patch the binary in a lot of different ways, from simply replacing certain instructions with `nop` to converting any PAC branching instruction to normal branches. Code 5.4 showcases the different possibilities that the final binary patcher allows.

At its core, there are two important concepts. The "patch locations" are responsible for determining where in the binary patches should be applied[6]. On the other hand, "patchers" are responsible for emitting assembly at specified patching locations, in other words they describe the contents of a patch. Since the relevant ELF sections are created by emitting a normal assembly file, patching the binary is achieved by simply emitting the relevant instructions, in between two `incbin` directives[7]. This has the advantage of patches being able to use any assembler directive. Code 5.5 shows an example of how a single patched instruction is emitted.

[6] This can be at potentially multiple locations, for example when patching specific instructions.

[7] There is a more complicated system to ensure all of SecureROM is included correctly, while allowing parts to be patched. This will not be described in detail here.

```
.global rom_platform_irq
.type rom_platform_irq,@function
rom_platform_irq:
    .incbin ".../rom", 11716, 92
    blr X8
    .incbin ".../rom", 11812, 116
```

Code 5.5: Example of patching a single instruction.

One major difficulty for finding the correct locations to patch is that often, similar instructions or the same instruction with different operands, need the exact same or a very similar patch. In one example, a good number of `blraaz` instructions with differing operands had to be patched with their equivalent non PAC counterpart, `blr`. To overcome this, one can specify a patch location as a regular expression (regex). The regex is matched against every instruction inside SecureROM and any instruction that matches, is patched. As seen in Line 2 of Code 5.4, this allows easily patching all `blraaz` instructions, regardless of operands. Additionally, patch locations can be specified as an offset from a symbol. The symbol locations are exported from IDA and available in the patcher through a `rom_` prefix. This makes the patcher more robust to work with different SecureROM versions, since no hardcoded addresses - that often change around - are needed.

A similar difficulty arises with patchers. The contents of a patch might depend on the specific location where it is applied[8]. Coming back to the previous example, the operands of `blraaz` had to be kept the same. Some patchers allow their arguments to be templated, which allows access to any information about the replaced instruction. As seen in Line 2 of Code 5.4, this allows the patching of `blraaz` to work flawlessly.

A difficulty specific only to patchers, is when a single instruction should be replaced with multiple instructions. When patching specific functions, this is often not necessarily needed, as many functions would be long enough to provide space for the replacement instructions. However, to make sure that patching works well, a function is always assumed to be only a single instruction and hence this issue emerges quite often. The solution is as follows. First, the instruction to be patched, is replaced with a branch to a new label[9]. Next, the new instructions are "wrapped" by the standard stack frame setup code on ARM[10], as seen in Code 5.6. Finally, the new instructions are placed at the end of the text section - labelled accordingly, to ensure that it can still be reached with a single instruction[11].

```
stp x29, x30, [sp, #-0x10]!
mov x29, sp
// new instructions appear here
ldp x29, x30, [sp], #0x10
ret
```

Code 5.6: Wrapping instructions to create a stack frame.

Another difficulty is patching using only assembly instructions. For example, implementing

[8] While the previous paragraph might have made it seem, like this could be fixed with a regex replace - instead of just a search - it would not work. Patching does not happen on an assembly level, but rather a text address level. Whether a specific address is patched or not, changes whether it is emitted as an `incbin` or the patched assembly directive.

[9] Depending on whether one might want to return to the original code flow or not, the branch is either a normal jump (`b`) or a call instruction (`bl`).

[10] While this might be a bit inefficient, it is much simpler to reason about and all of these patches are not in hot paths.

[11] Since instructions on ARM are fixed size at 4 bytes, branch instructions can have at most a 24-bit immediate value.

FETA - our custom allocator, see Section 5.8 - would be very tedious to achieve with just assembly. Additionally, it often makes sense to replace the implementation of functions entirely with a custom one. To this end, there are patchers for patching an address with a whole external function. The function should be defined in C and exported, so that the linker is able to find it. The patching works similar to multiple instructions. Again, to make sure that the function can actually be called even if it would be "far" away, we have to use some address trickery[12], as seen in Code 5.7. These are the instructions placed at the end of the text section, as explained before. The example also showcases, that we can potentially change the parameters the function receives. As seen on Line 11 of Code 5.4, the arguments of the newly called function can even be templated in a similar manner to normal instructions.

```
// PatchFunction("func", "0x42")
mov x0, 0x42
adrp x9, func
add x9, x9, :lo12:func
blr x9
```

Code 5.7: Example of instructions for calling a function.

Two specific instances where the binary had to be patched are detailed here, since they were interesting to fix.

5.2.2.1 Secret Pointer Authentication Code (PAC) Instructions

When running the binary, it would crash at addresses that were suspiciously close to those found for the kernel. This turned out to be a bug in QEMU, when receiving a signal that contained an empty fault address. The fault address was empty, because there was a jump to a non canonical address[13]. This happens, because of `blraaz` instructions, branching to pointers, that have not been authenticated. The PAC branch tries to verify the signature and in doing so "corrupts" the top bits of the pointer[14].

When this occurred, we were surprised since the PAC keys seem to be set accordingly for the PAC branch instructions to work. However, it seems that none of the function pointers get authenticated on startup. Thankfully, [4] already examined PAC on iPhones in detail and managed to find implementation defined bits of `SCTLR_EL1` that enable or disable specific PAC keys. SecureROM seems to only enable B keys for instructions. Hence, an easy solution to this problem is replacing all `blraaz` with `blr`.

5.2.2.2 Mysterious Heap Corruption

Another rather curious crash was observed when trying to initialize the heap. This is done, by providing a large chunk of memory to `heap_add_chunk`[15]. Unfortunately, this lead to a heap panic, claiming that the checksum of a heap allocation was invalid. After further investigation, we figured out that `heap_add_chunk` creates two guard allocations, right at the start and right at the end of the heap. The allocation with the invalid checksum was the one at the end and indeed its checksum was

[12] This sort of trickery is very common in ARM and would be emitted by the compiler itself, if we wrote in a higher level language.

[13] In simple terms, on X86, a canonical address has all top 16 bits either as a one or zero.

[14] This is a simplified explanation, but it should give the correct idea of what happens.

[15] This function name might be slightly misleading. The chunk in its name is not related to usual heap chunks, but rather refers to adding a big chunk of memory to be made available the heap.

set to all zeroes at some point in the initialization process. It happened right after the unallocated part of the heap was zeroed out.

Fortunately, [2] had previously done research on iPhones and made our lives easier in figuring out what was going wrong. The issue lies with the `bzero` function, specifically its fast code path. It uses `dc zva` to zero out large chunks - specifically `0x40` bytes - of memory at a time. The number of bytes to zero is implementation defined, but could be read out via the `DCZID_EL0` register. Since SecureROM is compiled for a specific chip, as [2] suggests, it is likely that such a check is optimized out or ignored entirely. Furthermore, QEMU uses a default of 512 bytes, which explained the zeroing of the checksum. It was simply zeroing too many bytes and hence the guard chunk got caught in the crossfire. Unfortunately, we still wanted PAC to be enabled, so choosing a CPU in QEMU with the correct number of bytes was not possible. To work around this, `bzero` was patched to only use the fast path with very large inputs, ones which should never be reached. The implemented patch can also be seen on Line 4 of Code 5.4.

5.2.3 Using Symbols from Static Analysis

To be able to efficiently and selectively fuzz SecureROM, one key feature is the ability to call any symbols found through static analysis from traditional C. Exporting the symbols with signature and location is quite simple, since IDA provides an API to retrieve both signature and symbol boundaries[16]. Difficulties arise when exporting type information and making the symbols visible to the compiler, and hence C.

While IDA has an API for exporting type information as well, it unfortunately cannot be compiled directly with any standard C compiler. One of the issues encountered, was types depending on each other. To make the compiler happy, the types are carefully iterated and for every type, any dependency is emitted first. For example, before emitting the struct we will see in Code 6.2, the types `DERByte`, `DERSize` and more have to be emitted. As seen in the example, this requires not only looking at struct fields, but also "drilling" down into them recursively.

Another difficulty is making the symbols visible to the compiler, while simultaneously making patching work well. A simple way to make the compiler see our symbols, is by having them as normal symbols in assembly. An example can be seen in Code 5.5, where the `rom_platform_irq` function is made available. One can see however, that this requires the originally large SecureROM binary to be split up into small chunks - one for every symbol. Every chunk is then included separately, in the generated assembly file.

To efficiently maintain these chunks and also allow patching of the binary, `ChunkTree` - a custom data structure - is used. It maintains a tree of chunks, that together span a specific address range. The chunks never overlap and there is never a single byte not contained in a specific chunk. Initially, the whole address range is spanned by a single chunk. Whenever a new chunk is added, the chunks currently overlapping with the new one are found and either split or shortened[17]. This way, we can be sure that all of SecureROM is included and there are no holes that can appear. To emit the symbol information as seen in Code 5.5, each chunk can have an associated symbol that starts at the chunks starting address. Thus, a first step is adding a chunk for every exported symbol. In addition, a chunk can have a string of assembly associated with it. If this is the case, instead of emitting an `.incbin` directive for the chunk, the assembly itself will be emitted. This forms the backend of the binary patcher, by adding chunks of the patched assembly. An example of the three chunks emitting the assembly seen in Code 5.5 is shown in Code 5.8.

[16] The process slightly differs for functions and globals, but is largely the same.

[17] Depending on whether the new chunk is in the middle of an existing one or at one of the ends.

```
[]Chunk{
        Chunk{Start: 0x100002dc4, End: 0x100002e20, Symbol: "rom_platform_irq"},
        Chunk{Start: 0x100002e20, End: 0x100002e24, ASM: "blr X8"},
        Chunk{Start: 0x100002e24, End: 0x100002e98}
}
```

Code 5.8: Example of chunks.

5.2.4 Symbolized Backtrace

A symbolized stacktrace[18] combined with signal handlers proved to be invaluable while debugging issues. Additionally, there is also some debugging code left in, that seems to log certain events with a unique ID. By hooking the logging function, these can be collected as well and provide additional information while debugging. Especially when trying to figure out why an image was rejected by the IMG4 verification, the logs were very useful, as they can usually pinpoint the exact property that was incorrect. Code 5.9 shows an example output of a handled signal including the symbolized stacktrace.

```
16:21:11 ERROR ../src/debug/signals.c:75: Received signal SIGSEGV @ 0x69000c800, register dump:
R00: 000000069000c800    R01: 000000032007bcd9    R02: 0000000000000001    R03: 000000069000c800
R04: 0000000000000000    R05: 0000000000000000    R06: 0000000000000001    R07: 0000000000000078
R08: 000000032007bcd8    R09: 0000000000000100    R10: 0000000000000100    R11: 0000000000000001
R12: 0000000000000020    R13: 0000000000000000    R14: 0000000000401768    R15: 0000000000000010
R16: 000000010000e488    R17: 000000550084ee80    R18: 00000055007fd692    R19: 0000000000420f10
R20: 0000000000000001    R21: 0000000000000001    R22: 000000019c010f00    R23: 0000000000001000
R24: 0000000000000000    R25: 000000032007bcd8    R26: 0000000000000001    R27: 0000000000000000
R28: 0000000000000000    R29: 0000005500800150    R30: 000000010000e5a0    R31: 0000005500800150
               STACKTRACE:
               [0x000000010000e59c] rom_usb_core_handle_usb_control_receive+0x114
               [0x000000010000eb5c] rom_usb_core_event_handler+0x94
               [0x0000000000403c98] ./main_usb() [0x403c98]+0x403c98
               [0x000000010000dc54] rom_usb_controller_start+0x1c
               [0x000000010000e434] rom_usb_core_start+0x2e0
               [0x000000010000ad04] rom_usb_init_with_controller+0x48
               [0x000000010000ee14] rom_getDFUImage+0x24
               [0x0000000000401ed8] ./main_usb() [0x401ed8]+0x401ed8
               [0x000000550084b080] /lib/libpthread.so.0(+0x7080) [0x550084b080]+0x84b080
               [0x0000005500942438] /lib/libc.so.6(+0xd0438) [0x5500942438]+0x942438
```

Code 5.9: Example stacktrace and handled signal.

Unfortunately, even though the symbols where added like any assembly symbol would be, neither a debugger nor other stacktrace libraries would correctly identify them. Additionally, any stacktrace libraries that were tried, did not even manage to recover the stacktrace. This is likely because the functions had no frame information associated with them and hence it was not possible for the libraries to figure out where the next return address was located. One solution could be to also export the frame information of every function in IDA and generate DWARF[19] information ourselves. This should then not only work to provide a stacktrace as seen before, but also allow a debugger to recover it.

However, this proved to be a lot more work than the simpler approach we took. Thankfully,

[18] "symbolized" here means, that the stacktrace shows the symbols from static analysis instead of just the addresses where something happened.

[19] DWARF is the latest format for providing debugging information in binaries.

any non leaf or thunk[20] function in SecureROM set up their stackframe normally. Therefore, the stacktrace can be recovered quite simply, by just following along the frame pointer. This basic algorithm is shown in Code 5.10. The symbol names are recovered by keeping a list of all symbols alongside their address range. If an address in the stacktrace is inside one of the symbols address range, the name is recovered. Sadly, this does not work in providing the stacktrace inside a debugger, but it is still useful regardless.

```c
typedef struct frame_info {
    struct frame_info* prev_frame;
    void* prev_pc;
} frame_info_t;

void backtrace(void* curr_pc, void* curr_fp, void** addr, size_t max_len)
{
    frame_info_t* prev_frame = curr_fp;
    for (int i = 0; i < max_len; i++) {
        addr[i] = curr_pc-4; // because call!
        prev_frame = prev_frame->prev_frame;
        if ((uint64_t)prev_frame <= 0x400 || (uint64_t)prev_pc <= 0x400)
            break; // ensure invalid addresses fail
        curr_pc = (uint64_t)prev_frame->prev_pc;
    }
}
```

Code 5.10: Algorithm for calculating the backtrace.

5.3 IMG4 Input Generation

Before the built ELF can be fuzzed, we have to create or find suitable input data. Since a fuzzer randomly changes input bytes, there are a few conditions on how the input data should look like. It should be as small as possible, as otherwise it might take forever to find the "right" input bit to flip, that creates an interesting[21] test case. Furthermore, having multiple test cases that are very similar - or more precisely have very similar coverage - does not help the fuzzing process and might make it a lot slower[22].

Considering these two requirements, normal IMG4 files from Apple cannot be chosen, as they are very large - up to multiple megabytes - and very similar to each other. Therefore, a custom tool was developed to generate IMG4 input files that fulfil the requirements. It can not only generate completely valid IMG4 files[23], but also invalid or even broken - in terms of the format - ones. To achieve this, the tool starts out with a simple valid file, that then is randomly mutated. The mutations can happen on either a high level - changing properties such as the name, the unique chip ID or signed with a fake root certificate - or a lower level - changing length, tag or contents of the encoded DER. After all mutations, it determines whether the input is still valid - in other words, the signature and digest are verified - and - depending on the result - the executed payload is modified. If the input is valid, the payload will return `RetValidPayload` (a constant number), whereas an invalid input will return `RetInvalidPayload`. This way, an invalidly signed input, that

[20] So called "thunk" functions are functions that directly call another function, so usually consist of a single branch instruction.

[21] In fuzzing, interesting usually means crashing, as that is the ultimate goal.

[22] This can be the case, if the fuzzer wants to try out all test cases, but since some of them are very similar, it does not gain any insight after trying one of them.

[23] Of course signed with a custom root certificate, not Apple's one.

passes all checks can be determined and the binary can crash. As such, certain classes of logic bugs should also be possible to detect.

On the backend of implementing such an input generator, there has to be a ASN.1 DER encoder library. Unfortunately, none of the existing libraries (for go) supported all the necessary features, such as large tags[24] or decoding the weird set format of IMG4 - see Section B.1. Therefore, we had to implement our own ASN.1 DER encoding library, that supports all needed features. Since a decoder was very useful - especially for reverse engineering the format - the library can also decode ASN.1 DER.

5.4 Fuzzing Setup

For our fuzzing setup we use AFL++ [8] as our fuzzer with a fairly standard configuration [9]. Every fuzzing cluster has twelve fuzzing instances running, where each number has a fixed label and associated parameters. The parameters for the different instances can be seen in Table 5.2 and have been chosen under guidance of the AFL++ documentation. cmpcov, cmpcovlib and cmplog all use different methods to instrument compare instructions. This can help with fuzzing complex input formats, by making AFL++ aware of failing branches. Except main, all other fuzzers run in non-deterministic mode, so they mutate randomly. This allows for the fuzzing to be distributed across multiple cores relatively easily, since every instance should be fuzzing different inputs. To combine the efforts of the individual instances, interesting inputs - ones with new coverage - are synchronized occasionally. main runs in deterministic mode, since this allows some additional mutations. Instance 2, qasan, does not run in our tests, since QASan [7] does not work with the kernel module described in Section 5.6.2. For more details on what the parameters do see the AFL++ documentation [1].

#	Label	Parameters
0	main	`-M`
1	cmpcov	`AFL_COMPCOV_LEVEL=2 -c 0 -S`
2	qasan	`AFL_USE_QASAN=1 -S`
3	cmpcovlib	`AFL_COMPCOV_LEVEL=2 AFL_PRELOAD=libcmpcov.so -S`
4	cmplog	`-c 0 -l AT -S`
5 - 11	fuzzer5 . . . 11	`-S`

Table 5.2: Fuzzing instances and their parameters.

To enable binary only fuzzing, AFL++ uses a customized version of QEMU that has various additions to a normal QEMU checkout. Specifically, it uses QEMU's user mode, which works by jitting instructions and relaying any system calls to the kernel. Most of AFL++'s additions are inside the JIT and focus on adding instrumentation. This includes instrumenting which basic blocks are hit, which compare instructions are executed and what is compared.

5.4.1 AFL++ Persistent Mode

As mentioned before, modern fuzzers implement a variety of tricks to speed up the fuzzing process. One such trick is called persistent mode in AFL++. It works as follows in the binary-only version

[24] Tags larger than 31, which is not actually that large.

of AFL++.

We provide AFL++ with the so called *persistent address*. Whenever a fuzzing run ends - either normally or due to a crash, AFL++ sets the instruction pointer to the this address, instead of actually terminating QEMU. In a way, this creates a fake loop inside the binary, which is much more efficient than constantly restarting the process. Due to this, a single fuzzing run is also called a *persistent iteration* in persistent mode. Usually, this should be set to the first instruction of the function to be fuzzed, which is also the case with emmutaler.

Normally when fuzzing with AFL++, AFL++ provides the new test case for a fuzzing run via stdin. While this would still work in persistent mode, to speed up the process even more, AFL++ loads a library from us, that copies the new test case to the correct location directly. This depends on what we are currently fuzzing, but is usually one of the arguments of the function that starts at the persistent address. Additionally, AFL++ also saves the state of the registers and memory before the first execution of the persistent address and restore it at the end of every persistent iteration. This ensures that every iteration has the same initial state and should make it easier to reproduce crashes.

5.4.2 AFL++ and QEMU Issues

Unfortunately, AFL++ did not work out of the box and we encountered multiple issues, both on QEMU's and AFL++'s end.

Persistent mode was broken on aarch64, since AFL++ had an off-by-one error. AFL++ checked whether the instruction pointer matched the persistent address before the instruction pointer was incremented and actually pointing at the current instruction. Therefore, the persistent iteration would only start at the second instruction and hence neither the frame pointer nor link register would be saved correctly on the stack. Fortunately, this was fixed easily and AFL++ in persistent mode now works correctly on aarch64 as well.

Section 5.2.2.1 describes one issue encountered with QEMU while developing emmutaler. Another issue with QEMU is unfortunately still unsolved. Due to unknown circumstances, QEMU can reach an assert titled "code should not be reached" inside the main CPU loop[25]. For IMG4 fuzzing, we worked around this issue, by snapshotting parts of libc. However, this can still occasionally occur during USB fuzzing. Fortunately, it will not be counted as a crash by AFL++ and we hence ignore it silently.

5.5 IMG4 Fuzzing

Once the tooling to interface with SecureROM was finished, setting up IMG4 fuzzing was relatively straight forward. Our C code is straightforward. It reads an input file up to `0x10000` bytes in size and directly passes it to `image_load` from SecureROM. If the image was loaded successfully, we then run it and check its return value. As explained in Section 5.3, we either abort or continue normally depending on the return value.

Before we can load the image, we also have to do some preparations. We need to setup the SecureROM heap correctly, since some parts of image parsing rely on a working heap implementation. This is straightforward again though, since we can directly use the normal heap initialization functions inside SecureROM. They just require a large chunk of memory. Additionally, we also need to setup certain fuses so that correctly signed input files are actually loaded correctly. This was again straightforward, as the fuses are memory mapped, thus we can write directly to them

[25] Clearly it is possible to reach this.

and set them to their correct value. To our knowledge, this is the only global state `image_load` depends on.

5.5.1 Allowing Out Of Bounds Accesses

During fuzzing we noticed that a lot of test cases failed to parse very early and hence did not test much of the binary at all. A more detailed breakdown of this is provided in Chapter 6, but we present an idea here to combat this issue. The basic idea why it fails so quickly, is the nature of our input format. Since DER relies on tag-length-value with length being variable, a slight change in a length field can have big impacts. In particular, it is very easy for the fuzzer to flip a bit and cause the length to be much larger than is allowed. SecureROM specifically makes sure that any parsed DER item does not exceed its parent. However, all DER parsing functions also have a useful parameter to ignore those checks. Therefore, we created a special version of SecureROM that patches the DER functions to always ignore bounds checks. In Chapter 6 fuzzing done using this version will be labeled as OOB, standing for out-of-bounds.

5.6 Improving Fuzzing Speed

To get a good initial fuzzing speed, we took the same measures one would normally take to speed up a binary. For example, we compiled the binary with `-O3` and kept logging to a minimum in builds made for fuzzing. Furthermore, we integrated additional steps to speed up fuzzing as much as possible. As described in Section 5.4.1, persistent mode should have a significant impact on fuzzing performance. Detailed results of its impact will be presented in Section 6.5.

5.6.1 Profiling Speed Issues

Unfortunately, even with persistent mode, fuzzing was not as fast as others have observed. While detailed comparisons follow in laters sections, we describe how we profiled these issues and what we did to achieve even greater speed up here.

Using `perf` we can quickly see what the remaining bottlenecks are. The first that springs into the eye is related to PAC instructions. A significant amount of time is spent calculating and verifying the PAC signatures inside QEMU. This makes sense, as on actual Apple chips, these cryptographic operations would be implemented in hardware and not in software like in QEMU. The easy fix here, was to just patch out the signature creation and verification inside QEMU. Patching any PAC instructions inside SecureROM would have been quite a bit more involved, since some of them cannot just be replaced with `nop`.

The next major bottleneck was a bit surprising. Almost all time was spent inside `memcpy`. After some more analysis, we discovered that the snapshotting implementation of AFL++'s persistent mode was the main culprit of this bottleneck. In essence, it snapshots any page that is declared as writeable and restores it at the end of every single persistent iteration. Not only does SecureROM have quite a lot of writable pages by itself (for example both data and bss are quite large), we allocated more than enough pages for special regions - such as IO. In total, AFL++ was snapshotting around 80 MB of memory before every single persistent iteration. Even by modifying AFL++ to only snapshot certain pages, we still had to snapshot a large chunk of libc to prevent crashes[26]. In the end we had to look for another way of speeding up AFL++.

[26] As explained in Section 5.4.2.

5.6.2 Kernel Module for Fast Memory Snapshotting

One thing we recognized, is that only a small part of the snapshotted pages actually are modified during a persistent iteration and hence need to be restored. Therefore, we looked into a COW implementation. Fortunately, [29] already implemented exactly this as a kernel module and AFL++ even had support for said module. However, when we tried to use it, it was completely broken. Not only did it not compile, it contained multiple UAF bugs and race conditions, leading to many kernel panics.

We fixed the encountered UAF bugs and eliminated the race conditions. Additionally, we replaced some data structures with ones that are faster to lookup up from. Lastly, we also rewrote a lot of parts to be compatible with a larger range of kernel versions. This was especially necessary, since our test machines could vary wildly in their kernel versions.

Unfortunately, the kernel module is still not bug free. If a snapshotted page is the first being accessed, it will not be restored correctly. We believe this to be due to the fact that the kernel applies COW to page table entries as well and hence we do not catch when the page is actually requested. However, this does not impact our fuzzing at all, since none of the snapshot pages are accessed first.

5.7 USB Fuzzing

Whereas IMG4 fuzzing was relatively straight forward once the foundational tools were available, USB fuzzing proved to be a lot more complicated. For once, the DFU protocol had no documentation at all and documentation for interfacing with the standard USB protocol implemented by SecureROM was equally sparse. Thanks to a great reverse engineering effort, most of the USB stack is now fully understood and we can fuzz it as efficiently as possible. However, there are still quite some limitations, as will be evident in the following sections.

Apart from the difficulties encountered to get USB fuzzing working, the process works exactly the same as for fuzzing IMG4. While we do not have an implementation that also disables some bounds checking here, we had three fuzzing clusters, each running the instances as described in Section 5.4. Each cluster was targeting the same binary and had the same parameters on each instance. This allows us to get an idea on how long it takes to find a bug such as checkm8 on average. The results of this process are discussed in Section 6.4.

5.7.1 Custom USB Driver

One possible way to fuzz USB messages would be read and write the registers used by the USB driver inside SecureROM. While this would certainly work, it requires a lot of reverse engineering and can easily break between different version of SecureROM. Fortunately, the USB implementation inside SecureROM is already very modular and we can instead write a custom driver that acts more like a paravirtualized one. By replacing a single instruction inside SecureROM, we replace the existing driver with ours.

On the sending side, our driver is quite simple. A single function - `transmit_usb_buffer` - is responsible for “transmitting” a buffer of specified length. To this end, we just call `usb_core_handle_usb_control_receive` inside SecureROM with the specified buffer. This is the same function the normal driver calls, when it has finished reading out a buffer from a real USB connection.

While not really needed for fuzzing, the ability to receive responses is also implemented, as it allows for easy testing of whether it actually works. Again, this is very simple, as we just implement the `usb_do_endpoint_io` function of the driver and will receive a response. To not have to worry about having to wait for the response, any response received is written to a global variable inside usb_do_endpoint_io and also read out inside transmit_usb_buffer. This works, since usb_core_handle_usb_control_receive only returns after having called our usb_do_endpoint_io implementation[27].

Although the driver is relatively simple, one big issue is the multithreaded nature of the USB handling. During normal operation, getDFUImage initializes the USB driver, starts it and then waits for the DFU protocol to signal it is done. When started, the original driver starts a separate task to send and receive USB messages. In our implementation, we switch the purpose of the threads around. The USB driver for sending and receiving messages is running on the main thread, while getDFUImage is started on a separate thread. This makes fuzzing a lot easier, since we can directly call the sending functions with AFL++'s persistent mode. Code 5.11 shows a simplified version of the USB thread function used for getDFUImage. Code 5.12 shows the simplified function ran by the main thread to process USB messages to be transmitted to SecureROM.

Since in the main thread, we might be in the middle of sending messages when getDFUImage returns, we need to ensure that we keep returning to getDFUImage for as long as we still have messages. For various reasons, we achieve this by just looping inside the dedicated USB thread. Creating a new thread is slow, especially when fuzzing and as such it is easier to keep looping inside the thread. This should have no impact on the semantics of our implementation of the USB DFU handling, as a similar loop is present inside SecureROM. As long as DFU mode is activated, it will keep calling getDFUImage, until it is able to boot a valid image.

```
void usb_main_thread(void* args)
{
    while (!main_should_exit) {
        int res = rom_getDFUImage(rom_img_start, kIMG_SIZE);
        struct image_info* img_info = image_create_from_memory(rom_img_start, res);
        load_and_test_image(img_info, kDEFAULT_TYPE, rom_img_start, res);
        free(img_info);
    }
}
```

Code 5.11: Simplified USB thread function.

```
void process_messages(void *buffer, uint64_t length)
{
    for (int i = 0; i < length / sizeof(usb_msg_t); i++) {
        usb_msg_t* curr = &((usb_msg_t*)buffer)[i];
        switch (curr->type) {
            case SETUP:
            case DATA:
                transmit_usb_buffer(curr->data, curr->data_size, curr->type == SETUP);
                break;
            /* Handle other types ... */
        }
    }
}
```

Code 5.12: Simplified USB message processing loop, running on the main thread.

However, there needs to be synchronization between the main thread and the USB thread to exit the loop inside the USB thread. This is not as trivial as it might seem from the previous

[27] While this might seem delicate and could easily break, it works fine for debugging and is not necessary for fuzzing.

code listings, since getDFUImage waits on an event from the DFU protocol implementation inside SecureROM. It also stops the driver and turns of the USB stack whenever it returns. To fix the first problem of getDFUImage being stuck waiting on an event, we can send an abort request. However, our driver might not be ready yet to receive such a request, since getDFUImage might have just returned. The main thread then waits on the USB thread to exit, which might not happen, since it might have called getDFUImage. An example of such a bad interleaving can be seen in Code 5.13.

```
rom_getDFUImage(...) // returns | USB
// USB driver is now inactive
while (!main_should_exit) { // main_should_exit still false | USB
send_abort_nowait() // called & returns | MAIN
// abort did not get sent, since USB driver still inactive!
thread_join(usb_thread) // called | MAIN
rom_getDFUImage(...) // called | USB
// USB driver is only now active, but we are deadlocked!
```

Code 5.13: Bad interleaving for trying to exit out of the USB loop.

Our solution to this involves complex system of multiple locks and events. For example, the main thread sends aborts for as long as the USB thread has not exited the loop yet.

5.7.2 To Panic or not To Panic

Another rather substantial problem of the DFU handling inside SecureROM is, that it can panic very easily. If the DFU protocol has received less than 0x10 bytes of data for the next stage image, before finishing the DFU protocol, it will panic. Finishing the protocol is also rather easy to trigger, as even an abort will do that. Therefore, it is quite common to see SecureROM panic, not only during fuzzing, but also when we were debugging and testing our implementation.

When fuzzing in standalone mode, a panic would not be a big deal, as we can overwrite the panic handler and exit normally. However, when fuzzing in persistent mode, a panic is very undesirable, as no locks or threads are cleaned up. Since the process will not actually exit in persistent mode, the next fuzzing run would completely halt and be unable to proceed. Fortunately, we know exactly which locks are held when we reach such a panic. Therefore, our custom panic handler unlocks these locks, ensures the USB thread exits cleanly and then finally exits normally. This ensures that we can start another fuzzing run.

5.7.3 Input File Format

Even with a custom USB driver, AFL++ has to still be provided with input files it can mutate. Therefore, a custom format was designed that is optimized for fuzzing. It consists of arbitrarily many so called messages of the same size, which are packed together in the file. Although they are called messages, they are not limited to a USB message, but can also for example cause a USB event. The definition of a single message can be seen in Code 5.14.

```
typedef struct usb_msg {
    uint8_t type;
    uint8_t data_size;
    uint8_t data[0x40];
} usb_msg_t;
```

Code 5.14: USB input format.

The input format wastes as little space as possible, so that flipping a bit in the input should always have a directly visible effect. To this end, both `type` and `data_size` are only a single byte in size, since they cannot use more. As seen, data is at most `0x40` bytes, so the maximum size easily fits in a single byte. While even less space could be wasted by combining `type` and `data_size` into a single byte, this was avoided for simplicity and future expansion.

Additionally, `data` is fixed size to not have huge changes in the input interpretation, while only changing a limited number of bits. If it was variable sized, using `data_size` as information on how large it is, a small change in `data_size` would have a large impact on all subsequent messages. Since the fuzzer does not know to account for the change in that field - for example by expanding or shrinking `data` - subsequent messages might completely change. Not only does this make the fuzzing a lot less controlled, it will also make reproducing test cases a lot harder. It would be difficult to identify quickly which messages are important.

Coming back to the message definition, the following types are implemented:

- `SETUP`: The data of this message should be treated as a USB setup packet[28] and sent to the USB driver accordingly.
- `DATA`: The data of this message should be treated as a USB data packet and sent to the USB driver accordingly.
- `EVENT`: The first four bytes of `data` are treated as an `usb_event_t` and passed directly to `usb_core_event_handler` from SecureROM. `usb_event_t` has the following - known[29] - possible values: `CABLE_CONNECTED`, `CABLE_DISCONNECTED` and `USB_RESET`.
- `SLEEP`: The first 8 bytes of `data` are treated as an `uint64_t`, indicating the number of milliseconds to be slept for.
- `NOP`: Do nothing.

While for `SETUP`, `DATA` and `EVENT` it is clear why they are needed, one could certainly argue that the other two are not. `SLEEP` could prove itself useful, if a potential crash is dependent on race conditions. To limit any fuzzing degradation - be it in slow test cases or AFL++ trying to create hanging test cases, the maximum number of milliseconds that are allowed is capped at 10. This should give enough time to trigger most race conditions, while simultaneously not slowing fuzzing down by much. `NOP` on the other hand could definitely be removed, but it was also trivial to implement.

5.7.4 Persistent Mode for USB Fuzzing

As described in previous sections, there were quite a few problems getting persistent mode working to a satisfactory degree with threading. However, we encountered the same issue again as seen when fuzzing IMG4, most time was spent snapshotting and restoring all writable memory. Unfortunately, the kernel module does not support threading at all. Nevertheless, we still tried to get it running, to no avail. When it was turned on, QEMU would either crash or we would hang somewhere. Still, we gained a significant speedup by using persistent mode instead of fuzzing in standalone mode, as is discussed in Section 6.5.

[28] In the DFU protocol, USB packets can be of two types: setup and data. According to the USB standard, a setup packet is always followed by zero or more data packets.

[29] There were no additional ones found during static analysis.

5.8 Fuzzing-Enabling Thread-safe Allocator (FETA)

To see whether the USB fuzzing is successful, we wanted to see whether it would find the UAF exploited by checkm8. However, when trying to trigger the UAF manually, we noticed that, while we were able to trigger the UAF, we could not get it to crash. At the point where the USB driver writes to the freed pointer, the freed buffer is allocated again. If there is no complicated heap feng shui previous to that, the buffer is allocated at the same location as before. Therefore, the freed pointer points to a completely valid heap chunk again and the write in the USB driver succeeds without an issue. To be able to detect a UAF regardless of the heap layout - and hence without the need for the fuzzer to first find a working heap feng shui setup - we created the "Fuzzing-Enabling Thread-safe Allocator" (FETA).

It is a drop-in replacement for malloc and free, so we patched those calls to instead go to their respective implementations in FETA. The main goal of FETA is to detect any heap issues a fuzzer might encounter, such as overflows or UAF, while also being thread-safe. FETA achieves this, by mapping a certain number of pages for every requested heap allocation. When a heap allocation is freed, the corresponding pages are unmapped. Because of this, any usage of the freed allocation, will immediately trigger a segmentation fault, even if it is only a read of said allocation[30]. To ensure that a previous allocation is not mapped again - and hence a UAF would not be detected, addresses of new allocations only increase and the pages are mapped at fixed addresses.

Furthermore, to prevent any overflows or over reads[31], the returned address of the allocation is calculated, such that the end of the allocation is exactly at the end of the mapped pages. Additionally, between any two neighbouring allocations, there is an unmapped guard page, ensuring that an overflow or over read crashes.

Lastly, FETA stores a pointer to a chunks metadata information right before the returned pointer. The metadata allows FETA to snapshot the state of the heap and restore the heap to a previous snapshot. However, this is not used currently, since AFL++ itself implements snapshotting and hence FETA does not need to snapshot as well.

Figure 5.1 visualizes some example allocations seen in Code 5.15. We assume a standard page size of 0x1000 and that the initial allocation will happen at 0x20000.

```
void* chunk1 = malloc(0x100);
void* chunk2 = malloc(0x1000);
void* chunk3 = malloc(0x201);
free(chunk2);
```

Code 5.15: Example allocations.

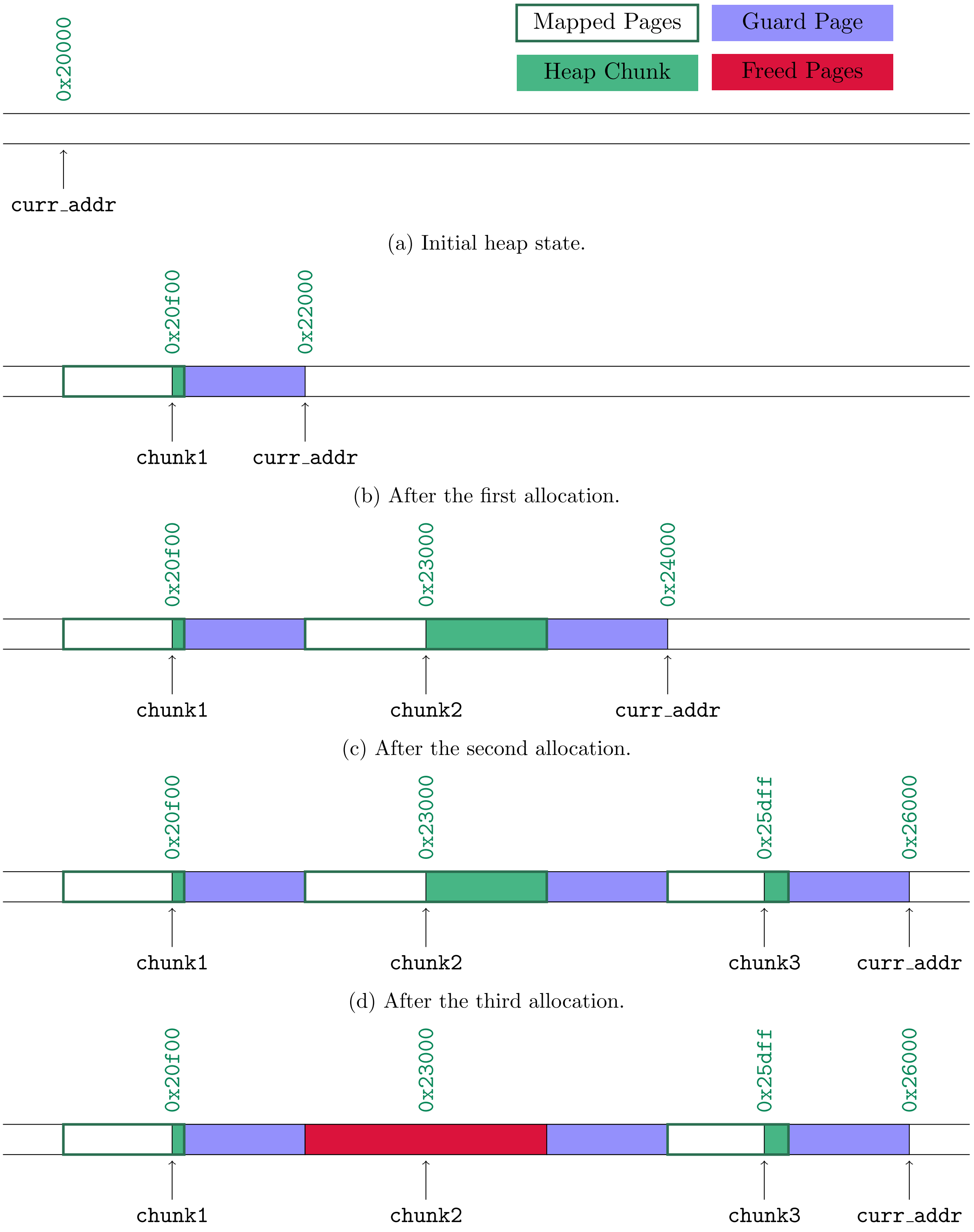
5.9 Coverage Information

While AFL++ already provides some coverage information, it is very limited and - for example - gives no indication whether covered blocks[32] are hit often or not. To be able to see whether the coverage guided fuzzing was actually successful in not only reaching most blocks, but also reaching them often, we had to use something else. Fortunately, QEMU provides an API for developing plugins that allow to hook a lot of the internals of QEMU [21].

[30] Even leaks by means of reading freed allocations can help in exploitation.

[31] Reading more than the allocated chunk, which again leads to a potential heap leak.

[32] Whenever we refer to blocks hereinafter, we refer to the basic blocks in a standard control flow graph.



(a) Initial heap state.

(b) After the first allocation.

(c) After the second allocation.

(d) After the third allocation.

(e) After the free.

Figure 5.1: Visualization of example allocations.

The basic idea behind measuring and visualizing the coverage of the fuzzing process is hence as follows. First, AFL++ saves any input that generates new coverage to a file, hereinafter referred to as *interesting input*. Those items can be manually ran one by one through QEMU with a custom plugin, whose job it is to save the coverage information to a file. Lastly, the resulting coverage files can be imported into IDA using a modified version of the Lighthouse plugin [10]. This allows the coverage to be visualized on a control flow graph (CFG) in addition to having just the raw statistics.

Thanks to QEMU's plugin API, the plugin is actually very simple. One can specify an address range whose coverage should be captured. It then allocates an integer array, large enough to hold a single 16-bit integer for every address in the given range. This array, is actually backed by a file of the same size, by mapping a file descriptor directly into memory. Due to this, even if QEMU were to crash or hang[33], the coverage information should still be persisted to disk. QEMU is then told to increment the integer element corresponding to a specific address, whenever an instruction is executed at that address.

With this approach, the coverage information not only contains which instructions were executed, but also how often. In essence, interesting inputs give exactly all paths the fuzzer was able to find throughout the binary. Therefore, we aim to gain some insight in the success of our fuzzing, by looking at how different functions or blocks fare in relation to each other and all found paths. For example, a block that is executed in most paths, is much more likely to have been entered with a variety of possible states and hence a crash is more likely to be found - if it exists. To be able to use the more detailed coverage information, we introduce two metrics.

The first metric, *input relative percentage* or *i-percentage* for short, gives the percentage of executions, for which a particular basic block was executed at least once. Using this, we can deduce on what percentage of paths a particular basic block lies. A complex basic block that lies on a small percentage of paths could indicate that the block was not exposed to many possible states and hence there could still be crashes lurking there. Equation 5.1 shows the formula for calculating the i-percentage of basic block bb. We designate $\mathcal{I}$ as the set of all interesting inputs and $\text{execs}_I(bb')$ as the number of times basic block bb' was executed during the input $I \in \mathcal{I}$.

$$\text{percentage}_i(bb) = \frac{|\{I \in \mathcal{I} \mid \text{execs}_I(bb) > 0\}|}{|\mathcal{I}|} \tag{5.1}$$

There are two drawbacks to the i-percentage metric. Often parsing functions have loops or in the case of USB fuzzing, our wrapper itself has a loop. Therefore, some basic blocks might be exposed to different states much more often than others, but since they are only counted once per input, this information is lost. The other drawback is, that if a function is executed very little, all percentages will be very small making it hard to discern any differences between basic blocks. To combat these drawbacks, we introduce the second metric, the *function relative percentage* or *f-percentage* for short. It gives the percentage of times a basic block is executed, relative to how many times the function was entered. Equation 5.2 shows the formula for calculating the f-percentage of basic block bb relative to function F. We designate $\text{execs}(bb')$ as the total number of times the basic block bb' was entered across all inputs and $\text{entry}(F)$ as the basic block of function F executed upon entry.

$$\text{percentage}_f(bb) = \frac{\text{execs}(bb)}{\text{execs}(\text{entry}(F))} \tag{5.2}$$

The results are presented in Section 6.6 and we use the i-percentage for the entry basic block of a function, whereas all other basic blocks will use the f-percentage.

[33] Meaning QEMU itself would crash, which unfortunately, still happened from time to time.

Chapter 6

Evaluation

In this chapter we present the results obtained with emmutaler and during its development. To start, we present the results of our static analysis, including three particularly interesting mitigations present in SecureROM we found. To our knowledge, two of these mitigations are novel and have not been discussed elsewhere. Then, we discuss the fuzzing results in detail, including the discovery of the checkm8 bug. Furthermore, we compare the different AFL++ modes and discuss their impact on fuzzing speed. Finally, we analyze the achieved coverage in detail and relate it back to the fuzzing results.

6.1 Static Analysis

In total, out of 724 functions[1], 554 have been identified by name. Almost all of the identified functions have correct type signatures now and are fully understood. Thanks to the type signatures, most of the functions have very readable decompiled code. An example of a smaller, now readable function is shown in Code 6.1. For a more complex function that includes virtual function calls, see Appendix C.

```
__int64 sub_100009BCC(char *a1)
{
  __int64 v3; // x0

  sub_1000127BC();
  if (a1 == aKsat || a1 == &unk_19C0107C0)
    sub_100008F90();
  if (*((_QWORD *)a1 + 3) || *((_QWORD *)a1 + 4))
    sub_100009C50(a1 + 24);
  sub_100009C50(a1 + 8);
  v3 = sub_100001C14(a1);
  sub_100012810(v3);
  return sub_10000FEF4(a1);
}
```

(a) Function before any static analysis.

```
void task_destroy(struct task *a1)
{
  enter_critical_section();
  if (a1 == &bootstrap_task || a1 == &idle_task)
    panic();
  if (a1->queue_node.prev || a1->queue_node.next)
    list_delete(&a1->queue_node);
  list_delete(&a1->task_list_node);
  arch_task_destroy(a1);
  exit_critical_section();
  heap_free(a1);
}
```

(b) Function after static analysis.

Code 6.1: An example function responsible for cleaning up after a task exits.

Not only do most functions have type signatures now, 202 types were identified and reversed.

[1] At least that is how many functions we found. While there could be more - there were more found while writing this thesis - it would not be a substantial amount.

As seen in Code 6.1, this allows IDA to correctly display structure member references and made reversing easier. An example, of such a type can be seen in Code 6.2.

```
struct Img4DecodeImplementation
{
  int (*do_ccdigest)(DERByte *, DERSize, uint8_t *, uint64_t, uint64_t);
  int (*parse_cert_chain)(DERByte *, DERSize, uint64_t *, uint64_t *,
    uint8_t *, uint8_t *, uint64_t, image4_wrapper_context *);
  int (*verify_pkcs1_sig)(const DERItem pubKey, DERItem sig, const DERItem toVerify, void
↪   *hasher);
  int (*verify_payload_properties)(img4 *, image4_wrapper_context *);
  hash_info *hasher_info;
  DERItem *digest_oid;
  DERItem *alg_oid;
};
```

Code 6.2: A type which is used by the IMG4 implementation.

In general static analysis went quite well and anything useful for fuzzing was reverse engineered completely. In addition, multiple interesting and previously unknown mitigations were discovered during static analysis. Most of those mitigations were introduced with the A12 chip and seem to indicate a general lockdown of SecureROM to increase security.

6.1.1 Mitigation 1: Running at EL0

Up until A11, SecureROM would run at EL1/EL3 until jumping to the next stage, which would potentially downgrade to a lower exception level. However, with A12, SecureROM quickly drops down to EL0 after entering the main function. To get around the restrictions present in EL0, there are multiple syscalls implemented starting with A12.

- `svc 0`: This syscall immediately returns from its handler, but continues execution as EL1. Through this method, SecureROM can execute privileged instructions, even when the rest of the code flow runs as EL0. A helper method allows returning back to EL0, as soon as the privileged instruction is executed. Additionally, thanks to helper methods, this operation can be nested and only the outermost instance will actually return to EL0.

- `svc 1`: IRQ handling is now done at EL0. As soon as an IRQ is received, execution returns to EL0 and the handling of the IRQ proceeds as before. Once the IRQ is correctly handled, SecureROM uses this syscall to return from handling the IRQ. This syscall returns from handling IRQs

- `svc 2`: Only present on A14, this syscall disables interrupts. On older chips, this was done by manipulating system registers.

- `svc 3`: This is the counterpart to the previous syscall, it enables interrupts. Again, this is only present on A14, as on older chips it was done by manipulating system registers.

This mitigation is interesting, as it is useless against an attacker gaining code execution. Such an attacker, could directly execute the syscall for gaining EL1/EL3 and then the mitigations would be defeated. However, they certainly help against attacks that rely on writing to some privileged memory, for example [14].

6.1.2 Mitigation 2: Pointer Authentication Codes (PAC)

With A12 being the first chip to support ARMv8.3-A, it also introduced PAC into SecureROM. Although all PAC keys are initialized with random values, only the `IB` key - see Section 2.5 - is enabled, as explained previously. This means that any instructions using other keys neither creates nor verifies signatures of pointers. Instead, they just perform their non-PAC function[2]. In addition, SecureROM seems to be compiled with PAC enabled, since all branches and returns use the PAC versions of their instructions. Furthermore, it seems that on A14, even more keys are initialized than are specified by the ARM specification. However, since these seem to be in implementation defined system registers, it is impossible to tell what those keys are actually used for. One possibility might be, the same set of keys as present for EL0, also for EL3. This speculation is based on the fact, that there are exactly the same amount of unknown keys as known EL0 PAC keys.

By using PAC, there is an additional security layer on top of the already used stack canary to prevent stack overflows. However, since only the `IB` is enabled, neither data nor function pointers - such as in the implementation of the hashing - are protected. While this weakens security, it is probably a performance tradeoff. PAC keys for data and function pointers might not be enabled, since it would take too long to sign the pointers after initially setting the keys.

6.1.3 Mitigation 3: Complicated Fuse Readings

Up until A11, the security mode was stored in a single bit in one of the fuses. This means, that by somehow getting the processor to read out a 0 instead of a 1 for this bit would completely defeat the security of SecureROM. This could be achieved in a multitude of ways, for example glitching the read out.

The possibility of defeating the whole security model of a device by glitching a single read might explain the next mitigation. On A12 and up, specific fuses - those closely tied to the security model of the device - are read out in a much more complicated fashion. For example, consider reading out the value of the current security mode. Previously, this would be done by simply calculating `(rCFG_FUSE0_RAW >> 1) & 1`, where `rCFG_FUSE0_RAW` is the double word at the memory location where the raw value of config fuse 0 is mapped. Code 6.3 shows what the more complicated process looks like on newer devices[3].

6.2 Binary Creation

Although there are not many results to present for the binary creation, it worked perfectly for our use case. Not only can we easily run the resulting binary outside the fuzzing harness, we can also debug it easily. Especially with the leftover tracing code from Apple itself, many bugs were easy to diagnose and fix. Furthermore, we can even debug the binary using the already annotated SecureROM binary in IDA and hence have a much easier time diagnosing issues as well. Therefore, it stands to reason that emmutaler could not only be used for fuzzing, but also serve as the base for a general analysis and debugging platform for SecureROM, outside of Apple. Exploits are definitely easier to debug and test, when one does not have to constantly restart an iPhone.

However, in its current state, emmutaler still requires quite a lot of manual labour to work with different SecureROM versions. While some symbols can be automatically recognized in other versions, often they are not recognized, even though they differ by very little. Additionally, memory

[2] For example, `retaa` would just return, instead of also verifying the signature of the return pointer.

[3] Since effectively a single bit in a config fuse now uses all 32 bits of the fuse, secure mode is stored in fuse 1, and not bit 1 of fuse 0.

```
int cfg_fuse_get(uint32_t fuse_val, uint32_t fuse_num, bool is_raw)
{
  if ( fuse_val == 0xA050C030 )
    return 0;
  if ( fuse_val == 0xA55AC33C )
    return 1;
  report_invalid(fuse_num, is_raw);
  return -1;
}

uint32_t cfg_fuse_get_default(uint32_t fuse_val, uint32_t fuse_num, bool is_raw, uint32_t def)
{
  int result = cfg_fuse_get(fuse_val, fuse_num, is_raw);
  if (result == 1) return 1;
  if (result == 0) return 0;
  return def;
}

bool chipid_get_secure_mode()
{
  return cfg_fuse_get_default(rCFG_FUSE1_RAW, 1, true, 1);
}
```

Code 6.3: Reading of security mode in A12 and up.

regions are currently not taken automatically from SecureROM, but rather hardcoded inside the build scripts. Lastly, anything other than SecureROM, that is from the same codebase such as iBoot, also does not work at all.

6.3 IMG4 Fuzzing

As explained in Section 5.4, we have twelve instances per fuzzing cluster, each instance on its own core. Furthermore, we have two clusters, one running the target with more permissible bounds checks (OOB, see Section 5.5.1) and one without such modifications. Both clusters were left running for 1 week - 168 hours[4] - with all resources allocated to them. Each cluster ran on a machine with Ubuntu 18.04 using an `Intel i7-8700K CPU @ 3.70GHz` with `8 GB DDR4 RAM`.

While no bugs - neither already known or new ones - were uncovered during fuzzing, we can still draw some conclusions from this experiment. As seen later with coverage, allowing certain out-of-bounds accesses had no observable impact on fuzzing performance. We also saw that ASN.1 is a particularly hard format for a fuzzer to mutate, based on coverage data - presented in Section 6.6 - and manual analysis of generated test cases. This comes to no surprise, as its tag-length-value nature means small changes in either the tag or length field, can quickly make an input be invalid and rejected.

6.4 USB Fuzzing

The setup for USB fuzzing is identical to IMG4 fuzzing. The only difference is in the number and target of the fuzzing clusters. We have three fuzzing clusters, that all have the same target, so there is no difference between the clusters, unlike with IMG4 fuzzing. We ran three clusters in parallel,

[4] Or for Elon, 6.79 sols.

to ensure we would hit the checkm8 bug in a reasonable timeframe[5]. Additionally, this allows us to better generalize our fuzzing performance, as we can compare multiple clusters with each other.

6.4.1 checkm8 A13

One important result of the USB fuzzing, is that we managed to trigger the checkm8 bug. In particular, using FETA we managed to trigger the UAF exploited by checkm8. An example stacktrace generated when crashing due to the checkm8 bug can be seen in Code 5.9.

To ensure that we are not finding the bug because we tailored our starting inputs for the fuzzer too closely to checkm8, we started a fuzzing cluster that only had standard USB messages as inputs. In particular, no messages related to the DFU protocol were provided as inputs. While the fuzzing cluster was configured the same as before, so it had the same twelve instances with parameters as explained in Section 5.4, it ran on a machine running Arch Linux using an `AMD 7 Ryzen 2700X` with `32 GB DDR4 RAM`. We then measured the time it took each fuzzing instance to find a crashing input, called time-to-exposure (TTE). While the fuzzing instances exchange interesting inputs, they will not exchange crashes. Therefore, looking at the different instances gives us a good idea of the average TTE. The results of the TTE of the different instances are presented in Figure 6.1.

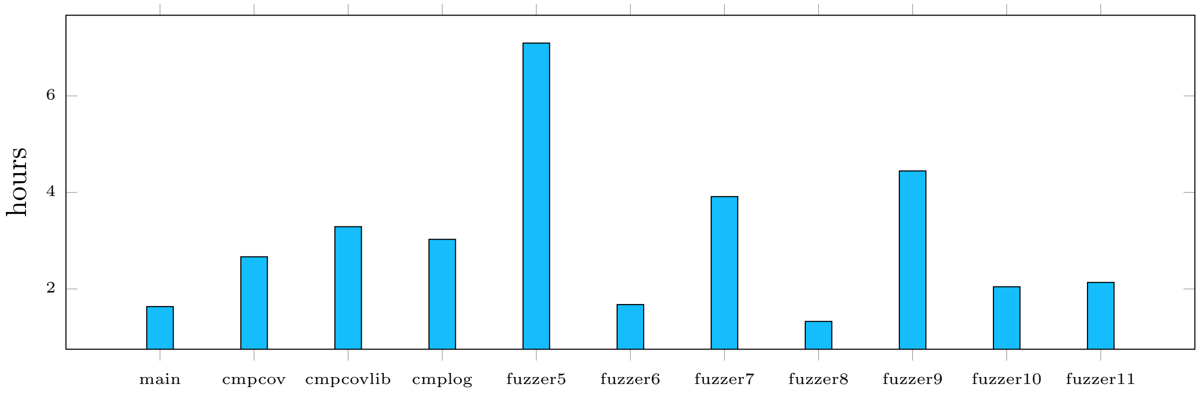


Figure 6.1: TTE for the checkm8 bug across all fuzzing instances.

As seen in the figure, TTE is very low across the board and one fuzzer managed to find the bug in less than 2 hours. Considering that the checkm8 bug is present on six generations of chips - from A8 till A13 -, it was quite surprising to find the bug so quickly. Furthermore, we have now been able to confirm that the main bug is still present on A13 chips. To our knowledge, this has not been reported anywhere yet and neither has the reason for why the full exploit chain does not work on A12 and A13. From our analysis, we presume this is due to the fact that the necessary heap feng shui setup is not possible anymore. Since there is no currently known way of causing allocations to stay allocated - so there is no memory leak -, the freed buffer will always be allocated at the same location and hence the UAF is useless.

Contrary to A12 and A13, A14 seems to actually fix the underlying bug. While we have not confirmed this with fuzzing, static analysis reveals that the pointer is now correctly set back to `NULL`.

[5] As seen in the next section, this was very much not needed, as the time-to-exposure of checkm8 was very low.

6.5 Improving Fuzzing Speed

To see the difference our speed improvements made, we ran three fuzzing clusters, fuzzing IMG4, with the following different configurations. One was running in standalone mode, one in persistent mode and one in persistent mode with the kernel module for snapshotting. Besides this, the clusters had the standard setup of twelve instances as described previously. However, each cluster ran on the same machine as for measuring the TTE described in Section 6.4.1. The average number of executions per second during the first four hours of fuzzing for the different instances and clusters is displayed in Figure 6.2. Clearly, the kernel module is the winner with respect to speed.

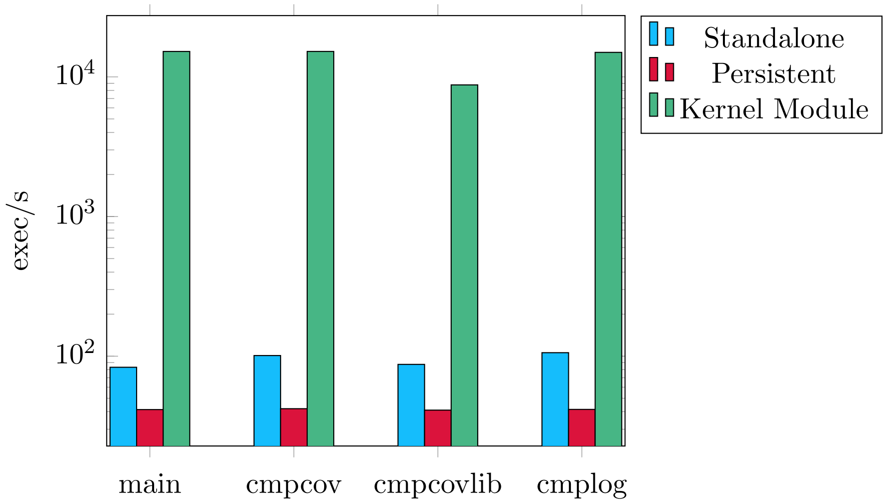


Figure 6.2: IMG4 fuzzing speed for different AFL++ modes.

However, an interesting observation stemming from said figure, is that persistent mode is slower than standalone mode. In standalone mode, AFL++ still uses something called the forkserver. In essence, AFL++ stops before executing main and forks itself. Every time the child exits and a fuzzing run ends, it forks again and creates a new child process. Since the kernel uses COW when forking, this is still quite efficient and hence it could explain the speed difference.

In contrast to the speed differences between the clusters, the speed differences between the instances are mostly negligible. Even when instrumenting every compare instruction not much speed is lost if any at all. While cmpcovlib certainly exhibits a dip in the number of executions per second, its instrumentation is also implemented in an external library and hence could indicate higher overhead.

Nevertheless, perf output shows that quite a lot of time is still spent not executing the binary, but rather in other areas, such as instrumentation. Another important point, is that the number of executions per second does not necessarily indicate the fuzzing speed. For example, if many fuzzing runs execute just a few basic blocks, the execution count will be much higher, compared to when many fuzzing runs execute a lot of basic blocks. However, since the inputs should be fairly consistent across fuzzing instances and clusters, the number of basic blocks executed per fuzzing run should be fairly consistent as well.

Contrary to IMG4 fuzzing, persistent mode yields a speedup of more than ten times when used with USB fuzzing. Again, we ran USB fuzzing with different configurations, only this time we could not test the kernel module as explained previously. Figure 6.3 presents the average number of executions during the first four hours of fuzzing for the different instances and clusters. A likely culprit for the slow operation of standalone mode is the need to create the additional thread for every fuzzing run. QEMU flushes all jitted pages whenever the first thread is created, which accounts for most of the time spent during fuzzing. In persistent mode, only the first fuzzing run creates the first thread and hence why it is not slowed down by this.

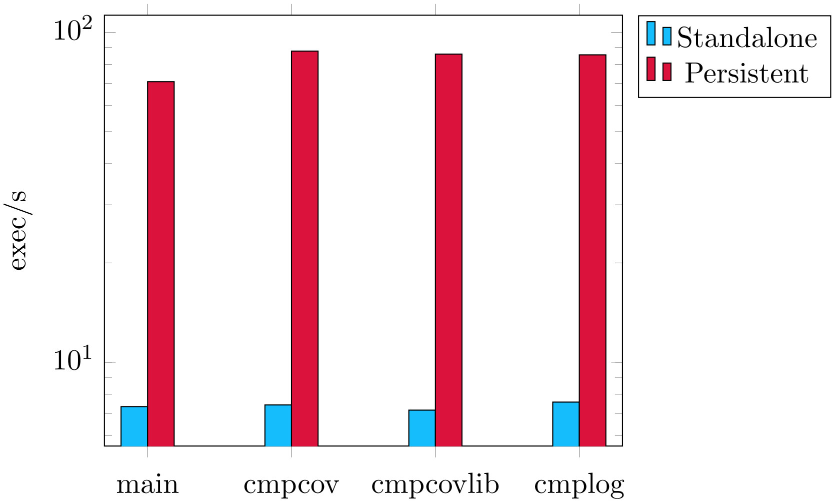


Figure 6.3: USB fuzzing speed for different AFL++ modes.

6.6 Coverage

Although coverage must not necessarily indicate how well the fuzzing performed, it certainly indicates how well *coverage-guided* fuzzing performed, since the idea is to achieve as much coverage as possible. To this end, we will go over the coverage obtained with fuzzing and comparing it between different fuzzing instances.

First, we have to introduce the key metrics which are relevant for our evaluation of achieved coverage. One cannot just look at the number of instructions hit at least once versus the number of total instructions as the single indication of success. Ideally we would want to reach full path coverage, something a percentage of instructions hit tells us very little about. Since this is difficult to measure - there can be infinite many paths with loops - we instead also look at the additional metrics introduced in Section 5.9. While this also tells us little about path coverage directly, it can help us understand which basic blocks are on many paths discovered by the fuzzer and hence give us an idea of path coverage.

Furthermore, we calculate the so called cyclomatic complexity (CC) - from [19] - for all functions. The basic idea is to calculate the number of possible basic paths in the control flow graph of a function. A function coverage table - such as the one seen in Table 6.5 - will also contain the number of basic blocks hit out of the total (BB Hit), the number of instructions hit of the total (Instr Hit) and the size of the function in bytes (Size). In addition, most figures and tables will be colored according to the legend seen in Table 6.2. The meaning of the percentage differs between tables and figures, tables use the percentage of instructions that are executed, while figures use the i-percentage for function entry blocks and the f-percentage for all other blocks. A special color is assigned to "ignored" basic blocks. A basic block is said to be "ignored", if it should be disregarded when viewing the CFG. This is due to the fact that it is in a code path that will - either due to patching or restriction of input - never be reached, regardless of how well the fuzzer works.

Description	Percentage	Color
Never Executed	0.0%	⬛
Covered	$(0.0\%, 0.5\%)$	🟥
	$[0.5\%, 2.0\%)$	🟥
	$[2.0\%, 10.0\%)$	🟥
	$[10.0\%, 25.0\%)$	🟪
	$[25.0\%, 60.0\%)$	🟪
	$[60.0\%, 100.0\%)$	🟦
	100.0%	🟦
Ignored	N/A	🟧

Table 6.2: Color of basic blocks in CFGs with coverage information.

In total - between both IMG4 and USB fuzzing - over 30% of instructions were executed at least once. This includes many instructions that were in completely unrelated parts - for example MMU code - and hence this percentage is not that useful on its own. Therefore, we will analyze the coverage in more detail in the following. For the full coverage information obtained while fuzzing, see Appendix D.

6.6.1 IMG4 Fuzzing

First we focus on the coverage obtained during IMG4 fuzzing. Table 6.5 shows the main functions involved in IMG4 parsing and the obtained coverage between normal and OOB fuzzing - as explained in Section 5.5.1. As can be seen, there is little difference in coverage and the normal version actually has slightly higher coverage. At first this is unexpected, since we specifically crafted the OOB version to get better coverage. A likely explanation is that, although previously rejected inputs would now pass a bounds check, parts of the input are now missing and hence get immediately rejected.

Function	Executions	BB Hit	Instr Hit	Size	CC
image4_load	18'956	59 / 154	264 / 581	2368 B	94
image4_validate_property_callback_interposer	463	33 / 72	172 / 323	1468 B	43
_image4_get_partial	31'623	16 / 18	77 / 85	348 B	12
Img4DecodePerformTrustEvaluatation	6'274	31 / 35	146 / 158	636 B	29

(a) Coverage for normal IMG4 fuzzing.

Function	Executions	BB Hit	Instr Hit	Size	CC
image4_load	44'985	58 / 154	263 / 581	2368 B	94
image4_validate_property_callback_interposer	1'067	33 / 72	172 / 323	1468 B	43
_image4_get_partial	75'144	16 / 18	77 / 85	348 B	12
Img4DecodePerformTrustEvaluatation	14'724	31 / 35	146 / 158	636 B	29

(a) Coverage for OOB IMG4 fuzzing.

Table 6.5: Coverage of IMG4 functions.

Because OOB has no significant benefit to coverage, we only look at coverage for normal fuzzing from here on. While _image4_get_partial and Img4DecodePerformTrustEvaluation are both covered well, the same cannot be said for the other two functions. To further our understanding of the other functions, we take a look at their CFGs, presented in Figure 6.5 and Figure 6.4.

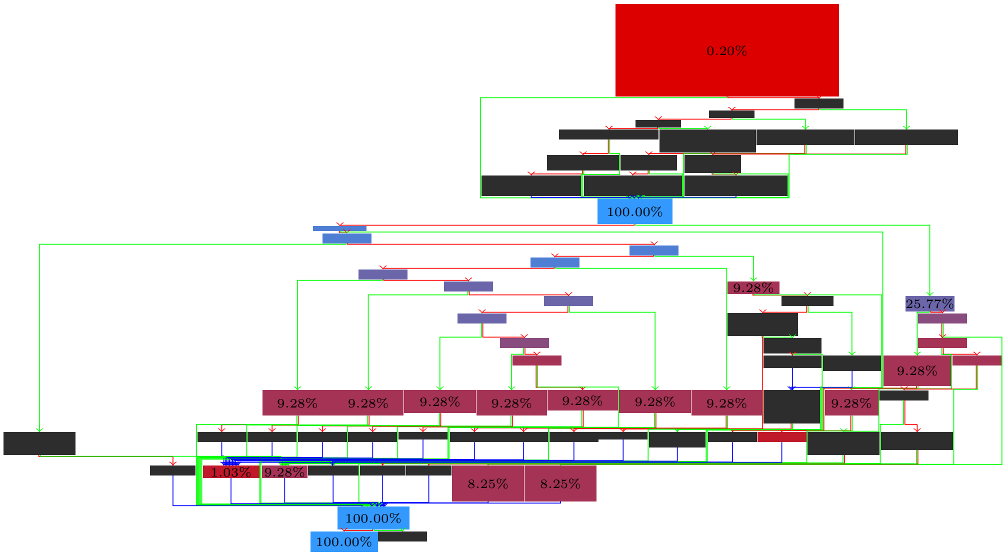


Figure 6.4: Annotated CFG of image4_validate_property_callback_interposer.

image4_load is the main function called from our fuzzer for parsing and validating images. Img4DecodePerformTrustEvaluation performs signature verification and matching of properties against the environment. Therefore, we can see that only 0.2% of inputs have a successful signature. This is not further surprising, as creating a fake signature is incredibly unlikely. In fact, it is very likely impossible to modify anything in a validly signed input and have it still be accepted. At least our fuzzing was unable to perform such a modification and our static analysis corroborates this. Since there is not much processing after signature verification, this is also not necessarily an indication that the fuzzing was not as successful here. It is still surprising however, that only 8.8%[6] of interesting inputs - reaching the signature verification - are not from the pool of initial seed inputs. This confirms our suspicion that ASN.1 and IMG4 in particular, is a difficult format for a fuzzer to mutate without creating ill-formed inputs.

However, in Figure 6.4 we can see that less than 0.3% of inputs actually reach `image4_validate_property_callback_interposer`, which is definitely of great interest. This function is responsible for matching manifest properties against the environment and hence the perfect candidate for any logic bugs - should they exist. Since some of the checks also copy bytes - for example the digest - it might even contain other kinds of vulnerabilities. It makes sense that only a very small number of inputs reach here, because almost everything must match the specification for it to not be rejected earlier. Nevertheless, as seen in Figure 6.4, most paths are covered equally. This is probably just due to the fact that this function is called for every property though. It can also be seen, that most of the basic blocks here are never executed. Hence, coverage in the IMG4 case could still be improved by quite a lot.

6.6.2 USB Fuzzing

Next we focus on coverage achieved during USB fuzzing. Table 6.7 shows the coverage information of the most important functions involved in USB handling in SecureROM. With most basic blocks being covered for all functions, it seems that coverage achieved here is better than in the IMG4 case. However, to really evaluate coverage, we also have to look at their CFGs. Here, we focus on the main function handling USB requests, `usb_core_handle_usb_control_receive`. Its annotated CFG is presented in Figure 6.6.

Function	Executions	BB Hit	Instr Hit	Size	CC
`usb_core_handle_usb_control_receive`	525’020	93 / 98	344 / 357	1504 B	62
`usb_core_event_handler`	16’334	22 / 25	69 / 75	312 B	13
`usb_dfu_handle_interface_request`	45’178	28 / 28	107 / 107	460 B	12
`usb_dfu_data_received`	9’686	7 / 9	42 / 54	220 B	3

Table 6.7: Coverage of USB Functions.

At a first glance, the coverage might seem a lot worse here than in the IMG4 case. However, we can see that coverage is distributed pretty evenly across all possible paths. In other words, the fuzzer seems to be able to hit most message types pretty evenly. Additionally, some USB messages are processed further by other functions and hence provide additional interesting paths to the fuzzer. Therefore, the paths in this function that correspond to such messages will have higher coverage. All in all, we believe coverage achieved here is already quite good and we believe that there are no significantly new insights to be gained from further improving coverage.

Another interesting thing to note, is that the function exit block does not have a 100% coverage as in previous examples. This is the case, since the USB handling might panic - as previously explained - and hence the return might not be reached. As we can see from the coverage however, this only happens a fraction of the time and thus is negligible, as we assumed.

[6] This was calculated via some IDA scripting.

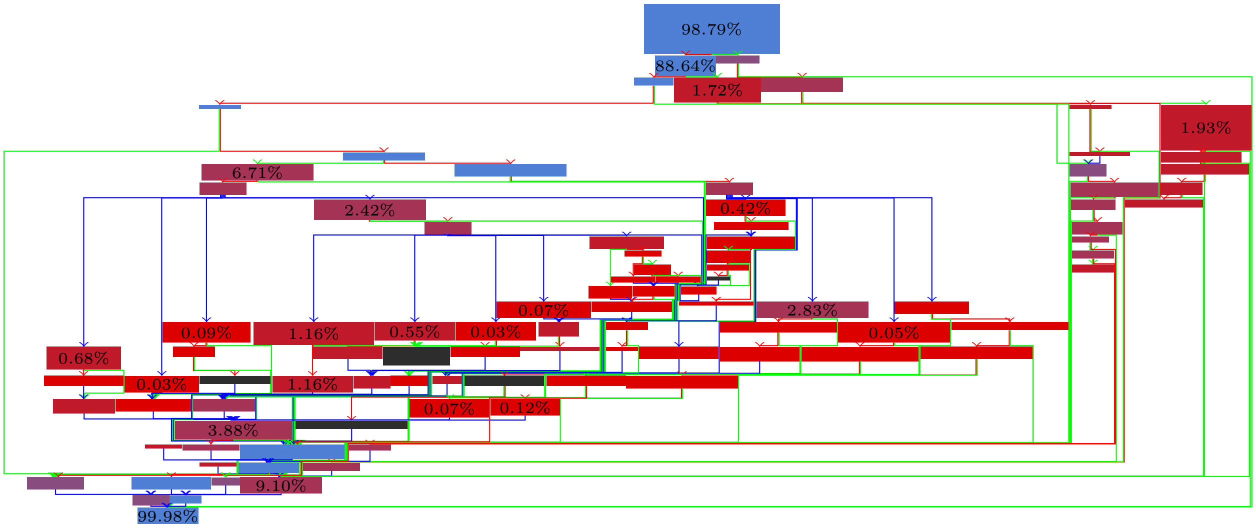


Figure 6.6: Annotated CFG of `usb_core_handle_usb_control_receive`.

6.6.3 Combined Coverage

Finally, we present the results for combined coverage across both IMG4 and USB fuzzing. In total, we reached a coverage of 31.76% across all instructions found inside SecureROM. This includes instructions that had no chance of being executed, since they occur in irrelevant parts[7] or in code that was patched out[8]. If we only include instructions that can theoretically be reached from the methods we call of SecureROM, then the coverage jumps to 46.37%. We can get an even better idea of coverage, by only including basic blocks that are not related to calling `panic` or have not been patched out. Almost all blocks calling `panic` are unreachable[9], our fuzzing only managed to trigger one specific case as mentioned before. With this, we get a coverage of 75.95%. Still, this includes code that can never be reached, for example due to certain global state. Taking that into consideration, this suggests fuzzing worked quite well and covered most instructions. It is unrealistic to ever hit 100% - especially in large code bases, there is almost always unreachable code - and 75.95% is quite decent for such a large binary as SecureROM. It could nevertheless still be improved upon in the future, but likely not without making the fuzzer work nicer with ASN.1.

In addition, Table 6.9 shows some additional interesting functions, besides the previously seen. These are all covered in both USB and IMG4 fuzzing and hence the combined coverage is looked at here. As seen in the table, the DER and certificate parsing functions are covered quite well. However, `verify_payload_properties` - which is one of the most complex parsing function - is only executed 241 times. While almost all basic blocks are hit, a greater number of executions would not hurt.

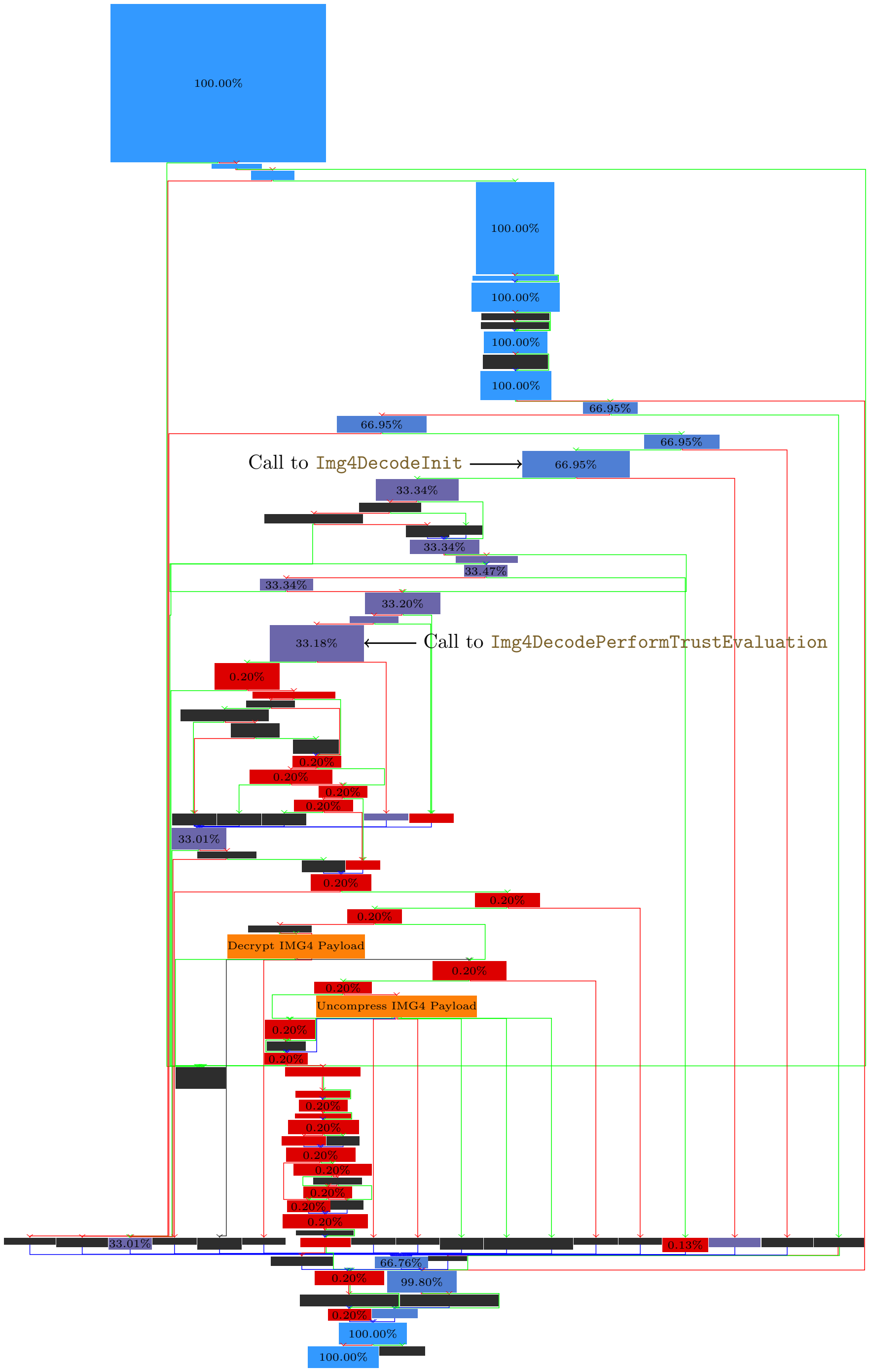
[7] For example, any startup, MMU, filesystem or NVME code.

[8] For example, the fast path of `bzero`.

[9] Often they are just sanity checks, such as making sure `malloc` returns a pointer.

Function	**Executions**	**BB Hit**	**Instr Hit**	**Size**	**CC**
`DERDecodeItemPartialBuffer`	1’923’891	24 / 26	76 / 81	324 B	16
`DERParseSequenceContent`	409’970	32 / 35	127 / 133	532 B	19
`_DERParseInteger64`	225’237	14 / 14	32 / 32	128 B	8
`parse_chain`	15’571	24 / 25	132 / 133	532 B	20
`verify_parse_chain`	30’785	22 / 23	96 / 97	388 B	18
`verify_payload_properties`	241	23 / 34	115 / 135	564 B	23

Table 6.9: Coverage of USB and IMG4 functions.



Call to `Img4DecodeInit`

Call to `Img4DecodePerformTrustEvaluation`

Figure 6.5: Annotated CFG of `image4_load`.

Chapter 7

Discussion

While we achieved some great results with emmutaler, there are still limitations to it. In this chapter, we discuss its limitations and give some ideas for future possibilities. In particular, we discuss the possibilities for future fuzzing speed improvements.

7.1 Limitations

One big limitation of emmutaler is that it is not as general as we would have liked to have. Trying to fuzz a different SecureROM from the one we currently look at involves quite a bit of manual labour, such as finding and annotating important symbols. The problem when fuzzing different SecureROM versions is the IO regions for fuses, devices and other things. Not only do these change frequently, they also do not change in a consistent way. Furthermore, sometimes the way they are accessed also changes in a major way.

Another limitation of emmutaler is that it currently only works for SecureROM. Neither iBoot nor the ROM of SEP are supported, even though they share a lot of code with SecureROM. Furthermore, we are limited to fuzzing very specific parts of SecureROM that need to be reverse engineered and understood in depth beforehand. Anything hardware related, such as the MMU configuration, can currently not be fuzzed. This is unfortunate, as some bugs have been found in for example the memory controller [28].

As mentioned before, QEMU can crash during the fuzzing process. We did not quantify what impact it has on fuzzing speed or other factors, but it is likely not insignificant. As such, another limitation is currently QEMU itself and its - as of writing this - inexplicable crashes.

Lastly, we had to create a custom heap implementation for the fuzzer to find the checkm8 bug. However, it is likely that this bug would have also not been found by AFL++ in a more standard Linux binary, if it required the same amount of heap-feng-shui to be triggered. As such, this is not much of a limitation of emmutaler, but more so of AFL++. Additionally, FETA while built specifically for this, is much more general. Not only can it detect any UAF bugs immediately, it also detects any heap overflows. One limitation of FETA however, is the amount of memory it needs. It can allocate up to one page of memory too much, even for larger chunks.

7.2 Future Possibilities

There are many possibilities for future work, especially to address many of the aforementioned limitations. A great starting point could be to not only allow fuzzing of different SecureROM versions more easily, but also expanding to iBoot and SEPROM. While iBoot might not be as valuable - it is not the root of the chain of trust - it also has many more attack vectors, such as a more complicated USB protocol. SEPROM is interesting as well, but its threat model makes it a less interesting target. Previously disclosed vulnerabilities needed an exploit in SecureROM first.

Another area of future work, could be the ability to fuzz more parts of SecureROM. A good candidate is likely the fuzzing of NVME namespaces and filesystems itself. This is the last big area from our threat model, that we have not fuzzed yet. It could also be interesting to think about how one could fuzz the hardware itself, such as a memory controller. In a similar vain, more targeted ASN.1 fuzzing could prove to be fruitful. As it stands currently, the fuzzer seems to have a hard time mutating ASN.1 without it being rejected immediately. An alternative might be to fuzz functions that occur later in the IMG4 verification process directly, such as `image4_validate_property_callback_interposer`.

Although FETA already works quite well, it could be improved to also detect heap underflows reliably. Additionally, it is currently quite inefficient with regards to memory. Future work could look at improving its memory efficiency. However, FETA also raises an interesting question. If we can find a UAF that has been present in six generations of chips by switching to a specialized allocator, what other bugs are we missing? For example, there might be another allocator design, such that memory leaks could be detected more easily and hence we could fuzz for one in SecureROM. As previously mentioned, a memory leak would likely allow the checkm8 bug to be exploitable on A12 and A13. Future work should not only limit itself to allocators though. Time-of-check to time-of-use (TOCTOU) bugs - or race conditions in general - are very difficult for even humans to properly trigger and identify. A conventional fuzzer has it even worse and might just classify racy inputs as variable coverage and hence discard them. One could look into designing a drop-in replacement threading library, that makes it significantly easier for fuzzers to find and report such bugs. While this might seem far-fetched, TOCTOU bugs require a specific access pattern to be triggered, which is similar to checkm8, that also needs a specific heap access pattern to be triggered.

Lastly, investigating why QEMU crashes mysteriously could be of interest as well. While our cursory investigation has been fruitless, a more involved and focused analysis could provide more insight. Not only would this make USB fuzzing more reliable, inputs would also experience less variability in their coverage, as QEMU would not die seemingly at random.

7.3 Fuzzing Speed Improvements

One particularly interesting area for future work is the improvement of fuzzing speed. While IMG4 fuzzing is quite fast already, there are still quite a few limitations encountered. For example, the kernel module is still not fully working correctly and it does not support threading at all. Having threading support would hopefully help speed up the USB fuzzing process by a lot.

Furthermore, even with a fully working kernel module, quite a lot of time was still spent not running SecureROM itself, but other things, such as instrumentation. Future work could explore static or dynamic rewriting - for example as was done in [5] - to further increase possible fuzzing speeds. Additionally, if QEMU crashes can be resolved, we need to snapshot a lot less memory[1] and hence could gain a significant speed up there.

[1] To avoid mysterious QEMU crashes, we currently have to snapshot a large chunk of libc as explained before.

Chapter 8

Related Work

To our knowledge, we are the first to publicly apply coverage-guided fuzzing - a subcategory of greybox fuzzing - to smartphone boot loaders. However, there have been previous works, applying fuzzing to smartphone boot loaders. At opposite ends of the spectrum, there are previous works for both whitebox and blackbox fuzzing, both of which are presented below. Lastly, we also present another work related to binary rewriting, which is a large part of emmutaler.

8.1 Blackbox USB Fuzzing

Blackbox fuzzing generates random inputs with the hope of triggering bugs through observable error behaviour, such as crashes. While this might seem counter intuitive, it takes a lot less time to setup and can be applied to any program.

[14] uses blackbox fuzzing to fuzz USB handling inside SecureROM[1]. In essence, they send all possible USB requests and observe any crashes that might occur. Although it seems very primitive compared to our much more guided fuzzing approach, it is very easy to setup and lead to vulnerabilities being found as well. Especially for finding bugs in the interaction with actual hardware, taking a more blackbox approach might be interesting in the future.

Furthermore, looking at how easy it is for our fuzzer to trigger the checkm8 bug, it could probably have been found by a blackbox fuzzer as well. Nevertheless, such a blackbox fuzzer would still need a way to get feedback on inputs. Since we had to use FETA to notice the checkm8 bug, the same would have to be done for a blackbox fuzzer as well.

8.2 Whitebox Fuzzing of Android Boot Loaders

On the other end of the spectrum we have whitebox fuzzing. Rather than blindly testing random inputs, whitebox fuzzing relies on computationally intensive analysis, such as symbolic execution, to find bugs. This trades fast analysis, for a more thorough one. It can sometimes find deeply hidden bugs, that would not be uncovered by even greybox fuzzing.

[22] applies whitebox fuzzing - in particular taint analysis - to Android boot loaders. Their tool can semi-automatically identify taint sources and sinks, with which a taint analysis engine can then be used to find bugs. It successfully identifies multiple vulnerabilities in a range of boot loaders. However, it would have probably not been successful in identifying the checkm8 bug. A UAF is difficult to detect with the type of taint analysis done by the referenced paper. Nevertheless,

[1] At the time it was known as BootROM.

it would be interesting to apply it to other parts of SecureROM, such as NVME namespaces or IMG4 parsing. In particular, fuzzing IMG4 could be enhanced with taint analysis - such as [11], since ASN.1 seems to be difficult to fuzz effectively. Alternatively, we could use taint analysis in combination with the idea to fuzz later functions. Running the fuzzer directly on later functions will likely result in lots of unreachable crashes, since later functions often make certain assumptions about the input. By using taint analysis together with the fuzzer, we could filter out which inputs are actually possible for the later functions.

8.3 Binary Rewriting on ARM

Our binary patcher and symbol extraction tools are an integral part of emmutaler. [5] takes a much different approach in patching binaries and enabling them to be fuzzed. In essence, it statically symbolizes ARM binaries and hence allows them to be rewritten almost arbitrarily. While it could be used as an alternative for QEMU to enable fuzzing for SecureROM, it could also serve as a replacement for our binary patcher. If the binary is fully symbolized, we would not have to work around ARM pointer limitations or needing to patch within a single instruction only. Symbolization could also help in identifying functions between SecureROM versions, as often the high level layout stays the same.

Chapter 9

Conclusion

In summary, we presented emmutaler, a set of tools to enable fuzzing and debugging of SecureROM. Thanks to a significant reverse engineering effort and many custom tools, we can effectively fuzz IMG4 parsing and USB handling of SecureROM using emmutaler. Having identified a number of challenges in bringing state of the art fuzzing to boot loaders, we discussed how emmutaler addresses them. Our experiments with AFL++ showed that not only can we achieve great coverage, we can even find a security vulnerability, checkm8. Thanks to our custom allocator FETA, even use-after-free bugs - needing otherwise complicated heap feng shui - can be detected. Additionally, by finding checkm8 we have shown that modern coverage-guided fuzzing is an important tool which could potentially have led to checkm8 being discovered and fixed earlier. Furthermore, we presented some mitigations present in SecureROM - uncovered during static analysis - and why they might have been implemented. We also found a number of issues with AFL++ - such as bugs and fuzzing speed - and fixed them or demonstrated improvements. Finally, we note that emmutaler could be used as a foundation for enabling fuzzing of a larger range of Apple's boot loaders, such as iBoot or SEPROM. So, to ensure reproducibility and encourage further research, we will open-source emmutaler and provide open access to the results of our static analysis at github.com/galli-leo/emmutaler.

Bibliography

[1] Afl++ documentation. URL: https://aflplus.plus/docs/ (visited on 09/02/2021).

[2] L. Aronsky. Accelerating iOS on QEMU with hardware virtualization (KVM). 2020. URL: https://alephsecurity.com/2020/07/19/xnu-qemu-kvm/ (visited on 04/17/2021).

[3] axi0mX. EPIC JAILBREAK: Introducing checkm8 (read "checkmate"), a permanent unpatchable bootrom exploit for hundreds of millions of iOS devices. Most generations of iPhones and iPads are vulnerable: from iPhone 4S (A5 chip) to iPhone 8 and iPhone X (A11 chip). Sept. 27, 2018. URL: https://twitter.com/axi0mX/status/1177542201670168576. https://twitter.com/axi0mX.

[4] B. Azad. Examining Pointer Authentication on the iPhone XS. 2019. URL: https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html (visited on 07/27/2021).

[5] L. D. Bartolomeo. *ArmWrestling: efficient binary rewriting for ARM*. Master's thesis, ETH Zürich, 2021.

[6] K. Conger. Apple confirms ios kernel code left unencrypted intentionally. URL: https://techcrunch.com/2016/06/22/apple-unencrypted-kernel/ (visited on 09/09/2021).

[7] A. Fioraldi, D. C. D'Elia, and L. Querzoni. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*, pages 23–30, 2020. DOI: 10.1109/SecDev45635.2020.00019.

[8] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. Afl++ : combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[9] Fuzzing binary-only targets - afl++. URL: https://github.com/AFLplusplus/AFLplusplus#fuzzing-binary-only-targets (visited on 09/02/2021).

[10] gaasedelen. Lighthouse - A Code Coverage Explorer for Reverse Engineers. URL: https://github.com/gaasedelen/lighthouse (visited on 07/17/2021).

[11] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen. {Greyone}: data flow sensitive fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2577–2594, 2020.

[12] Hekapoo. Apple ROM collection. URL: https://securerom.fun (visited on 07/13/2021).

[13] Introduction to apple platform security. 2021. URL: https://support.apple.com/en-gb/guide/security/seccd5016d31/web (visited on 09/02/2021).

[14] Joshua Hill. Shattered dreams: adventures in bootrom land. HITBSecConf2013. 2013. URL: `http://conference.hitb.org/hitbsecconf2013kul/materials/D2T1%20-%20Joshua%20'p0sixninja'%20Hill%20-%20SHAttered%20Dreams.pdf` (visited on 12/14/2020).

[15] Learn the architecture: AArch64 Exception model. URL: `https://developer.arm.com/documentation/102412/0100/Privilege-and-Exception-levels` (visited on 08/23/2021).

[16] J. Levin. **OS Internals Volume II: Kernel Mode*. Technologeeks.com, 2020.

[17] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan. PAC it up: towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA. USENIX Association, Aug. 2019. ISBN: 978-1-939133-06-9. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand`.

[18] H. Martin. Asahilinux wiki – sw:boot. 2021. URL: `https://github.com/AsahiLinux/docs/wiki/SW%3ABoot` (visited on 09/15/2021).

[19] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. DOI: `10.1109/TSE.1976.233837`.

[20] Oid repository - home. Orange SA. URL: `http://www.oid-info.com/#oid` (visited on 07/13/2020).

[21] QEMU Project Developers. QEMU TCG Plugins. URL: `https://qemu.readthedocs.io/en/latest/devel/tcg-plugins.html` (visited on 07/17/2021).

[22] 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 *26th USENIX Security Symposium (USENIX Security 17)*, pages 781–798, 2017.

[23] J. Scahill and J. Begley. The CIA Campaign to Steal Apple's Secrets. URL: `https://theintercept.com/2015/03/10/ispy-cia-campaign-steal-apples-secrets/` (visited on 09/09/2021).

[24] Siguza. APRR. 2018. URL: `https://siguza.github.io/APRR/` (visited on 07/15/2021).

[25] A. Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007:11–20, 2007.

[26] Unknown. Abstract syntax notation one (asn.1). URL: `https://portal.etsi.org/CTI/ApproachToTesting/SpecLanguages/ASN.1.htm` (visited on 09/09/2021).

[27] Z. Whittaker. Lenovo used shady 'rootkit' tactic to quietly reinstall unwanted software. 2015. URL: `https://www.zdnet.com/article/lenovo-rootkit-ensured-its-software-could-not-be-deleted/` (visited on 09/02/2021).

[28] H. Xu. Attack Secure Boot of SEP. Paper presented at MOSEC 2020, Shanghai, China. 2020.

[29] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2313–2328, Dallas, Texas, USA. Association for Computing Machinery, 2017. ISBN: 9781450349468. DOI: `10.1145/3133956.3134046`. URL: `https://doi.org/10.1145/3133956.3134046`.

[30] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security*, Aug. 2020. URL: `https://comsec.ethz.ch/wp-content/files/parmesan_sec20.pdf`.

Appendix A

SecureROM Memory Map

A.1 Metadata Inside SecureROM

Below is a visualization of the metadata section found inside SecureROM, that provides information about the layout of different section.

0x200	Banner (e.g. SecureROM for t8030si, Copyright 2007-2018, Apple Inc.)	
0x240	Style (e.g. ROMRELEASE)	
0x280	Tag (e.g. iBoot-4479.0.0.100.4)	
0x300	Pointer to Banner	Pointer to Style
0x310	Pointer to Tag	Text Start
0x320	Text End	Text Size
0x330	Data RO Start	Data Start
0x340	Data End	BSS Start
0x350	BSS End	Stacks Start
0x360	Stacks Size	EL1 Stack
0x370	EL0 Stack	Page Tables Start
0x380	Page Tables Size	Heap Guard

A.2 A13 SecureROM Memory Map

The full memory map of SecureROM on the A13 chip is presented below for reference.

0xffffffffffffffff	0x240000000	– unmapped –
0x23fffffff	0x230000000	Memory Mapped IO Registers
0x22fffffff	0x19C030000	– unmapped –
0x19C02ffff	0x19C028000	[heap]
0x19c027fff	0x19c020000	– unmapped –
0x19c01ffff	0x19c01c000	[stack]
0x19c01bfff	0x19c00d100	.bss
0x19c00d0ff	0x19c00c000	.data
0x19c00bfff	0x19c000000	Page Tables
0x19bffffff	0x100030000	– unmapped –
0x10002ffff	0x100000000	.text

Appendix B

IMG4 Format Specification

B.1 SET format

A lot of the IMG4 values are ASN.1 `SET`, corresponding to a kind of dictionary, where every item of the `SET` has a key, encoded in the tag of the item, and a corresponding value, encoded in the contents of the item. The key - a four character string - is encoded, by converting every character to its ASCII value, then concatenating them to form a 32-bit number. For example, the key `MANP`, is first converted to `0x4d, 0x41, 0x4e, 0x50` and then to `0x4d << 24 | 0x41 << 16 | 0x4e << 8 | 0x50 = 1296125520`. The inverse of this operations is labeled `long_to_bytes` in the specification below.

B.2 IMG4 Format Full Specification

Below is the full ASN.1 specification of the IMG4 format.

```
IMG4Format DEFINITIONS ::= BEGIN

    ObjectID ::= IA5String (SIZE (4))

    Keybag ::= SEQUENCE {
        type INTEGER,
        iv OCTET STRING,
        key OCTET STRING
    }

    Keybags ::= SEQUENCE OF Keybag

    ManifestBytes ::= SEQUENCE {
        objectID ObjectID,
        value OCTET STRING
    }

    ManifestInt ::= SEQUENCE {
        objectID ObjectID,
        value INTEGER
    }

    ManifestBool ::= SEQUENCE {
        objectID ObjectID,
        value BOOLEAN
```

```
}

PayloadManifestInfo ::= SEQUENCE {
    objectID ObjectID,
    contents SET {
        -- long_to_bytes(1145525076) == "DGST"
        digest [[PRIVATE] 1145525076] EXPLICIT ManifestBytes OPTIONAL
        -- long_to_bytes(1162560857) == "EKEY"
        enableKeys [[PRIVATE] 1162560857] EXPLICIT ManifestBool OPTIONAL
        -- long_to_bytes(1162891855) == "EPRO"
        effectiveProduction [[PRIVATE] 1162891855] EXPLICIT ManifestBool OPTIONAL
        -- long_to_bytes(1163085123) == "ESEC"
        effectiveSecurityMode [[PRIVATE] 1163085123] EXPLICIT ManifestBool OPTIONAL
    }
}

MANP ::= SEQUENCE {
    -- long_to_bytes(1112425288) == "BNCH"
    bootNonceHash [[PRIVATE] 1112425288] EXPLICIT ManifestBytes OPTIONAL
    -- long_to_bytes(1112494660) == "BORD"
    boardID [[PRIVATE] 1112494660] EXPLICIT ManifestInt OPTIONAL
    -- long_to_bytes(1128616015) == "CEPO"
    certificateEpoch [[PRIVATE] 1128616015] EXPLICIT ManifestInt OPTIONAL
    -- long_to_bytes(1128810832) == "CHIP"
    chipID [[PRIVATE] 1128810832] EXPLICIT ManifestInt OPTIONAL
    -- long_to_bytes(1129337423) == "CPRO"
    certificateProduction [[PRIVATE] 1129337423] EXPLICIT ManifestBool OPTIONAL
    -- long_to_bytes(1129530691) == "CSEC"
    certificateSecurityMode [[PRIVATE] 1129530691] EXPLICIT ManifestBool OPTIONAL
    -- long_to_bytes(1162037572) == "ECID"
    uniqueChipID [[PRIVATE] 1162037572] EXPLICIT ManifestInt OPTIONAL
    -- long_to_bytes(1396985677) == "SDOM"
    securityDomain [[PRIVATE] 1396985677] EXPLICIT ManifestInt OPTIONAL
    -- long_to_bytes(1885565552) == "pcrp"
    pcrp [[PRIVATE] 1885565552] EXPLICIT ManifestBytes OPTIONAL
    -- long_to_bytes(1936617326) == "snon"
    sepNonce [[PRIVATE] 1936617326] EXPLICIT ManifestBytes OPTIONAL
    -- long_to_bytes(1936881262) == "srvn"
    srvn [[PRIVATE] 1936881262] EXPLICIT ManifestBytes OPTIONAL
    -- long_to_bytes(1095585357) == "AMNM"
    allowMixNMatch [[PRIVATE] 1095585357] EXPLICIT ManifestBool OPTIONAL
}

IM4P ::= SEQUENCE {
    objectID ObjectID,
    payloadID ObjectID,
    version IA5String,
    contents OCTET STRING,
    keybags OCTET STRING (
        CONTAINING Keybags
        ENCODED BY {
            joint-iso-itu-t asn1(1)
            ber-derived(2)
            distinguished-encoding(1)
        }
    )
}

MANB ::= SEQUENCE {
        objectID ObjectID,
        contents SET {
            -- long_to_bytes(1296125520) == "MANP"
            properties [[PRIVATE] 1296125520] EXPLICIT MANP
            -- long_to_bytes(1768056163) == "ibec"
            ibec [[PRIVATE] 1768056163] EXPLICIT PayloadManifestInfo
        }

    }

    IM4M ::= SEQUENCE {
        objectID ObjectID,
        version INTEGER,
        contents SET {
            -- long_to_bytes(1296125506) == "MANB"
            body [[PRIVATE] 1296125506] EXPLICIT MANB
        },
        signature OCTET STRING,
        -- Certificate is the standard ASN.1 X.509 Certificate
        certChain SEQUENCE OF Certificate
    }

    IMG4 ::= SEQUENCE {
        objectID IA5String,
        payload IM4P
        manifest [0] EXPLICIT IM4M OPTIONAL
    }

END
```

Appendix C

Longer Before and After Comparison

To give a better idea of how big of a difference static analysis made, we showcase a larger function below. On the left is the function as pulled directly from IDA without any modifications. On the right is the function after a lot of reverse engineering and cleanup.

```
__int64 __fastcall sub_100013DEC(unsigned int
↪ a1, __int64 a2, _QWORD *a3, __int64
↪ (**a4)(void), __int64 a5)
{
  __int64 result; // x0
  __int64 (*v11)(void); // x9
  __int64 (__fastcall *v12)(__int64, __int64);
↪   // x8
  __int64 v13; // x8
  int v14; // w26
  char v15; // [xsp+Fh] [xbp-81h] BYREF
  __int64 v16; // [xsp+10h] [xbp-80h] BYREF
  __int64 v17; // [xsp+18h] [xbp-78h] BYREF
  __int128 v18[3]; // [xsp+20h] [xbp-70h]
↪   BYREF

  v16 = 0LL;
  v17 = 0LL;
  memset(v18, 0, sizeof(v18));
  result = 6LL;
  if ( !a2 )
    return result;
  if ( !a3 )
    return result;
  result = 6LL;
  if ( !a4 || !*a3 )
    return result;
  if ( !*a4 )
    return 6LL;
  if ( !a4[3] )
    return 6LL;
  if ( !a4[1] )
    return 6LL;
  if ( !a4[2] )
    return 6LL;
  v11 = a4[4];
  if ( !v11 || !*(_QWORD *)(a2 + 24) )
    return 6LL;
```

```
__int64 Img4DecodePerformTrustEvaluatation(
        uint32_t actualType,
        img4 *img4,
        img4_property_callbacks
      ↪   *validate_props,
        Img4DecodeImplementation *hasher,
        image4_wrapper_context *wrapper_ctxt)
{
  __int64 result; // x0
  hash_info *hasher_info; // x9
  DERByte *manifest_der_data; // x0
  __int64 (__fastcall
↪   *get_integrated_manifest_hash)(img4 *,
↪   image4_wrapper_context *); // x8
  _BYTE *integrated_hash; // x8
  unsigned __int64 hash_size; // x3
  int should_verify_properties; // w26
  bool payload_exists; // [xsp+Fh] [xbp-81h]
↪   BYREF
  DERSize leafPubKeyLen; // [xsp+10h]
↪   [xbp-80h] BYREF
  DERByte *leafPubKey; // [xsp+18h] [xbp-78h]
↪   BYREF
  DERDecodedInfoFind found_manifest_item[2];
↪   // [xsp+20h] [xbp-70h] BYREF
  const DERItem pubKey; // 0:x0.16
  const DERItem toVerify; // 0:x4.16

  leafPubKeyLen = 0LL;
  leafPubKey = 0LL;
  memset(found_manifest_item, 0,
↪   sizeof(found_manifest_item));
  result = 6LL;
  if ( !img4 )
    return result;
  if ( !validate_props )
    return result;
  result = 6LL;
```

```
 if ( *(_QWORD *)v11 > 0x30uLL )
   return 7LL;
 result = (*a4)();
 if ( (_DWORD)result )
   return result;
 *(_BYTE *)(a2 + 1) = 1;
 v12 = (__int64 (__fastcall *)(__int64,
↪  __int64))a3[1];
 if ( v12 && (v13 = v12(a2, a5)) != 0 &&
↪  !(unsigned int)sub_100011A6C(a2 + 328,
↪  v13, *(_QWORD *)a4[4]) )
 {
   v14 = 0;
 }
 else
 {
   result = ((__int64 (__fastcall *)(_QWORD,
  ↪  _QWORD, __int64 *, __int64 *, __int64,
  ↪  __int64, __int64 (**)(void),
  ↪  __int64))a4[1])(
              *(_QWORD *)(a2 + 296),
              *(_QWORD *)(a2 + 304),
              &v17,
              &v16,
              a2 + 312,
              a2 + 320,
              a4,
              a5);
   if ( (_DWORD)result )
     return result;
   if ( *(_QWORD *)a4[4] > 0x30uLL )
     return 7LL;
   result = ((__int64 (__fastcall *)(_QWORD,
  ↪  _QWORD, __int64))*a4)(
              *(_QWORD *)(a2 + 264),
              *(_QWORD *)(a2 + 272),
              a2 + 376);
   if ( (_DWORD)result )
     return result;
   result = ((__int64 (__fastcall *)(__int64,
  ↪  __int64, _QWORD, _QWORD, __int64,
  ↪  _QWORD, __int64 (**)(void),
  ↪  __int64))a4[2])(
              v17,
              v16,
              *(_QWORD *)(a2 + 280),
              *(_QWORD *)(a2 + 288),
              a2 + 376,
              *(_QWORD *)a4[4],
              a4,
              a5);
   if ( (_DWORD)result )
     return result;
   v14 = 1;
 }
 result = sub_100013064(a2, a2 + 40, a2 +
↪  56);
 if ( (_DWORD)result )
   return result;
```

```
 if ( !hasher ||
↪  !validate_props->validate_property )
   return result;
 if ( !hasher->do_ccdigest )
   return 6LL;
 if ( !hasher->verify_payload_properties )
   return 6LL;
 if ( !hasher->parse_cert_chain )
   return 6LL;
 if ( !hasher->verify_pkcs1_sig )
   return 6LL;
 hasher_info = hasher->hasher_info;
 if ( !hasher_info )
   return 6LL;
 manifest_der_data = img4->manifest_der.data;
↪
 if ( !manifest_der_data )
   return 6LL;
 if ( hasher_info->hash_size > 0x30uLL )
   return 7LL;
 result = hasher->do_ccdigest(
            manifest_der_data,
            img4->manifest_der.length,
            img4->manifest_der_digest,
            hasher_info->hash_size,
            hasher);
 if ( result )
   return result;
 img4->did_digest = 1;
 get_integrated_manifest_hash =
↪  validate_props->get_integrated_manifest_hash;
↪
 if ( get_integrated_manifest_hash
   && (integrated_hash =
  ↪  get_integrated_manifest_hash(img4,
  ↪  wrapper_ctxt)) != 0LL
   && !memcmp(img4->manifest_der_digest,
  ↪  integrated_hash,
  ↪  hasher->hasher_info->hash_size) )
 {
   should_verify_properties = 0;
 }
 else
 {
   result = hasher->parse_cert_chain(

              ↪  img4->manifest_items.cert_chain.data,
              ↪

              ↪  img4->manifest_items.cert_chain.length,
              ↪
                &leafPubKey,
                &leafPubKeyLen,
                &img4->cert_manifest,
                &img4->cert_manifest.length,
                hasher,
                wrapper_ctxt);
   if ( result )
     return result;
  result = sub_100012F88(a2 + 40, a1 |
↪   0xE000000000000000LL,
↪   0x2000000000000011LL, v18);
  if ( (_DWORD)result )
    return result;
  *(_OWORD *)(a2 + 72) = *(__int128 *)((char
↪   *)&v18[1] + 8);
  if ( v14 )
  {
    result = ((__int64 (__fastcall *)(__int64,
  ↪   __int64))a4[3])(a2, a5);
    if ( (_DWORD)result )
      return result;
  }
  if ( *(_QWORD *)a4[4] > 0x30uLL )
    return 7LL;
  result = sub_100013144(a2, &v15);
  if ( (_DWORD)result )
    return result;
  if ( v15 )
  {
    result = ((__int64 (__fastcall *)(_QWORD,
  ↪   _QWORD, __int64, _QWORD, __int64
  ↪   (**)(void)))*a4)(
               *(_QWORD *)(a2 + 8),
               *(_QWORD *)(a2 + 16),
               a2 + 184,
               *(_QWORD *)a4[4],
               a4);
    if ( (_DWORD)result )
      return result;
    *(_BYTE *)a2 = 1;
  }
  result = sub_100013C14(a2 + 56, 0LL, *a3,
↪   a5);
  if ( !(_DWORD)result )
    return sub_100013C14(a2 + 72, 1LL, *a3,
  ↪   a5);
  return result;
}
```

```
    hash_size =
  ↪   hasher->hasher_info->hash_size;
    if ( hash_size > 0x30 )
      return 7LL;
    result = hasher->do_ccdigest(

             ↪   img4->manifest_items.contents.data,
             ↪

             ↪   img4->manifest_items.contents.length,
             ↪
               img4->manifest_item_digest,
               hash_size,
               hasher);
    if ( result )
      return result;
    pubKey.length = leafPubKeyLen;
    pubKey.data = leafPubKey;
    toVerify.length =
  ↪   hasher->hasher_info->hash_size;
    toVerify.data =
  ↪   img4->manifest_item_digest;
    result = hasher->verify_pkcs1_sig(pubKey,
  ↪   img4->manifest_items.signature,
  ↪   toVerify, hasher);
    if ( result )
      return result;
    should_verify_properties = 1;
  }
  result =
↪   DERImg4DecodeParseManifestProperties_0(img4,
↪   &img4->man_b, &img4->man_p);
  if ( result )
    return result;
  result = DERFindInSequence(
             &img4->man_b,
             (actualType |
           ↪   (ASN1_PRIVATE|ASN1_CONSTRUCTED)),
           ↪
             ASN1_CONSTRUCTED|ASN1_SET,
             found_manifest_item);
  if ( result )
    return result;
  img4->image_manifest =
↪   found_manifest_item[1].content;
  if ( should_verify_properties )
  {
    result =
  ↪   hasher->verify_payload_properties(img4,
  ↪   wrapper_ctxt);
    if ( result )
      return result;
  }
  if ( hasher->hasher_info->hash_size >
↪   0x30uLL )
    return 7LL;
  result = Img4DecodePayloadExists_0(img4,
↪   &payload_exists);
  if ( result )
```

```
    return result;
  if ( payload_exists )
  {
    result = hasher->do_ccdigest(
               img4->payload_der.data,
               img4->payload_der.length,
               img4->payload_der_hash,
               hasher->hasher_info->hash_size,
             ↪
               hasher);
    if ( result )
      return result;
    img4->verified_stuff = 1;
  }
  result =
↪   Img4DecodeEvaluateDictionaryProperties_0(&img4->man_p,
↪   0LL, validate_props->validate_property,
↪   wrapper_ctxt);
  if ( !result )
    return
  ↪   Img4DecodeEvaluateDictionaryProperties_0(
  ↪
             &img4->image_manifest,
             1LL,

           ↪   validate_props->validate_property,
           ↪
             wrapper_ctxt);
  return result;
}
```

Appendix D

Full Coverage Information

Below is a table of coverage across all covered function achieved during both IMG4 and USB fuzzing.

Function	Executions	BB Hit	Instr Hit	Size	CC
arm_read_cntpct	4'765'040	1 / 1	2 / 3	12 B	1
arm_clean_dcache	74	1 / 1	1 / 1	4 B	1
platform_irq	0	1 / 12	2 / 48	212 B	4
rGPIO_PINSTRAPS_VALID_CFG_0	102'047	1 / 1	4 / 4	24 B	1
_image4_get_partial	155'084	16 / 18	77 / 85	348 B	12
der_expect	719'343	3 / 3	16 / 16	64 B	2
der_restrict	294'371	4 / 5	17 / 19	76 B	2
der_expect_ia5string	294'371	10 / 12	52 / 57	228 B	6
image4_validate_property_callback_interpose	2'577	33 / 72	172 / 323	1468 B	43
get_integrated_manifest_hash	30'785	1 / 1	6 / 6	24 B	1
image4_load	93'915	59 / 154	264 / 581	2368 B	94
image4_load_copyobject	93'915	14 / 21	55 / 80	340 B	8
image4_verify_number_relation	1'205	8 / 10	37 / 40	160 B	6
image4_verify_boolean_relation	890	4 / 5	30 / 31	124 B	2
sub_10000658C	241	5 / 7	38 / 40	160 B	3
platform_cache_operation	74	5 / 28	19 / 75	300 B	17
platform_get_iboot_flags	8'132	1 / 1	17 / 17	68 B	1
platform_get_usb_product_id	9'889	1 / 1	9 / 9	36 B	1
platform_get_usb_product_string	9'889	1 / 1	3 / 3	12 B	1
platform_get_usb_serial_number_string	9'889	5 / 6	79 / 80	328 B	3
hash_calculate	8'169	4 / 6	35 / 59	240 B	2
platform_get_usb_more_other_string	9'926	13 / 14	109 / 110	444 B	7
platform_consume_nonce	8'169	3 / 5	23 / 28	112 B	1
report_no_boot_image	41'678	1 / 1	1 / 6	32 B	1
some_kind_of_report	51'063	1 / 1	1 / 6	32 B	1
sub_100006E10	37	1 / 1	3 / 3	20 B	1
platform_get_entropy	1'361	1 / 5	1 / 27	108 B	1
get_integrated_hash	93'915	1 / 1	4 / 4	24 B	1
platform_reset	37	1 / 2	4 / 5	20 B	2
platform_get_board_id	102'047	2 / 3	14 / 18	80 B	1
platform_get_nonce	8'169	3 / 4	20 / 22	96 B	2

Function	Executions	BB Hit	Instr Hit	Size	CC
platform_get_sep_nonce	8’133	1 / 1	1 / 1	4 B	1
platform_get_boot_manifest_hash	93’915	3 / 7	6 / 15	68 B	2
memzero_if_io_reg	93’915	4 / 10	8 / 21	100 B	3
platform_get_mix_n_match_prevention_status	93’915	1 / 1	4 / 4	24 B	1
cfg_fuse_get	428’374	4 / 4	12 / 12	56 B	1
sub_100008284	212’430	1 / 1	11 / 11	48 B	1
chipid_get_current_production_mode	8’336	1 / 1	6 / 6	32 B	1
cfg_fuse_get_default	428’374	1 / 1	14 / 14	56 B	1
chipid_get_raw_production_mode	102’047	1 / 1	6 / 6	32 B	1
chipid_get_secure_mode	110’383	1 / 1	6 / 6	32 B	1
chipid_get_security_domain	103’804	1 / 1	21 / 21	92 B	1
chipid_get_board_id	102’047	1 / 1	4 / 4	24 B	1
platform_get_security_epoch	102’047	1 / 1	4 / 4	24 B	1
chipid_get_chip_id	102’047	1 / 1	2 / 2	8 B	1
_chipid_get_chip_revision	8’132	1 / 1	6 / 6	32 B	1
chipid_get_ecid_id	102’047	1 / 1	5 / 5	28 B	1
nullsub_7	555	1 / 1	1 / 1	4 B	1
_panic	37	3 / 5	65 / 93	384 B	1
sub_100009198	8’169	4 / 6	22 / 33	132 B	1
security_protect_memory	8’169	7 / 9	28 / 34	148 B	6
security_allow_modes	27’097	2 / 4	8 / 18	80 B	3
security_set_untrusted	204	1 / 1	7 / 7	36 B	1
security_allow_memory	10’759	4 / 6	23 / 26	108 B	3
security_set_production_override	204	1 / 1	8 / 8	36 B	1
security_get_effective_production_status	204	1 / 1	11 / 11	44 B	1
security_set_boot_manifest_hash	204	2 / 3	12 / 14	60 B	1
security_set_mix_n_match_prevention_status	204	1 / 1	8 / 8	36 B	1
iteratively_return_to_el0	37	3 / 4	11 / 12	48 B	2
current_task	9’931’267	2 / 3	7 / 11	44 B	1
wait_queue_wake_one	9’808	2 / 4	15 / 20	80 B	3
list_remove_head	9’808	3 / 4	12 / 15	60 B	2
event_init	9’889	1 / 1	5 / 5	20 B	1
event_signal	9’808	3 / 5	18 / 21	84 B	3
sub_10000A160	9’886	7 / 10	22 / 27	108 B	6
event_wait	9’886	2 / 7	14 / 27	108 B	3
system_time	7’147’521	1 / 1	7 / 7	28 B	1
sub_10000AC64	9’926	3 / 3	19 / 19	88 B	2
usb_init_with_controller	9’926	6 / 8	25 / 30	120 B	5
usb_free	9’808	2 / 3	5 / 6	24 B	1
platform_init_usb	9’926	1 / 1	8 / 8	32 B	1
usb_quiesce	9’808	3 / 3	16 / 16	64 B	2
nullsub_9	2’094	1 / 1	1 / 1	4 B	1
nullsub_10	10’027	1 / 1	1 / 1	4 B	1
sub_10000ADB4	783	1 / 1	7 / 7	28 B	1
nullsub_11	19’697	1 / 1	1 / 1	4 B	1
nullsub_12	9’889	1 / 1	1 / 1	4 B	1

Function	Executions	BB Hit	Instr Hit	Size	CC
sub_10000ADD8	11'732	1 / 1	2 / 2	8 B	1
sub_10000ADE0	10'776	1 / 1	2 / 2	8 B	1
mib_get_string	16'338	2 / 3	21 / 22	88 B	1
image_load	93'915	13 / 23	61 / 78	336 B	12
platform_get_fuse_modes	8'132	1 / 1	13 / 13	52 B	1
sub_10000B300	9'931'268	1 / 1	2 / 2	8 B	1
timer_get_ticks_0	7'147'561	1 / 1	1 / 1	4 B	1
timer_ticks_to_usecs	7'147'504	1 / 1	11 / 11	44 B	1
ticks_per_usec	7'147'502	5 / 6	23 / 27	112 B	3
synopsys_otg_controller_init	3'308	1 / 1	1 / 3	12 B	1
usb_controller_register	19'734	4 / 4	17 / 17	76 B	2
usb_controller_init	9'889	4 / 6	18 / 26	108 B	1
usb_controller_start	9'889	2 / 3	9 / 13	56 B	1
sub_10000DC70	9'808	2 / 3	9 / 13	56 B	1
usb_controller_set_address	3'669	2 / 3	9 / 13	56 B	1
usb_controller_get_connection_speed	9'915	3 / 3	13 / 13	56 B	1
usb_controller_do_endpoint_io	73'958	2 / 3	9 / 13	56 B	1
usb_controller_stall_endpoint	525'567	2 / 3	9 / 13	56 B	1
sub_10000DD88	501	2 / 3	9 / 13	56 B	1
sub_10000DDC0	608	2 / 3	9 / 13	56 B	1
sub_10000DDF8	783	2 / 3	9 / 13	56 B	1
usb_controller_abort_endpoint	559'477	3 / 3	13 / 13	56 B	1
sub_10000DEA0	783	1 / 1	3 / 3	12 B	1
usb_core_send_zlp	24'007	2 / 3	11 / 16	64 B	1
alloc_ep0_device_io_request	73'958	3 / 4	28 / 32	132 B	2
usb_core_init	9'899	3 / 3	58 / 58	240 B	2
usb_alloc_string_descriptor	49'445	5 / 6	25 / 29	116 B	2
usb_create_string_descriptor	39'556	6 / 7	19 / 20	84 B	3
usb_core_start	9'889	19 / 29	141 / 194	780 B	13
usb_core_register_interface	9'889	3 / 3	9 / 9	40 B	2
usb_core_handle_usb_control_receive	627'812	93 / 98	344 / 357	1504 B	62
usb_core_event_handler	20'242	22 / 25	69 / 75	312 B	13
usb_core_complete_endpoint_io	73'958	3 / 3	14 / 14	56 B	2
usb_core_do_transfer	25'401	3 / 5	21 / 34	136 B	3
sub_10000ECEC	9'808	6 / 7	24 / 25	104 B	4
sub_10000ED54	9'808	13 / 14	36 / 38	156 B	7
getDFUImage	9'922	5 / 5	23 / 23	96 B	3
usb_dfu_init	9'889	3 / 3	56 / 56	228 B	2
usb_dfu_handle_interface_request	54'687	28 / 28	107 / 107	460 B	12
usb_dfu_data_received	11'518	7 / 9	42 / 54	220 B	3
handle_bus_reset	3'166	3 / 3	8 / 8	36 B	1
usb_dfu_exit	9'808	5 / 5	20 / 20	84 B	3
siphash_aligned	1'575'369	14 / 22	77 / 96	432 B	12
heap_alloc	64'643	8 / 10	59 / 64	256 B	4
uk_heap_fn	46'746	2 / 3	4 / 7	36 B	2
required_size	46'746	2 / 3	8 / 15	60 B	1

Function	Executions	BB Hit	Instr Hit	Size	CC
maybe_verify_block_padding	46’746	6 / 8	17 / 21	84 B	4
calloc	73’958	3 / 3	7 / 7	28 B	2
verify_block_checksum	603’033	2 / 3	21 / 22	92 B	1
heap_free	172’167	17 / 22	104 / 127	516 B	13
calculate_block_checksum	972’336	1 / 1	6 / 6	28 B	1
heap_memalign	2’002	1 / 18	1 / 166	664 B	8
verify_block_padding	138’683	8 / 10	27 / 29	116 B	5
free_list_add	185’429	3 / 8	26 / 41	164 B	3
heap_add_chunk_0	91’937	9 / 16	78 / 101	424 B	11
heap_add_chunk	91’937	2 / 2	5 / 5	20 B	1
heap_set_cookie	91’937	2 / 2	12 / 12	52 B	1
sub_1000108E8	46’746	4 / 5	14 / 15	60 B	3
round_size	232’175	4 / 5	15 / 19	76 B	3
free_list_remove	93’492	6 / 9	26 / 35	140 B	6
split_tail	46’746	6 / 7	70 / 74	296 B	3
malloc	46’746	1 / 1	2 / 2	8 B	1
free	58’456	1 / 1	1 / 1	4 B	1
memset_s	69’820	1 / 1	8 / 8	32 B	1
weird_memset	69’820	4 / 5	14 / 18	72 B	3
_do_printf	439’265	58 / 128	202 / 466	1864 B	72
sub_1000114BC	439’265	3 / 3	22 / 22	88 B	2
snprintf	439’232	2 / 3	14 / 18	72 B	1
sub_100011648	37	2 / 2	5 / 5	20 B	1
puts	185	4 / 4	16 / 16	64 B	2
_putchar	444	3 / 3	15 / 15	60 B	2
strlcat	24’397	2 / 2	3 / 3	12 B	1
strlcpy	16’278	2 / 3	3 / 10	40 B	1
memcpy	5’667’507	12 / 21	59 / 108	432 B	11
_bzero	29’383’810	1 / 4	8 / 30	120 B	3
_memset	342’034	9 / 9	42 / 42	168 B	6
memcmp	331’278	5 / 5	13 / 13	52 B	3
sub_100011ACC	24’397	8 / 8	35 / 35	140 B	5
sub_100011B58	16’278	6 / 7	21 / 26	104 B	3
strlen	1’332’108	3 / 3	6 / 6	24 B	2
sub_100011BD8	24’397	5 / 5	9 / 9	36 B	3
mib_find_node	142’556	17 / 20	54 / 60	240 B	9
_mib_find	142’556	4 / 6	18 / 25	100 B	2
mib_probably_retrieve_value	142’556	6 / 10	22 / 34	136 B	4
mib_get_size	142’556	2 / 3	7 / 11	44 B	1
mib_get_cached	3’245’842	5 / 6	26 / 30	120 B	4
mib_get_value	142’556	5 / 7	21 / 34	136 B	1
init_entropy_source	8’169	5 / 11	25 / 79	324 B	6
sub_1000122CC	8’169	1 / 1	12 / 12	48 B	1
random_get_bytes_internal	8’169	7 / 20	42 / 101	408 B	9
random_get_bytes	8’169	1 / 1	9 / 9	36 B	1
enter_critical_section	7’362’621	4 / 7	14 / 24	96 B	2

Function	Executions	BB Hit	Instr Hit	Size	CC
exit_critical_section	7’352’768	3 / 5	12 / 19	76 B	1
platform_get_current_production_mode	8’336	1 / 1	1 / 1	4 B	1
platform_get_raw_production_mode	102’047	1 / 1	1 / 1	4 B	1
platform_get_security_domain	103’804	1 / 1	1 / 1	4 B	1
platform_get_secure_mode	110’383	2 / 3	8 / 11	44 B	1
platform_get_hardware_epoch	93’915	1 / 1	1 / 1	4 B	1
platform_get_chip_id	102’047	1 / 1	1 / 1	4 B	1
platform_get_chip_revision	8’132	1 / 1	1 / 1	4 B	1
platform_get_ecid_id	102’047	1 / 1	1 / 1	4 B	1
j__platform_init_usb	9’926	1 / 1	1 / 1	4 B	1
platform_get_usb_vendor_id	9’889	1 / 1	2 / 2	8 B	1
platform_get_usb_manufacturer_string	9’889	1 / 1	3 / 3	12 B	1
platform_get_usb_device_version	9’889	1 / 1	2 / 2	8 B	1
DERImg4DecodeFindInSequence_0	1’446	7 / 9	32 / 34	136 B	4
DERImg4DecodeContentFindItemWithTag_0	1’446	3 / 3	18 / 18	72 B	2
DERImg4DecodeTagCompare_1	163’856	10 / 10	29 / 29	116 B	6
parse_main_img4_der	62’145	10 / 13	47 / 52	216 B	8
parse_payload_der	58’478	8 / 12	39 / 50	204 B	9
parse_manifest_der	58’060	11 / 11	39 / 39	160 B	9
parse_recovery_der	31’237	9 / 10	28 / 30	124 B	7
DERImg4DecodeProperty_0	6’118	11 / 12	59 / 61	248 B	10
DERFindInSequence	723	7 / 9	52 / 55	220 B	5
DERImg4DecodeParseManifestProperties_0	241	11 / 13	45 / 49	224 B	9
_Img4DecodePayloadExists_0	31’658	8 / 8	14 / 14	56 B	5
Img4DecodeGetPayload	204	6 / 10	24 / 28	112 B	7
Img4DecodeGetPayloadType	30’972	10 / 11	28 / 29	116 B	8
Img4DecodePayloadCompressionExists	204	6 / 8	10 / 14	56 B	5
Img4DecodeCopyPayloadDigest	241	12 / 18	51 / 70	280 B	11
_Img4DecodeManifestExists_0	30’843	4 / 4	10 / 10	40 B	3
Img4DecodeCopyManifestDigest	204	12 / 19	42 / 59	236 B	11
sub_100013820	1’205	6 / 9	31 / 35	140 B	6
Img4DecodeGetPropertyBoolean_0	890	4 / 5	29 / 31	124 B	3
sub_100013928	241	9 / 14	38 / 46	184 B	9
verify_payload_properties	241	23 / 34	115 / 135	564 B	23
Img4DecodeEvaluateDictionaryProperties_0	482	14 / 15	68 / 70	284 B	11
Img4DecodeInit	62’145	8 / 8	47 / 47	188 B	7
Img4DecodePerformTrustEvaluatation	30’807	31 / 35	146 / 158	636 B	29
verify_signature_for_reals	9’368	23 / 27	115 / 121	484 B	19
verify_pkcs1_sig	12’278	14 / 14	38 / 38	152 B	13
parse_chain	15’571	24 / 25	132 / 133	532 B	20
verify_chain_signatures	8’830	23 / 25	97 / 99	396 B	20
verify_parse_chain	30’785	22 / 23	96 / 97	388 B	18
parse_cert_chain	30’785	1 / 1	3 / 3	12 B	1
crack_chain_root	19’075	1 / 1	14 / 14	56 B	1
Img4DecodeComputeDigest	43’394	9 / 10	19 / 21	84 B	5
sub_100014898	43’372	2 / 2	10 / 10	40 B	1

Function	Executions	BB Hit	Instr Hit	Size	CC
crack_chain	19'075	14 / 15	56 / 57	228 B	10
parse_extensions	29'008	16 / 16	78 / 78	316 B	12
DERDecodeItem	95'315	1 / 1	2 / 2	8 B	1
DERDecodeItemPartialBuffer	1'923'891	24 / 26	76 / 81	324 B	16
DERDecodeItemPartialBuffer_0	719'343	22 / 22	69 / 69	276 B	15
_DERParseBitString_0_0	37'579	3 / 4	12 / 16	64 B	2
_DERParseBoolean_0	890	3 / 4	13 / 15	60 B	2
DERParseInteger	224'032	6 / 7	30 / 31	124 B	3
_DERParseInteger64	225'237	14 / 14	32 / 32	128 B	8
DERDecodeSeqInitAgain	35'022	6 / 7	38 / 39	160 B	3
DERDecodeSeqContentInit	2'410	1 / 1	7 / 7	28 B	1
DERDecodeSeqNext	1'758'476	6 / 7	37 / 38	152 B	3
DERParseSequence	187'246	6 / 7	42 / 43	176 B	3
DERParseSequenceContent	409'970	32 / 35	127 / 133	532 B	19
DEROidCompare	140'240	4 / 4	10 / 10	40 B	1
memcmp_0	50'280	2 / 2	11 / 11	44 B	1
_ccn_cmp_0	17'385'931	4 / 4	13 / 13	52 B	3
_ccn_mul	18'045'342	8 / 8	43 / 43	172 B	5
sub_10001ACEC	17'704	3 / 3	13 / 13	52 B	3
ccdigest	51'533	2 / 3	53 / 54	216 B	1
_ccdigest_init	51'536	1 / 1	18 / 18	72 B	1
_ccdigest_update	51'535	9 / 13	54 / 78	312 B	7
_cc_mux2p	26'516'946	1 / 1	13 / 13	52 B	1
sub_10001B520	8'935	4 / 6	53 / 55	220 B	2
sub_10001B5FC	8'935	13 / 29	77 / 194	776 B	14
_ccn_add_0	34'753'992	10 / 11	44 / 45	180 B	6
sub_10001B9B8	214'442	3 / 3	15 / 15	60 B	2
sub_10001B9F4	26'733'840	8 / 9	165 / 166	664 B	3
sub_10001BC8C	27'671	14 / 16	40 / 44	176 B	8
sub_10001BCB4	10'016	1 / 1	10 / 10	40 B	1
sub_10001BD64	5'480'700	1 / 1	10 / 10	40 B	1
_ccn_sub_0	26'920'458	10 / 11	44 / 45	180 B	6
_ccn_write_uint_size	9'368	1 / 1	8 / 8	32 B	1
cc_clear	69'822	1 / 1	6 / 6	24 B	1
sub_10001BE78	8'852	11 / 13	89 / 92	368 B	6
sub_10001BFE8	9'368	11 / 16	152 / 203	812 B	8
sub_10001C314	9'368	4 / 6	49 / 51	204 B	2
sub_10001C3E0	8'935	4 / 6	34 / 43	172 B	4
sub_10001C48C	9'368	12 / 14	86 / 90	360 B	8
_ccn_shift_right_1	300'968	11 / 16	43 / 76	304 B	9
sub_10001CB70	17'376'996	2 / 3	41 / 42	168 B	1
sub_10001CC18	51'535	12 / 12	162 / 162	648 B	7
sub_10001D4AC	9'368	1 / 1	8 / 8	32 B	1
sub_10001D4CC	150'484	5 / 6	137 / 138	552 B	2
sub_10001D6F4	141'632	1 / 1	26 / 26	104 B	1
sub_10001D760	99'219	7 / 8	1280 / 1281	5124 B	3

Function	**Executions**	**BB Hit**	**Instr Hit**	**Size**	**CC**
`get_hash_impl`	104’661	1 / 1	3 / 3	12 B	1
`_cc_muxp`	150’484	1 / 1	8 / 8	32 B	1
`_ccn_n_0`	223’377	4 / 4	11 / 11	44 B	3