TU WIEN Informatics

TECHNISCHE UNIVERSITÄT DARMSTADT

Intel Baseband Fuzzing and Security Analysis on iOS

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

IT-Sicherheit

by

Stefan Sterz, BSc
Registration Number 2428994

to the Department of Computer Science

at the TU Darmstadt

Advisor: Prof. Dr.-Ing. Matthias Hollick
Assistance: Dr.-Ing. Jiska Classen

at the TU Wien

Advisor: Prof. Dr.techn. Thomas Grechenig

Vienna, 22nd September, 2021

Stefan Sterz	Matthias Hollick

TU WIEN Informatics

TECHNISCHE UNIVERSITÄT DARMSTADT

Intel Baseband Fuzzing and Security Analysis on iOS

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Stefan Sterz, BSc
Registration Number 01426147

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr.techn. Thomas Grechenig

at the TU Darmstadt

Advisor: Prof. Dr.-Ing. Matthias Hollick
Assistance: Dr.-Ing. Jiska Classen

Vienna, 22nd September, 2021

Stefan Sterz	Thomas Grechenig

Double Degree IT Security between Technische Universität Darmstadt and Technische Universität Wien
www.tu-darmstadt.de ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Stefan Sterz, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. September 2021

Stefan Sterz

Acknowledgements

I would like to express my deepest gratitude to my family and my girlfriend, Larissa, for their support throughout my studies and to my friends for helping me reach this goal.

Special thanks to Prof. Dr.-Ing. Matthias Hollick, Dr.-Ing. Jiska Classen, Prof. Dr.techn. Thomas Grechenig, and Clemens Hlauschek MSc, for their advice and guidance.

Furthermore, I want to thank Larissa Rackl, Magdalena Steinböck, and Simon Hayden for spellchecking and proofreading my thesis.

Abstract

In the past three decades, mobile phones have become a pervasive presence in our lives. The Baseband (BB) processor, the chip enabling cellular connectivity, has evolved as well. If this processor's firmware is insecure, an attacker could gain access to a zero-click remote code execution that, combined with a local privilege escalation, could grant complete control of any given smartphone. The existence of such a vulnerability is very likely because most of the firmware has existed for many years, and much of it has been written before many security measures were widely established.

This thesis analyzes the protocol used by the main Central Processing Unit (CPU) or Application Processor (AP) to communicate with the BB processor called Apple Remote Invocation (ARI). The BB and its firmware are then fuzzed via this protocol in four different ways. Further, memory exploitation mitigations and the complexity growth between five *Intel* BBs generations released over four years are described. These efforts culminate in the discovery of several crashes in the BB and a kernel-level vulnerability within iOS.

Kurzfassung

In den letzten drei Jahrzehnten wurden Mobiltelefone ein immer größerer Teil unseres Alltags. Der Baseband-Prozessor, der Chip, der Kommunikation über Mobilfunk ermöglicht, hat sich ebenfalls stark weiterentwickelt. Wenn die Firmware dieses Chips unsicher ist, ist es einem Angreifer möglich, in Kombination mit einer lokalen Rechteausweitung und ohne Nutzerinteraktion, die volle Kontrolle über ein Smartphone zu erreichen. Dass eine solche Schwachstelle existiert, ist sehr wahrscheinlich, weil der Großteil des Codes der Baseband-Firmware schon vor vielen Jahren programmiert worden ist, bevor viele Sicherheitsmaßnahmen weit verbreitet waren.

In dieser Arbeit wird das Protokoll zwischen *Intel*-Baseband-Prozessoren und dem Hauptprozessor unter *iOS* namens Apple Remote Invocation (ARI) untersucht. Mittels vier verschiedener Fuzzing-Ansätze dieses Protokolls wird die Firmware des Baseband-Prozessors untersucht. Außerdem werden die Abschwächungen für Sicherheitslücken und das Komplexitätswachstum in fünf Baseband-Generationen, die über einen Zeitraum von vier Jahren veröffentlicht wurden, zusammengefasst. All diese Ansätze führen zu der Entdeckung vieler Abstürze des *Intel*-Baseband-Prozessors und einer Sicherheitslücke innerhalb des *iOS*-Kernels.

Contents

CHAPTER 1

Introduction

This chapter will quickly motivate the topic of this thesis in Section 1.1. Then, a list of specific contributions is provided by Section 1.2. Finally, the rest of this thesis is outlined in Section 1.3.

1.1 Motivation

Over the last two decades, mobile phones have become ever more present in our daily lives. With their growing popularity and usage, cellular networks had to evolve and improve as well. Nowadays, phones support several standards such as Global System for Mobile Communications (GSM), Universal Mobile Telecommunications System (UMTS), Long-Term Evolution (LTE), and, more recently, 5G. To support these complex standards efficiently, a Real-Time Operating System (RTOS) is required. Thus, modern smartphones contain at least two processors. The Application Processor (AP) is the device's main processor and handles most of the interactions with the user, the main Operating System (OS), and other interfaces. The Baseband (BB) processor, which runs the RTOS, is solely responsible for cellular communication.

Due to the proprietary nature of most commercially available BBs' firmware, it is not possible to perform a code audit as an independent security researcher. Furthermore, since the purpose of a mobile phone is cellular communication, users rarely turn this functionality off. Since cellular standards are convoluted and ever-evolving, severe vulnerabilities are likely to arise. Especially parsers for complex protocols such as GSM or LTE have been affected by memory corruption flaws and similar security issues [100, 67, 52]. Due to the widespread usage of smartphones, such a vulnerability might have a severe impact.

Intel is one of the most prominent BB manufacturers.[1] Their *XMM* modem series has been featured most notably in *Apple*'s *iPhone* smartphones [95]. Since the *iPhone* line of mobile phones usually exhibits some of the most current technologies, *Intel*'s modems are amongst the most exciting security analysis targets.

1.2 Contributions

In this thesis, an analysis of several firmware generations running on *Intel* modems will be provided. More specifically, their implementations for four generations of *iPhones* and one for the *Apple Watch* will be investigated. Contributions in this thesis can be generally split into two categories. The first focuses on statically reverse-engineering parts of the *Intel* BB. This step is necessary to understand how the BB relates to the AP and handles certain tasks. Specific contributions of this thesis include:

- A more detailed overview of the *Apple* cellular stack running on the AP and its relation to the BB.
- An overview of processor architectures employed in the *Intel* BB stack over five generations. The RTOS used by the BB and a short look at the Digital Signal Processor (DSP) architecture used by certain BB generations.
- An analysis of the complexity of the BB's firmware. Two approaches are used: using disassemblers to detect the number of functions and reconstructing a code base skeleton.
- An outline of memory exploitation mitigations present in the five BB generations and how they changed over time.
- An overview of diagnostic tools on *iOS*, their features and a look at the proprietary log format produced by them.
- Finally, further insights were uncovered about the management protocol between the AP and BB called Apple Remote Invocation (ARI).

Secondly, the insights gained through reverse engineering were used to implement several approaches to fuzzing (parts of) the BB. Two approaches to fuzzing a part of the BB in an emulator were implemented first. Then, two approaches of fuzzing the communication between *iOS* and the BB were implemented on a physical device. More specifically, this thesis contributes:

- A general emulation setup for the function responsible for parsing ARI messages in the BB, as well as a custom fuzzer using the setup to analyze the parser. Further, the emulated parser was also fuzzed using American Fuzzy Lop Plus Plus (AFL++).

[1] In 2019, *Apple* acquired a majority of *Intel's* smartphone modem division [7].

- The implementation of two in-process fuzzers for sending fuzzed ARI messages to the BB.
- A kernel-level vulnerability within *iOS* that has since been reported to, acknowledged and fixed by *Apple* in *iOS* 14.6.

Additionally, an evaluation of the implemented fuzzers is provided. This thesis also contributes several smaller scripts that can help carry out similar analysis tasks in the future. Thus, it provides a starting point for analyzing new BB generations.

1.3 Outline

The next two chapters of this thesis provide a foundation that help to understand the following chapters. In Chapter 2, background information about several techniques involved in this thesis is provided. Followed by Chapter 3, which presents previous work on the topic of BB security research and fuzzing.

Chapter 4 focuses on static reverse-engineering tasks carried out throughout the thesis and details the insights into the BB gained this way. Then, Chapter 5 uses these insights to implement emulation-based and in-process fuzzing approaches to analyze the BB dynamically. These fuzzing approaches are then evaluated in Chapter 6.

Finally, Chapter 7 provides a discussion of the results of the thesis. It also outlines additional topics that can be used for future work on *Intel* BBs. Chapter 8 concludes the thesis.

CHAPTER 2

Background

In this chapter, a summary of several analysis techniques and general background information about Basebands (BBs) is provided. First, in Section 2.1, an overview of a general BB architecture is given. Then, Section 2.2 details how the communication between the BB and the Application Processor (AP) is typically handled. Section 2.3 introduces several concepts within the realm of firmware emulation. Further, Section 2.4 and Section 2.5 detail static and dynamic analysis techniques used in this thesis. Finally, Section 2.6 describes the attacker model considered in the following chapters.

2.1 Baseband Architecture

In general, a BB runs on a separate processor with a Real-Time Operating System (RTOS) [100, 19, 46, 52]. Figure 2.1 shows the general architecture of a modern cellular stack. Universal Mobile Telecommunications System (UMTS) or Long-Term Evolution (LTE) stacks are run on top of the RTOS to handle such complex cellular standards. Connecting to the cellular network is achieved by the BB via its physical interface (PHY).

Typically, there are two approaches to issue commands to a given BB. First, a BB might be controlled via a serial interface, such as Peripheral Component Interconnect Express (PCIe) [84]. The protocol depends on the BB itself and the Operating System (OS) running on the AP. The second approach to interact with a BB is to use shared memory. Note that these approaches are not mutually exclusive, meaning that they might be used in conjunction. These two methods are solely the two most common methods, and different vendors implement others.

Cellular capabilities that a BB supports are based on standards mainly published by the 3rd Generation Partnership Project (3GPP). These standards are issued in so-called *Releases*. As of writing, there are about 23 releases, with two additional ones currently under development (releases 17 and 18) [1]. The latest finalized or “frozen” major version

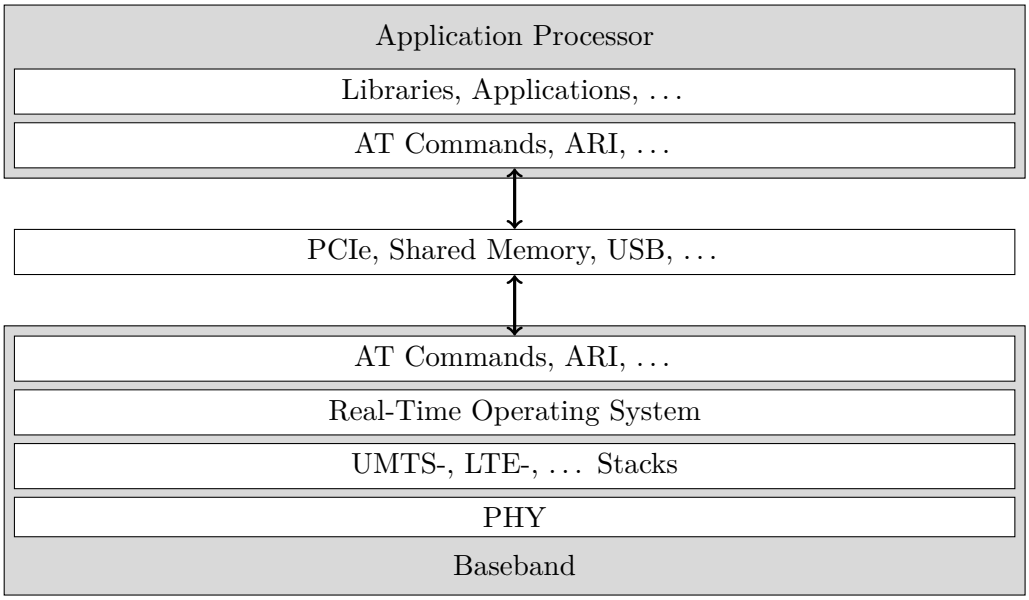

Figure 2.1: A generalized cellular stack.

is release 16. A release can contain multiple standards, and a given cellular protocol like LTE is defined across multiple releases. It is also possible that a particular release might be withdrawn at a later date. For example, the latest *Intel* LTE modem called XMM 7660 is based on release 14 [56]. The following timeline shows how the development of the modem correlates with the publication of the release:

17.09.2014 Work on release 14 was started by 3GPP [1].

09.06.2017 Release 14 was "frozen" [1].

16.11.2017 *Intel* planned to release the XMM 7660 modem in 2019 [54].

Q3 of 2019 The *Intel* XMM 7660 modem was officially released. Variants of the *iPhone* 11 use the XMM 7660 modem. It was released on the 20$^{\text{th}}$ of September 2019 [6].

2.2 Application Processor to Baseband Communication

Several protocols exist to facilitate the serial interface between the BB and the AP. In this section, a short introduction to two of these protocols will be given. The first one is AT commands, which are standardized and, thus, publicly documented. The second one is Apple Remote Invocation (ARI), is a proprietary protocol used by devices made by *Apple* that utilize *Intel* BBs.

2.2.1 AT Commands

AT commands were first introduced by Dennis Hayes in the 1980s [90]. Since then, AT commands have been part of several European Telecommunications Standards Institute (ETSI) and, by extension, 3GPP standards. For example, standard commands for the Short Message Service (SMS) [32], the Global System for Mobile Communications (GSM) [27], and LTE [28] exist. However, manufacturers tend to extend the AT command set with custom commands (e.g., creating a dump of the BB's internal memory) [97].

According to the ETSI [27, 28] standards, a given command typically starts with "AT," followed by a list of basic commands separated by a semicolon. Each command can also have a set of parameters. An AT command is terminated by a carriage return. Once a command has been issued, the BB responds with information responses containing the issued commands' results. A response is terminated by either `OK` or `ERROR` if an error has occurred with verbose responses enabled. Alternatively, if numeric responses are enabled, `0` or `4` will be returned, respectively. The standard also explicitly mentions that line feed and carriage return characters must be included in the responses.

2.2.2 Apple Remote Invocation

ARI is a proprietary management protocol that is used between *Intel* BB and mobile devices made by *Apple*. There is no official public documentation of this protocol. However, Kröll et al. [59] were able to reverse-engineer it. A summary of the parts relevant to this thesis is given in Section 4.8.1.

2.3 Baseband Firmware Emulation

This section will give an overview of several emulation techniques relevant to this thesis. While emulation of the BB's firmware is a central part of this work, detailing all emulation approaches introduced over the years would go beyond the scope of this thesis. Thus, only a selection is presented. A more detailed survey on emulation techniques and tools was created by Fasano et al. [35].

Due to the current BB chips' proprietary and enclosed nature, emulating their firmware allows more accessible and faster analysis than physically examining them. This process is called "re-hosting" because the firmware is moved to an emulated host processor. However, accurately emulating such a system can be rather challenging as well. For one, the involved software stack is complex, and documentation is sparse. Further, it is necessary to understand wireless and serial interfaces and shared memory between the BB and AP to emulate all aspects that might influence the state of the BB.

An alternative is partial emulation, which makes it possible to take a specific part of a given firmware and re-host this part [67, 68] in particular. Instead of figuring out how to emulate all preconditions for the firmware to even boot up in the emulator, it is possible to emulate only the section that will be analyzed. To make partial emulation work, it is typically necessary to modify functions that rely on the hardware to succeed.

One approach to emulate a given system either fully or partially is to use the *Quick Emulator (QEMU)* [15]. It offers fast and efficient processor emulation. Furthermore, it supports many processor architectures, even relatively obscure ones. However, emulating binaries that are not formatted in a way that QEMU understands can be difficult. Several projects have arisen over the years to support this use case. For example, *LuaQEMU* extends QEMU with a just-in-time compiler for the *Lua* language. It exposes many internal programming interfaces to this compiler, allowing it to swiftly adapt QEMU's behavior. Thus, it is possible to quickly emulate a given binary without having to modify it.

Another such project is *Unicorn* [80], which allows adding hooks. Hooks are callback functions that get executed if the emulation encounters a given condition. For example, a "code hook" can detect if a given address is reached during the emulation and invoke the callback. The function then has access to the emulator state. This method allows for exact control of the emulation. For instance, a hook can be used to skip a code section that *Unicorn* cannot easily emulate itself. Such situations may include accessing shared memory or making use of the BB's transceiver.

2.4 Software Analysis

The methods utilized in this thesis can broadly be categorized into static and dynamic analysis methods. Static analysis approaches focus on the binaries that make up the firmware themselves without executing them. On the other hand, dynamic approaches analyze software while it is being executed, either in an emulator or on the BB itself.

2.4.1 Static Analysis

Since most commercially available BB processors and their firmware are proprietary, it is impossible to analyze their firmware's source code directly. However, it is typically necessary to understand the basic structure of the BB and details such as memory management to analyze security vulnerabilities. This section will discuss several tools to aid with this task.

Disassembler and Decompilers

Disassemblers and decompilers [26] are standard tools used to reverse-engineer software. A disassembler translates the raw binary data back into assembly instructions. It needs to know the processor architecture on which the binary is supposed to run to work accurately. It is typically also necessary to distinguish between parts of a file intended to be executed and other data. Fortunately, many tools automate identifying these parameters [4, 53, 3, 99].

A decompiler [26] tries to reconstruct the source code of a program. Of course, much of the information typically lost in the compilation process cannot be restored, such as variable names or function definitions. Symbol tables provide a way to preserve some of

that information in a compiled binary. This thesis will use the terms "symbol tables," "debug symbols," or only "symbols" interchangeably. However, firmware vendors typically remove this information from a binary for production builds. Such a binary is also referred to as a "stripped binary."

Thus, disassembling and decompiling a stripped firmware typically misses a significant amount of function signatures. For instance, previous work by Friebertshauser et al. [37] and Pang et al. [83] has shown that state-of-the-art tools can still miss around 25 % of function signatures depending on the exact settings. Worse, functions that are identified are often false positives. Therefore, static analysis of a stripped firmware can be unpredictable, and findings achieved this way might not be entirely factual.

Tools such as *IDA Pro* [53], *Binary Ninja* [99], or *Ghidra* [3] combine disassemblers and decompilers. They also typically contain functionality to aid with detecting the processor architecture a given binary is supposed to run on and much more. This thesis will utilize *IDA Pro*, *Binary Ninja*, and *Ghidra* for static analysis. Furthermore, with *IDA Pro*, the *Thumbs Up* plug-in [20] is used to improve function detection. *Thumbs Up* uses basic machine learning techniques in addition to *IDA Pro's* linear sweep analysis. Machine learning can improve function detection because the function prologues and epilogues differ between compilers. By learning the prologues and epilogues as patterns, more function can be detected accurately. This approach can provide dramatic improvements over *IDA Pro's* default recursive analysis.

JTool 2

While the main focus of this thesis is on *Intel's* BB implementation, it is helpful, and at times necessary, to understand how *iOS* interacts with it. Some binaries and dynamic libraries can be easily extracted from a given *iOS* device itself. However, in the case of the *iOS* kernel extracting it from an *iOS* update can be useful. The kernel on *iOS* is delivered as a kernel cache. This kernel cache contains the kernel itself in addition to all of its extensions in a compressed format [96].

JTool 2 [61] can decompress this format to import it into *IDA Pro* or *Ghidra*. In general, this tool parses and analyzes the Mach-O binary format. Mach-O is the native format for executables on *macOS* [9] and is also used on *iOS*. Another valuable feature of *JTool 2* is adding symbols to a kernel panic report produced by *iOS*. Of course, this is conditioned on the corresponding kernel containing the necessary symbols.

2.4.2 Dynamic Analysis

Dynamic analysis is at the heart of this thesis. Fuzzing and emulation are both techniques that are commonly categorized as or used to facilitate dynamic analysis techniques. This section will focus on some tools that are used in addition to fuzzing and emulation.

Frida

FЯIDA [81] is a framework that allows for the dynamic analysis of executables. It injects a *JavaScript* engine into a running process and then offers a simple Application Programming Interface (API). A script can use this API to call functions, read and write memory regions and registers, add hooks, and much more. All this functionality makes FЯIDA a very versatile tool. In this thesis, FЯIDA will collect samples of ARI messages, fuzz the BB from the AP side, and try to reproduce crashes encountered during fuzzing. These crashes might stem from the FЯIDA-based in-process fuzzer or the emulation-based fuzzer.

Diagnostic Tools

There are several diagnostic tools available on *iOS* that can be used to interact with the BB. Since there is little public documentation available for these tools, reverse-engineering them was also part of this thesis. They are described in more detail in Section 4.7.

2.5 Fuzzing

The original idea behind fuzzing or fuzz testing [72] is relatively simple. Random input data is provided to the program to test whether a program conforms to expectations. This input is also referred to as a fuzz case. The program is then observed to see whether it crashes or exhibits other unexpected behavior. Unlike formal verification or testing through manually defined test cases, fuzzing can be carried out much more efficiently.

Over the years, many approaches have been created to improve upon this basic approach. Some rely on gaining information at runtime by instrumenting a binary. Others rely on insights into the format of the input provided to a program. Furthermore, the way a program is executed and provided with input has evolved too. In the following, a couple of approaches will be outlined that are relevant to this thesis.

2.5.1 Coverage-guided Fuzzing

Coverage-guided fuzzing tries to improve fuzzing results by checking which parts of a program are executed or covered if a given input is provided [24, 103, 36]. Through maximizing the coverage during fuzz testing, a program is explored more deeply, and more complex states are reached.

A program is typically instrumented with custom code to collect coverage data. This code can then collect different information about how a program responds to a given input. One of the earliest approaches to coverage-guided fuzzing used basic blocks as coverage information [24]. Basic blocks are sequences of instructions that do not contain any branching points or jumps. Another option is to use “edges.” Edges are tuples consisting of two basic blocks that are separated by a branch or a jump. Fuzzers like

American Fuzzy Lop (AFL) [103] and American Fuzzy Lop Plus Plus (AFL++) [36] favor this approach.

Coverage-guided approaches are sometimes also called evolutionary fuzzers [24, 103, 36]. The name stems from the way coverage-guided fuzzers produce new inputs. The fuzzer maintains a list of inputs. It will then modify or "mutate" an entry from the list and provide it as input to the program. The "mutated" fuzz case will only be retained if it, in turn, generates new coverage. This approach is very effective when it comes to maximizing coverage.

2.5.2 Grammar-based and Protocol Fuzzing

Grammar-based [43, 12] and protocol fuzzers use knowledge of the format provided to the program to generate new fuzz cases. Grammar-based fuzzers generate new inputs based on a description of a formal grammar. Typically context-free grammars are used. The goal is to explore a given binary more deeply. Coverage guidance can be added to improve the effectiveness of a grammar-based fuzzer further.

Protocol fuzzing [21] is similar to grammar-based fuzzing in that a description of input is needed to generate inputs. Fuzzing a protocol has unique challenges, mainly because a bug may only be triggered if several packets are exchanged in a sequence. To use an example, protocols such as Transport Layer Security (TLS) [87] use a handshake to derive cryptographic keys to secure a connection. The first packet in this protocol is a *ClientHello* message, which contains specific cryptographic parameters. Fuzzing only this first packet would result in abysmal coverage of the TLS implementation as only the first part of the handshake would be analyzed. Therefore, a more appropriate approach to fuzz such protocols requires understanding the underlying protocol to create a series of inputs.

2.5.3 Over-the-Air, Emulation, and In-Process Fuzzing

Due to its hardware requirements, the *Intel* BB's firmware cannot be run natively on standard *x86* computers. Thus, a different approach is necessary. One approach could be to set up a cellular base station and send malicious packets to the BB over the air [78]. There are several advantages to this approach. Crashes and errors found this way are very likely to be reproducible because the testing setup is essentially indistinguishable from a real-life setting. Very little work needs to be done to understand the internals of the BB itself. The fuzzer could use publicly available standards to produce malicious inputs.

However, over-the-air fuzzing also has significant drawbacks [78]. For one, it might miss certain memory corruptions. If the BB does not crash or return unexpected results, issues will remain undetected. Even more importantly, setting up a base station is challenging itself. Many countries regulate who can operate a base station. Interference with actual base stations must be prevented. If not, the results could range from a simple outage of the cellular network in the surrounding area to damage caused to devices within the range

of the fuzzing base station. Therefore, the use of a Faraday cage would be mandatory. This thesis will not pursue this approach for these reasons.

Another option is to use emulation to re-host the firmware [67, 52, 34]. It can then be provided with inputs through the emulator. Hardware requirements can be met either by software re-implementations or by modifying the firmware and its execution. This approach allows for a much deeper look into the inner workings of the firmware. Memory can be read and written to at will. Thereby also enabling the addition of sanitization [89, 92] to the fuzzers, which means that memory corruptions will be spotted much quicker. Further, execution speed does not depend on the actual hardware anymore and can be parallelized, increasing fuzzing speeds, thus, making the fuzzer more efficient.

The downside of emulation-based fuzzing is the up-front cost of creating a proper emulation of the program. Re-implementing hardware dependencies in software and patching functions within the firmware can add much overhead before fuzzing can even start.

Finally, in-process fuzzing [49] allows hooking into software running on the AP to fuzz communication between the BB and the AP. It is possible to forgo the overhead incurred with emulation by using this approach. Nor is it necessary to go through the lengthy process of setting up a base station.

It does require some work to figure out how exactly communication between the AP and BB is facilitated. Another disadvantage is the lack of introspection into the BB. Similar to the over-the-air fuzzing approach, it is not possible to inspect memory in the BB. Nor is it possible to instrument the firmware to collect coverage or add similar improvements to the fuzzer.

Traditionally the term “in-process” fuzzing stems from the way a target is fuzzed. The fuzzer runs in the same process as its target instead of a separate one. While the fuzzers described in this thesis as “in-process” fuzzers do not hook functions within the BB’s firmware but instead processes running on the AP, the term is still used here. This decision is based on the term “in-process” fuzzing is much more widely used than alternatives such as “on-device” fuzzers. Moreover, other than the fuzzing target, they function much like a traditional “in-process” fuzzer.

2.6 Attacker Model

It is necessary to understand what capabilities a potential attacker has and what goals they could pursue to understand the relevance of this work. This section will give a short overview of the attacker considered in this thesis. Further, the effects of a successful attack will be discussed.

This thesis assumes that the attacker already has control over the main OS running on the AP. The attacker can execute code and has access to all interfaces and processes on the phone itself. Thus, the attacker can interact with the BB directly or via hooking into

a running process that uses cellular capabilities. Due to the architecture of most modern phones, the attacker can already disable cellular communication on the phone and can reset the BB at will.

Usually, the goal is to gain control over the AP. However, the attacker model outlined here already assumes such control. While this model might appear unrealistic, results gained by assuming it are still helpful. Analyzing the handler for the protocol used between the AP and the BB is not meaningfully different from analyzing handlers for glsgsm, UMTS, LTE, and more.

Since the attacker is already quite potent within the bounds of the main OS, their goal is mainly focused on extending their control past access to the current device itself. For instance, it does not make sense for the attacker to control the BB to surveil SMS messages. After all, they have access to this information through other means, such as monitoring process and databases within *iOS*.

Thus, two scenarios reveal themselves. First, an attacker might want to use the BB to send malformed messages to the cellular network. While the attacker can easily send messages within the bounds of the cellular specifications by issuing typical commands to the BB, malformed transmissions would require an exploit on the BB. Therefore, the attacker could benefit from better access to the BB to attack cellular base stations or other components within the cellular network.

Secondly, especially in the case of *iPhones*, some functionality is not publicly documented, such as the *Off-Grid Radio Service* or *OGRS*. This service is part of an unreleased feature that would allow devices to communicate directly without the need for a cellular network [63]. As part of this thesis, it was uncovered that some of the code for this feature made it into the BB's firmware final release. An attacker might use this to infect other *iPhones* even if they are not connected to a cellular network.

CHAPTER 3

Related Work

This chapter provides an overview of existing literature in several fields relevant to this thesis. Section 3.1 discusses existing security research on the *Intel* XMM series of modems. Then, Section 3.2 provides an outlook on work on other Baseband (BB) implementations and cellular software stacks. Lastly, several works on fuzzing techniques are summarized in Section 3.3. While the following sections detail related work on security research of BB implementations by different vendors, including *Intel*, none of them analyzed Apple Remote Invocation (ARI) to date. Thus, this thesis adds to existing literature by approaching *Intel* BBs via this interface.

3.1 Intel XMM Modems

Since this thesis focuses on *Intel's* XMM modem series, work surrounding this BB family is of particular interest. Especially work involving the XMM 7360 and XMM 7480 modems is relevant here, as they are direct predecessors to the XMM 7660 modem at the center of this work.

3.1.1 Reverse Engineering XMM Modems

In a talk at SyScan 360 in 2018 [47], Guy presented his efforts to reverse-engineering the XMM 7360 modem's firmware. This modem is present in some *iPhone* 7 devices. Since this binary does not include debug symbols, which would dramatically improve analysis, several approaches to generate such symbols are presented. In the end, the most successful approach was to port symbols from a related *Android* firmware image. Furthermore, approaches to fuzzing the AT interface of the BB are described.

3.1.2 Vulnerabilities in XMM Modems

The Public Warning System (PWS) [29] is a cellular standard used to inform the broader public about a given threat scenario, such as a missile strike or a natural disaster. Golde and Weinmann [39] have taken a look at how this mechanism is implemented in the XMM 7360 and XMM 7460 modems. They uncovered a series of vulnerabilities, such as a logic flaw, an integer underflow, and a lack of bounds checking. These vulnerabilities ultimately culminated in a buffer overflow in the parser responsible for PWS messages. It allowed an attacker to write arbitrary values into the memory of the BB.

Following his talk from 2018, Guy presented his BB research progress at *REcon 19* [46]. In this presentation, a heap buffer overflow is described. It was found by manually reverse-engineering the BB. Further, efforts to emulate the BB firmware to fuzz the parsers it contains are delineated. While full emulation of the firmware was unsuccessful, partial emulation showed some promise.

3.2 Baseband Security Research

Research that revolves around BBs is of particular pertinence to this thesis. The following section focuses on four different areas. First, work that has been carried out about the communication between the BBs and the Application Processor (AP) is summarized. Secondly, BB reverse engineering efforts are detailed. Thirdly, work analyzing cellular stacks, in general, is discussed. Finally, literature on emulating BB firmware is outlined.

3.2.1 Application Processor to Baseband Protocols

(U)SimMonitor [102] uses the AT command interface of *Android* and *iOS* devices to extract security-relevant data. This data includes the current network provider, the International Mobile Subscriber Identity (IMSI), and cryptographic keys used to encrypt cellular communication. A potential attacker could trick a user into downloading a maliciously crafted application that includes *(U)SimMonitor*. It could then extract and upload this information to an attacker-controlled server. In turn, the attacker could use this information to identify and spy on a given user.

In their work from 2018, Tian et al. [97] analyzed AT commands in over 2000 *Android* smartphones. Their approach first extracted AT commands from firmware images available from the websites of the corresponding cell phone vendors. Next, the commands would be replayed to selected *Android* devices, which exposed an AT interface over USB. Using this interface, they could read and write the BB's memory and, in some cases, modify its firmware. Furthermore, since AT commands are usually unauthenticated, phones with exposed AT interfaces are vulnerable to information leakage. With the *LG* G4, it was possible to enter *Android's* USB debugging mode without user interaction, even if a passcode or pattern was enabled.

ATFuzzer [58] implements an evolutionary grammar-based fuzzer for the AT interface of *Android* phones. Inputs are generated based on a set of grammars that are iteratively

spliced to create new grammars. A fitness function is then used to select a new set of grammars for the next round. The fitness function is loosely based on the timing an AT command needs to execute. This approach was able to uncover several issues across ten *Android* devices by six different vendors.

Makkaveev [68] details his approach to emulate and fuzz a *Qualcomm* BB. *Qualcomm* uses its proprietary protocol called Qualcomm Mobile Station Modem Interface (QMI) to facilitate communication between the BB and the application processor. The blog post details how the BB's firmware was decompressed and how the QMI handlers were found by reverse-engineering. Through fuzzing these handlers, it was possible to find a heap-based buffer overflow within the BB.

3.2.2 Reverse Engineering of Basebands

Weinmann [100] reverse-engineered *Intel* and *Qualcomm* BBs. While fuzzing is discussed in the paper, he argues that it leads to many crashes that are not exploitable vulnerabilities. The preferred approach in this work is to identify functions related to dynamic memory management such as `memcpy()` or `memmove()`. Once these functions have been identified, he searches for calls that use a non-static length parameter. This approach enabled the discovery of several memory corruption issues throughout both BB stacks. However, not all vulnerabilities are described in the paper.

In 2013 at *30C3*, Weinmann [101] gave a follow-up talk about *Qualcomm's Hexagon* architecture which is used in their BBs. He describes the general structure of the processor, calling conventions, and how Return Oriented Programming (ROP) [16] can be achieved on *Hexagon*. Additionally, security mitigations within the BB, such as Address Space Layout Randomization (ASLR) [25], are summarized.

At *REcon 2016* [40] Golde and Komaromy outlined their approach to reverse-engineering the *Shannon* BB. *Shannon* BBs are developed and manufacture by *Samsung* and, thus, feature most prominently in *Samsung's Galaxy* series of smartphones. They describe how they were able to disassemble and analyze the binary that contains the firmware. Further, a stack-based buffer overflow was demonstrated.

A blog post from 2017 by Miru [76] demonstrated how an exploited BB could escalate privileges to the AP. A vulnerability in the Non-Volatile Random-Access Memory (NVRAM) implementation of a *MediaTek* BB allowed an attacker to overwrite files and executables of the *Android* file system. If such an executable was then invoked, an attacker could not only run code on the AP but potentially gain root privileges.

Cama discussed another vulnerability in the *Shannon* BB at *INSOMNI'HACK 2018* [19]. He details his approach to reverse-engineering the firmware and uncovering a bug within the General Packet Radio Service (GPRS) implementation. It was also possible to trigger this vulnerability over the air and write data to the *Android* file system. Lastly, attempts to implement a debugger using this vulnerability are discussed.

At *Black Hat USA 2018*, Grassi et al. [45] demonstrate how they reverse-engineered a *Huawei* BB. A vulnerability present in a part of the Short Message Service (SMS) parser allowed an attacker to overflow a buffer on the stack. They implemented an exploit based on this vulnerability which changed the device's International Mobile Equipment Identity (IMEI) by running code within the BB.

An effort to implementing a debugger for a *Qualcomm* BB was presented at *DEF CON 26* by Burke [17]. By using known exploits in the security architecture of *Android* phones, it was possible to circumvent the signature checks used when booting the BB. Thus, it was possible to modify the firmware. Then, replacing the handlers for some rarely used AT commands, made it possible to arbitrarily read and write memory.

Muiruri et al. [77] presented their efforts in reverse-engineering a *MediaTek* BB at *OPCDE Kenya 2018*. They could add hooks to AT command handlers within the BB because there is no signature check for the BB's firmware. Using this approach, they were able to implement a debugger based on AT commands. With the help of this debugger, they could identify a Denial of Service (DoS) attack.

3.2.3 Analysis of Cellular Stacks

In 2009 Mulliner and Miller [79] gave a talk at "Black Hat USA," showing how they injected SMS messages into an *iPhone*, an *Android*, and a *Windows Mobile* device. Their approach inserts a layer into the communication path between the BB and the application processor. It acts as a driver for the modem's serial interface and then injects the necessary commands to simulate the reception of an SMS message. Using this technique, they were able to inject messages generated by a fuzzing framework. This led to the discovery of several vulnerabilities throughout all three platforms.

Levin describes several details of how *iOS* interacts with the *Qualcomm* or *Intel* BB in his 2018 book [62]. The description also includes a brief discussion of the BBs' firmwares' structure. Further, Peripheral Component Interconnect Express (PCIe) interfaces to the BB are summarized. ARI is mentioned, but no details about its structure are given.

SMS Simulator [91] simulates SMS delivery for local testing on an *iOS* device. It uses an undocumented interface within *iOS* to inject SMS messages without involving the BB at all. Therefore, the tool can be used with *Qualcomm*- and *Intel*-based devices because it acts on a higher layer of the cellular stack.

At *RC3*, Classen [22] detailed her approach towards creating a fuzzer for *CommCenter*, the process within *iOS* handling cellular communication. She used FЯIDA to hook certain functions within *CommCenter*. These hooks could then either manipulate incoming or inject new ARI messages. Further, several factors affecting the fuzzing performance of FЯIDA-based fuzzers are discussed.

3.2.4 Emulation of Baseband Firmware

Grassi and Kira analyzed a BB implementation by *MediaTek* [44]. They reverse-engineered the BB using older versions of the firmware for which the source code had been leaked online. They discuss how they used Quick Emulator (QEMU) [15] to partially emulate the BB and use this emulation for fuzzing. Finally, vulnerabilities are discussed briefly, such as a heap buffer overflow.

BaseSAFE [67] is a cellular fuzzing framework that allows for efficient fuzzing of closed-source firmware blobs such as BB stacks. It uses *Unicorn* to re-host parts of a BB's firmware and then uses the American Fuzzy Lop Plus Plus (AFL++) to fuzz it. Additionally, a sanitized heap was implemented to find memory corruptions faster. A memory dump of a *MediaTek* BB is fuzzed to demonstrate the capabilities of this approach. These efforts resulted in discovering vulnerabilities in the Long-Term Evolution (LTE) implementation that could be reproduced over the air.

At *BlackHat USA 2020*, Hernandez and Muench [52] demonstrated their efforts to reverse-engineer and emulate a *Shannon* BB. To emulate the BB, they used the *Panda* [82] framework, which is based on QEMU. Additionally, the *avatar*^{2} framework was used to model peripherals. After successfully emulating the firmware, they fuzzed parts of the Global System for Mobile Communications (GSM) implementation, culminating in discovering a buffer overflow vulnerability.

3.3 Dynamic Binary Analysis with Fuzzing

Fuzzing, in general, was first introduced by Miller et al. [72] in the context of UNIX utilities. They developed *fuzz* to analyze the reliability of different programs. *fuzz* produces a random series of characters and is the first publicly known fuzzer. At the time many programs could not handle randomly generated inputs. However, the main goal was not to find security vulnerabilities per se but to test whether a program would hang or crash easily. Many programs failed this original test.

Since then, fuzzers have been adapted and improved to allow for more in-depth analysis in several domains. Especially when it comes to finding memory corruption vulnerabilities, fuzzing has enjoyed much popularity over the past two to three decades. The following sections will give an overview of selected literature on the topic of fuzzing. In particular, four fuzzing techniques will be discussed: stateful fuzzing, grammar-based fuzzing, coverage-guided fuzzing and in-process fuzzing.

3.3.1 Stateful Fuzzing

RESTler [14] utilizes stateful fuzzing. The idea behind *RESTler* is that it tries to find vulnerabilities that depend not only on one input but on a series of inputs. It analyzes the API description of a cloud service and automatically creates REST-request sequences. This method allows *RESTler* to automatically find vulnerabilities in such

services. Atlidakis et al. tested their approach successfully with several public cloud services.

Stateful fuzzing is also discussed by Chen et al. [21]. They propose an approach that first captures state changes in a given protocol and then employs heuristics to generate new inputs to a given program. Thus, they explore a program more deeply than would be the case if they would only fuzz one packet of a protocol at a time.

A mechanism to help fuzzers overcome challenges when exploring complex states in a program is presented by Aschermann et al. [13]. Using their tool called *IJON*, a human analyst can add annotations to a program. The fuzzer can then use these annotations to solve challenges it otherwise could not. This technique leads to a much more in-depth exploration of possible program states.

3.3.2 Grammar-based Fuzzing

The concept of grammar-based fuzzing was summarized in Section 2.5.2. This section will now summarize approaches to implement grammar-based fuzzers.

Grammar-based fuzzers tend to be less efficient than entirely random fuzzers. Gopinath and Zeller [43] first implemented a grammar-based fuzzer in *Python*. Then, they iteratively optimized and re-wrote their implementation until their fuzzer surpasses a random lexical fuzzer.

Nautilus [12] is a grammar-based fuzzer that uses a formal grammar's description to generate inputs for a given program. It can also be extended with scripts to make it possible to use non-context-free grammars, making the fuzzer more versatile. Further, *Nautilus* also uses coverage data to improve fuzzing efficiency.

3.3.3 Coverage-guided Fuzzing

Coverage-guided or evolutionary fuzzers, as discussed in Section 2.5.1, first mutate a given input and provide it to the fuzz testing target. After the target has finished processing the input, the fuzzer decides whether the input should be considered for further fuzzing based on coverage data. The following section discusses two of the most popular evolutionary fuzzers that are also used in this thesis.

One very popular general-purpose fuzzer is American Fuzzy Lop (AFL) [103]. It uses instrumentation to add custom code to a given program at each branch point. This code tracks how often a branch is taken by increasing a counter referenced by the current and previous basic block. Using this coverage data, AFL maintains an input queue. This queue is filled from a corpus that needs to be provided by the user. AFL will then mutate this corpus in several different ways. If a given input produces new coverage information, it will then be added to the queue. Coverage information is also used to analyze crashes and minimize the input needed to cause a crash.

Since development on AFL has mostly been inactive since 2017, a community fork named AFL++ [36] has gained increasing notoriety. It added several improvements such as more sophisticated mutation strategies, more flexible instrumentation, and additional coverage metrics. It also offers an Application Programming Interface (API) to implement custom fuzzing strategies. In combination, these additions make AFL++ a much more versatile fuzzer than AFL and a very efficient one as demonstrated in several benchmarks.

3.3.4 In-process Fuzzing

In-process fuzzers use function and data hooks in already existing processes to fuzz a given target. A more in-depth discussion of the general topic is given in Section 2.5.3. The following section introduces three in-process fuzzers that have centrally influenced the in-process fuzzers presented in this thesis.

Frizzer [69] is a general purpose coverage guided black box fuzzer. It is based on FЯIDA and can collect coverage through FЯIDA's *Stalker* feature. It can fuzz a binary while it is running on a given host system without the need for prior instrumentation. The binary can even reside on a different system than the fuzzer itself, as long as it can be instrumented via FЯIDA.

ToothPicker [49], which is partially based on *Frizzer*, instruments the *iOS Bluetooth* daemon using FЯIDA. It then mutates packets within a *Bluetooth* connection using *Radamsa* [50]. Further, it can also collect and use coverage data. *ToothFlipper* uses an even more straightforward approach: It simply flips certain bits and bytes in a given packet. The resulting fuzzer is very stateful and can reach very complicated states within the *Bluetooth* daemon.

Improving on the concepts of *ToothPicker*, *fpicker* [48] supports several modes of operation. The most noteworthy mode is AFL++ mode which runs an AFL++ instance on the device that is being fuzzed. *fpicker* uses a custom mechanism based on semaphores and shared memory to improve the overhead incurred by the FЯIDA API to communicate with the instrumented process. The tool supports fuzzing processes on *macOS*, *iOS*, and *Linux*.

CHAPTER 4

Baseband Reverse Engineering

Before parts of the Baseband (BB) can be emulated, some parameters have to be understood. This chapter aims to demonstrate the knowledge gained about *Intel* BB implementations by reverse-engineering its firmware. First, a short overview of the cellular stack on *iOS* is given in Section 4.1. Section 4.2 details how the firmware can be acquired and Section 4.3 gives some information about the processor model and memory layout. An in-depth look at the size and complexity of the different BB generations is presented in Section 4.4. Then, Section 4.5 briefly discusses the Real-Time Operating System (RTOS) used by the BB. Further, Section 4.7 provides a short introduction to diagnostic tools on *iOS*. Finally, Section 4.8 describes Apple Remote Invocation (ARI) and several handlers related to it within the BB and *iOS*.

4.1 Overview

Device ID	Product Name	Modem	Baseband	Architecture
iPhone 9,1	iPhone 7 (Global)	XMM 7360	ICE16	ARMv7-A
iPhone 10,1	iPhone 8 (Global)	XMM 7480	ICE17	ARMv7-A
iPhone 10,3	iPhone X (Global)	XMM 7480	ICE17	ARMv7-A
iPhone 11,2	iPhone XS	XMM 7560	ICE18	x86
iPhone 11,8	iPhone XR	XMM 7560	ICE18	x86
iPhone 12,1	iPhone 11	XMM 7660	ICE19	ARMv7-A
iPhone 12,8	iPhone SE (2020)	XMM 7660	ICE19	ARMv7-A
Watch 4,4	Apple Watch Series 4	XMM 7560 Variant	IBIS18	ARMv7
Watch 5,4	Apple Watch Series 5	XMM 7560 Variant	IBIS18	ARMv7
Watch 6,4	Apple Watch Series 6	XMM 7560 Variant	IBIS18	ARMv7

Table 4.1: Selected *Apple* devices and their BB versions.

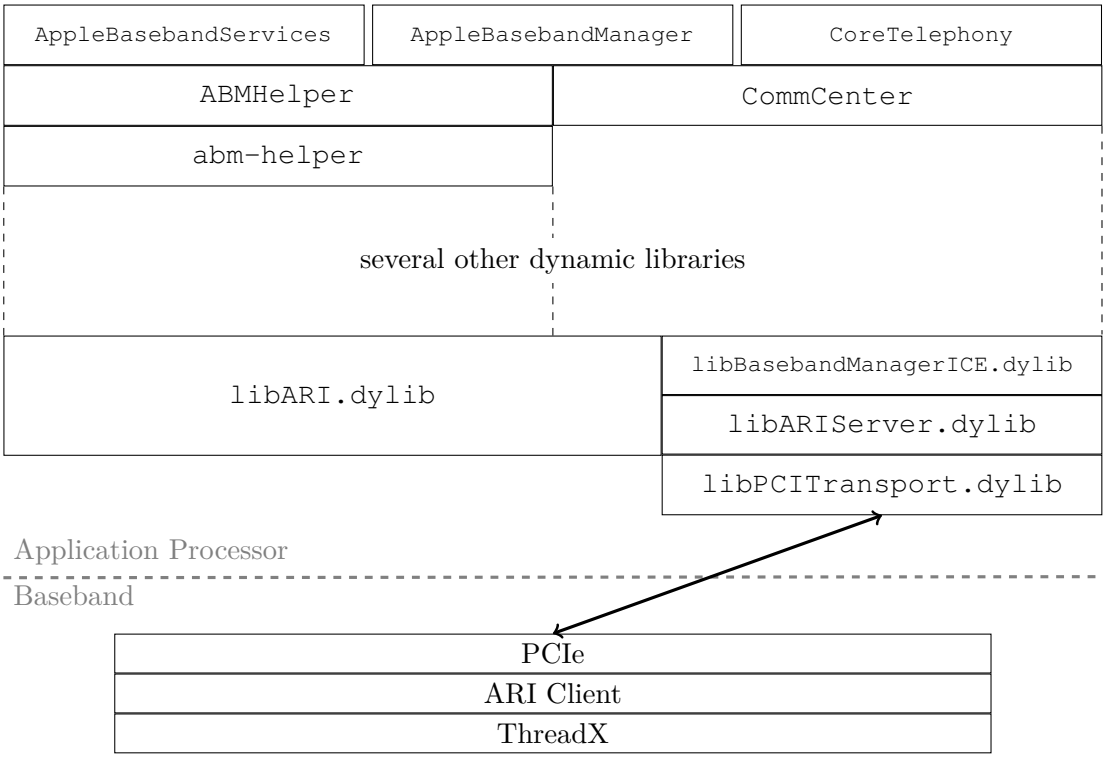

Figure 4.1: An overview of selected Frameworks, Libraries and Programs in the *Apple* cellular stack in combination with *Intel* BBs.

This chapter will give an overview of four different versions of *Intel* BBs in connection with *Apple* devices. The main focus will be on the newest version, *ICE19*, used by *iPhones* and *iPads* with XMM 7660 modems. It is a successor to the *ICE16*, *ICE17* and *ICE18* BBs, utilized in previous *iOS* device generations. Lastly, *IBIS18* BBs are present in *Apple Watch* Series 4, 5, and 6 devices. Table 4.1 shows an overview of different selected devices made by *Apple*. Each device has its identifier, the product name, the modem version and the BB firmware generation listed. It also shows that the Apple Watch uses a modified version of the XMM 7560 modem, reflected by the differing BB version. Finally, also the processor architecture of the given chip is stated.

Further, given that this thesis focuses on *Intel* BBs used primarily in *iOS* devices, it is necessary to understand certain parts of *iOS* that interact with the BB. Selected parts of the cellular stack and the BB relevant to this work can be seen in the schematic in Figure 4.1. In general, the cellular stack is built on top of at least four frameworks: `AppleBasebandServices`, `AppleBasebandManager`, `ABMHelper`, and `CoreTelephony`. The first three are private frameworks used by *Apple* to build public frameworks, applications and more.

Then, there are at least two significant executables build on top of these frameworks. Firstly, `CommCenter` handles phone calls, Short Message Service (SMS) messages and

other cellular functions. A small amount of its functionality is exposed to developers through `CoreTelephony`, such as querying the cellular service provider [8]. Secondly, there is `abm-helper` which is used to manage the BB. It monitors whether the BB has crashed and needs to be reset and generally handles tracing and diagnostic capabilities.

Both of these executables import several dynamic libraries that can be accessed through FЯIDA at runtime. Note that the exact libraries loaded by the executables depend on whether an *Intel* or a *Qualcomm* BB is used. Two of these are of particular importance: `libARIServer.dylib` and `libARI.dylib`. They handle the communication with the BB via Peripheral Component Interconnect Express (PCIe). Lastly, `libBasebandManagerICE.dylib` manages interactions with the BB in `CommCenter`.

4.2 Firmware Acquisition

Acquiring the firmware of the BB is the first step to analyzing it. In the case of *iOS*, it can be extracted from *iOS* updates available through `ipsw.me`.[1] *iOS* updates are issued in `.ipsw`-files, these are simply ZIP archives and can be extracted easily with any standard decompression tool (e.g., the `unzip` command).

From there, the firmware is usually found in the aptly named "Firmware" folder. The BB firmware image is typically stored as a `.bbfw`-file, which again is a ZIP archive. The build manifest included in the `.ipsw`-file gives some insight into which firmware is used for the different parts of the *iOS* device. For the BB, it lists the relative path to the `.bbfw` archive. Furthermore, for each binary contained within the BB firmware image, a base64-encoded `SHA2-384` digest is listed. An excerpt of a `BuildManifest.plist` demonstrating this can be found in Section A.2. In this fashion, BB firmware images from *iOS* 13.5, 14.1, 14.4, and 14.5 were acquired over the course of this thesis.

Depending on the exact firmware version, the `.bbfw`-file contains a different set of files. A comprehensive comparison of which files are present in which BB version is presented in Table 4.2. *ICE19* likely uses `psi_ram2.bin` and `psi_ram2.bin` for different revisions of the modem. Note that the absence of files, such as `2GFW.elf`, does not indicate that the modem does not support Global System for Mobile Communications (GSM) but that GSM implementation is likely part of a different binary. The most crucial binary in this archive is the "System Software" contained in `SYS_SW.elf`. It is present across all BB generations and contains the central part of the firmware.

4.3 Processor and Memory

The processor architecture and memory layout need to be understood to a certain extent to emulate the BB successfully. Some general information can be gained directly from the Executable and Linkable Format (ELF) [98] file, such as the endianness and processor

[1] *iOS* update images are available through `https://ipsw.me`.

File	ICE16	ICE17	ICE18	ICE19	IBIS18	Meaning or Purpose
2GFW.elf		8.9 kB	117 kB			2G (GSM) firmware
3GFW.elf		737 kB	2 MB		445 kB	3G firmware
ant_cfg_data.elf				506 kB		Configuration data for the antenna
AudioFW.elf		14 kB*	721 kB		13 kB	Audio processing firmware
bbcfg.bin	133 kB	82 kB	805 kB	1.8 MB	42 kB	Baseband configuration data
CDMA2KFW.elf			105 kB			CDMA2000 firmware
custpack.elf			17 kB	384 kB		-
Debug_info.elf		274 kB	1.1 kB		281 B	Debugging related information
DPC.elf			818 kB*			-
ebl.bin	77 kB	73 kB	243 kB	144 kB	69 kB	Embedded boot loader [60]
GNSS_FW.elf					69 kB	Global navigation satellite system firmware
irx_coefficient.elf					1.4 MB	-
legacy_rat_fw.elf				595 kB		Firmware for the legacy radio access technology
LTEFW.elf	4.2 MB*	5 MB*	5.5 MB*		4.7 MB*	LTE firmware
psi_ram2.bin				397 kB		Stage zero boot loader [60]
psi_ram.bin	128 kB	160 kB	396 kB	396 kB	128 kB	Stage zero boot loader [60]
restorepsi2.bin				397 kB		Possibly restores stage zero boot loader
restorepsi.bin	128 kB	160 kB	396 kB	396 kB	128 kB	Possibly restores stage zero boot loader
RFFW.elf		414 kB*	14 MB*	17 MB*	8.4 MB*	-
RPCU.elf			1.6 MB*			-
SYS_SW.elf	27 MB	29 MB	49 MB	63 MB	13 MB	System Software (main firmware)
TDSFW.elf	1.7 MB	1.8 MB	1.7 MB			-
TPCU.elf				1.1 MB*		-
upc.elf				19 MB		-

Table 4.2: Files contained in the different BB firmwares. The asterisk indicates that the binary uses the *Xtensa* architecture.

architecture. However, this information is not always trustworthy as it can be overwritten easily.

4.3.1 Processor Architecture

Inspecting the ELF-headers of `SYS_SW.elf`, which contains the main firmware, for *ICE18* BBs will indicate that this is a 32-bit *ARM* binary with little-endian byte order. However, only about nine functions will be automatically detected when loading this file into a disassembler like *IDA Pro* [53]. In contrast, when automatically analyzing an *ICE19* BB, about 100.000 functions are found. It turns out that the XMM 7560 modem shipped with some *iOS* devices (e.g., the *iPhone* XS) shipped with *x86* based BBs [60]. *IDA Pro's* "MetaPC" processor type can be used to disassemble all *x86* instructions. Re-running the analysis with this processor type yields a similar result to *ARM* BBs like *ICE17* or *ICE19*.

Even though *ICE18*'s main firmware is based on the *x86* architecture, other firmware parts are based on the *Xtensa* [94, 41] architecture. When running `cpu_rec` [4] on the `DPC.elf` file to identify the Central Processing Unit (CPU) architecture, we can find two code segments that use *Xtensa* instructions. Two other files, `RPCU.elf` and `AudioFW.elf`, contain *Xtensa* segments as well. Strings within these files further confirm the use of this architecture.

Usage of *Xtensa* cores can also be found in *ARM*-based BB generations. For instance, the Long-Term Evolution (LTE) stack's firmware (`LTEFW.elf`) is typically implemented as an *Xtensa* binary. In *ICE19* BBs, inspecting the `TPCU.elf` file yields strings that indicate an *Xtensa* instruction set, and so does `cpu_rec`.

The *Xtensa*-based binaries often implement digital signal processing tasks, such as handling the audio for voice calls. It stands to reason that they were used to increase the efficiency of the overall BB. *Xtensa* cores are often used specifically to implement Digital Signal Processors (DSPs) [18]. Therefore, they can handle these assignments much more efficiently, thanks to the modular nature of the *Xtensa* architecture.

4.3.2 Processor Model

Particular details about the processor cannot be understood by analyzing the ELF, such as the exact processor model. Luckily running the `strings` command on the `SYS_SW.elf` file, in the case of *ARM*-based BBs, returns some very informative strings, such as the one in Listing 4.1.

```
1 [...]
2 ../../3p_threadx/cortex-a5r-smp/src/tx_thread_shell_entry.c
3 [...]
```

Listing 4.1: String contained in the *ICE19* BB indicating an *ARM* Cortex-A5 processor.

This string indicates that the BB's CPU is based on *ARM's* Cortex-A5 processor, which uses the ARMv7-A instruction set. Similar strings are also contained in *ICE16* and *ICE17* BBs. According to *ARM's* specification [64], it offers a modular core count, which means the physical processor might have between one and four cores. The firmware checks the number of cores across all BB generations. However, the number of cores differs between generations. Listing 4.2 shows a check within the *ICE17* BB that indicates that the given processor has three cores.

```
1 char buf[20];
2
3 /* ... */
4 if ( possible_core_id >= 3 )
5     sub_85DCD5AC(buf, "Invalid Core Id!", 20);
```

Listing 4.2: Check for valid core id in ICE17 BBs.

There are further indications that the *ICE17* BBs contain three cores. The same is the case for *ICE16*. While *ICE18* BBs contain similar checks, their change of processor architecture makes a direct comparison difficult. However, *ICE19* BBs use *ARM* Cortex-A5 processors again. Here, the check has been modified to allow up to four cores, as can be seen in Listing 4.3.

Thus, the actual core count differs between *ARM*-based BBs as well. *ICE16* and *ICE17* BBs appear to have shipped with a variant containing three cores. *ICE19* BBs, on the

```
/* ... */
if ( possible_core_id >= 4 )
    return sprintf_like("%s Error: Invalid CoreID : %d",
        off_8A420E78 + 29, possible_core_id);
/* ... */
```

Listing 4.3: Check for a valid core id in ICE19 BBs.

other hand, utilize all four possible cores. More code snippets indicating the core count in all four BB generations can be found in Section A.3.

4.3.3 Memory Layout

For the memory layout needed for the emulator, the `LOAD` segments of the ELF files are beneficial. They indicate the virtual addresses to which a given segment within the ELF file must be mapped and its size and alignment. Additionally, every segment contains a flag indicating whether it can be executed, is readable, or writeable. Using these offsets comes with the upside that addresses in the emulator's memory are identical to the ones used by *IDA Pro* and *Ghidra*.

Since this thesis mainly focuses on partial emulation, the presented information is sufficient to emulate certain firmware parts, as will be shown in Chapter 5. However, this is not a complete memory layout by any means. For example, the BB has access to non-volatile memory stored on the device's main file system. In other words, the Application Processor (AP) has access to this memory. It is mapped as a set of files in *iOS* in the */private/var/wireless/baseband_data/bbfs* directory. However, details about this mechanism are yet to be determined.

4.4 Baseband Complexity

This section provides an overview of the general structure and size of the BB's main firmware. First, the number of functions detected across the three BB generations will be discussed. This analysis will show how the firmware's complexity has changed over time. Secondly, the general structure of the firmware's source code was partially reconstructed from strings within the BB. This reconstruction will provide further insights into the complexity of the BB.

4.4.1 Function Detection

Several tools were used to show how many functions are present within the BB's main system firmware (`SYS_SW.elf`). Previous work [37, 83] has shown that instruction and function detection can vary between different static analysis tools. Thus, this section can also serve as a comparison between the performance of the chosen tools. The tools

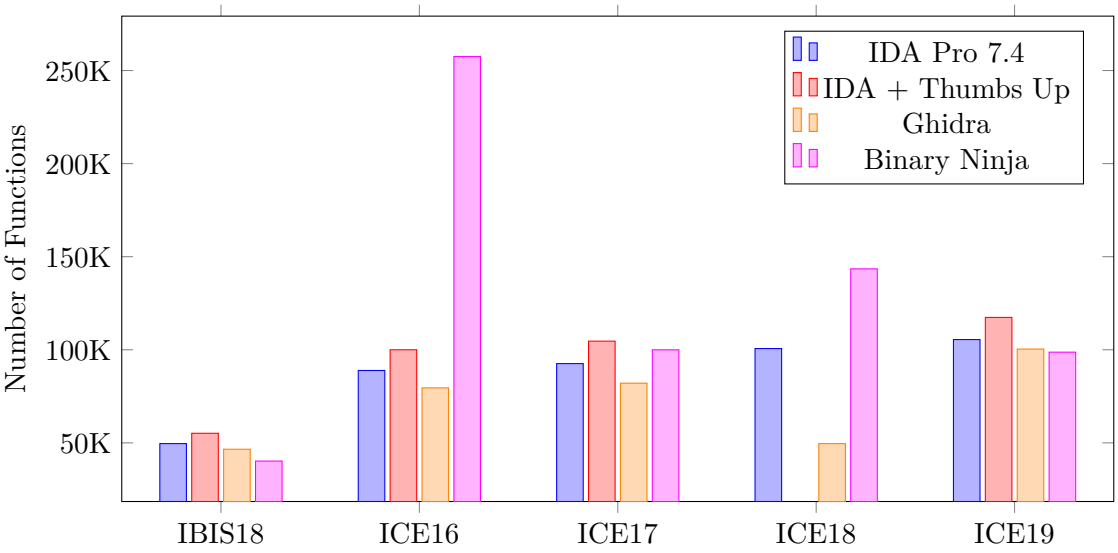

Figure 4.2: Function detection between different BB generations and disassemblers.

include *IDA Pro* version 7.4 [53], *Ghidra* version 9.2 [3], and *Binary Ninja* version 2.4 [99]. Further, to improve function detection in *IDA Pro*, the *Thumbs Up* [20] script was used.

The import settings of each disassembler were left to the default settings in most cases, with two notable exceptions. First, in *ICE18* BBs, every tool trusted the information in the ELF and had to be manually set to use an *x86* disassembler. Secondly, *Binary Ninja* would not start analysis on the segments defined by the ELF headers automatically. ELF segments with the same semantics as code regions had to be converted to sections. In other words, each writable and executable segment within a given ELF file was marked as a target for analysis via a script.

Figure 4.2 shows the reported number of functions after having imported the `SYS_SW.elf`. Sadly we lack a baseline to compare the results too. As of writing, there is no symbolicated *ICE* or *IBIS* BB firmware publicly known. Thus, it is unknow what the actual function count could be.

However, the differences between the imports, especially between *Ghidra* and *IDA Pro*, line up with previous work [37] on disassembler performance. Thus, the results are probably relatively close. One caveat remains, the likely substantial percentage of false positives.

The plot shows a steady growth between the BB versions. From *ICE16* to *ICE17*, the function count increased by about 4 000 functions. Then from *ICE17* to *ICE19*, about 12 000 to 13 000 thousand functions were added. This result shows that the number of functions increases between 4.5 % and 14 % with each generation. Note that comparing the function count to *ICE18* is difficult due to switching to an *x86* based infrastructure. However, the detected count is very close to *ICE19* BBs, it misses only about 5 000

functions.

It is also clear that *IBIS18* is a lot smaller than its *ICE* siblings. All disassemblers seem to have found only half as many functions compared to *ICE19*. This result is also reflected by the smaller overall size of the firmware archive. The *IBIS18* archive is slightly bigger than a quarter of the size of the *ICE19* firmware (12.4 vs. 42.6MB).

While *Ghidra* and *IDA Pro* mostly behave as expected, showing a steady increase and relatively similar results, *Binary Ninja* did not. The most apparent oddity is the function count for *ICE16*, which is more than twice the number of functions than any other BB generation. Importing this binary was also showed strange behavior in that it took the longest to finish (about 30 hours). It likely interpreted parts of the BB that were pure data as a code region.

Interestingly, the *Binary Ninja* numbers for the other *ARM*-based BBs mostly align with the other disassemblers. Further, while a raw import of the *ICE19* BB shows it to be smaller than *ICE17*, after manually adjusting a couple of function signatures (fewer than 20), the number of functions detected by *Binary Ninja* increased by 14.78 % to 113 328. This number brings it much closer to the number reported by *IDA Pro* in combination with *Thumbs Up* (117 394 functions).

Finally, *Ghidra* detects surprisingly few functions in *ICE18* BB. It reports only 49 587 functions which is less than half of the about 100 000 functions reported by other disassemblers. Similarly, *Binary Ninja* again reports surprisingly many functions (143 484).

It might be that the information in the ELF impacts the analysis even though it was manually overwritten. Several different settings were tried, and the imports were re-run to ensure that it was not a simple configuration error. However, neither in the case of *Ghidra* nor *Binary Ninja* did results change significantly. When looking at the results reported by Friebertshauser et al. [37] they show that changing settings within *Ghidra* seems to have little effect, which is consistent with the results of this work.

4.4.2 Source Code Structure

Another way to understand the structure and complexity of the BB is to analyze specific strings within the BB. By making inferences from these strings, the general structure and size of the underlying codebase can be understood. A *Python* script was implemented to extract such a structure to automatize this process.

This section will first discuss the two approaches utilized by this script. Secondly, the results of this analysis will be presented. This approach was inspired by Hernandez's [51] approach to recreating the *Shannon* BB's source code structure.

Source Code Files

Some analysis must first be carried out to implement a script that can extract source code paths and reconstruct the codebase's folder structure. Listing 4.4 shows several strings

```
1 ../../../modem/msw_platform_services/runtime_services/utils/src/
    util_mem_s.c
2 ../../msw_platform_services/runtime_services/icc/src/icc.c
3 modem/msw_platform_services/runtime_services/utils/src/util_string_s.
    c
4 /var/xbs/bwa/3A54AE22-4095-447F-B510-5BCB148C5C9C/ICE19BaseBandFW
    -20407/srcroot/libsec/bblibsec/ice_rf_self_test.c
5 ../../3p_ice/src/ibi_adapters/ibi_adapter_grp23_trc_info.cpp
6 ../../3p_ice/src/ibi_adapters/ibi_adapter_grp34_ice_ipc.cpp
7 ../../media_dsp/source/../../media_dsp/source/handler_layer/
    ahb_error_task/ahb_error_task.c
```

Listing 4.4: Source code path examples in the *ICE19* firmware.

that point towards source code files. From the file extensions, it is possible to deduct that the firmware has a mixed C/C++ codebase. These file extensions can then be leveraged to filter the strings within the BB to only contain valid source code paths. Additionally, since references to simple file names cannot be placed in an overarching folder structure, strings that do not include a *Unix* path separator (`/`) can also be excluded.

However, most strings that point to source code files are relative paths often containing references to parent directories. Thus, information from absolute paths or relative paths without references to parent directories was used. Most of the files are likely contained in a directory with the name "modem." If a path starts by going back up the directory tree twice, this is replaced with the "modem" directory.

Further, the entirety of the source code seems to be contained in a folder named "srcroot." Thus, the "modem" folder is placed inside the "srcroot" directory. All other references to parent directories are then resolved relative to "srcroot." For every path, a new empty file is created at the given location. All files are then collected in a `.zip`-file.

This approach results in a realistic-looking source code structure. There are, of course, exceptions to the rule and the script does produce a small amount of strange-looking sub-structures. For instance, with *ICE19*, a folder with the name "l" containing a single file is created, the file's name contains a colon. It seems unlikely that this folder is present in the actual codebase.

Libraries

In addition to the paths related to source code files, the firmware for each BB generation also contains references to library files (`.lib`). Sadly, these references do not include absolute or relative paths but only file names. Therefore, it is impossible to add them to the source code structure in the same way as the source code paths.

However, they do provide an insight into how the firmware has changed over time. Thus, the script to produce a folder structure includes an option to collect this information as well. In the re-created codebase, the libraries are then added to a folder named "libs".

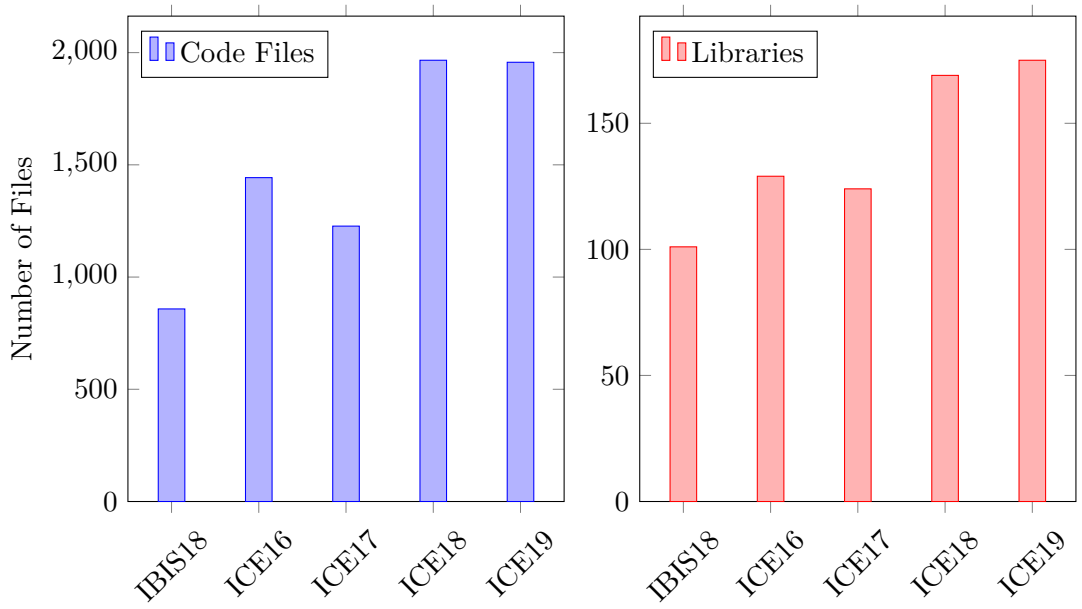

Figure 4.3: Comparison between the number of source code file paths in the BB's firmware.

Structure Analysis Results

One advantage of this approach is that the entire firmware image is taken into account. In Section 4.4.1, only `SYS_SW.elf` was analyzed. This circumstance was mainly due to the complexity of importing these files into the respective tools. An analysis process typically took several hours, and some failed several times due to a lack of memory before completing successfully. This script-based approach is much simpler and, thus, faster. Hence, all files in a given firmware will be taken into account.

Figure 4.3 shows how the BB's firmware has grown between different generations. The chart also confirms that *IBIS18* is much smaller than the other BBs. At 858 source code files and 101 libraries, it is about half the size of *ICE19* at 1 957 source code files and 175 libraries.

Interestingly, *ICE17* contains considerably fewer references to source files (1 227 vs. 1 443) and slightly fewer libraries (124 vs. 129) than *ICE16*. Section 4.4.1 showed that *ICE17* contains around 4 000 more functions than *ICE16*. One possible explanation is that certain log strings might have been stripped from *ICE17*. Analyzing the structure of the BB based on the files included in their given firmware image (see: Table 4.2) also shows that significant changes have occurred.

In addition to quickly analyzing all files in a given firmware image, this approach also makes it possible to compare *ICE18* to the *ARM*-based BBs more directly. For instance, taking a look at the contents of the "modem" shows that *ICE18* only misses two out

of 40 folders compared to *ICE19*. This result shows that the code base has remained relatively similar in its structure between the two generations.

Thus, we can conclude that from *ICE16* to *ICE18*, the firmware's structure has gained about 36 .24 % more source files (1 443 vs. 1 966) and 31 % more libraries (169 vs. 129). Finally, *ICE18* and *ICE19* have remained relatively stable, with the source code file count slightly decreasing from 1 966 to 1 957 and libraries moderately increasing from 169 to 175.

4.5 Real-Time Operating System

Intel BBs use a version of *ThreadX* [70][2] as their RTOS, as indicated by several strings in the BB's firmware. *ThreadX* was initially developed by *Express Logic* and has been acquired by *Microsoft* in 2019 [38]. Beginning with version 6.0, *ThreadX* has been re-branded as *Azure RTOS ThreadX*, and the source code has been made available through *GitHub*.

```
1 Copyright (c) 1996-2013 Express Logic Inc. * ThreadX SMP/Cortex-A5/
    RVDS Version G5.6.1.5.1 SN: 3582-266-1319 *
```

Listing 4.5: Copyright string from a *ICE19* BB indicating that *ThreadX* is used.

The *ICE16*, *ICE17* and *ICE19* BBs run a version for *ARM* Cortex-A5 processors with the version number "G5.6.1.5.1". This conclusion can be inferred from the copyright string in Listing 4.5, taken from an *ICE19* BB. An essentially identical string can be found in the latest versions of *ICE16* and *ICE17* BBs.

```
1 Copyright (c) 1996-2014 Express Logic Inc. * ThreadX Xtensa Version
    G5.6.5.7 SN: 3906-198-3201 *
```

Listing 4.6: Copyright string from the *ICE18* BB for an *Xtensa* version of *ThreadX*.

ThreadX is used in *ICE18* and *IBIS18* BBs as well. While the main firmware does not include the copyright string present in *ICE16*, *ICE17*, and *ICE19* BBs, some other files contain a copyright string for an *Xtensa* version with the number "G5.6.5.7" as shown in Listing 4.6. *ICE19* uses this version for its *Xtensa* cores too.

4.6 Memory Exploitation Mitigations

Especially within *ICE19* BBs, an effort to secure against memory corruption exploits is noticeable. This effort can be seen in Table 4.3, which gives an overview of the mitigations discussed in this section. For instance, the bootloaders within the firmware contain the ability to set up an Address Space Layout Randomization (ASLR) [25] mechanism. The stage zero bootloader in `psi_ram.bin` or `psi_ram2.bin` can either generate offsets

[2] ThreadX on GitHub: `https://github.com/azure-rtos/threadx/`

Mitigation	ICE16	ICE17	ICE18	ICE19	IBIS18
Stack Canaries	?	?	?	✓	?
ASLR				✓	
Safe `string` Library		✓	✓	✓	✓

Table 4.3: Memory exploitation mitigations deployed in different BB generations.

or use injected ones, which means that the AP might have the ability to deactivate ASLR. In `ebl.bin`, regions from the main firmware contained in `SYS_SW.elf` are re-located to regions related to these offsets.

ICE19 BBs also use stack canaries or cookies [25] to protect the return address during function execution. After the function prologue, a value is loaded from the address `0x8671F000` and is stored onto the stack. Then, the function body is executed. Before the function returns, the value on the stack is checked. If the value remained unchanged, the function returns normally. Otherwise, a failure handler is executed. The assembly code of this mechanism is shown in Listing 4.7.

```
1     ; function prologue
2     LDR             R6, =0x8671F000
3     ; ...
4     LDR             R0, [R6]
5     STR             R0, [SP,#0x20+var_20]
6     ; function body
7     LDR             R0, [SP,#0x20+var_20]
8     LDR             R1, [R6]
9     CMP             R0, R1          ; canary check
10    BEQ             loc_8A783CCA    ; success: branch to return
11    POP.W           {R2-R8,LR}
12    B.W             stack_sec_fail  ; failure: branch to handler
```

Listing 4.7: Setup and check of a stack canary taken from a function in ICE19.

Listing 4.8 shows the stack canary failure handler. It writes the value `0xDEADBEEF` to the address stored in the link register (`LR`) and then returns. In essence, this will cause a crash. Presumably jumping to `0xDEADBEEF` is recognized as an invalid write operation and is, thus, aborted.

```
1 .stack_sec_fail
2     LDR       R1, =0xDEADBEEF
3     STR.W     R1, [LR]
4     BX        LR
```

Listing 4.8: Stack canary failure handler in ICE19.

ICE17 introduced the usage of a safer version of the `string` library [46] to make buffer overflows and similar vulnerabilities less likely. It contains string and memory routines

that can lead to issues because they do not consider the size of the buffer they are writing or reading. For instance, instead of using functions such as `memset()` or `memcpy()`, `memset_s()` and `memcpy_s()` are used. The only BB generation considered in this thesis that does not utilize this improved version is *ICE16*. These functions try to prevent out-of-bounds write errors by specifying two length parameters, the first states the size of the buffer that will be written to and the second the number of bytes to write. While such checks can be useful to detect some issues, they still depend on the developers to use them correctly. If the size of the destination buffer is not correctly provided, illegal writes might still be possible.

4.7 Diagnostic Tools on iOS

On *iOS*, several command-line tools that can help with diagnosing issues with the BB exist. While these tools are available on a jailbroken *iPhone*, their full functionality is typically not accessible.

4.7.1 DumpBasebandCrash

The name of this tool would indicate that it can dump either memory or a kind of crash report from a BB once it crashed. At first, running this command does not seem to affect anything. There is no output at all nor an indication of whether any files have been written or changed. It turns out that this tool will only work if a crash has been detected recently. In this case, `.istp` files will be written to a directory within the */var/wireless/Library/Logs/CrashReporter/Baseband* directory. If the *iOS* BB debug profile is installed the files are additionally copied to */var/mobile/Library/Logs/CrashReporter/Baseband*[3].

4.7.2 abmlite

Another tool that is available on current *iPhones* is `abmlite`. It is based on the `ABMHelper` private framework, which contains strings hinting at a more powerful version of `abmlite` called `abmtool`. However, since this is an internal tool, `abmtool` is not available to the broader public. The publicly available `abmlite` is limited to similar functionality as `DumpBasebandCrash`. Its main component is the `logdump` sub-command which offers four dump types: `bb`, `core`, `oslog`, and `tel`.

The `bb`, `core`, and `tel` commands are very similar. They all create a set of `.istp` files, again in the */var/wireless/Library/Logs/CrashReporter/Baseband* directory. However, `core` and `tel` also create a dump of the device log limited to activity related to `CommCenter`. Lastly, the `oslog` option fails without providing a reason for why it failed. It likely outputs a file containing parts of the regular system log.

[3]Debug profiles are available from `https://developer.apple.com/bug-reporting/profiles-and-logs/`.

4.7.3 .istp Files

It is necessary to understand a proprietary file format called ISTP to understand existing diagnostics tools on *iOS*. This format is not publicly documented, and since the ability to use `abmlite` or `DumpBasebandCrash` hinges on an understanding of this format, several attempts were made to reverse-engineer ISTP.

First, it is necessary to understand how the `.istp`-log files are created. About every five seconds two new files are written to the scratch folder (*/var/wireless/Library/Logs/AppleBasebandManager/BBTrace.scratch*). The first is the `.istp` file itself, and the second is a `.meta` file. This second file is a text file containing a timestamp at which the corresponding `.istp` file was created and the number of bytes it contains. Typically `.istp` files are each about one megabyte in size. The name of both files is the hexadecimal representation of a counter.

Once a `logdump` command is issued through `abmlite`, the `.istp` files in the scratch folder get moved to a log folder. The `.meta` files are merged into an `info.txt` file. Next to the file sizes and creation timestamp, this file includes the BB version and a reason for the dump's creation (i.e., whether it was triggered through a tool or a BB crash). It appears that the reason for the crash has only been provided since the update to *iOS* 14.

Several approaches to investigate the format of the `.istp` files are detailed in the following sections. None have yielded a completed picture of the structure of these files, but several details have been understood so far. It is also possible to rule out that ISTP is identical to various well-known protocols.

Suspected Internal Tooling

Intel seems to make available several tools for their partners that make it possible to understand the *ISTP* format. One such tool is the *Intel® System Debugger* [57]. Another tool is the *Intel System Trace Tool (STT)* [55]. However, it was impossible to obtain them as part of the thesis because *Intel* only shares these tools with partner companies after signing a confidential non-disclosure agreement. The latter tool, *STT*, is also explicitly mentioned in the *Intel* XMM 7160 modem [93] documentation.

Static Analysis and Reverse Engineering

`binwalk` [79] was used to rule out that ISTP files were encrypted or compressed with a known algorithm. Entropy analysis showed that it is unlikely that ISTP files are encrypted. Figure 4.4 shows the entropy distribution of a sample `.istp` file taken from an *ICE19* BB. If they were encrypted, the expectation would be that any given `.istp` file would have constant high entropy because encrypted files should appear to be random.

Further, analysis with `binwalk` also indicated that `.istp` files are unlikely to be compressed with a well-known algorithm. It cannot be ruled out that compression might

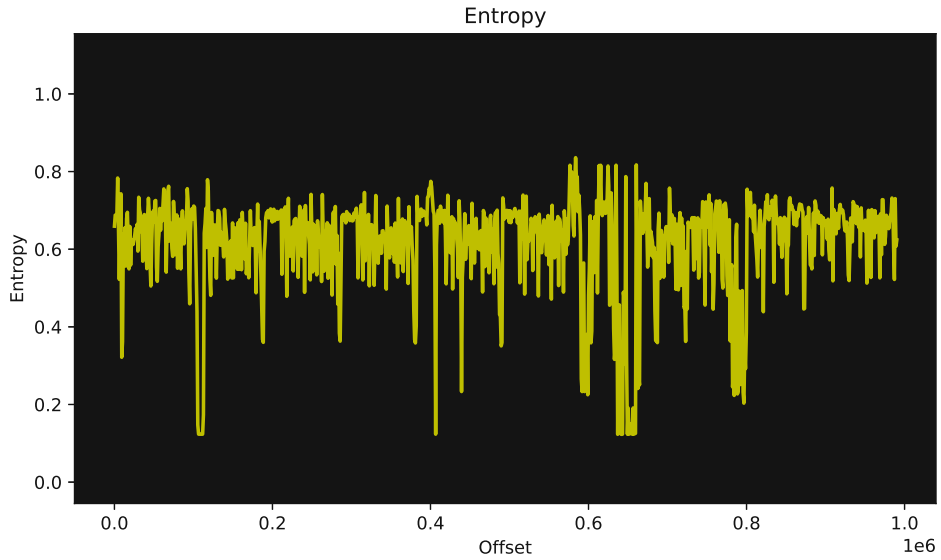

Figure 4.4: Entropy graph of a sample `.istp` file.

be applied to at least parts of the files. However, ISTP versions used in BBs before *ICE19* were not compressed either.

By analyzing the structure of these files and frameworks, libraries, and tools interacting with them both on the BB and *iOS*-side, some information can be gained. When opening such a file in a hex editor, the following binary pattern is present repeatedly in all files: ten `0xFF` bytes followed by two `0x0F` bytes and terminated by two bytes with the value `0xB018`. The pattern likely serves as a header or a separator between different packets. This binary pattern also appears in the firmware, and using *IDA Pro's* cross-referencing

```
1  int write_istp_header_()
2  {
3    void * src = (void *) 0x895A2C18; // address of the
4                                      // 0xFFFF FFFF FFFF FFFF FFFF  0
                                          F0F B018
5                                      // byte sequence
6
7    if ( * (void *) dword_8A01B5E4 == 32 )
8      src -= 48;
9
10   return memcpy_s((void *) 0x82FF9C40, 48, v0, 48);
11 }
```

Listing 4.9: Code copying the presumed ISTP header sequence in the BB.

```
switch (work_mode) {
    // [...]
    case 11:
        result = mbt_mode_ind?(0x87b03610, r1, r2_1);
    case 12:
        result = mipi_mode?(&data_87b03610, r1, r2_1);
    case 14:
        result = mon_dbg_mode?(0x870a2070, r1, r2_1);
    case 15:
        result = etm_mode?(&data_87b03610, r1, r2_1);
    case 0x17:
        result = "Nothing defined for OCT2!\r\nOK";
    case 0x18:
        result = current_settings?(0x870a2070, r1, r2_1)
    case 0x19:
        result = at_history?(0x870a2070, r1, r2_1);
}
```

Listing 4.10: A switch parsing the work mode of a AT+XSYSTRACE command.

feature reveals that it is used in at least one function that copies it to a fixed offset in the BB's memory. It can be seen in Listing 4.9. Analyzing the BB further with *IDA Pro* suggested a protocol implemented over shared memory between the BB and *iOS*. In terms of an attack from the BB to the AP, this protocol provides little surface. *iOS* only copies the ISTP "packets" to files but does not parse them in any way.

Further following the reverse call graph for this function leads to a complex handler covering several tracing and debugging related tasks. When comparing strings in that handler with the AT command specifications discussed in Section 2.2.1, it seems likely that this function is an AT command handler. For instance, the handler and functions it often calls print strings ending in "\r\nOK", which is how AT command responses are required to end.

Comparing the handler to the AT commands described in the documentation for the XMM 7160 modem [93], it becomes apparent that the handler is (part of) the AT+XSYSTRACE command. It can enable and configure how the BB emits traces. There are several other AT commands connected to tracing and debugging features, such as AT+TRACE and AT+XSIO. The latter of these is explicitly described as being responsible for configuring the ISTP format.

The document further specifies that other output formats exist, such as "MIPI-1" or "MIPI-2". The Mobile Industry Processor Interface Alliance (MIPI) [4] is an organization that designs and specifies interfaces for mobile devices to encourage the integration of components by different member companies. Most interestingly, they specify a whole suite of tracing and debugging interfaces and protocols.

[4] More information about MIPI is available at www.mipi.org.

```
1  ICE_AWD: IceAwdListener_LtePdschStat bler = %d mcs_index= %d rb_count
       = %d
2  ICE_AWD: IceAwdListener_LtePdschStat sf_count = %d
3  ICE_AWD: IceAwdListener_LtePdschStat bler = %d mcs_index= %d rb_count
       = %d
4  7480.S2.001.32.9.012.1.1228
5  ICE_AWD: awd_mem_free: ptr=%p err_stat=%d alloc_total_blocks=%d
6  ICE_AWD: TASK: message %x:%X received!
7  ICE_AWD: awd_mem_free: ptr=%p err_stat=%d alloc_total_blocks=%d
8  7480.S2.001.32.9.012.1.1228
9  7480.S2.001.32.9.012.1.1228
10 oct sleep cycles 2114
```

Listing 4.11: An excerpt of strings in a *ICE17* ISTP dump.

Investigating the handler for `AT+XSYSTRACE` in terms of support for one of these protocols leads to discovering a switch. An excerpt of which can be seen in Listing 4.10. The `break` statements have been left out to shorten the excerpt. Depending on the variable `work_mode`, different actions are carried out. "Work mode" `12` triggers "MIPI Mode." Thus, the assumption that ISTP can either be switched to a MIPI protocol or can contain MIPI-compatible messages is palpable.

Next, it is possible to find the ARI magic bytes within these files. Interpreting the surrounding bytes as ARI headers appears to yield valid headers at first. Thus, as part of this thesis, a carving tool was written to extract these messages. When analyzing them further in *ARIstoteles* [59], it becomes clear that these messages are not valid. The occurrence of the ARI magic bytes is, therefore, likely coincidental.

ISTP Versions

While it is impossible to fully understand the ISTP format as of writing, the format has likely changed between BB versions. For one, it is not possible to find the header or separator in the ISTP dumps of previous BB generations.

Further, while only the separator sequence may have changed, it seems that also the format of the packets or at least their content has changed significantly. Inspecting dumps that stem from an *ICE17* BB reveals many readable strings, such as the BB modem number, format strings for logs and status messages. Listing 4.11 shows an excerpt from an *ICE17* ISTP dump. Interestingly, some strings seem to log memory allocations, which are particularly important when creating a memory corruption based-exploit. *ICE19* dumps do not include any of this information directly. Therefore, *ICE19* produces a much more opaque ISTP log.

Other Tracing Formats in Connection with ISTP

The following section will shortly give an insight into other tracing protocols that seem to be connected to ISTP. These are either related to *Intel* BBs by strings found in the firmware or by reverse-engineering the tracing protocols related AT command handler.

MIPI System Trace Protocol The MIPI alliance publishes several standards on tracing protocols. Most interestingly, they released a specification named MIPI System Trace Protocol (STP) [74]. It provides several common features for protocols used for traces that might include information such as processor instructions and data flow. However, it is easy to rule out that ISTP is purely based on MIPI STP because the documentation for MIPI STP explicitly mentions that the most extensive series of consecutive one bits is 75 one bits. The separator between ISTP packets has 80 consecutive one bits.

MIPI System Software - Trace Another protocol specified by MIPI is the System Software - Trace (SyS-T) protocol. It can be used in combination with MIPI STP and is geared towards exchanging information between a mobile device and a debugging and testing system. SyS-T is a much more concrete and extensive protocol than MIPI STP. Fortunately, MIPI provides sample code,[5] including a printer that can transform a SyS-T trace to Comma-Separated Values (CSV). However, running this example implementation over several `.istp` files did not yield any results.

Other MIPI Protocols While it can be ruled out that ISTP is identical to any of the previous two MIPI protocols, there are several more candidates. Of those candidates, the MIPI Trace Wrapper Protocol (TWP) [75] is particularly compelling. It can combine several trace sources into a single trace and then output it over a shared interface or store it in a buffer. ISTP may be similar to TWP in that it may combine several different traces into a single output format.

MIPI standards had at least some impact on the BB. While none of those above protocols are directly referenced in the firmware, one is explicitly mentioned. MIPI Parallel Trace Interface (PTI) [73] is a standard for a physical interface to connect a given device to a debugging and tracing system. This standard is also mentioned in the documentation for the *Intel* XMM 7160 modem [93]. Therefore, it is not unlikely that another MIPI specification influenced the ISTP format.

TraceX *TraceX* [71] is part of the *Azure RTOS ThreadX* suite of products. It is a Windows-based analysis tool and comes with its specification for a trace format. A screenshot of the utility is shown in Figure 4.5 with a demo trace file. Since the BB uses *ThreadX* as its RTOS, using *TraceX* would require little effort to integrate with the existing system, in theory. However, trying to load `.istp` files into the program does

[5] The MIPI code is available at: `https://github.com/MIPI-Alliance/public-mipi-sys-t`

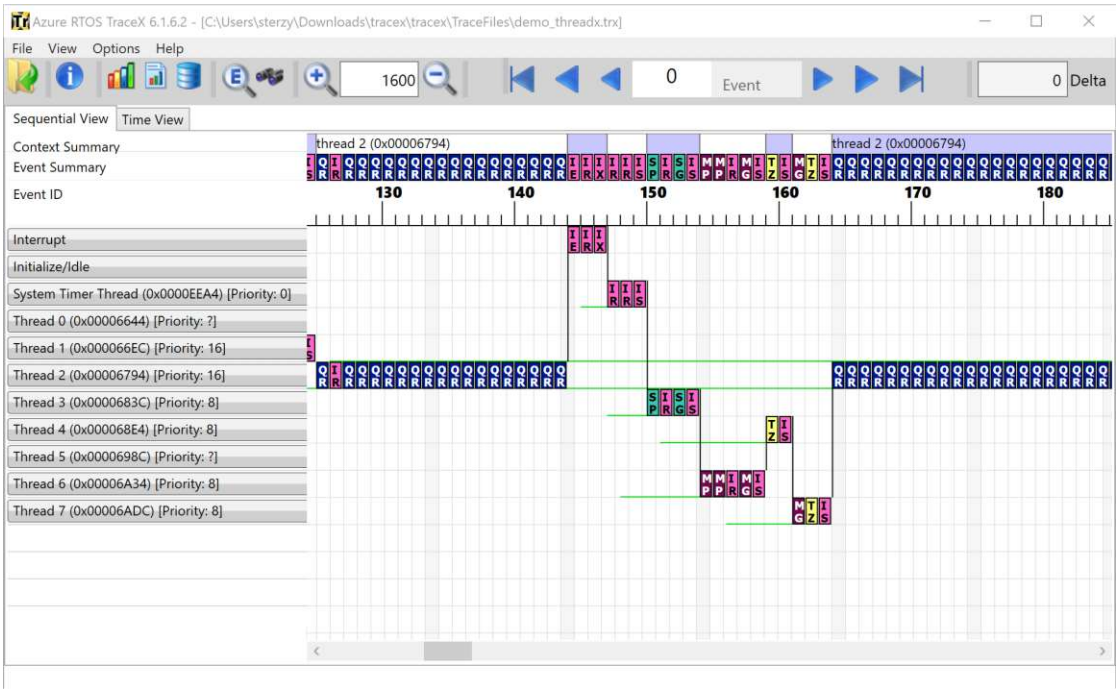

Figure 4.5: A screenshot of the *TraceX* application with a loaded trace file.

not yield any results. Comparing the *TraceX* tracing protocol to ISTP also shows that it is structurally very dissimilar from ISTP. While ISTP is likely not based on *TraceX*, ISTP might be able to encapsulate *TraceX* events.

ARM CoreSight and Embedded Macro Cells Another technology that might be related to ISTP is *ARM*'s *CoreSight* [11] tracing and debugging architecture. More specifically, the non-invasive tracing architecture is based on *Embedded Trace Macrocells* or *ETMs*. These *ETMs* are embedded into processor cores and can provide real-time instruction and data tracing capabilities. This assumption is supported by the availability of an "ETM Mode" in the BB's tracing-related AT command handler. Further, the *CoreSight* architecture also includes a *Program Trace Macrocell* or *PTM*. It implements a program flow trace functionality. While *ICE19* `.istp` files contain only very few strings, a prevalent one is "PTM".

Qualcomm DIAG Finally, something worth mentioning here is that `.istp` files are specific to *Intel* BBs. When using `abmlite` or `DumpBasebandCrash` on a *Qualcomm*-based device, `.diag.qmdl` files are produced. These have a completely distinct structure and are typically much smaller than an ISTP dump. In a talk at *RC3*, Esage [5] presented her efforts at reverse-engineering this protocol.

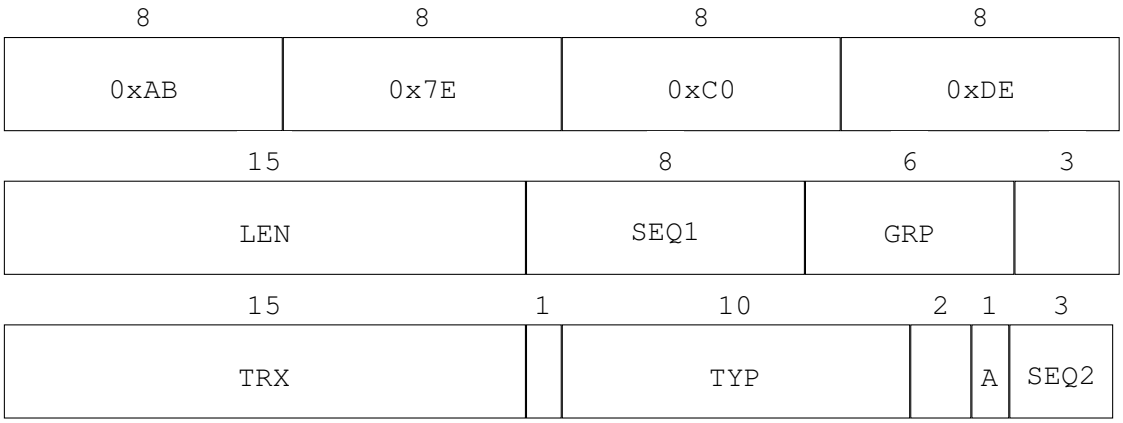

Figure 4.6: The ARI header.

4.8 Apple Remote Invocation

Since ARI has not been analyzed publicly regarding the BB side, it is an exciting target for fuzzing. Additionally, it exemplifies several properties that tend to produce memory corruption issues, such as several fields indicating a packet's length (of parts). This section will first take a brief look at the ARI format and then show how the corresponding parser has been identified. In a later chapter, this handler will then be emulated and fuzzed.

4.8.1 ARI Packet Format

In this thesis, only a quick summary of the ARI format will be given. It will only include parts that are relevant to the topics discussed in Chapter 5. However, throughout this thesis, several contributions to the understanding of ARI were made, such as the concept of groups and the transaction identifier. In the following, a short overview based on work by Kröll et al. [59] will be given. The next subsections will then detail the additional contributions made by this thesis. Since the BB is a 32-bit little-endian system, ARI will be treated as such in this section. Please note that this differs from how *iOS* treats ARI.

Every ARI packet has a header that is twelve bytes long. The format of this header is displayed in Figure 4.6. The first four bytes consist of a fixed value (`0xAB7EC0DE`). They are then followed by a length field (`LEN`) that is 15 bits long. Note that this length refers to the packet's body only and does not include the twelve header bytes.

Further, the ARI header contains a sequence number split over two fields, the first containing eight bits (`SEQ1`) and the second three (`SEQ2`). It is unclear if the sequence number is utilized by the BB as messages with an invalid sequence are sometimes still accepted. The header also includes a six-bit group identifier (`GRP`), a 15-bit transaction identifier (`TRX`), and a ten-bit long type field (`TYP`). Moreover, an acknowledgement flag is included (`A`). `TRX` and `A` were reverse-engineered as part of this thesis. The remaining

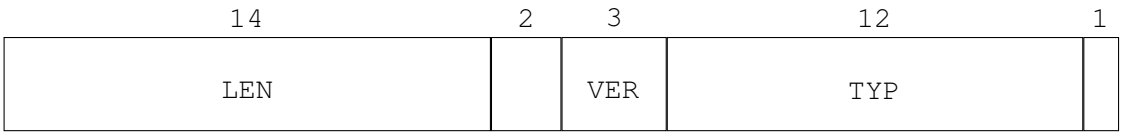

Figure 4.7: The header of a TLV field withing an ARI packet.

bits that are unlabelled in the figure are likely unused in the currently existing ARI implementations.

The body of an ARI packet is made up of Type-Length-Value (TLV) fields, which are explicitly referred to as "TLVs" in the logging strings of the BB. Each TLV starts with a four-byte-header, which is shown in Figure 4.7. It consists of a fourteen-bit long length field (`LEN`). This length field is followed by two bits that are currently unused. Next is a three-bit long version field (`VER`), followed by ten bits of type information (`TYP`). The last bit in the TLV header is again likely unused.

4.8.2 ARI Parsing in the Baseband

The fixed value in the header was used to find the ARI parser in the BB's firmware. Using *IDA Pro's* ability to search for hexadecimal values in a binary file, only four results showed up for the value `0xAB7EC0DE`. Two of which were used within the body of a function, found by the automatic analysis of *IDA Pro*. Analyzing these functions, it became apparent that one was parsing ARI messages. It contained many calls to a logging function and would also log different parts of the ARI and TLV headers. From here on out, this function will be referred to as `extract_ari_msg()` (it can be found at offset `0x8a84d818`). Reconstruction of the ARI format was accomplished by using these logging strings.

ARI Transactions

One field that was identifiable in the ARI header via analyzing the format strings was the transaction identifier or `TRX`. Listing 4.12 shows the call to the ARI logging function with the `extract_ari_msg()` function. The format string for the function call references several fields within the ARI header.

In Listing 4.12 `ari_buf` is an array of 32 bit integers, containing the ARI message that is currently being parsed. The length of the current message has been parsed previously and is stored in the `length` variable. Further, the `group`, `type`, and `trx` are filled with the values for the current group, type, and transaction identifier respectively.

ARI Groups

Further analyzing the surrounding functions and strings within the BB reveals that ARI packets are handled by different functions, depending on their group. Each of these functions first calls the logging function associated with ARI and then `extract_ari_msg()`.

ID	Name	Meaning or Purpose
01	bsp	*Baseband Signaling Protocol*
02	call_cs	Phone calls (Circuit Switched)
03	call_ps	Phone calls (Packet Switched)
04	sms	SMS
05	cbs	*Cell Broadcast Service*
06	ss	*Supplementary Services*
07	net_plmn	*Networking - Public Land Mobile Network*
08	net_rat	*Networking - Radio Access Technology Settings*
09	net_cell	*Networking - Cell Information/Configuration*
10	net_dc_ims	*Networking - Data Channel, IP Multimedia Subsystem (ISM)*
11	sim_access	Access information on the Subscriber Identification Module (SIM).
12	sim_sec	SIM security matters (e.g., entering PIN)
13	sim_tk	SIM toolkit [30]
14	sim_pb	SIM phone book
15	cps	*Common Profile Storage*
16	call_cs_voims	Phone calls (CS) voice over ISM
17	ims_me	*ISM Mobile Equipment Capabilities*
18	sys_res	*System Resources*
19	cls	*Combined Location System*
20	sys_diag	*System Diagnostics*
21	ss_lcs	*Supplementary Services - Location Services*
22	xcc	Clock related
23	trc_info	*Tracing Info*
29	ogrs	Off-Grid Radio Services
31	embms	evolved Multimedia Broadcast/Multicast Service
33	pri	-
34	ice_ipc	*ICE Inter Process Communication*
50	ibi_vinyl	related to the embedded SIM
51	ice_awds	-
60	ice_audio	*ICE Audio*
61	rf_power_sar_nbd	*related to Radio Frequency, Power Management etc.*
62	bspnbd	miscellaneous features (e.g., Debugging, Non-volatile Memory)
63	security	Security

Table 4.4: Overview of ARI groups and their purpose. *Cursive text* indicates that the information is mostly based on the abbreviated name and information from *ARIstoteles* [59].

```
1  uint16_t group = ((uint16_t) ari_buf[1] >> 3) & 0x3F;
2  uint16_t type  = ((uint16_t) ari_buf[2] >> 6;
3  uint32_t trx   = ari_buf[2] >> 17;
4
5  logger_(4, "(%s:%d) Decode  %s(%d-0x%03x) len(%zu) buf(%p) trx(0x%08x
       )",
6          "AriMsg", 269, *(const char **)(v27 + 32),
7          group,
8          type,
9          length + 12,
10         ari_buf,
11         trx);
```

Listing 4.12: A call to a logging function within the ARI handler.

Depending on the result of the parsing function and the group of the ARI message, different actions are carried out. A complete list of ARI groups and their purpose can be seen in Table 4.4.

Over the years, ARI remained relatively stable. Only a few groups have been added or removed since its introduction with *ICE16*. Groups 01 to 23 are present across all BB generations, including *IBIS18*. This fact also applies to Groups 33, 34, 50, 51, and 60 to 63. *ICE16* and *ICE17* do not have handlers for Group 31, which is related to the *evolved Multimedia Broadcast/Multicast Service* [33].

ICE19 is the only BB that supports group 29, related to the *Off-Grid Radio Service* or *OGRS*. *OGRS* was an attempt to provide *iPhone* users with the capability to text and make phone calls without a cellular network or Wi-Fi [63]. Instead, phones would connect directly. Given that ARI groups such as group 29 were added later on, it is likely that other missing groups, such as groups 23 to 28, are reserved for later use.

ARI and AT Commands

While ARI replaced AT commands in the case of *Intel* BBs, AT commands still impacted the design of ARI. There are message types within ARI that are successors of AT commands. Such as `CsiXsioSetReq`, which is based on the `AT+XSIO` command. `AT+XSIO` was used in previous *Intel* modems to configure tracing [93]. Similarly, using *ARIstoteles* [59] shows that `CsiXsioSetReq` is part of ARI group 23 called "`trc_info`" or "tracing info."

Secondly, according to certain standards such as the specification for the "AT command set for User Equipment (UE)," [31] some AT commands can be forwarded to the SIM directly. Strings within the BB indicate that ARI supports sending AT commands towards the SIM via ARI group 13 (`sim_tk`). These strings belong to the ARI type `IBISimTkProactiveCmdIndCb` according to *ARIstoteles* [59], which includes TLV types such as `run_at_command_t18`.

ARI as an SDK

It turns out that ARI is implemented in the form of a Software Development Kit (SDK). This fact means that the parser in the BB is based on the same source code as the AP side. The SDK is split into two parts: `ari_host` is used by *iOS* in the `libARI.dylib` and `libARIServer.dylib` libraries, and the BB utilizes `ari_remote`. Knowledge of this matter is helpful because `libARI.dylib` and `libARIServer.dylib` contain symbols that allow much easier reverse-engineering. Using this information sheds light on the otherwise obscure nature of the BB's firmware.

Sadly this fact was realized relatively late into the process of statically reverse-engineering the BB. Thus, some of the results will be presented in the way they were originally uncovered in the following. For instance, the `extract_ari_msg()` function's canonical name is `AriMsg::AriMsg()`.

Function Parameters

In order to be able to emulate the `extract_ari_msg()` function, it is necessary to understand the input parameters to the parsing function. According to *IDA Pro*, parameters are stored in the `R0` to `R2` registers in conformance with standard *ARM* calling conventions. Due to their usage throughout the handler, the second parameter is revealed to be a pointer. It points to a buffer, which should contain the ARI message. The third parameter is the length of this buffer, i.e., the length in the ARI header plus twelve for the header bytes.

Using *IDA Pro's* cross-referencing feature revealed that the first parameter is an array of four-byte long values. Different array entries are used throughout the handler to store error codes or meta-information about the ARI message, such as its length. Different calls to the parser pass different lengths of arrays. It is likely used to pass metadata back to the calling function by reference. The resulting function signature can be seen in Listing 4.13.

```
1 int * extract_ari_msg(uint32_t *obj, uint32_t *buf, uint32_t len)
```

Listing 4.13: Signature of the function that parses an ARI messages.

Memory Management

The ARI parser needs to allocate and free memory dynamically on the heap. Since the goal is to emulate solely this parser, setting up memory handling for the parser is necessary. To do so, functions within the parser that handle heap memory management have to be identified. Throughout this thesis, several approaches to overcome this obstacle have been used.

During the development of the emulation-based fuzzer, parsing a valid ARI message would lead to crashes of the emulator always at the same address. Inspecting the code that caused the crashes in *IDA Pro* showed that the instruction in question was an `MRC`

instruction. It moves a Register from a co-processor to the primary processor, in this case from co-processor 15.

On *ARMv7*, co-processor 15 or CP15 is used for many tasks and often for memory management tasks [10]. Further inspecting the execution path leading to this instruction revealed an allocation function at address `0x8a8873a8`. It calls a logging function with a format string that showed which function parameter specifies the length for the buffer to be allocated. Additionally, it would log the string "MemAlloc," further confirming that this function was used to allocate memory. Thus, to overcome emulation issues and use *BaseSAFE* [67] to its fullest extend, this function is hooked via *Unicorn* as described in Section 5.2.3.

The signature of this function can be seen in Listing 4.14. It takes two parameters. The first is a reference to an `AriOsa` object. However, in the BB, the argument is mainly replaced by the static value `2`. More importantly, the second parameter is the length of the buffer that needs to be allocated.

```
1 int * AriOsa::MemAlloc(AriOsa * this, int len)
```

Listing 4.14: Signature of the memory allocation function used by ARI.

Another dynamic memory-related function could be found by comparing the ARI handler in the BB to the one available in `libARI.dylib`. It frees a list of TLV fields. For example, if certain allocations operations failed or the length value in a given TLV exceeds the length of the entire ARI message. Its signature can be seen in Listing 4.15.

```
1 void AriMsg::FreeTlvList(char* arg1)
```

Listing 4.15: Function signature of `AriMsg::FreeTlvList()`.

4.8.3 libARI and libARIServer

On *iOS*, ARI is implemented by two dynamic libraries, `libARI.dylib` and `libARIServer.dylib`. While `libARI.dylib` does most of the heavy lifting when it comes to parsing, `libARIServer.dylib` is responsible for managing the actual communication with the BB. One of the runtime dependencies of `libARIServer.dylib` is `libARI.dylib`. Moreover, `CommCenter` uses both libraries, and when hooking the `CommCenter` with FЯIDA, function from both libraries can be called.

One discovery made by investigating `libARI.dylib` on iOS 14.3 is the acknowledgement option. It contains several methods connected to the `AriMsg` object, which parse different fields from a given ARI message. One such method is the `AriMsg::GetBufAckOpt()` with the symbol `_ZN6AriMsg12GetBufAckOptEPKhm`, which parses the acknowledgement option.

Thanks to prior work by Classen [22], several functions relevant to this thesis have been previously identified. The `SendRaw()` function can send raw binary buffers from *iOS* to

the BB. Its full signature can be seen in Listing 4.16. The first parameter is a pointer to a buffer containing an ARI message. The second parameter is the length of the message, and the third contains a kind of destination. It is not entirely clear what this parameter represents. However, when analyzing `libARIServer.dylib`, the value passed through this parameter is first used in a function call containing the string "AriMsgTO." In practice, this value was intercepted by hooking the `SendRaw()` function. It turns out that while this parameter is not static, there are only a handful of values that are used by `libARIServer.dylib`.

```
1 uint64_t AriHostRt::SendRaw(char *buf, uint32_t len, uint32_t ariTo)
```

Listing 4.16: Signature of the `SendRaw()` function.

Secondly, all ARI messages coming from the BB pass through the `AriHostRt::InboundMsgCB()` [22]. While this function is of lesser importance to this thesis, it can collect ARI messages alongside `SendRaw()`, serving as an initial corpus for fuzzing. The signature for this callback can be seen in Listing 4.17. It takes two parameters: `buf` contains a pointer to the buffer containing the ARI message and, `len` which is the length of the buffer.

```
1 AriHostRt::InboundMsgCB(char *buf, uint32_t len)
```

Listing 4.17: Signature of the `InboundMsgCB()` function.

Now that functions that send, receive, and parse ARI messages are known within *iOS* and the BB, the idea is to analyze the BB in terms of security vulnerabilities via ARI. After all, the general structure of the protocol is complex, and all necessary parameters for fuzz-testing are known. Thus, it is likely that fuzzing might help find vulnerabilities quickly. The next chapter implements several fuzzers to accomplish this goal.

CHAPTER 5

Baseband Security Analysis with Fuzzing

This chapter mainly discusses the two types of fuzzers implemented throughout this thesis. First, a short overview of the chapter is presented in Section 5.1. The following two sections first detail the emulation-based fuzzer in Section 5.2 and then the in-process fuzzer in Section 5.3.

5.1 Overview

In this thesis, demonstrates two approaches to fuzzing the Baseband (BB). First, an approach based on the *Unicorn* [80] bindings provided by *BaseSAFE* [67] is shown. After having identified the Apple Remote Invocation (ARI) parsing function in the BB, it implements an emulation of the handler. Two approaches to fuzzing this emulation are used. First, American Fuzzy Lop Plus Plus (AFL++) [36] is used in *Unicorn* mode. Secondly, a custom multi-threaded fuzzer is created to fuzz ARI specifically.

The second approach to fuzzing the BB this thesis demonstrates is using FЯIDA to hook functions in `CommCenter`. From there, two different approaches are taken. Either ARI messages sent from *iOS* to the BB are modified before they are sent, or the fuzzer generates ARI messages and send them to the BB.

As a target for the fuzzers, the ARI handler within the *ICE19* BB was chosen for several reasons. ARI is a relatively complex protocol with characteristics that could lead to memory corruption bugs, such as several length fields. Secondly, ARI has not been publicly analyzed before, making it more likely to still contain vulnerabilities. Furthermore, *ICE19* is the latest generation of *Intel*-made BBs and contains some of the most advanced mitigation techniques (see Section 4.6). While an *ICE19* BB might crash quickly if it encounteres malformed input, an older BB might act on it. This would make

it more susceptible to attacks. Thus, exploits that work on *ICE19* are more likely to work on older generations than the other way around.

5.2 Emulation-based Fuzzing

This section first provides an overview of the requirements for the different emulation-based fuzzers in Section 5.2.1. Then, in Section 5.2.2, a summary of the fuzzers' architecture is given, and the fuzzers' implementation will be described in Section 5.2.3. Finally, in Section 5.2.4, a quick introduction is given to how the fuzzers can be used and how the emulation can aid with understanding crashes in the ARI handler.

5.2.1 Requirements

This section lays out several requirements that the emulation-based fuzzers will aim to fulfill. While most of these specifications apply to both fuzzers, the custom and AFL++-based fuzzer, some only apply to the custom fuzzer.

Fast Fuzzing Speeds Fuzzing, in its essence, is a brute force search through the space of all possible inputs to a given program. In this case, all possible ARI messages can be sent to the BB. Since the variable part of ARI messages can be $2^{15} + 8$ bytes long, the search space is at least $2^{(2^{15}+8)*8}$ messages big. A search is successful if a given message causes a crash. Luckily, it is not necessary to search the space exhaustively. Thus, fuzzing speeds of several execution cycles per second are the goal for the fuzzer. This goal is in line with speeds reached by *BaseSAFE* [67], which reached around 1.5 thousand fuzz cases per second on an *Intel* i7-6700 @ 3.40 GHz processor on a single core.

Coverage-guided Fuzzing The fuzzers should collect coverage data during fuzzing and use it to maintain the corpus that will be fuzzed. Coverage data can also help to distinguish crashes. If two inputs that cause a crash have the same execution path and the cause of the crash is identical, then the fuzzer found the same crash twice instead of two distinct issues. Avoiding duplicate crashes decreases the time spent on analyzing the crashes after the fuzzer found them.

Parallelization Fuzzing is inherently parallelizable, and modern hardware encourages the use of parallelized tasks. By running several instances of the program that is being fuzzed at once, performance can be improved dramatically. This thesis will heavily use parallelization within the custom fuzzer and reach the goals stated above.

ARI-aware Input Generation (Custom Fuzzer Only) The custom fuzzer should generate ARI messages that are likely to reach more complex states in the parser by using the information gained during static analysis of the BB. Leveraging information about the ARI format should help avoid failing trivial checks by the parser. For instance, fuzzing the first four magic bytes of the ARI format does not yield any results. This fact

is correct because the parser in the BB checks these four bytes before doing anything else. If the check fails, it returns. Thus, fuzzing these four bytes has no effect. Leveraging knowledge like this, the custom fuzzer should improve fuzzing effectiveness.

5.2.2 Architecture

In this section, the architecture of the two emulation-based fuzzers is stated. Both utilize the same emulation setup to then use two different fuzzing strategies. While the AFL++ fuzzer essentially follows the same structure as the one outlined in the *BaseSAFE* paper [67], the custom fuzzer uses a more involved architecture.

Emulation Setup

At the core of both fuzzers is the emulator running the BB's ARI parser. A common library is used to parse the `SYS_SW.elf` file. It then creates a *Unicorn* [80] instance and maps the Executable and Linkable Format (ELF) segments into the emulator's memory. Finally, it uses the sanitized heap implementation from *BaseSAFE* [67] to set up a heap and further allocates some memory for the stack.

After initializing the emulator, several hooks are set to modify its behavior or collect metadata about the emulation run. Finally, the fuzzed input will be provided to the parsing function, and the function will be called. The emulation will then exit for one of the following reasons:

- A crash occurs, indicating that either unmapped memory was read or written.
- The emulation will timeout. Each run will last at most 10 seconds. If the handler needs more time to parse the message, this will be detected as a hang and the execution will be aborted.
- Everything seems to have worked out, and the handler reaches its end.

Upon exiting, the reason will be provided to the fuzzer, which will take appropriate action. This common emulation core will also be used in a separate tool to replay an ARI trace. By doing so, additional information can be presented that would otherwise slow down emulation.

AFL++-based fuzzer

Additions for the AFL++ fuzzer are relatively minor. It must accept a fuzzed input via an input file to call the program with the `afl-fuzz` utility. Thus, a callback needs to be provided, which maps the contents of said file to the emulator's memory. Further, AFL++ needs to know when a crash occurred or if the handler has exited legitimately. Another callback for validating crashes and an array of exit addresses can be provided to AFL++ to accomplish this.

All of this information can be passed to AFL++ via one single function call. Additional parameters to this function call specify whether the crash validation callback should be called every time the emulator exists or only upon encountering crashes. Finally, it is possible to specify how often the program should be called without spawning a new child process.

Using these bindings, AFL++ will then provide its coverage collection instrumentation. It will also be possible to run several instances of AFL++ in parallel. Thus, all requirements outlined in Section 5.2.1 are handled mainly by AFL++ itself.

Custom Fuzzer

Rust [88] is chosen as the language to implement the custom fuzzer for several reasons. First, *Rust* helps to accomplish the fuzzing speed requirement because it does not incur any overhead due to the need for a runtime or garbage collection. Secondly, *Rust* enforces thread safety, making it much easier to create a multi-threaded program. Finally, all the necessary libraries already exist for *Rust*. For instance, *BaseSAFE* [67] provided bindings not only for AFL++ but also for *Unicorn* [80] in general.

The custom fuzzer will use simple block coverage instead of AFL++'s more detailed edge coverage. During the fuzzing run, the starting address of each block will be collected. These addresses will be hashed with a fast 32-bit variant of a Cyclic Redundancy Check (CRC) function to derive a single value representing one execution path. This function was selected because of its efficiency. The fuzzer should spend as little time as possible on calculating this hash.

Achieving parallelization in the custom fuzzers is done by using threads. Managing the communication between different threads is accomplished using channels. The idea is to spawn several worker threads, which will execute the emulator. Workers should receive their input from a generator thread. After testing an input, the emulation result alongside coverage data is sent to a triaging thread. It decides whether the input should be considered for further fuzzing and whether a crash is indeed a new crash. The input that caused a new crash will then be stored in an output directory. It will also be added to the existing corpus used for input generation by the generator thread.

A rough outline of this architecture can be seen in Figure 5.1. Rectangles with a solid outline represent the three different kinds of threads. Dashed outlines, on the other hand, symbolize data structures.

As a mutation strategy, a simple bit-flipper is implemented first as a proof-of-concept for our custom fuzzer. The intention was to create a more elaborate fuzzing strategy later. While this strategy was implemented, for various reasons, it could not be integrated with the custom fuzzer. A more detailed description of the envisioned mutation strategy can be found in Section 5.2.3.

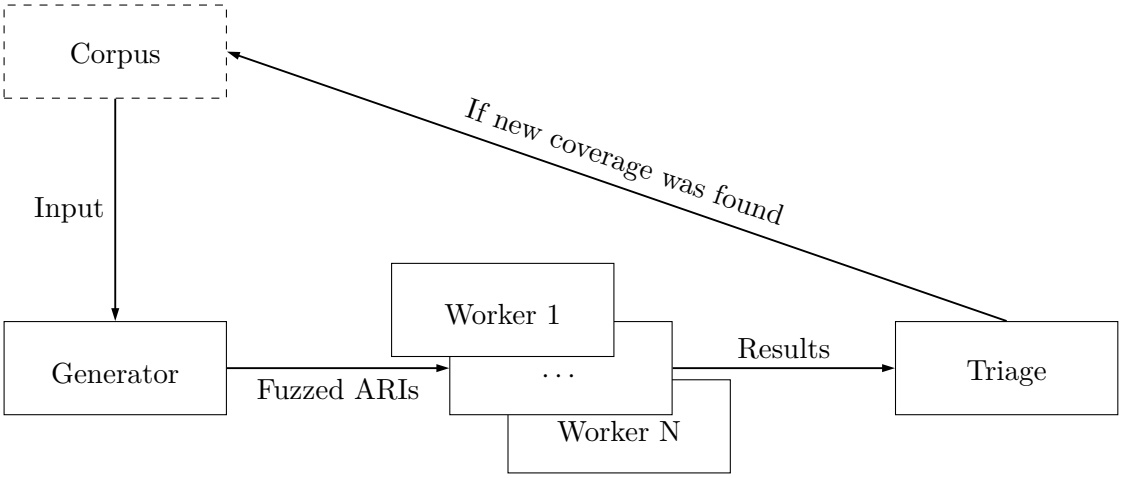

Figure 5.1: A simplified view of the custom fuzzer's architecture.

5.2.3 Implementation

Implementation details of the emulation-based fuzzers are given in this section. The memory mappings needed for the emulator are described to start. It then continues to summarize hooks needed to make the emulation work. Details about the custom fuzzer's implementation in regards to multithreading and coverage collection are given.

Memory Mappings

The first step towards emulating the ARI parser is to map code segments from the BB's firmware into the emulator's memory. The `SYS_SW.elf` file is parsed via *libgoblin* [65]. A *Unicorn* instance is set up based on the architecture and endianness specified in the ELF file.[1]

Next, the `LOAD` segments specified in the ELF headers are mapped into the memory of the emulator. The ELF specifies whether a given segment should be readable, writable, or executable. By respecting this information, the emulator should be able to detect illegal writes and reads quicker. Moreover, vulnerabilities found in the emulator are more likely to also be present in the real BB.

`LOAD` segments will be mapped to the addresses specified by the ELF file to make analysis easier. Since tools like *IDA Pro* [53] or *Ghidra* [3] respect this information, addresses in the emulator and these disassemblers will match up. Therefore, finding the corresponding instructions in *IDA Pro* or *Ghidra* is easy if a crash occurs in the emulator at a given address. There are two exceptions to this mapping rule: Firstly, addresses and sizes of memory segments must be 4 kB aligned. Otherwise, *Unicorn* will return an error. Secondly, if a memory segment overlaps with one already mapped into the emulator's

[1] While we discussed in Section 4.3.1 that information from an ELF header is not always accurate, the processor architecture is correct in the case of *ICE19*.

```
let skip_hk = move |mut em: UnicornHandle<'_, _>, addr, _| {

    // read the return address from the link register
    let lr = em.reg_read(RegisterARM::LR as i32).expect("...");

    // set the program counter to the value to the return address
    // exiting the current function call immediately
    em.reg_write(RegisterARM::PC as i32, lr).expect("...");
};

// set hook up so that it is executed as soon as the logger
// is called
emu.add_code_hook(0x8A835280, 0x8A835280, skip_hk).expect("...");
```

Listing 5.1: *Unicorn* hook to skip a function call.

memory, it will be ignored. A consequence of these exceptions might be the absence of some segments that the firmware would typically expect.

The fuzzer should use the sanitized heap from *BaseSAFE* [67] to detect double frees, use-after-free, and out-of-bound reads and writes on the heap faster. However, integrating it with the emulation setup is not straightforward. The starting address of the heap depends on where the `LOAD` segments are placed. It should be placed on top of all the code sections. Since the segments are mapped at runtime, this address is also only known at runtime. Mapping the segments, of course, already requires an instantiated emulator. The problem is that *BaseSAFE*'s bindings do not offer a way to add a sanitized heap to an existing emulator instance and require specifying the address when the emulator is instantiated.

Thus, a workaround is needed to integrate the heap properly. An empty heap is allocated and then used to instantiate the emulator to overcome this issue. After all `LOAD` segments have been allocated, a memory area for the heap is allocated. This area is one mebibyte in size. Then another instance of a sanitized heap is created. It uses a copy of *BaseSAFE's* `heap_unalloc` hook to detect invalid access to the heap's memory. Finally, the empty heap in the emulator is replaced by the freshly instantiated heap.

Lastly, one mebibyte of space is allocated for the stack. In the BB, the stack grows from the highest to the lowest address. Thus, the `SP` register, which stores the stack pointer, is set to the highest address within the allocated memory region.

Function Patching Hooks

Certain functions within the BB rely either on hardware that the emulator cannot simulate or on state information not present when invoking the ARI parser directly. The behavior of these functions has to be modified to make the emulation work. A skip hook was implemented to delete specific function calls from the handler. It can be seen in

```
let malloc_hk = move |mut em: UnicornHandle<'_, _>, addr, _| {
    // read the size of requested buffer from the R1 register
    let size = em.reg_read(RegisterARM::R1 as i32).expect("...");

    // allocate the buffer
    let buf_ptr = uc_alloc(&mut em, size).expect("...");

    // set the return value to the address of the buffer
    em.reg_write(RegisterARM::R0 as i32, buf_ptr).expect("...");

    // and return
    let lr = em.reg_read(RegisterARM::LR as i32).expect("...");
    em.reg_write(RegisterARM::PC as i32, lr).expect("...");
};
```

Listing 5.2: Memory allocation hook for the `AriOsa::MemAlloc` function.

Listing 5.1. Essentially, it will return instantly from the current function whenever it is called.

The example in Listing 5.1 also shows how the hook is used to skip the logging function used by the ARI handler. By hooking the address of the first instruction associated with the logger, it is never executed. This hook is set for two reasons. First, the logger does not need to be executed while fuzzing because it does not affect the parsing function. Secondly, it requires proper dynamic memory management. Every memory allocation or deallocation function within the logger must be identified and hooked to make this work. While string formatting functions, such as the logger, can contain problematic allocations and de-allocations skipping the logger is easier and more likely to accomplish the goal of finding a crash within the ARI parser itself. Moreover, format string vulnerabilities are likely easily identified via static reverse engineering. For instance, by checking whether the provided format string is a constant or a variable value.

Besides the logger, another category of functions that need to be hooked is functions within the ARI handler that allocate and deallocate memory. These functions need to be overwritten with their equivalents from *BaseSAFE* [67] to utilize the sanitized heap. *BaseSAFE* offers the `uc_alloc` function to allocate memory and the `uc_free` function to deallocate it. By using these helper functions, additional hooks are placed on the memory regions of the heap. If memory is accessed illegally, these hooks will trigger a crash of the entire program.

As discussed in section Section 4.8.2, the ARI handler uses a memory allocation function located at `0x8a8873a8`. A hook was implemented to integrate this function with the sanitized heap. It can be seen in Listing 5.2. The hook simply reads the buffer length from the `R1` register. Then it allocates a buffer of the corresponding size with `uc_alloc`. It sets the return value to the address of the buffer. Finally, it returns from the allocation function before any of its instructions are executed.

```
// instantiate a linked list to collect coverage
let path = Rc::new(RefCell::new(LinkedList::new()));
let mut prev_addr = 0;
let mut_path = path.clone();

let tracing_hk = move |_em: UnicornHandle<'_, _>, addr, _| {
    // borrow the list dynamically and push the address
    // of the current block and its size onto it
    mut_path.borrow_mut().push_back((addr, prev_addr.clone()));
    prev_addr = addr;
};

// set the hook
let tracer = emu.add_block_hook(tracing_hk).expect("...");

// run the emulator
let result = emulate_ari(emu, input.clone());

// remove the hook from the emulator
emu.remove_hook(tracer).expect("...");

// borrow and clone the list
let trace = path.borrow().clone();

// return the values
(result, trace, input.clone())
```

Listing 5.3: The coverage data collection hook.

Coverage Collection in the Custom Fuzzer

Collecting coverage information in the custom fuzzer is done by using the block hook provided by *Unicorn*. As is shown in Listing 5.3, a `LinkedList` is wrapped in an `Rc` and a `RefCell` struct to enable dynamic borrow rules, ensuring memory safety at runtime. The `LinkedList` is then used by the hook to store a tuple consisting of the starting addresses of the current and previous blocks. Finally, the the LinkedList's content is cloned to avoid mutability issues between the hook and the rest of the program. It is then returned alongside the emulation result.

The fuzzer will then iterate over the `LinkedList` and create a hash value based on the starting address of the blocks. This value will then be considered as representing the coverage of a single run. By comparing this hash to previous hashes, the custom fuzzer decides if a new crash was already encountered before or not. If the crash was not encountered before, the input will be added to the corpus considered for further mutation. Moreover, if the same crash was encountered before, but the new input is shorter than the old, it will also be added to the corpus.

Even though basic block coverage is used to decide whether an input should be kept or not, for comparison reasons, edges are also collected. The tuples in the `LinkedList` each represent an edge. These are then shifted and XORed as described by the American Fuzzy Lop (AFL) whitepaper [103] and collected in a `HashSet` to maintain a collection of unique edges encountered throughout the run.

Multithreading in the Custom Fuzzer

While *Rust* offers several features that encourage multithreading, communication between threads via channels is not among them. Within the custom fuzzer, *crossbeam* [23] is used to provide this functionality. Upon starting the fuzzer, at least three threads are started, the generator thread, one or more worker threads, and a triaging thread. They have the following purposes:

Generator Thread By using a mutation strategy, it generates new inputs. In a first step, it simply flips bits in a given ARI message at random. After generating a new input, it adds the result to a bounded channel. This channel will store at most 500 fuzzed inputs. Limiting this channel is necessary because otherwise, the fuzzer will not terminate promptly. The current naive strategy is faster than the worker threads so that the channel will grow uncontrollably. Thus, when requesting the fuzzer to exit, it will first try to emulate all fuzz cases left in the channel. In other words, the longer the fuzzer runs, the longer it will take to exit. Worse, the channel could run out of memory and crash the entire fuzzer unpredictably.

Worker Threads A worker thread sets up an emulator instance and then waits for the input from two different channels. The first channel contains the mutated inputs from the generator threads. These are mapped into the emulator's memory, and then the emulator is executed. After emulation has concluded, the coverage data alongside the emulation result are sent to the triaging thread through another channel. The second channel is a `tick` channel, which outputs an event every second. Upon receiving such an event, the fuzzer frees its current emulator instance and instantiates a new one. If this reset is not present in the custom fuzzer, the performance will decrease drastically over time. It is unknown why that happens, but creating a new emulator instance every second fixes this issue.

Triaging Thread Upon receiving the output from a worker thread, the triaging thread evaluates the coverage data and emulation results. If a crash is encountered with a new execution path, the corresponding input is stored in an output folder. Furthermore, the triaging thread also listens to a `tick` channel that outputs an event every 2.5 seconds. When it receives such an event, the thread will print a message that contains statistics about the current fuzzing run. Finally, the statistics are also written to a Comma-Separated Values (CSV) file to enable straightforward data collection for benchmarking purposes.

Signals are used to exit the fuzzer. Upon receiving a `SIGINT` signal for the first time, the fuzzer stops generating new inputs for the worker threads. It also closes the input channel to the worker threads. As soon as they have finished parsing all fuzzed inputs left in the generator thread channel, they will exit. The triaging thread behaves the same way, processing all outputs still provided by the workers and then exiting. Finally, if another interrupt signal is received, the fuzzer will forcibly shut down.

ARI Generation

Currently, the custom fuzzer simply flips bits in a provided ARI message. However, as part of this thesis, several versions of an ARI generator have been implemented. The first *Rust*-based version was implemented for the custom fuzzer. It was supposed to be used as a more advanced input generation strategy, but sadly it was not possible to integrate it into the custom fuzzer yet. Using this generation strategy would have meant needing to re-write the sanitized heap provided by *BaseSAFE*. However, it did inspire the generator used with the in-process fuzzer and, thus, will be described in the following.

A Linear Congruential Generator (LCG) with a period length of 2^{64} is used to generate a lot of random bytes quickly. The output of this LCG is then split and interpreted as two new seeds for an LCG with a 2^{32} period. One LCG generates the header for an ARI message, and another generates Type-Length-Values (TLVs) for the body of the ARI message.

When generating a new ARI message, the generator will first generate a random ARI header by calling the 32-bit LCG twice. It will then interpret the received data as an ARI header and parse its length. It will call a TLV generator until the entire length of the message is filled with TLVs to complete the message.

Generating a TLV works similarly to generating the ARI message. First, the 32-bit LCG is called with the seed for generating TLVs. These four bytes are then interpreted as the TLV header. The remaining length of the TLV is filled with bytes from the LCG.

There are two cases in which the length of the TLV will be modified. First, the randomly generated length of a TLV might exceed the remaining length of the ARI message. Secondly, after appending the currently generated TLV, the remaining length of the ARI message might be four bytes or less. Thus, there would not be enough space to append another TLV. In both cases, the TLV's length will be extended to fill the remaining length of the ARI packet.

5.2.4 Applications

The fuzzers and emulation setup implemented in the previous section have two purposes. Most importantly, to fuzz the ARI handler in the BB and uncover security vulnerabilities. A secondary purpose is to better understand how and why the ARI parser can and does crash. Since we lack introspection into the BB, the emulation can give a deeper look

```
[*] Loading corpus and checking it...
[*] Emulating "../../resources/corpus/0x00000004-0x0000b48f.ari"...
[*] Time: 2s            Execs: 6839            FCS: 3419.50           Corpus size: 49                Edges: 88
[*] Time: 5s            Execs: 14667           FCS: 2933.40           Corpus size: 49                Edges: 88
New crash encountered, writting ari message to "output_dir/READ_UNMAPPED-9fae6a04.ari"
New crash encountered, writting ari message to "output_dir/READ_UNMAPPED-9fb274fe.ari"
New crash encountered, writting ari message to "output_dir/READ_UNMAPPED-026cade8.ari"
[*] Time: 7s            Execs: 21101           FCS: 3014.43           Corpus size: 53                Edges: 100
[*] Time: 10s           Execs: 28863           FCS: 2886.30           Corpus size: 54                Edges: 100
New crash encountered, writting ari message to "output_dir/READ_UNMAPPED-3cd481a9.ari"
[*] Time: 12s           Execs: 35498           FCS: 2958.17           Corpus size: 56                Edges: 107
New crash encountered, writting ari message to "output_dir/READ_UNMAPPED-d09b000d.ari"
[*] Time: 15s           Execs: 43079           FCS: 2871.93           Corpus size: 58                Edges: 107
[*] Time: 17s           Execs: 49542           FCS: 2914.24           Corpus size: 58                Edges: 107
[*] Time: 20s           Execs: 57249           FCS: 2862.45           Corpus size: 60                Edges: 107
New crash encountered, writting ari message to "output_dir/READ_UNMAPPED-80a7d8c3.ari"
[*] Time: 22s           Execs: 63453           FCS: 2884.23           Corpus size: 61                Edges: 107
New crash encountered, writting ari message to "output_dir/READ_UNMAPPED-cddbf003.ari"
New crash encountered, writting ari message to "output_dir/READ_UNMAPPED-f34b1425.ari"
[*] Time: 25s           Execs: 71112           FCS: 2844.48           Corpus size: 66                Edges: 107
```

Figure 5.2: The custom fuzzer running with four worker threads.

into how memory and register values change during execution. In the following, a short introduction to the use-cases of the developed software will be given.

By adapting the emulation setup for AFL++, a sophisticated fuzzer can be run relatively quickly. Ideally, several instances of AFL++ are run in parallel to fully utilize the underlying hardware. AFL++ has a set of recommendations [2] to optimize fuzzing runs, which should be respected for ideal results.

Figure 5.2 shows the custom fuzzer running with four worker threads. During a fuzzing run, the fuzzer reports the total time the fuzzer has been running, the number of times the ARI handler has been executed, and the current average fuzz cases per second. Further, when a crash is encountered, it outputs a corresponding message.

Finally, the emulation setup was used to implement a separate program that would simply replay a given ARI message to the emulator. It would then report whether the emulator crashed and what the cause of the crash was. Optionally a trace of basic blocks can be printed. A hook can be added that prints the top five entries of the stack, or a hook from *BaseSAFE* [67] can be added to understand register values and current instructions.

5.3 In-Process Fuzzing

In addition to the emulation-based fuzzers, two in-process fuzzers were developed, an in-place and an injection fuzzer. The requirements for these fuzzers are stated in Section 5.3.1. Then, an overview of their architecture is given in Section 5.3.2, followed by a description of their implementation in Section 5.3.3. Lastly, their application for fuzzing the BB is detailed in Section 5.3.4.

5.3.1 Requirements

The requirements for the in-process fuzzers differ somewhat from the emulation-based fuzzers. One crucial difference between the in-process fuzzers and the emulation-based fuzzers is that no coverage data can be collected. In the following, the requirements are described in more detail.

Fast Fuzzing Speeds Similar to the emulation-based fuzzer, it is essential to mutate messages quickly. In a talk at *RC3*, Classen [22] showed that a bit flipping in-process fuzzer for Short Message Service (SMS) could achieve about 22000 fuzz cases per second on an *iPhone* 8. Since the fuzzer described in this section is more intricate than solely flipping bits, the goal is to reach about 10000 fuzz cases per second on an *iPhone* SE 2020. This goal only applies to the injection fuzzer because the in-place fuzzer does not actively insert messages.

ARI-aware Input Generation ARI-aware input mutation and generation are necessary to explore the BB efficiently. Avoiding trivial checks within the BB is pertinent to reach more complex bugs within the BB. Thus, knowledge achieved during static analysis should be used to improve input mutations.

Deterministic Input Generation Since the in-process fuzzer lacks introspection into the BB, coverage collection is not an option. While this means that the fuzzer is less effective, it can also be used as an advantage. Ideally, the fuzzer would generate an entirely deterministic series of inputs if provided with a corpus and a random seed value. This requirement enables it to only store this seed value and the initial corpus before fuzzing. Generated inputs themselves do not need to be stored as they can be re-created given the initial corpus and the corresponding seed.

Minimal Serialization In-process fuzzers consist of two components: the in-process harness and an outside manager running on a different device. This manager component can handle input generation and then forward them to the harness, or the harness sends inputs to the manager to store them. However, for this communication, the inputs need to be serialized and de-serialized by FЯIDA. This (de-)serialization is one of the most significant slow-down factors for in-process fuzzers [22]. Thus, minimizing the need for serialization is essential.

5.3.2 Architecture

This section gives an overview of the general architecture of the in-process fuzzers. Further, the differences between the in-place and injection fuzzer are given. Lastly, adaptions for the standalone mode of the injection fuzzer are given.

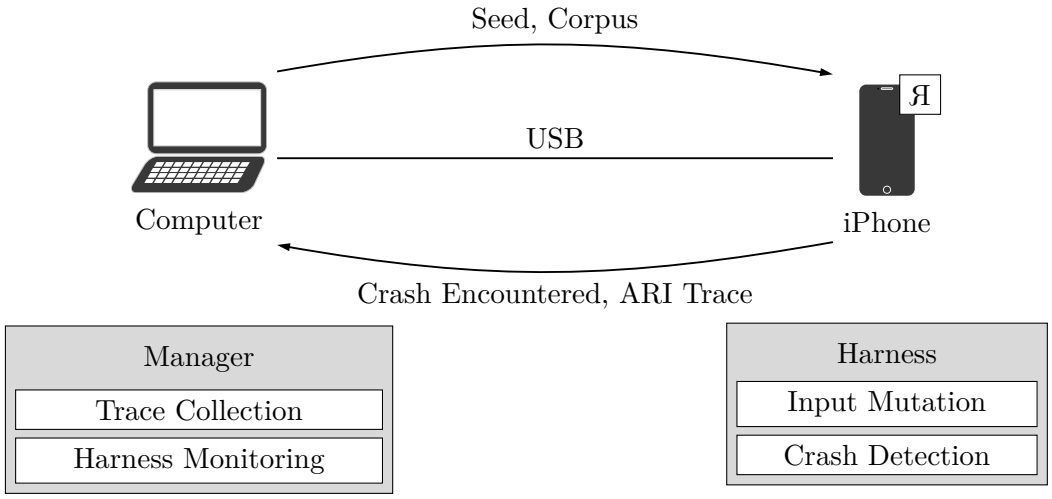

Figure 5.3: A simplified view of the in-process fuzzers' general architecture.

General Architecture

In general, the in-process fuzzers follow the same structure as *ToothPicker* [49], consisting of a manager and a harness. One main difference is that none of the in-process fuzzers in this thesis use an external fuzz case generator like *radamsa* [50]. Instead, input generation and mutation are carried out in the harness itself. Figure 5.3 shows the general architecture of the in-process fuzzers. They run directly within FЯIDA on an *iPhone* that is connected to a computer via USB. The tasks of the different components are outlined below.

Manager The manager uses the FЯIDA *Python* Application Programming Interface (API) to inject the harness into `CommCenter`. It provides the harness with an initial corpus and a randomly generated seed value. After having set up the harness, it awaits incoming messages from it. These messages will either contain information that the BB has been reset or crashed or contain fuzzing traces. If the BB has crashed, the manager shuts down `CommCenter` and exits. Fuzzing traces will be saved in an output folder.

Harness By hooking messages in `libARIServer.dylib`, the harness will fuzz the BB. It is responsible for mutating ARI messages and providing the fuzzed input to the corresponding functions. It also hooks functions that reset the BB to see if *iOS* triggers a reset. Further, a global variable specifies whether messages can be sent to the BB via ARI. If this variable is `2`, then ARI is ready to be used. The harness also exposes a function to the manager to check the state of this variable.

In-place Fuzzer

The in-place fuzzer hooks the `SendRaw()` function to see when it is called. It then modifies the ARI message that is provided as an input to the function. After modifying the message, it reports it back to the manager to store it as part of the fuzzing trace. Thus, the fuzzer does not require a corpus, but the manager still needs to provide a random seed for the mutator. The idea behind the fuzzer is to exploit the statefulness of modifying ARI messages "mid-flight". Since the fuzzer modifies messages in a regular active connection, most trivial checks should be satisfied. This approach should enable the fuzzer to reach more complex states.

Injection Fuzzer

After receiving the corpus and the random seed from the manager, the injection fuzzer uses the `SendRaw()` directly. It chooses a random message from the corpus, modifies it, and calls the `SendRaw()` function with the mutated message. Thus, the injection fuzzer is independent of the actual active connection between the BB and *iOS*. Further, as long as the mutation strategy is deterministic, a fuzzing trace can be reproduced from the seed and the corpus alone.

Additionally, the injection fuzzer can also be used without the manager component in "standalone" mode. By providing the corpus directly in the script, the fuzzer can hook the necessary functions by itself. It stops itself if the BB is reset. The fuzzing trace can be reproduced from the corpus and the seed value. Thus, the trace does not need to be stored by a manager.

5.3.3 Implementation

Details about the implementation of the in-process fuzzers are given in this section. First, the custom mutator at the heart of the fuzzers is described. Later on, more insights into how the two fuzzers share symbols are given. Finally, some more information about the specificities of the injection fuzzers is described.

Mutation and Generation of ARI messages

By moving the mutation logic into the fuzzing harness, the fuzzers can avoid unnecessary (de-)serialization. It is necessary to write a mutator in *JavaScript* to accomplish this. The input generation strategy consists of two parts. First, a simple mutator that takes messages from the corpus and modifies it with some simple rules. Secondly, an adapted version of the ARI generator from the emulation-based fuzzing attempts.

The simple mutator has four different mutations it can carry out. Two trivial ones use an LCG to generate a random offset in the ARI message and flip a bit or an entire byte. Another inserts a random byte at a random offset. Lastly, the most complex one inserts a known value at a random offset. Known values include null values or four-byte integer maximum and minimum values. Each mutation is carried out with a known probability

and might occur in conjunction with any other mutation. All random values, such as the offset or whether a given mutation should be carried out, stem from a full period 32-bit LCG. The LCG is seeded with the value received from the manager. Alternatively, it can be seeded with a fixed default value.

Generating ARI messages in the in-process fuzzers differs slightly from the ARI generator described in Section 5.2.3. The most significant difference to the *Rust* version is that *JavaScript* does not support 64-bit integers. Thus, instead of using one 64-bit LCG to provide two seeds for the same 32-bit LCG for ARI and TLV generation, two different 32-bit LCGs are used. The first is used to generate ARI headers and to seed the second LCG. This second LCG is then used to generate a TLV header. Thus, the entire generator can again be seeded with one value.

There are two *JavaScript* versions of the ARI generator, to be exact. The first working the same way as the *Rust* version save for the changes described in the previous paragraph. However, this version would generate ARI messages much longer than packets observed "in the wild." Therefore, another version was created that chooses a length from a known list of observed ARI lengths. It then splices this value back into the generated header and proceeds the same way as the other version.

Since all the mutator's decisions depend on the output of a known LCG, they can be reproduced as long as the generator's seed is known. In other words, the generator is deterministic and given the seed for the generator, and the corpus, a set of fuzz cases can be re-created. Further, the modular *JavaScript* implementation of the mutator makes it trivial to create a *Node.js* script that can produce them independently of FЯIDA or the actual fuzzed device.

Symbol Sharing between Fuzzers

Both in-process fuzzers use the same symbols and hook the same functions. It makes sense to create a custom *JavaScript* class to share these symbols between them. This thesis leverages `frida-compile` [86] to do so. The symbols provided by Classen [22] were taken and merged into a single class. This class also provides direct access to all necessary functions. Some symbols changed between *iOS* versions. A fuzzer can import this class and specify which symbols it wants to use to support as many *iOS* versions as possible. This approach makes it very easy to maintain the fuzzers between different *iOS* versions. Implementing new fuzzers also uses much less boilerplate code, as it can simply import a class and instantiate it. The resulting object then provides all functions and symbols a new fuzzer needs. An example of how this symbol class can be used is available in Listing 5.4.

Injection Fuzzer Specifics

The injection fuzzer requires some additional setup to work. It can function in two modes: with a manager or standalone. A corpus is provided to the manager in the managed

```
// load and instantiate the symbol class
const ARISymbols = require('../js/ari-symbols').ARISymbols;
const syms = new ARISymbols();

// load the symbols for iOS 14.3
syms.setSymbols("arm64e_14.3");

// attach an interceptor to the `RequestReset` function
// it resets the baseband
Interceptor.attach(syms.RequestReset, {
  onEnter(_args){
    //...
  }
});

// ...

// send an ari payload to the baseband
syms.SendRawFunc(payloadBuf, length, to);
```

Listing 5.4: Example of using the symbols class.

mode, which parses it and sends it to the harness alongside a randomly generated seed value. The manager component will also tell the harness which symbols to use.

In standalone mode, all this information needs to be compiled into the harness itself. After doing so, the script can be invoked with FЯIDA directly. The point of this mode is to reduce the need for serialization as much as possible. There is one exception, the fuzzer still outputs fuzzing statistics. This exclusion is necessary to evaluate the effectiveness. However, this happens relatively rarely. The performance overhead is therefore minimal.

Another issue that needs to be solved in the inject fuzzer is finding a suitable value for the third parameter of the `SendRaw()` function. While the in-place fuzzer can re-use the value provided to the function by the original callee, the inject fuzzer has to provide this fuzzer itself. However, this value can be chosen from a handful of values observed "in the wild" as described in Section 4.8.3.

5.3.4 Applications

The in-process fuzzer and the injection fuzzer can both be executed by running their accompanying *Python* manager component. In the case of the injection fuzzer, a corpus needs to be collected first. A corpus collection script based on the identical function signatures as the fuzzers was implemented to accomplish this. It collected the ARI messages, whether they were received from the BB or sent to it, and the value for the `ariTo` parameter.

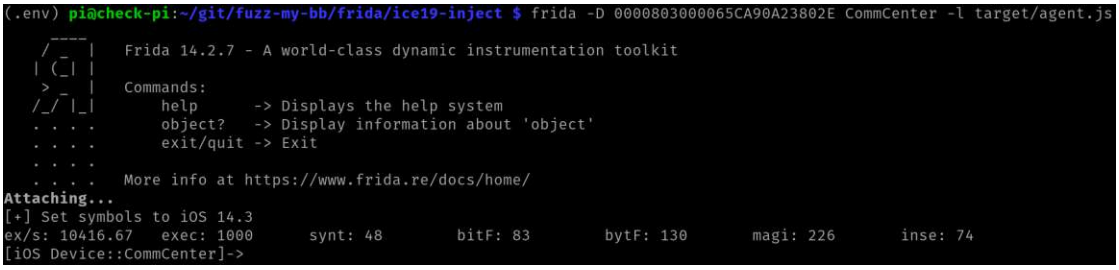

Figure 5.4: The injection fuzzer running in standalone mode.

When it comes to collecting a corpus, typically, it is recommended to include as much variety as possible. This method ensures that good initial coverage of the fuzzed software is possible. When it comes to fuzzing the BB, some exceptions have to be made. Since fuzzing SMS messages or phone calls would trigger actual interactions with the cellular network, fuzzing would cost money. Thus, it is necessary to craft a corpus carefully. By only triggering specific communication with the BB, this can be achieved.

The injection script is equipped with a helper function to provide all necessary input parameters to run the standalone fuzzer. The corpus needs to be encoded into an array of hexadecimal strings. Further, the *iOS* version and the seed value for the LCG can be specified as well. After compiling the script with `frida-compile`, it should be injected into the `CommCenter` process. Figure 5.4 shows the injection fuzzer running in standalone mode. The last line shows the FЯIDA prompt indicating that a crash was encountered. It is recommended to kill the `CommCenter` process with `frida-kill` before executing that fuzzer to achieve better crash reproducibility.

Finally, two helper scripts were implemented. The first script replays a given trace of ARI messages to the BB. Replaying traces is necessary to confirm that a crash is reproducible. Since ARI is a management protocol, the components involved in crashes have many responsibilities. Each may possess a certain amount of state information that may be relevant to the crash. Thus, ARI traces produced during fuzzing might not always cause the same behavior when replayed. It is necessary to replay an ARI trace, sometimes even multiple times, to establish that a crash is related to the trace.

The second script re-creates an ARI trace from a seed value and an initial corpus. This script can be used to produce the ARI trace of the standalone injection fuzzer. Thereby, it circumvents the need for communication with a management component and the need for serialization caused by this communication altogether.

CHAPTER 6

Evaluation

In this chapter, several performance characteristics of the fuzzers introduced previously are presented in Section 6.1. These characteristics include benchmarks in terms of fuzzing speed and coverage data. Further, this chapter briefly discusses some additional use cases for the tools developed throughout this thesis in Section 6.2. Finally, an analysis of the fuzzing results is given in Section 6.3.

6.1 Performance Evaluation

The following sections provide insights into the performance of the fuzzers. First, several performance characteristics of the emulation-based fuzzers are given. Then, some insights into the in-process fuzzers are provided.

6.1.1 Emulation-based Fuzzers

This section is divided into benchmarks for the American Fuzzy Lop Plus Plus (AFL++)-based and the custom fuzzer. The same corpus was used for all benchmarks in this category to make some comparisons possible. All of the benchmarks were run on an *Intel* i7-6700K @ 4.00 GHz. Each section first introduces how the benchmarks were implemented and then presents their results.

AFL++-based Fuzzer

A Dockerfile to compile and run the benchmarks was created to make them more easily reproducible. It always uses the latest version of AFL++, which at the time of writing is 3.14c. Then, a script sets several environment variables that follow *Google*'s *Fuzzbench* [42] setup. It then invokes a single instance of the AFL++-based fuzzer and runs it for precisely one hour. The intention is to make the exact settings for the benchmarks more easily accessible. However, the actual benchmarks were run without the Docker container

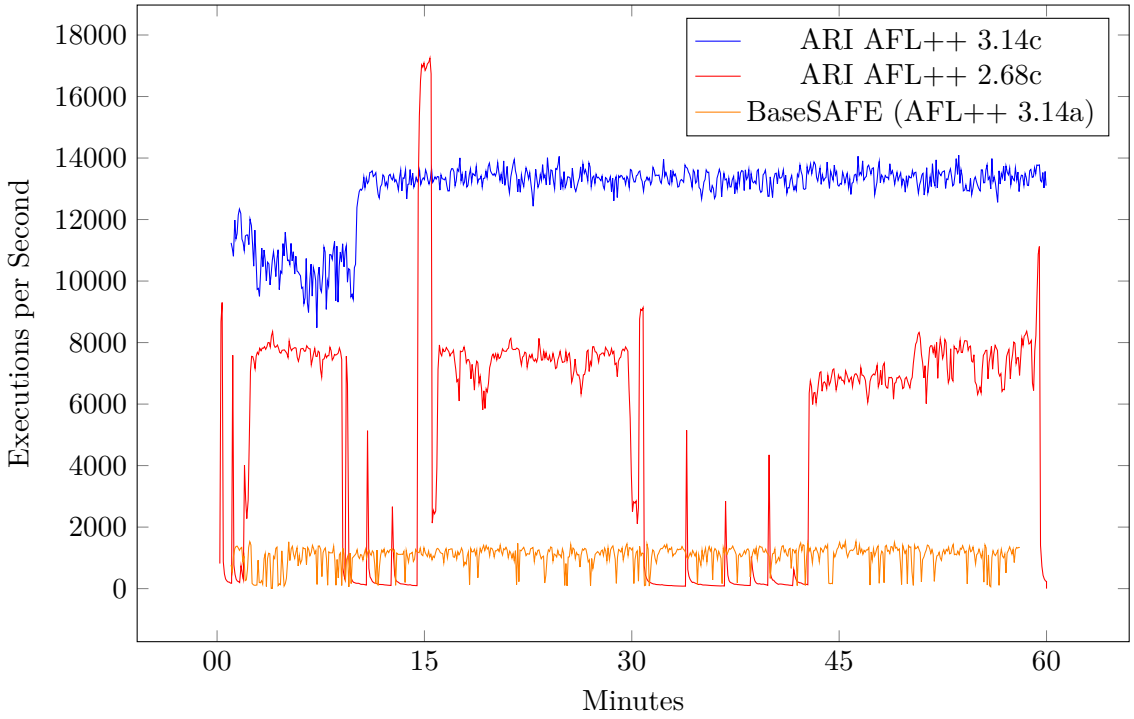

Figure 6.1: Executions per second by the AFL++-based fuzzers over time.

to reduce overhead. See Section A.4 for a comparison between benchmarks with and without Docker.

A benchmark was run with the `errc` example from *BaseSAFE* [67] to establish a baseline for the AFL++-based fuzzer. Finally, a benchmark was run with version 2.68c because that was the current version of AFL++ during the development of the custom fuzzer. It is presented here to give some context.

The plot in Figure 6.1 shows the fuzzing speed of the three different benchmarks over time. One observation that becomes immediately apparent is that the AFL++ 2.68c benchmark has a much less stable execution speed than the other two. The execution speed fluctuations were reported in the original *BaseSAFE* paper [67], but the repository for *BaseSAFE* has since been updated to AFL++ version 3.14a. Once the fuzzing speed has reached a plateau in the 3.14c benchmark, it is higher than the 2.68c fuzzer. This result is also reflected in the average execution speeds between the two versions: 12 865 .92 executions per second for version 3.14c and 5 094 .16 for version 2.68c, an increase of 152 .56 %.

Another observation is that the *BaseSAFE* benchmark is much slower than either fuzzer, with an average of 1 053 .82 executions per second. The original paper reported about 1 500 executions per second on an *Intel* i7-6700 @ 3.40 GHzz [67]. However, other than the code in the repository, it is not mentioned how AFL++ was executed. This information

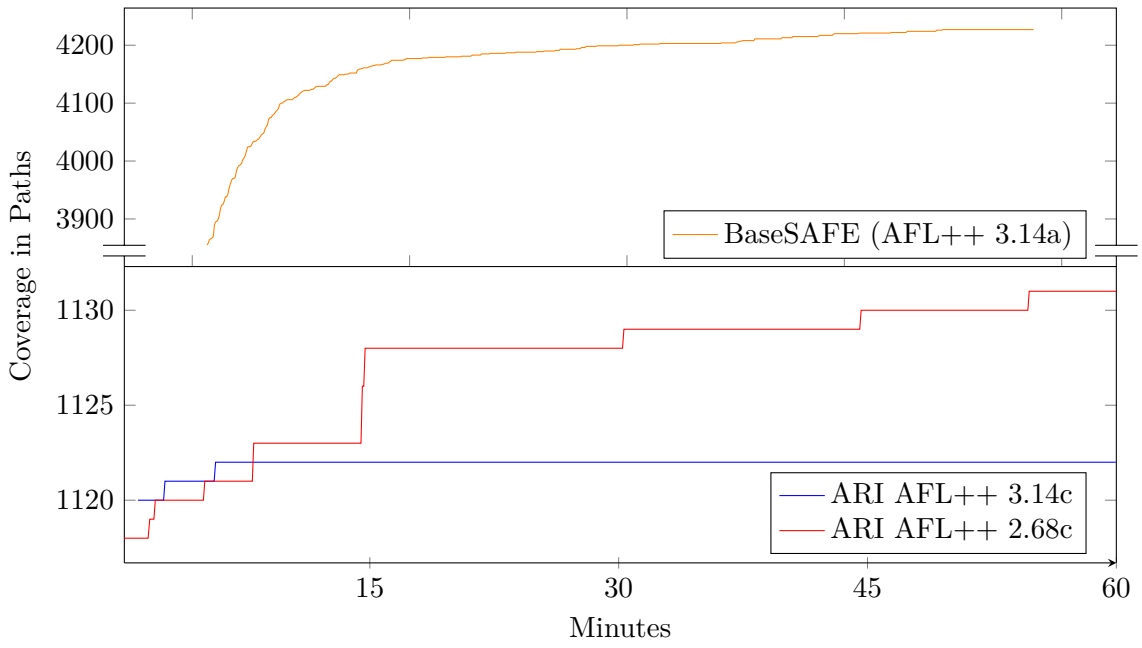

Figure 6.2: Total paths found by the AFL++-based fuzzers over time.

matters because AFL++ performance can differ depending on the configuration of the machine it is executed on, such as the Central Processing Unit (CPU) scheduler and memory allocators [2]. Thus, these results are not as comparable as they might appear at first glance. The difference in execution speed between the *BaseSAFE* benchmark and the Apple Remote Invocation (ARI) fuzzers is likely explained by the fuzzed target's complexity.

This assumption is also supported by the plot shown in Figure 6.2. It shows that AFL++ could find several times the number of paths within the Long-Term Evolution (LTE) Radio Resource Control target. An interesting result in this graph is that version 2.68c uncovered more paths in the same time frame even though its average fuzzing speed was much slower. AFL++ 3.14c also plateaued very early. Since this benchmark is based on a single run, this result is possibly due to randomness used within the fuzzer.

Thus, the benchmark on version 3.14c was re-run three times to compare the results. They can be seen in Figure 6.3, which shows a very similar pattern to path discovery between runs. While this is not conclusive proof that AFL++ version 3.14c discovers fewer paths than version 2.68c, it is still a surprising result. The paths discovered by the fuzzers are likely error paths because the corpus only contains valid ARI messages.

Custom Fuzzer

Similar to the AFL++ fuzzer, a Docker setup was created to compile and run the benchmark for the custom fuzzer. However, the actual benchmarks were again executed

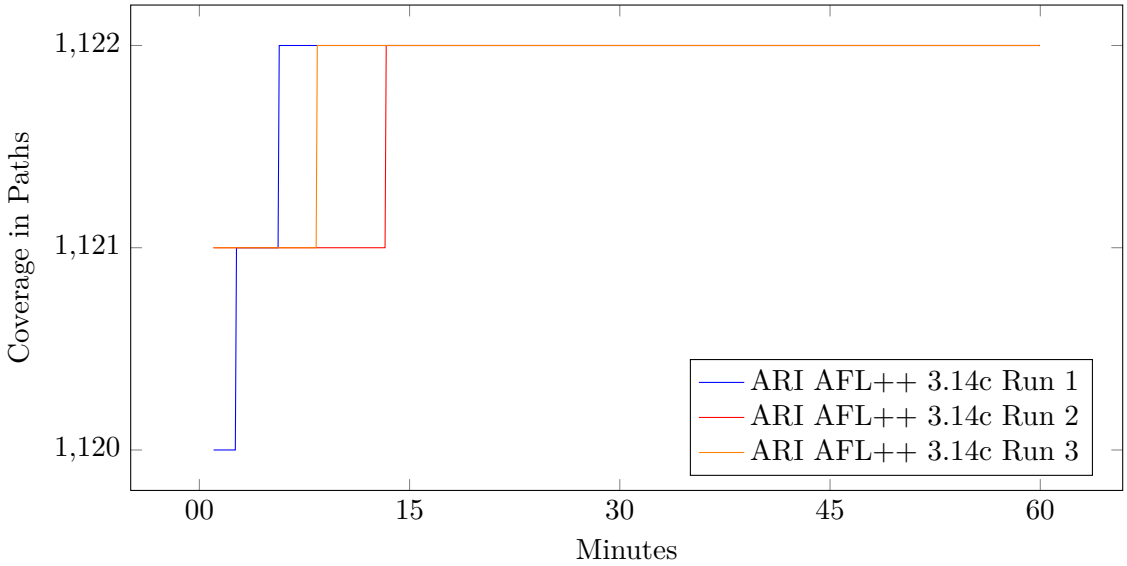

Figure 6.3: Total paths found by AFL++ 3.14c over time between different runs.

without Docker. This set of benchmarks focuses on three key points. The first is to find the ideal number of worker threads for the machine that carries out the benchmarks. Secondly, fuzzing speed will be compared between the custom fuzzer with the ideal number of worker threads and AFL++ 2.68c and 3.14c. Finally, edge coverage was collected to compare it to the data collected from AFL++ 3.14c. Sadly, version 2.68c does not output how many edges have been found and thus cannot be included in this comparison.

Four benchmarks were run, each with a different number of workers to uncover the ideal setup. Figure 6.4 shows that one worker can achieve an average of 638.61 executions per second. This speed is not increased linearly when new workers are added. For instance, eight workers can achieve a speed of 3 017.38 executions per second, increasing the execution speed only by about 3.7 times compared to the expected eightfold improvement.

Figure 6.4 further shows that a peak is reached with ten worker threads. This benchmark reached an average execution speed of 3 111.25, a slight improvement of 3.2 % over the eight worker benchmark. Performance dramatically decreases with 12 workers down to 2 730.43 executions per second or a decrease of about 12.24 %. This decrease is likely caused by some threads waiting to read from and write to their corresponding channels. Thus, it can be concluded that the ideal number of workers for an eight-core machine is ten.

When compared to AFL++ as shown in Figure 6.5, it becomes clear that AFL++ version 3.14c is definitively the fastest fuzzer. While the custom fuzzer also lags behind version 2.68c due to its fluctuating nature, the custom fuzzer can surpass it at times. It is necessary to mention that fuzzing performance with AFL++ can be further increased by

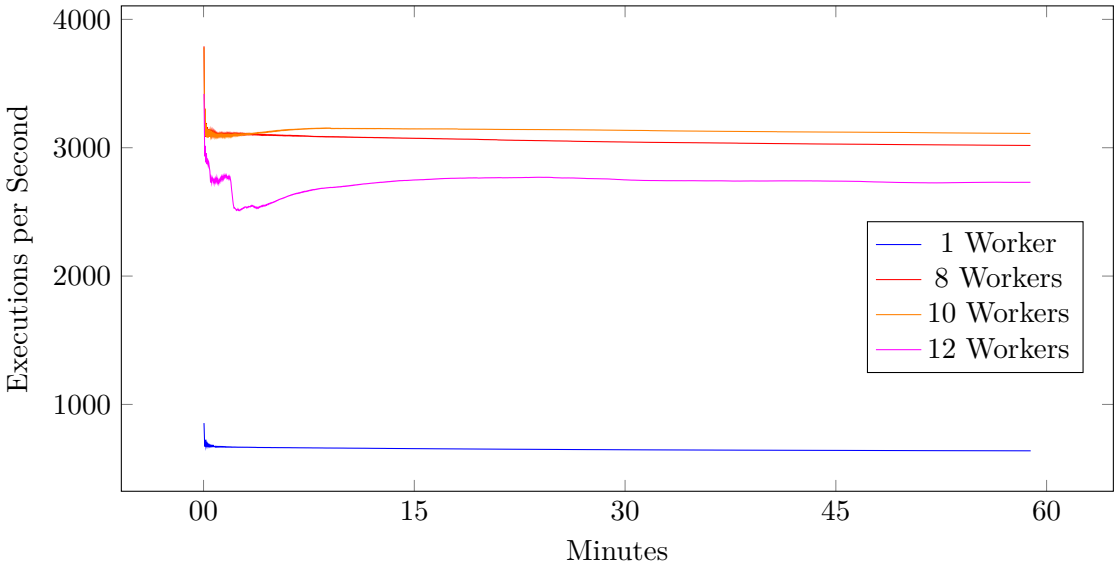

Figure 6.4: Ideal number of worker threads on an eight-core machine.

creating several parallel instances. AFL++ with both versions, 2.68c and 3.14c, would likely be much faster in that case than the custom fuzzer. This fuzzing speed increase is likely due to AFL++'s many optimizations. One of which applies explicitly to *Unicorn*, the underlying emulator. AFL++ caches already translated blocks [67], while the custom fuzzer does not.

Finally, Figure 6.6 shows edge coverage between the custom fuzzer and the three runs of AFL++. Both fuzzers plateau before reaching the thirty-minute mark. However, the custom fuzzer finds about 12.6 % or, in absolute numbers, 13 more edges than AFL++. Several factors may explain this result. While the custom fuzzer only randomly flips bits, AFL++ also tries to minimize the inputs. However, ARI can only be minimized so far. The custom fuzzer excludes the first four magic bytes from fuzzing, while AFL++ lacks this awareness. Minimizing the inputs and not excluding the first four bytes also means that it is more likely that fuzzing operations will occur in those bytes. This circumstance is due to them making up a more significant percentage of any given input.

6.1.2 In-process Fuzzers

The following sections presents some performance characteristics of the in-process fuzzers. Since they cannot instrument code running within the Baseband (BB), providing coverage information is impossible. Thus, this section focuses on the speed at which input can be provided to the BB. All benchmarks were run on an *iPhone* SE 2020.

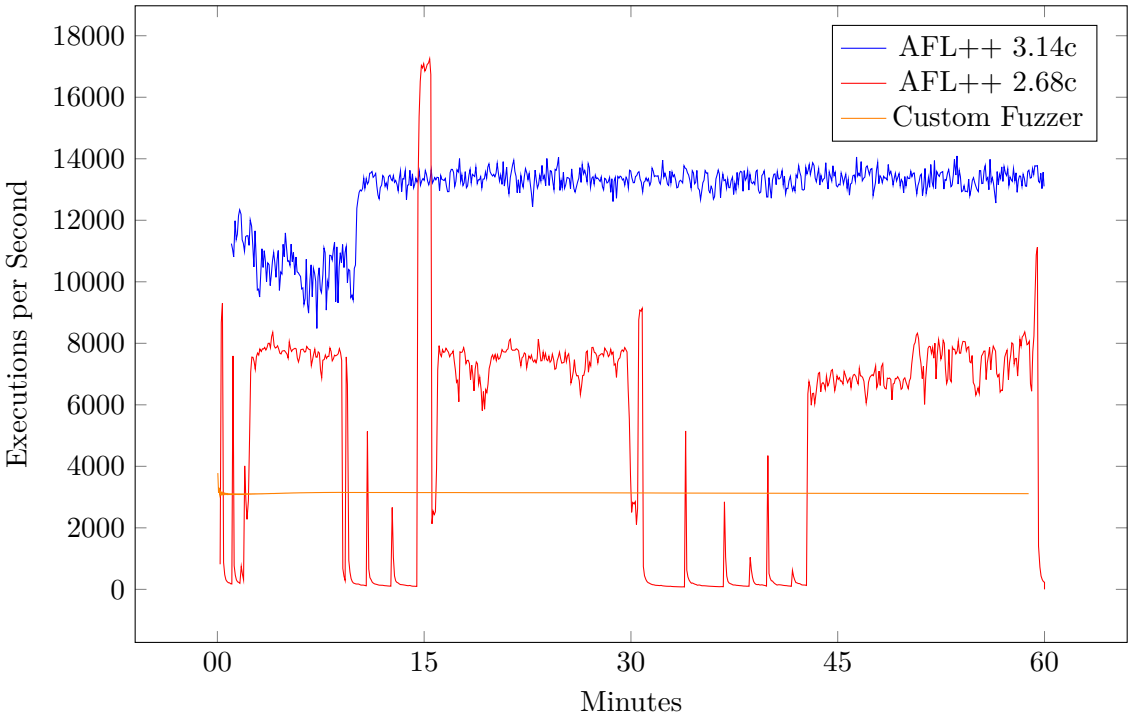

Figure 6.5: Speed comparison between the custom fuzzer and AFL++.

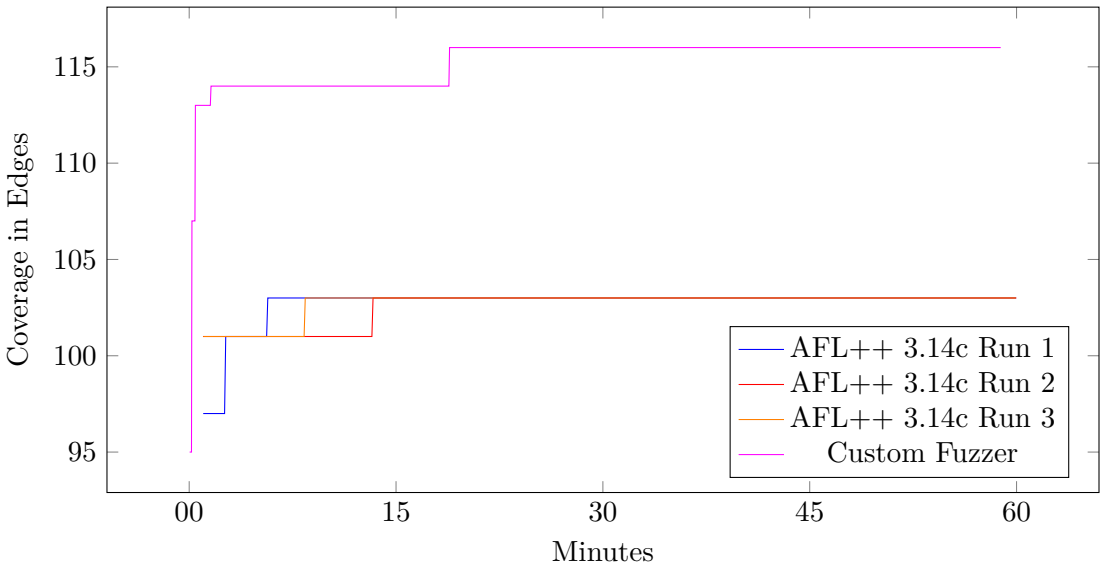

Figure 6.6: Comparison of coverage between the custom fuzzer and AFL++.

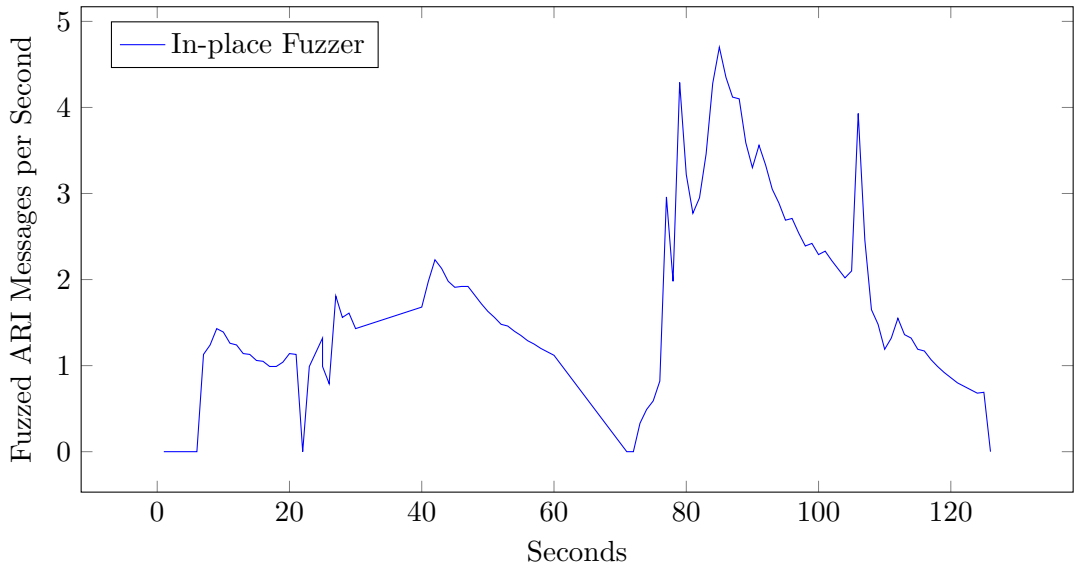

Figure 6.7: In-place fuzzer performance over time.

In-place Fuzzer

Benchmarking the in-place fuzzer is rather challenging because it requires active interaction between the BB and *iOS*. Further, since ARI is a management protocol, it does not transfer audio from calls or data from Short Message Service (SMS) messages. This reality rules out the possibility of simply starting an hour-long phone call and fuzzing it. Instead, it was run for five minutes while manually opening several different websites to give an insight into how the fuzzer performs. This benchmark is not comparable to other fuzzers because it is so much slower. It is also inherently hard to reproduce since it depends on the websites being visited and many other factors that are hard to control. Thus, the point of this benchmark is to show that fuzzing speed is not the only important variable, and improvements such as more stateful fuzzing are also tremendously significant to overall performance.

The fuzzer caused the phone to become very unstable. Cellular functionality broke down completely after about two minutes. Thus, it was not possible to create new fuzz cases after that. Hence the decline in fuzzed ARI messages around that mark, shown in Figure 6.7. Overall, the fuzzer only fuzzed about 124 messages or about one message per second in the two-minute time frame it was able to run. This result shows that typically only a few ARI messages are needed to trigger a severe crash. A severe crash, in this case, describes a crash of the BB that is unrecoverable until a reboot of the entire phone occurs.

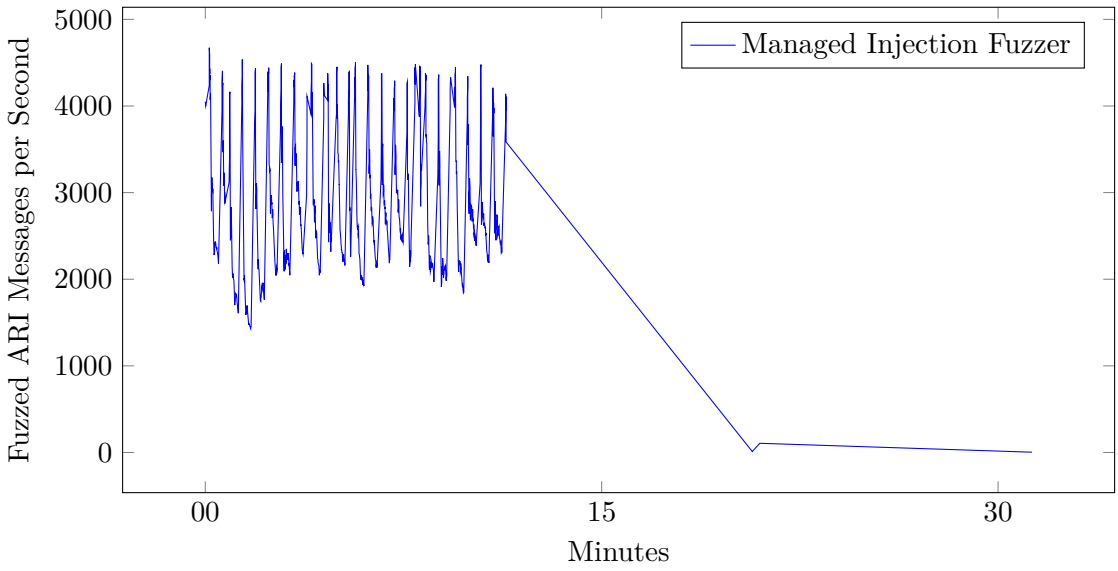

Figure 6.8: Fuzzing run of the managed injection fuzzer.

Injection Fuzzer

The initial approach to benchmark the injection fuzzer was very similar to benchmarking the emulation-based fuzzers. However, running the injection fuzzers for an hour is not possible because a crash of the BB or a problem with the current ARI state would occur quite quickly. Restarting the fuzzers and re-running them would eventually lead to a crash that was unrecoverable for the device even after terminating all processes grouped with `CommCenter`. At that point, a reboot of the device is required, making an hour-long benchmark impossible.

An attempt at benchmarking the injection fuzzer with a manager component for an hour while continuously restarting it after a crash can be seen in Figure 6.8. After about ten to fifteen minutes, a crash is encountered that cannot be recovered from for a few minutes. The benchmark run breaks down entirely around the thirty-minute mark, as cellular functionality on the device is not restored until it is rebooted.

Thus, the benchmark was limited to five minutes, as seen in Figure 6.9. The standalone fuzzer was also benchmarked for five minutes. However, comparing the average over that timeframe is difficult. Due to its improved performance, the standalone fuzzer would encounter a crash quite quickly and take longer to recover from a crash, which means that the standalone fuzzer was not running for the same amount of time. Moreover, when it comes to maximum fuzzing performance, the standalone fuzzer reached up to 10 752 .69 fuzzed inputs at its peak, whereas the managed fuzzer never managed to exceed 4 672 .9 fuzzed inputs. Thus, the input data serializing and sending it back to the manager is responsible for a 56 .54 % decrease in fuzzing speed.

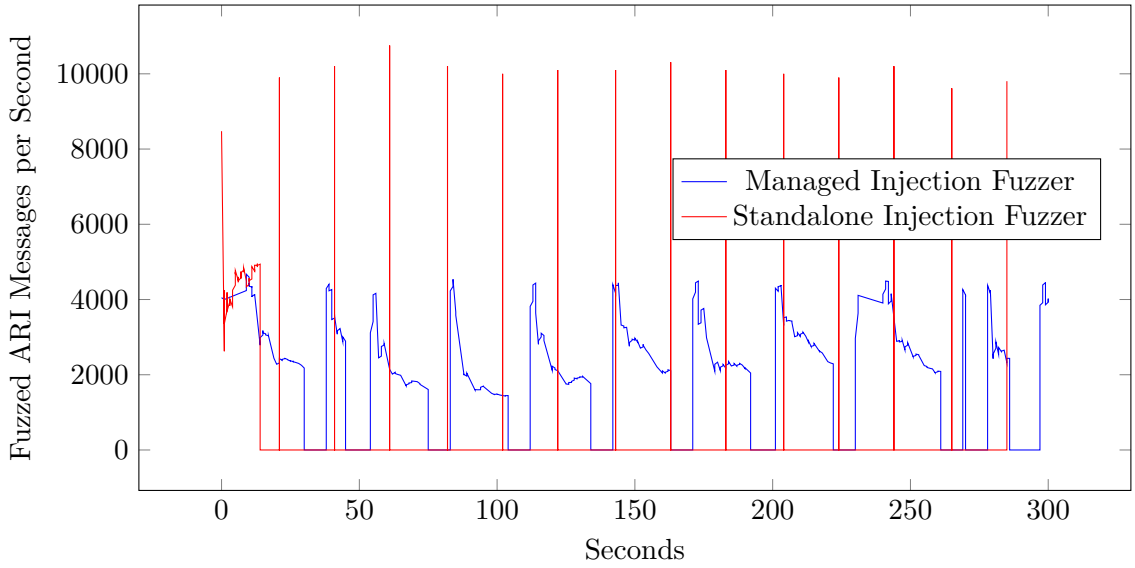

Figure 6.9: Fuzzing performance in standalone mode and with a manager.

6.1.3 ARI Generators

The three versions of the ARI generator were benchmarked to establish how much of a bottleneck generating ARI messages would be within the fuzzers. Each benchmark would generate 50 000 ARI messages and then write them each as a file to the disk. The three versions benchmarked were the Rust and two JavaScript versions. Further, the JavaScript versions are split between the length-limited and non-limited versions. All benchmarks were run on an *Intel* i7-6700K @ 4.00 GHz processor. *JavaScript* versions of the generator were run in *Node.js* version v14.16.0 and the *Rust*-based generator was compiled with `rustc` version 1.52.1.

Figure 6.10 shows the differences between the three versions on a logarithmic scale. First, it shows how much time the benchmark took overall, with the *Rust* and limited *JavaScript* versions of the fuzzer being the fastest at 1 .57 and 0 .84 seconds, respectively. The slowest being the non-limited *JavaScript* version at 11 .99 seconds. In that time, the *Rust* version generated about 31 880 .88 ARI messages per second compared to the non-limited version's 4 170 .49 or the limited version's 59 880 .24 messages per second. This throughput is equivalent to an output of 521 .42 megabytes per second for the *Rust* version, 75 .42 for the non-limited Javascript version and 36 .01 megabytes per second for the limited version.

One explanation for these differences is the additional overhead needed to run the *JavaScript* code within *Node.js* as compared to the native performance of *Rust*. Moreover, the limited version produces much less data. While it manages to produce more ARI messages per second than the limited fuzzer, its actual throughput in megabytes per second is much lower.

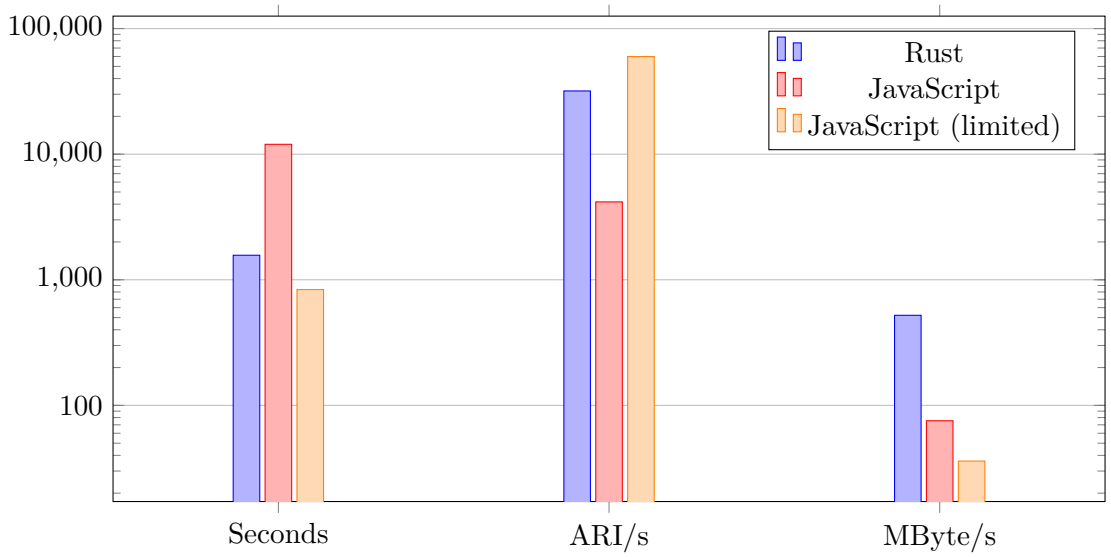

Figure 6.10: Comparison of the performance of the different ARI generators.

6.2 Reverse Engineering Use Cases

During this thesis, several tools were created to analyze the BB. These tools can be used outside of this work as well. The following section will highlight a few additional use cases for these tools. Each paragraph will first describe a given tool and then summarize how it was used in this thesis.

First, a set of shell and *Python* scripts was developed to quickly parse and output results from `radare2`'s [85] `ra-find2` utility. This set of scripts enables quickly searching for hexadecimal values in a set of files and then outputting the result in the JavaScript Object Notation (JSON) format. It is then possible to use a simple *Python* script to parse the results and give more context. During this thesis, these scripts were used to analyze `.istp` files. It was possible to find the four magic bytes related to ARI and the separator sequence this way. It was also trivial to add scripts to analyze these structures further.

Secondly, a script to find matching sequences within a set of files was created. The script loads two sets of files into memory and then tries to find matching sequences within them. It outputs the ten longest matches to the console and outputs all matches as JSON to a file. During this thesis, this script was used to find commonalities between two ISTP dumps. The idea is that similar actions, such as turning off and on airplane mode or sending SMS messages, should produce similar dumps. Since the files are about one megabyte in size, dumps can become big quite quickly. Consequently, finding matches becomes infeasible because the processing time increases dramatically.

Thirdly, the script used to reconstruct the BB source code structure in Section 4.4.2 is compatible across all BB generations. Thus, it can likely be used for future BB

generations, allowing for quick comparisons between generations. This script may help find new, possibly undocumented, features. Further, the general concept could easily be extended to other domains. The script's structure is modular enough to easily add support for firmware built from different code bases.

Finally, to debug and analyze the emulation setup, an additional hook was implemented that can print the current state of the stack. This hook can be used in addition to *BaseSAFE*'s register hook [67]. The register hook prints the current contents of the *ARM* registers `R0` to `R11`, `SP`, and `LR`. The stack hook can print the top five entries on the current stack pointed to by `SP`. During this thesis, these hooks were used to gain a better insight into the emulation setup and debug it. It can also be used with a utility program created during this thesis, which shows the trace of basic blocks that a given emulation run traverses.

6.3 Analysis of Fuzzing Results

This section summarizes the results of the different fuzzing campaigns carried out during this thesis. The vulnerability's cause and whether the given vendor has addressed it are outlined in the case of vulnerabilities.

6.3.1 Emulation-based Fuzzing Results

During the fuzzing campaigns with AFL++ and the custom fuzzers, several crashes were encountered. For instance, over an hour, the emulation-based fuzzer was able to find 372 crashes. All of these crashes are related to read-operations referencing an unmapped memory region. A replay script based on the symbols used for the in-process fuzzers was implemented to verify whether these crashes are reproducible on the actual device. Before replaying the trace, the `CommCenter` process was terminated with the `frida-kill` command. This termination was done to clear potential state information that could impact the crashes.

Due to the implementation of the emulation setup, only crashes caused by single ARI packets were considered. Most of the time, no results were encountered when replaying these messages. Neither the event log of the *iPhone* nor `DumpBasebandCrash` would show any signs that a BB crash was encountered.

However, sometimes a dump of `.istp` files was produced. Analyzing the dump showed that the reason for it was an `ARI_TIMEOUT`. Most likely, this timeout was caused by the injection of new messages itself. However, it is possible that the BB crashes "silently", meaning that *iOS* does not notice the crash. The exact cause is not understood to date due to a lack of information about the ISTP format.

Overall, emulation-based fuzzing in this thesis did not yield any exploitable bugs. While both AFL++ and the custom fuzzer can find crashes within the realm of emulation, reproducing them on an actual device failed. Even if crash logs were created, their cause

could not be understood. The most likely cause for the crashes is an incomplete emulation setup that misses the necessary global state for ARI to be parsed correctly. However, reconstructing such a state without a runtime view of the BB was too challenging and time-consuming to achieve. While these issues could not be resolved during this thesis, it was possible to emulate and fuzz a handler within a highly complex target reasonably well. Further, once issues such as understanding the ISTP format are resolved, it might be possible to show that the encountered crashes indicate severe issues within the BB.

6.3.2 In-process Fuzzing Results

The failure to find relevant crashes through emulation-based fuzzing was one of the motivating factors while implementing the in-process fuzzers. By using an actual device, the global state and other necessary parameters were already instantiated. Thus, crashes encountered during fuzzing were more reproducible.

Encountering crashes during fuzzing usually does not take much time. As shown in Section 6.1.2, typically, a crash is encountered after less than a minute or two. Crashes differ in severity. While some cause a BB, reset others disable cellular functionality altogether. So far, it was possible to recover more severe crashes by rebooting the device.

As with the crashes produced by the emulation-based fuzzers, traces from the in-process fuzzers that caused a crash were replayed using a FЯIDA script. This way, it was possible to quickly verify whether an ARI trace causes crashes within the BB. Crashes can either be identified by consulting the `info.txt` for the dump reason or by checking the devices log.

Listing 6.1 shows the device log of an *iPhone* SE 2020 during encountering a crash and resetting the BB. First, `abm-helper` detects a reset of the BB. Then, some event listeners for the BB's boot-state and reset status are registered, and the BB's bootloader is told to reload the firmware. The firmware is then transferred step by step to the BB, and a reboot is carried out. Finally, some flags are reset to indicate that the reset is complete.

However, while it was possible to find reproducible crashes, the lack of insight into the BB's inner workings hinders analysis. It is not possible to tell what the root cause is for any given crash. Consequently, it is also not possible to distinguish crashes from each other. A better understanding of the ISTP format would be helpful here as well.

The in-process fuzzers' goal was to fuzz the BB and uncover vulnerabilities within it and not within *iOS*. However, while fuzzing it, the BB responds in erratic ways to the received ARI messages and generally exhibits unexpected behavior. Thus, in effect, the in-process fuzzers fuzz not only the BB, but also components within *iOS*. The rest of this section details a vulnerability found while fuzzing with an in-process fuzzer.

While running the injection fuzzer with the first *JavaScript* version of the ARI generator, the *iPhone* would reboot after about 40 injected packets. Since the fuzzer was relatively fast, this would only take a couple of milliseconds between starting the fuzzer and the

```
abm-helper(ABMHelper)[434] <Notice>: #I baseband reset detected
[...]
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: #I ipc.mod.
    client registered for: (
     EventBasebandBootStateChange,
     EventBasebandResetDetected,
     kEventTraceDumpStateBegin,
     kEventTraceDumpStateOsLog
)
[...]
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: #I Allowing
    ARI reset requests
[...]
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>:
    16:1628534176.561[4.0]BBUICEInitializer::: Preparing at first
              with reset requested 1
[...]
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>:
    16:1628534176.712[4.0]BBUICEInitializer::: Preparing at second
             with reset requested 0
[...]
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: 3:
    Sending PSI enhanced command 'Load and execute EBL'
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: 3:
    BEGIN: Sending Images
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: 3:
                Product Type : 97 and Hardware Config: 0x2 in
                Reserved0
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: 3:
                EBL started, continue to image download.
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: 3:
                Loaded file 'TPCU.elf'
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: 3:
                Reading file 'TPCU.elf' (offset=0, length=52)...
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: 3:    0x34
    of 0x34 (100 percent)
[...]
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: #I Added
    detection with key 'Baseband Recovered Gate', reset=0
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: #I Sending
    EventBasebandBootStateChange to radio.mod.client at
    1628534178226
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: #I The
    baseband reset-flag is reset.
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: #I Sending
    EventBasebandBootStateChange to CommCenter at
    1628534178227
CommCenter(libBasebandManagerICE.dylib)[28527] <Notice>: #I Added
    detection with key 'Baseband Crash Recovery', reset=1
```

Listing 6.1: Excerpt from an *iPhone*'s system log showing a BB crash.

```
1  struct IOACIPCTagList
2  {
3    unsigned __int16 chainLength;
4    unsigned __int16 nextFreeTag;
5    unsigned __int16 lastIndex;
6    unsigned __int16 initiallyZero1;
7    unsigned __int16 initiallyZero2;
8    unsigned __int16 unknown1;
9    unsigned __int32 unknown2;
10   IOACIPCTagEntry *  entries;
11 };
```

Listing 6.2: The `IOACIPCTagList` data structure.

phone rebooting. This reboot prevented *iOS* from generating a crash log indicating the cause of the reset. Slowing down the fuzzer to only inject two messages per second prevented this behavior and also generate a crash report. The crash report indicates that a kernel panic had occurred due to an assertion failure. While it was possible to reproduce the issue across three *iOS* versions(13.5, 14.1, and 14.3), it was only reproducible on devices with *ICE19* BBs. On *ICE17* the issue could not be reproduced and *ICE16*, *ICE18*, and *IBIS18* could not be tested. This observation shows that while this is an issue within the *iOS* kernel, it originates from communication between the BB and the application processor.

The kernel cache of the device on which the issue was reproduced was loaded into *IDA Pro*. However, this kernel cache did not contain any debugging symbols, making analysis rather challenging. Thus, a trick[1] known in the *iOS* reverse-engineering community was used: *Apple* included a research kernel cache that still contained debugging symbols in specific builds of *iOS* 14.

Identifying the failing assert-statement was relatively easy using *IDA's* cross-referencing feature and the string provided to the assertion. It was possible to identify `IOACIPCTagList` as a data structure involved in the crash by using the research kernel cache. The panic occurred in its `freeTagChain()` method. After manual analysis, it was possible to reconstruct the fields of this data structure. `IOACIPCTagList's` fields are shown in Listing 6.2.

The `IOACIPCTagList` contains several variables that are used to implement other methods associated with the list. Additionally, it contains a pointer to a second data structure called `IOACIPCTagEntry`, which can be seen in Listing 6.3. However, for both data structures, it was not possible to understand the meaning of all fields.

The actual cause of the panic can be seen in Line 12 of Listing 6.4. Understanding how the `IOACIPCTagList` works is necessary to understand the `if`-statement in the line before the panic. When a new list is initialized, a length parameter is passed to the

[1] See: `https://twitter.com/tihmstar/status/1295814618242318337`

```
1 struct IOACIPCTagEntry
2 {
3   unsigned __int16 claimed;
4   unsigned __int16 index;
5   char unknown[20];
6 };
```

Listing 6.3: The `IOACIPCTagEntry` data structure.

initializer. This parameter indicates the number of tag entries stored in a given list. Thus, the first step in the initialization method is to allocate an array of `IOACIPCTagEntry` elements. Next, the `chainLength` field is set to the value of the length parameter, `nextFreeTag` is set to zero, and `lastIndex` is set to the length parameter minus one. Finally, for each entry of the list, the `index` and `claimed` fields will be set. The `index` variable will be set to the index of the next element in the list, except for the last element, where it will be set to the actual index of the last element. As for the `claimed` field, it will be assigned the value one. When a new free tag entry is requested through the `getFreeTag()` method, roughly the following steps will be taken:

1. If the tag at the index referenced by `nextFreeTag` has been claimed, an assertion will fail.
2. If the same tag's `index` field is set to its actual index, a null value indicating an error is returned.
3. Otherwise, assign the `index` field the value of the entry index and return the entry by reference.

Therefore, whether or not a tag is free or in use seems to depend on its `index` value. It follows that the check in Listing 6.4 assures that no tag entry is freed twice. *Apple* has confirmed the issue and addressed it with an *iOS* update to version 14.6[2]. Due to the lack of an *iOS* 14.6 jailbreak at writing, independently verifying whether the problem was removed was impossible. Further code snippets documenting this mechanism can be found in Section A.5.

Even though it was not possible to exploit this vulnerability in this thesis, *iOS* behaves very strangely. The entire device crashes when sending too much data and instantly reboots. While *Apple* did not assign a CVE, it is suspected that this vulnerability might be exploitable regardless. Such an exploit might lead to a remote code execution attack, which would be much more severe than a simple Denial of Service (DoS).

[2]The update notes acknowledging this issue can be found at `https://support.apple.com/en-us/HT212528`.

```
1  bool __fastcall IOACIPCTagList::freeTagChain(IOACIPCTagList * this,
       __int16 a2,
2   unsigned int tagChainToFree)
3  {
4    IOACIPCTagEntry * v3;
5    /*...*/
6
7    v3 = this->entries;
8    if ( v3[tagChainToFree].index != tagChainToFree )
9      panic(
10       "\"assertion failed %s:%u\"",
11       "/Library/Caches/com.apple.xbs/Sources/IOACIPCFamily/
           IOACIPCFamily-38.0.1/IOACIPCFamily/IOACIPCCore/
           IOACIPCTagList.cpp",
12       101LL);
13
14   /*...*/
```

Listing 6.4: The `freeTagChain` method causing the kernel panic.

CHAPTER 7

Discussion and Future Work

This chapter first discusses the results of the work presented in the previous chapter in Section 7.1. Section 7.2 then summarizes topics that are still questions for future work in regards to analyzing *Intel* Basebands (BBs).

7.1 Discussion

While Chapter 6 has discussed the performance of the different fuzzers used in this thesis, this section now looks at the upsides and downsides of the different approaches. First and foremost, it is necessary to acknowledge that the emulation-based fuzzing approaches, as described in Section 5.2, were ultimately unsuccessful. They did not yield any relevant security vulnerabilities. However, while creating the fuzzing setup, some insights into the BB were uncovered, such as functions used to allocate buffers within the BB. This information might be helpful for further investigations.

Another upside of the emulation-based fuzzers is the ability to instrument the emulator. By setting hooks, it was possible to leverage coverage guidance in the custom fuzzer and American Fuzzy Lop Plus Plus (AFL++). Inspecting the crashes that were encountered was also much more accessible and thorough. It was possible to inspect register values, Central Processing Unit (CPU) state, and the heap and stack of the running process.

One reason why emulation-based fuzzing might have failed is that the parser that was being emulated and fuzzed was too small. Apple Remote Invocation (ARI) itself has a wide-ranging impact on the BB, and its contents are evaluated throughout the firmware. However, the parser analyzed in this thesis focuses primarily on parsing the Type-Length-Values (TLVs) from given ARI messages. Creating a more sophisticated emulation setup could lead to discovering a hidden vulnerability much deeper in the BB.

In-process fuzzing, as discussed in Section 5.3, has several advantages. For instance, it is much easier to set up. Creating a FЯIDA script that can effectively fuzz a given process

in *iOS* or use functions within that process to fuzz other parts of the device (such as the BB) is much faster.

It was ultimately not possible to uncover a security vulnerability in the BB itself. However, the device's stability during fuzzing is so low that it is clear that these interfaces have likely not been internally analyzed before by *Intel* or *Apple*. For example, results from the in-place fuzzer (see: Section 6.1.2) show that even a tiny amount of fuzzed ARI messages leads to unrecoverable crashes. It requires the device to reboot and, thus, creating a Denial-of-Service attack. Similar situations were encountered with all fuzzers in minutes, attesting to the high instability of the examined implementations.

Furthermore, the discovery of a kernel-level vulnerability, as described in Section 6.3.2, further confirms the instability of the *Apple* cellular stack. Even though the goal of this thesis was to uncover issues within the BB itself, fuzzing it led to a crash within *iOS* itself. In reality, this is a much more concerning result than finding issues in the BB. While this particular vulnerability did not lead to code execution within the kernel, similar issues could enable a remote attacker to do exactly that. A zero-click remote code execution that can escalate into the *iOS* kernel would have a much more severe impact than an attack from the Application Processor (AP) to the BB. After all, the latter would typically require the former to be feasible in the first place.

The topic of the ISTP format has to be discussed. This proprietary logging and tracing protocol was the main factor why a security vulnerability within the BB could not be detected. Without understanding it, it is not possible to know the internal state of the BB. It was not possible to understand the protocol sufficiently for this kind of analysis. However, it was feasible to uncover specific hints about the nature of this format. These hints were detailed in Section 4.7.3 and might help future research on the topic.

Finally, when it comes to statically reverse-engineering the BB, this thesis uncovered several details. It was possible to compare the growth in complexity between five BB generations and document the processor architecture and model of the CPU and Digital Signal Processors (DSPs) of several BBs. Further, the memory exploitation mitigations in *Intel* BBs and several details of the ARI protocol were uncovered.

7.2 Future Work

From further reverse-engineering tasks to improvements to the fuzzers, various tasks remain for future work. This section lists several of these topics. First, focusing on open reverse-engineering questions and then detailing improvements to fuzzing strategies.

7.2.1 Reverse Engineering Tasks

This work presented an overview of several topics related to the *Intel* BB. However, the BB runs a very complex firmware. Thus, the following section details areas of interest for future work.

ISTP Format Since the ISTP format holds the key to understanding the internal workings of the BB, reverse-engineering it remains an important open topic. It would make it possible to distinguish crashes within the BB and enable analyzing them in more detail. This work might eventually lead to the discovery of severe vulnerabilities within the BB.

ARI Reverse-engineering While the general structure of ARI is well understood, the actual interaction between the BB and the AP is still missing detail. In other words, while it is clear which groups, types, TLV types and more, are included within ARI, their specific meaning is not always clear. Further analysis might help to better understand ARI and create tools that could interact with the BB directly.

Reverse-engineering Cellular Protocol Implementations This thesis has focused chiefly on the communication between the BB and the AP. However, another major part of the BB are the cellular stacks for Global System for Mobile Communications (GSM), Universal Mobile Telecommunications System (UMTS), Long-Term Evolution (LTE), and more. These have contained many vulnerabilities in the past [39, 46]. However, in the case of *Intel* BBs, only very little public work on the topic exists. Thus, reverse-engineering these software stacks could enable more interesting security research.

General Baseband Reverse-engineering As has been shown in this thesis, the BB stack is a highly complex target containing likely more than 100000 functions. Another topic for future work could be gaining a better understanding of the general structure of the BB and how different parts interact with each other. This information could aid greatly with future security research.

7.2.2 Better Emulation and Fuzzing

While it was possible to uncover vulnerabilities in the *iOS* kernel, the fuzzing campaigns did not uncover any security vulnerabilities in the BB. Several improvements could help achieve this task. Thus, this section focuses on these advancements.

More thorough Emulation At the moment, the emulation setup only focuses on the parser for ARI messages and nothing else. Especially the further handling of the information contained in the messages could be crucial to finding vulnerabilities. Thus, improving the emulation setup to include more parts of the BB could be an essential advancement. It could even go as far as emulating the entire BB. Similar work has been done for *Samsung*'s *Shannon* BBs by Hernandez and Muench [52].

More Flexible Sanitized Heap The custom fuzzer currently uses the sanitized heap implementation provided by *BaseSAFE* [67]. This heap is intended to be used with AFL++. When illegal memory access occurs, the entire process crashes intentionally because a `panic` occurs. This behavior is a problem for the custom fuzzer since it cannot

recover from such a crash. Thus, a more flexible heap that allows setting custom hooks via callbacks would be beneficial.

Better Mutations Currently, the custom fuzzer flips bits at random to generate new inputs. The in-process fuzzers use more sophisticated techniques such as splicing known problematic values into the input or generating random ARI messages. This method or an even more sophisticated approach could be implemented in the custom fuzzer to improve the effectiveness of the fuzzer.

AFL++ Custom Mutator An alternative approach to improve fuzzing effectiveness is to use the custom mutator mechanism supported by AFL++ [36]. It allows for the modification of AFL++'s behavior either via a *Python* script or a *C/C++* program. Thus, it would be possible to integrate knowledge gained through reverse-engineering ARI into AFL++, harnessing all the advantages AFL++ offers and gaining effectiveness through knowledge about the fuzzing target.

LibAFL More recently, *LibAFL* [66] was published. It is a library written in *Rust* that heavily influenced AFL++. Compared to AFL++, it takes a much more modular approach allowing users to only use the parts of the fuzzer they need. It also features a FЯIDA mode that allows the in-process fuzzer discussed in this thesis to implement more sophisticated mutations. Integrating *LibAFL* into this fuzzer might yield a much more effective fuzzer overall.

In-process fuzzing without exclusions Currently, the in-process fuzzers exclude certain parts of ARI, such as Short Message Service (SMS) messages and phone calls. When fuzzing these BB parts, actual phone calls are triggered, which means costs are incurred. Due to such costs and the fact that this might impact real cellular networks, the corresponding parts of ARI were excluded. Future work could set up a test bench with a fake base station to work around this issue.

Grammar-based Fuzzing All fuzzers implemented in this thesis could benefit from a grammar-based input generation technique to generate new inputs. Since ARI messages are categorized into groups and then sub-categorized into types, each with a specific set of associated TLV types, randomly generated messages often fail trivial checks. Thus, using a grammar describing these relationships could be more effective when creating inputs. It would also allow for a more sophisticated way of only fuzzing certain parts of ARI, such as SMS message handling.

Conversation-based Fuzzing This thesis has mostly fuzzed one ARI message at a time and has not considered any global state when generating and mutating messages. However, ARI has a global state that every message influences and behavior changes depending on this state. Furthermore, many ARI messages exhibit a request/response pattern. Considering the global state and patterns such as this one during fuzzing might

yield a fuzzer that can explore the BB much deeper. Therefore, it might be able to uncover vulnerabilities buried within.

Coverage-guided In-Process Fuzzing Once the ISTP format is understood well enough, it might be possible to implement a coverage-guided fuzzer. Since `.istp` files are written continuously, it might be possible to hook into functions responsible for handling them within *iOS*. It might be possible to parse incoming ISTP data to decide whether a given input has discovered a new state within the BB by leveraging these hooks.

x86 Fuzzing While *ICE18* is not the newest generation of *Intel* BBs, it still has much overlap with the code base of *ICE19*. One significant advantage to fuzzing *ICE18* is that it is an *x86*-based BB, which means that the emulation overhead needed to run parts of it could be much smaller than with *ARM* generations. Leveraging this fact could increase fuzzing speeds.

CHAPTER 8

Conclusions

In this thesis, five generations of *Intel* Basebands (BBs) were discussed and analyzed. An overview of their processor architecture, Real-Time Operating System (RTOS), and how they interact with the *Apple* cellular stack was provided. The growth in complexity, as well as memory exploitation mitigations, were discussed. Furthermore, several approaches to emulating and fuzzing the BB's firmware were detailed. Through fuzzing efforts, it was possible to uncover a kernel-level vulnerability in *iOS*. This chapter gives a more detailed summary of this thesis and then concludes it.

To begin, background information and analysis techniques used throughout this work were described in Chapter 2. Then, in Chapter 3, the current state-of-the-art of BB security research and reverse-engineering efforts was outlined. Further, several key works in the realm of fuzzing were summarized.

In Chapter 4, the results of statically reverse-engineering the BB were presented. An overview of the *Apple* cellular stack with *Intel* BBs was given. Next, how the firmware was acquired and what files are included in each BB generation was summarized. Then, the general processor architecture and model of each BB generation's Central Processing Unit (CPU) were discussed, as well as a brief summary of the use of *Xtensa* cores as Digital Signal Processors (DSPs) was given.

By leveraging the function detection of four different disassemblers, libraries, and source code paths referenced by strings in the BB, the firmware's complexity was compared between five BB generations. This comparison was followed by a brief discussion of the BB's RTOS. Then, an overview of several memory exploitation mitigations across the five BB generations was presented. The chapter concluded by summarizing the Apple Remote Invocation (ARI) protocol used for communication between the BB and the Application Processor (AP).

In Chapter 5, the insights from reverse-engineering the BB were leveraged to fuzz its firmware with the intention of finding a security vulnerability. In total, four fuzzers were

implemented. The first two focused on emulating parts of the BB's firmware. The first fuzzer uses a custom implementation to utilize knowledge about ARI, the second utilizes American Fuzzy Lop Plus Plus (AFL++). The following two fuzzers would hook functions within *iOS*' `CommCenter`. `CommCenter` is a process that is responsible for handling all kinds of cellular-based functionality, such as phone calls or Short Message Service (SMS) messages. These two fuzzers took two different approaches to send manipulated messages to the BB. The in-place fuzzer mutates or even replace specific messages mid-flight between the AP and the BB. The injection fuzzer, on the other hand, mutates messages from a corpus or generate new random messages and then send as many of them as it could to the BB.

The performance of the different fuzzers was evaluated in Chapter 6. This chapter also discussed reverse-engineering use cases of scripts created throughout this work. Then, a discussion of the fuzzing campaigns' results was given. The discussion included an analysis of a kernel-level vulnerability within *iOS* that *Apple* has since acknowledged and fixed. Finally, Chapter 7 briefly discussed the results of this thesis and outlined a variety of topics for future work.

Overall, the *Intel* BB implementation is a very complex system that is entirely closed off. Some parts of its implementation can be understood from convoluted public specifications, but documentation is minimal. In light of these circumstances, this thesis attempted to document how the *Intel* BB works as much as possible. It also tried to uncover security vulnerabilities within it. While the latter goal ultimately failed, a kernel-level vulnerability within *iOS* was discovered. It was also possible to demonstrate that the BB is extremely unstable when pressured under fuzz testing.

In conclusion, this thesis serves as a foundation for future research. Several results discussed in this thesis can be used for future security or reverse-engineering research, such as information about memory exploit mitigations or the CPU architecture. Future research is encouraged as it is highly likely that severe vulnerabilities are yet to be discovered in *Intel*'s BB implementation.

APPENDIX A

Appendix

This appendix contains further information about various topics discussed in the main thesis. Starting off with an overview of symbols and offsets for each of the given listings or functions discussed during this thesis in Section A.1. Then, Section A.2 provides an excerpt from a `BuildManifest.plist`. Section A.3 gives an overview of the amount of cores present in the *ICE* Basebands (BBs). Finally, Section A.4 and Section A.5 provide some additional information about fuzzing performance with and without the use of Docker as well as further information about the vulnerability discussed in Section 6.3.2.

A.1 Symbols and Offsets

Table A.1 shows the offsets and symbols used for the different code snippets and references to functions within the BB throughout this thesis. It includes the offset or symbol name of the given function or listing, as well as the BB and *iOS* version that the offset relates to. Additionally, it also lists where in this thesis the symbol was referenced. All offsets refer to the main firmware (`SYS_SW.elf`) of the given BB version after they have been mapped according to the information in the Executable and Linkable Format (ELF)-file. This corresponds to how tools such as *IDA Pro* [53] or *Binary Ninja* [99] load these files automatically.

For the functions contained within the *iOS* kernel cache and dynamic libraries Table A.2 lists their names and symbols. It also lists the respective *iOS* version. Further, a reference to where in this thesis the symbol was discussed is listed.

A.2 Build Manifest

Listing A.1 contains an excerpt from a `BuildManifest.plist` that shows how the BB firmware is referenced in the build manifest. It contains a relative path to the firmware

Offset	Baseband Version	iOS Version	In this Thesis
0x85E9CEDA	ICE17-4.03.05	14.5.1	Listing 4.2
0x86EDDE40	ICE17-4.03.05	14.5.1	Listing A.3
0x858E4C3C	ICE17-4.03.05	14.5.1	Listing A.4
0x8A420E46	ICE19-2.04.07	14.5.1	Listing 4.3
0x8642830C	ICE16-5.02.04	14.5.1	Listing A.2
0x8642831C	ICE16-5.02.04	14.5.1	Listing A.2
0x8642832C	ICE16-5.02.04	14.5.1	Listing A.2
0x019AF01C	ICE18-3.03.05	14.5.1	Listing A.5
0x019A5C12	ICE18-3.03.05	14.5.1	Listing A.6
0x8A0B2AF6	ICE19-2.04.07	14.5.1	Listing A.7
0x8a783c7e	ICE19-2.04.07	14.5.1	Listing 4.7
0x8a4ad204	ICE19-2.04.07	14.5.1	Listing 4.8
0x8a01b37a	ICE19-2.04.07	14.5.1	Listing 4.9
0x8a015d7e	ICE19-2.04.07	14.5.1	Listing 4.10
0x8a84c908	ICE19-2.04.07	14.5.1	Listing 4.12
0x8a84d96c	ICE19-2.04.07	14.5.1	Listing 4.12
0x8a84d818	ICE19-2.04.07	14.5.1	Listing 4.13
0x8a8873a8	ICE19-2.04.07	14.5.1	Listing 4.14
0x8a84c8bc	ICE19-2.04.07	14.5.1	Listing 4.15

Table A.1: Symbols and offsets used in this thesis in the BB.

Symbol	File	iOS Version	In this Thesis
_ZN9AriHostRt7SendRawEPhjj	libARIServer.dylib	14.3	Listing 4.16
_ZN9AriHostRt12InboundMsgCBEPhm	libARIServer.dylib	14.3	Listing 4.17
IOACIPCTagList::freeTagChain()	kernelcache.research.iphone12b	14.0.1	Listing 6.4
IOACIPCTagList::initialize()	kernelcache.research.iphone12b	14.0.1	Listing A.8
IOACIPCTagList::getFreeTag()	kernelcache.research.iphone12b	14.0.1	Listing A.9

Table A.2: Symbols and offsets within libraries in *iOS*.

archive as well as a base64-encoded SHA2-384 digest for each binary contained in the archive. This sample has been taken from a .ipsw-file that contained the *iOS* 14.3 update for an *iPhone* SE 2020 (iPhone12,8).

A.3 Processor Core Counts

This section provides more examples of listings that have been obtained by reverse-engineering the *ICE16* to *ICE19* BBs. Specifically, these snippets give further indication of the processor architecture of the different modem generations.

```
<!-- ... -->
<dict>
  <key>BuildIdentities</key>
  <array>
    <dict>
      <!-- ... -->
      <key>Manifest</key>
      <dict>
      <!-- ... -->
      <key>BasebandFirmware</key>
      <dict>
        <key>Info</key>
        <dict>
          <key>Path</key>
          <string>Firmware/ICE19-2.03.04.Release.bbfw</string>
        </dict>
        <key>SystemSW-DownloadDigest</key>
        <data>
        DT/hAOsjLl5VRfC4N2CWMUw0vELl6U3Z2TDB
        YO8JIpZ8SYUgOh5PLN+ecmb4FIJx
        </data>
<!-- ... -->
```

Listing A.1: Excerpt from a `BuildManifest.plist` contained in an *iOS 14.3* update.

Several format strings within the *ICE16* BB support the claim that it is based on a three-core variant of the *ARM* Cortex-A5 processor. Three of these format strings can be found in Listing A.2.

```
" Core 0: %d\r\n"
" Core 1: %d\r\n"
" Core 2: %d\r\n"
```

Listing A.2: Strings hinting at a three-core processor in the *ICE16* firmware.

Similar to *ICE16* BBs, *ICE17* BBs contain three cores. This statement is supported by the format string shown in Listing A.3. As discussed in Section 4.3, the processor for this generation was an *ARM* Cortex-A5 variant and this core count aligns with that proposition.

```
"Idle counters (reset at every wakeup)\r\n\
 Core 0: %d\r\n\
 Core 1: %d\r\n\
 Core 2: %d"
```

Listing A.3: A format string from the *ICE17* BB indicating the amount of cores.

Further indications for a processor with three cores can be found in Listing A.4. Here, a string containing an error message is returned, if the core id is bigger than two. Thus, three values for a core id are plausible: zero, one, and two.

```
1 if (v32[0] > 2 )
2     return "ERROR: cs_etb: coreID range error";
```

Listing A.4: Check if a given core id is valid within the *ICE17* BB.

Listing A.5 gives some indication about the processor architecture of the *ICE18* BBs. Unlike *ICE17* or *ICE19* BBs, they are based on an unknown *x86* processor that likely has two cores.

```
1 if ( possible_core_id >= 2 )
2     return sprintf_like("%s Error: Invalid CoreID : %d",
3         "sah_save_current_thread_info:", possible_core_id);
```

Listing A.5: Check if a given core-id is valid within the *ICE18* BB.

Further clues of two cores can be found in Listing A.6. While iterating over every possible core-id, it checks whether the given core is stalled. The variable called `v64` seems to indicate the core-id of the core on which the thread that is performing the check is running.

```
1 for ( int possible_core_id = 0; possible_core_id < 2; ++
    possible_core_id )
2 {
3     if ( possible_core_id != v70 &&
4          dword_7775778[possible_core_id] < dword_7775778[v71] )
5         printf_like("Core %u seems to be stall", possible_core_id);
6 }
```

Listing A.6: Check whether a core could be stalled in the *ICE18* BB.

ICE19 BBs seem to have been based on *ARM's* Cortex-A5 series of processors again. However, this time a variant with four cores as indicated by Listing A.7 is used. The code iterates over all possible core-ids and checks whether the corresponding core is stalled. If so, an error string is returned. The loop runs from core-id zero until core-id three, which suggests a processor with four cores.

```
1 for ( int possible_core_id = 0; possible_core_id < 4; ++
    possible_core_id ) {
2     if ( possible_core_id != a1 ) {
3         if ( v8[possible_core_id] < v8[a1] )
4             result = sprintf_like("Core %u seems to be stalled",
                  possible_core_id);
5     }
6 }
```

Listing A.7: Check which cores are stalled within the *ICE19* BB.

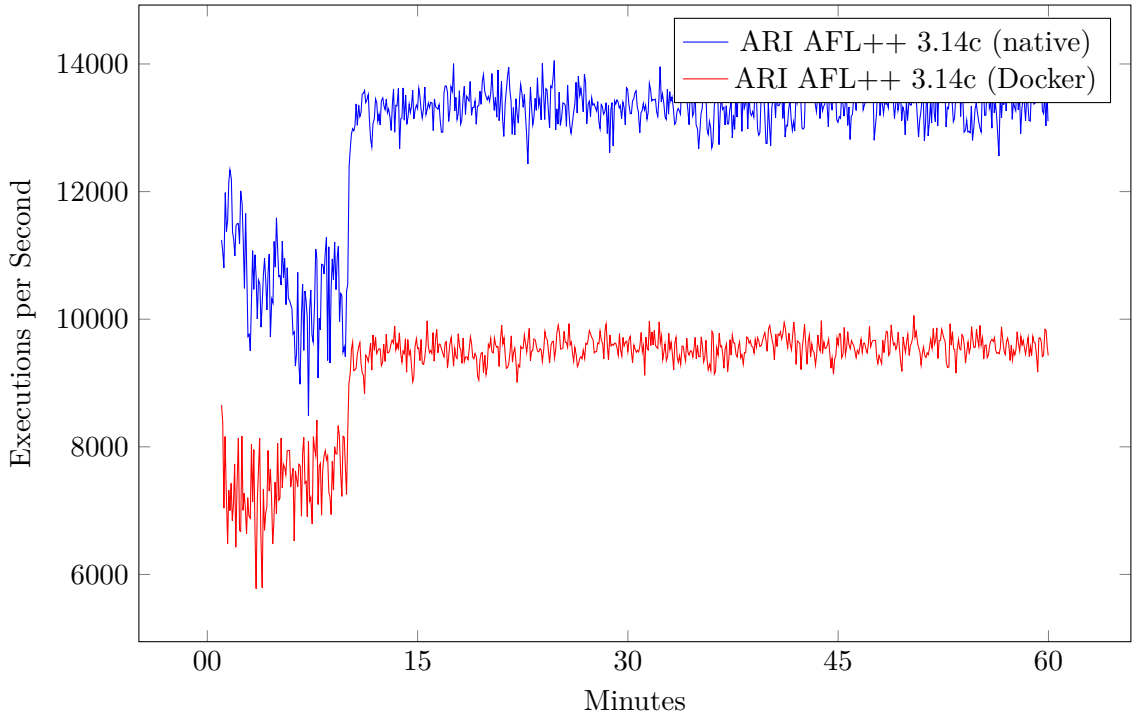

Figure A.1: Fuzzing speed of AFL++ 3.14c with and without Docker

A.4 Docker and Fuzzing Performance

The plot in Figure A.1 shows the performance difference of running American Fuzzy Lop Plus Plus (AFL++) version 3.14c with Docker and natively. Fuzzing performance decreased from 12865.92 executions per second to 9208.98. A reduction of 28.4%.

A.5 Further Code Snippets involving IOACIPCTagList

In this section, code snippets that have been reverse-engineered from an *iOS* 14.0 kernel cache will be provided. They document the mechanisms described in Section 6.3.2 and have been decompiled using *IDA Pro* and were cleaned up manually. Some parts have been removed for clarity, such as the initialization of fields whose meaning is not clear or simple sanity checks (e.g. is the length parameter for the initializer bigger than two).

Listing A.8 shows how an `IOACIPCTagList` is initialized. In Line 5 an array of `IOACIPCTagEntry` is allocated and if the allocation succeeded, the rest of the structure is initialized.

The `IOACIPCTagList::getFreeTag()` shown in Listing A.9 will update the `index` field to the actual index of the `IOACIPCTagEntry` if it is the next free tag in the list. Finally, the index and the address of the tag entry will be returned by reference.

```
1  __int64 IOACIPCTagList::initialize(unsigned int length)
2  {
3      /* ... */
4
5      __int64 result = operator new[](24LL * length);
6      this->entries = (IOACIPCTagEntry *)result;
7
8      if ( result ) {
9
10         this->chainLength = length;
11         this->nextFreeTag = 0;
12         this->lastIndex = length - 1;
13
14         /* ... */
15
16         for (int i = 0; i < lastEntry; i++) {
17             this->entries[i].index = i+1;
18             this->entries[i].claimed = 1;
19         }
20
21         this->entries[lastEntry].claimed = 1;
22         this->entries[lastEntry].index =  length - 1;
23
24         result = 1;
25     }
26
27     return result;
28 }
```

Listing A.8: Initialization of an `IOACIPCTagList` object.

```
1  __int64 IOACIPCTagList::getFreeTag(unsigned __int16 *a2,
       IOACIPCTagEntry *a3)
2  {
3      if ( v3[this->nextFreeTag].claimed != 1 )
4          IOACIPCTagList::getFreeTag(); // this causes a panic
5
6      if ( v3[this->nextFreeTag].index == this->nextFreeTag )
7          return 0;
8
9      *a2 = this->nextFreeTag;
10
11     /*...*/
12
13     *a3 = &this->entries[this->nextFreeTag];
14
15     /*...*/
16
17     unsigned __int16 support = this->nextFreeTag;
18     this->nextFreeTag = this->entries[support].index;
19     this->entries[support].index = support;
20     return 1;
21 }
```

Listing A.9: Returning a free tag from an `IOACIPCTagList` object.

List of Figures

List of Tables

List of Algorithms

List of Listings

Glossary

3GPP
3rd Generation Partnership Project. 5–7

AFL
American Fuzzy Lop. 11, 20, 21, 57

AFL++
American Fuzzy Lop Plus Plus. 2, 11, 19, 21, 49–52, 59, 67–72, 77, 83, 85, 86, 90, 95

AP Application Processor. ix, 1, 2, 5–7, 10, 12, 13, 16, 17, 28, 34, 38, 46, 84, 85, 89, 90

API
Application Programming Interface. 10, 21, 61

ARI
Apple Remote Invocation. ix, xi, 2, 3, 6, 7, 10, 15, 18, 23, 39, 42–51, 53–55, 57–65, 69, 71, 73–78, 83–86, 89, 90

ASLR
Address Space Layout Randomization. 17, 33, 34

BB Baseband. ix, 1–3, 5–13, 15–19, 23–43, 45–51, 53, 54, 58–62, 64, 65, 71, 73, 74, 76–80, 83–87, 89–94

CPU
Central Processing Unit. ix, 26, 27, 69, 83, 84, 89, 90

CRC
Cyclic Redundancy Check. 52

CSV
Comma-Separated Values. 40, 57

DoS
Denial of Service. 18, 81

DSP
Digital Signal Processor. 2, 27, 84, 89

ELF
Executable and Linkable Format. 25–30, 51, 53, 91

ETSI
European Telecommunications Standards Institute. 7

GPRS
General Packet Radio Service. 17

GSM
Global System for Mobile Communications. 1, 7, 19, 25, 85

IMEI
International Mobile Equipment Identity. 18

IMSI
International Mobile Subscriber Identity. 16

ISM
IP Multimedia Subsystem. 44

JSON
JavaScript Object Notation. 76

LCG
Linear Congruential Generator. 58, 62, 63, 65

LTE
Long-Term Evolution. 1, 5–7, 13, 19, 27, 69, 85

MIPI
Mobile Industry Processor Interface Alliance. 38–40

NVRAM
Non-Volatile Random-Access Memory. 17

OS Operating System. 1, 5, 12, 13

PCIe
Peripheral Component Interconnect Express. 5, 18, 25

PWS
Public Warning System. 16

QEMU
Quick Emulator. 8, 19

QMI
Qualcomm Mobile Station Modem Interface. 17

ROP
Return Oriented Programming. 17

RTOS
Real-Time Operating System. 1, 2, 5, 23, 33, 40, 89

SDK
Software Development Kit. 46

SIM
Subscriber Identification Module. 44, 45

SMS
Short Message Service. 7, 13, 18, 24, 44, 60, 65, 73, 76, 86, 90

TLS
Transport Layer Security. 11

TLV
Type-Length-Value. 43, 45, 47, 58, 63, 83, 85, 86

UMTS
Universal Mobile Telecommunications System. 1, 5, 13, 85

Bibliography

[1] 3GPP. *Releases*. 3GPP - A Global Initiative. 2020-12-11. URL: https://www.3gpp.org/specifications/67-releases (visited on 2021-09-17).

[2] Advanced Fuzzing League++. *AFLplusplus - Tips for performance optimization*. 2021-06-25. URL: https://github.com/AFLplusplus/AFLplusplus/blob/3.14c/docs/perf_tips.md (visited on 2021-09-17).

[3] National Security Agency. *Ghidra*. URL: https://ghidra-sre.org/ (visited on 2021-09-17).

[4] Airbus security lab. *cpu_rec*. 2021-06-09. URL: https://github.com/airbus-seclab/cpu_rec (visited on 2021-09-17).

[5] Alisa Esage. "Advanced Hexagon DIAG". Remote Chaos Expierence, 2020-12-29. URL: https://www.youtube.com/watch?v=94NwlrtGF7I (visited on 2021-09-17).

[6] Apple. *Apple introduces dual camera iPhone 11*. Apple Newsroom. 2019-09-10. URL: https://www.apple.com/newsroom/2019/09/apple-introduces-dual-camera-iphone-11/ (visited on 2021-09-17).

[7] Apple Inc. *Apple to acquire the majority of Intel's smartphone modem business*. Apple Newsroom. 2019-07-25. URL: https://www.apple.com/newsroom/2019/07/apple-to-acquire-the-majority-of-intels-smartphone-modem-business/ (visited on 2021-09-17).

[8] Apple Inc. *Core Telephony*. Apple Developer Documentation. URL: https://developer.apple.com/documentation/coretelephony (visited on 2021-09-17).

[9] Apple Inc. *Overview of the Mach-O Executable Format*. Apple Developer Documentation. 2014-03-10. URL: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Articles/MachOOverview.html (visited on 2021-09-17).

[10] ARM Ltd. *ARM Cortex-A Series Programmer's Guide for ARMv7-A - Coprocessor 15*. Arm Developer. URL: https://developer.arm.com/documentation/den0013/d/ARM-Processor-Modes-and-Registers/Registers/Coprocessor-15?lang=en (visited on 2021-09-17).

[11] Arm Ltd. *CoreSight Architecture*. Arm Developer. URL: https://developer.arm.com/architectures/cpu-architecture/debug-visibility-and-trace/coresight-architecture (visited on 2021-09-17).

[12] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. "NAUTILUS: Fishing for Deep Bugs with Grammars". In: *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019-02. DOI: 10.14722/ndss.2019.23412.

[13] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. "IJON: Exploring Deep State Spaces via Fuzzing". In: *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2020-05, pp. 1597–1612. DOI: 10.1109/SP40000.2020.00117.

[14] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. "RESTler: Stateful REST API Fuzzing". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019-05, pp. 748–758. DOI: 10.1109/ICSE.2019.00083.

[15] Fabrice Bellard. "QEMU, a fast and portable dynamic translator". In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '05. USA: USENIX Association, 2005-04-10, p. 41.

[16] Erik Buchanan, Ryan Roemer, Stefan Savage, and Hovav Shacham. "Return-oriented Programming: Exploitation without Code Injection". In: *Black Hat USA* (2008-08), p. 53. URL: https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf (visited on 2021-09-17).

[17] Seamus Burke. "A Journey Into Hexagon: Dissecting a Qualcomm Baseband". DEF CON 26. Las Vegas, Nevada, USA, 2018-08-09. DOI: 10.5446/39735.

[18] Cadence Design Systems, Inc. *Tensilica Customizable Processor and DSP IP*. URL: https://ip.cadence.com/ipportfolio/tensilica-ip (visited on 2021-09-17).

[19] Amat Cama. "A walk with Shannon: A walkthrough of a pwn2own baseband exploit". INSOMNI'HACK 2018. Geneva, Switzerland, 2018-03-23. URL: https://www.youtube.com/watch?v=6bpxrfB9ioo (visited on 2021-09-17).

[20] Check Point Software Technologies Ltd. *Karta / Thumbs Up*. 2021-06-08. URL: https://github.com/CheckPointSW/Karta (visited on 2021-09-17).

[21] Yurong Chen, Tian Lan, and Guru Venkataramani. "Exploring Effective Fuzzing Strategies to Analyze Communication Protocols". In: *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*. FEAST'19. New York, NY, USA: Association for Computing Machinery, 2019-11-15, pp. 17–23. DOI: 10.1145/3338502.3359762.

[22] Jiska Classen. “Fuzzing the phone in the iPhone”. Remote Chaos Expierence, 2020-12-28. URL: https://media.ccc.de/v/rc3-11358-fuzzing_the_phone_in_the_iphone (visited on 2021-09-17).

[23] Crossbeam. *crossbeam*. URL: https://github.com/crossbeam-rs/crossbeam (visited on 2021-09-17).

[24] Jared D. DeMott, Richard J. Enbody, and William F. Punch. “Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing”. In: *Black Hat USA* (2007), p. 17.

[25] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment*. 1st edition. Indianapolis, Ind: Addison-Wesley, 2006-11-20. 1174 pp. ISBN: 978-0-321-44442-4.

[26] Eldad Eilam and Elliot J. Chikofsky. *Reversing: secrets of reverse engineering*. Indianapolis, IN: Wiley, 2005. 589 pp. ISBN: 978-0-7645-7481-8.

[27] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); AT Command set for GSM Mobile Equipment (ME) (3GPP TS 7.07 version 7.8.0 Release 1998)*. 2003-03. URL: https://www.etsi.org/deliver/etsi_ts/100900_100999/100916/07.08.00_60/ts_100916v070800p.pdf (visited on 2021-09-17).

[28] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; AT command set for User Equipment (UE) (3GPP TS 27.007 version 13.6.0 Release 13)*. 2017-07. URL: https://www.etsi.org/deliver/etsi_ts/127000_127099/127007/13.06.00_60/ts_127007v130600p.pdf (visited on 2021-09-17).

[29] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); Public Warning System (PWS) requirements (3GPP TS 22.268 version 15.2.0 Release 15)*. 2018-10. URL: https://www.etsi.org/deliver/etsi_TS/122200_122299/122268/15.02.00_60/ts_122268v150200p.pdf (visited on 2021-09-17).

[30] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Specification of the SIM Application Toolkit (SAT) for the Subscriber Identity Module - Mobile Equipment (SIM-ME) interface (3GPP TS 11.14 version 8.18.0 Release 1999)*. 2007-06. URL: https://www.etsi.org/deliver/etsi_ts/101200_101299/101267/08.18.00_60/ts_101267v081800p.pdf (visited on 2021-09-17).

[31] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; AT command set for User Equipment (UE) (3GPP TS 27.007 version 10.3.0 Release 10)*. 2011-04. URL: https://www.etsi.org/deliver/etsi_

ts/127000_127099/127007/10.03.00_60/ts_127007v100300p.pdf (visited on 2021-09-17).

[32] European Telecommunications Standards Institute. *Digital cellular telecommunications system (Phase 2+); Use of Data Terminal Equipment - Data Circuit terminating; Equipment (DTE - DCE) interface for Short Message Service (SMS) and Cell Broadcast Service (CBS) (GSM 07.05 version 5.3.0)*. 1997-08. URL: https://www.etsi.org/deliver/etsi_gts/07/0705/05.03.00_60/gsmts_0705v050300p.pdf (visited on 2021-09-17).

[33] European Telecommunications Standards Institute. *Universal Mobile Telecommunications System (UMTS);LTE; Multimedia Broadcast/Multicast Service (MBMS); Protocols and codecs (3GPP TS 26.346 version 12.3.0 Release 12)*. 2014-10. URL: https://www.etsi.org/deliver/etsi_ts/126300_126399/126346/12.03.00_60/ts_126346v120300p.pdf (visited on 2021-09-17).

[34] Kaiming Fang and Guanhua Yan. "Emulation-Instrumented Fuzz Testing of 4G/LTE Android Mobile Devices Guided by Reinforcement Learning". In: *European Symposium on Research in Computer Security*. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 20–40. DOI: 10.1007/978-3-319-98989-1_2.

[35] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. "SoK: Enabling Security Analyses of Embedded Systems via Rehosting". In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 687–701. DOI: 10.1145/3433210.3453093.

[36] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++ : Combining Incremental Steps of Fuzzing Research". In: 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020.

[37] Jan Friebertshauser, Florian Kosterhon, Jiska Classen, and Matthias Hollick. "Polypyus – The Firmware Historian". In: *NDSS 2021 Binary Analysis Research (BAR) Workshop 2021* (2021).

[38] Sam George. *Microsoft acquires Express Logic, accelerating IoT development for billions of devices at scale*. The Official Microsoft Blog. 2019-04-18. URL: https://blogs.microsoft.com/blog/2019/04/18/microsoft-acquires-express-logic-accelerating-iot-development-for-billions-of-devices-at-scale/ (visited on 2021-09-17).

[39] Nico Golde. *There's Life in the Old Dog Yet: Tearing New Holes into Intel/iPhone Cellular Modems*. Comsecuris Blog. 2018-04-04. URL: https://comsecuris.com/blog/posts/theres_life_in_the_old_dog_yet_tearing_new_holes_into_inteliphone_cellular_modems/ (visited on 2021-09-17).

[40] Nico Golde and Daniel Komaromy. “Breaking Band”. REcon 2016. Montreal, Canada, 2016. URL: https://comsecuris.com/slides/recon2016-breaking_band.pdf (visited on 2021-09-17).

[41] Ricardo E. Gonzalez. “Xtensa: a configurable and extensible processor”. In: *IEEE Micro* 20.2 (2000-04), pp. 60–70. DOI: 10.1109/40.848473.

[42] Google. *FuzzBench: Fuzzer Benchmarking As a Service - AFL++ Setup*. 2021. URL: https://github.com/google/fuzzbench (visited on 2021-09-17).

[43] Rahul Gopinath and Andreas Zeller. “Building Fast Fuzzers”. In: *arXiv preprint* (2019-11-18). arXiv: 1911.07707.

[44] Marco Grassi and Kira (Xingyu Chen). “Exploring the MediaTek Baseband”. Offensive Con 20. Berlin, Germany, 2020-02-14. URL: https://www.youtube.com/watch?v=OSJdAGtn8Qw (visited on 2021-09-17).

[45] Marco Grassi, Muqing Liu, and Tianyi Xie. “Exploitation Of A Modern Smartphone Baseband”. In: *Black Hat USA* (2018), p. 17. URL: https://i.blackhat.com/us-18/Thu-August-9/us-18-Grassi-Exploitation-of-a-Modern-Smartphone-Baseband-wp.pdf (visited on 2021-09-17).

[46] Guy. “Burned in Ashes: Baseband Fairy Tale Stories”. REcon 2019. Montreal, Canada, 2019-06-30. URL: https://recon.cx/media-archive/2019/Session.022.Guy.Burned_in_AshesBaseband_Fairy_Tale_Stories-WXoRdFBuXytGB.mp4 (visited on 2021-09-17).

[47] Guy. “From 0 to Infinity”. SyScan 360. Singapore, 2018. URL: https://docs.google.com/presentation/d/19A1JWyOTueZvD8AksqCxtxriNJJgj0vPdq3cNTwndf4 (visited on 2021-09-17).

[48] Dennis Heinze. *fpicker*. 2021-05. URL: https://github.com/ttdennis/fpicker (visited on 2021-09-17).

[49] Dennis Heinze, Jiska Classen, and Matthias Hollick. “ToothPicker: Apple Picking in the iOS Bluetooth Stack”. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020.

[50] Aki Helin. *Radamsa - a general-purpose fuzzer*. URL: https://gitlab.com/akihe/radamsa (visited on 2021-09-17).

[51] Grant Hernandez. *Shannnon S5000 Skeleton*. 2020-01-31. URL: https://github.com/grant-h/shannon_s5000 (visited on 2021-09-17).

[52] Grant Hernandez and Marius Muench. “Emulating Samsung’s Baseband for Security Testing”. BlackHat USA 2020. Las Vegas, Nevada, USA, 2020-08-05. URL: https://www.blackhat.com/us-20/briefings/schedule/#emulating-samsungs-baseband-for-security-testing-20564 (visited on 2021-09-17).

[53] Hex Rays. *IDA Pro*. URL: https://hex-rays.com/ida-pro/ (visited on 2021-09-17).

[54] Intel. *Intel Introduces Portfolio of Commercial 5G New Radio Modems, Extends LTE Roadmap With Intel® XMM™ 7660 Modem*. Intel Corporation. 2017-11-16. URL: https://www.intc.com/news-events/press-releases/detail/191/intel-introduces-portfolio-of-commercial-5g-new-radio (visited on 2021-09-17).

[55] Intel. *Intel System Trace Tool (STT)*. URL: http://designintools.intel.com/product_p/iot25.htm (visited on 2021-09-17).

[56] Intel. *Intel XMM 7660 Modem Product Brief*. 2017. URL: https://www.intel.de/content/dam/www/public/us/en/documents/product-briefs/xmm-7660-brief.pdf (visited on 2021-09-17).

[57] Intel. *Intel® System Debugger*. URL: https://www.intel.com/content/www/us/en/develop/documentation/system-debug-user-guide/top.html (visited on 2021-09-17).

[58] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. "Opening Pandora's box through ATFuzzer: dynamic analysis of AT interface for Android smartphones". In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019-12-09, pp. 529–543. DOI: 10.1145/3359789.3359833.

[59] Tobias Kröll, Stephan Kleber, Frank Kargl, Matthias Hollick, and Jiska Classen. "ARIstoteles – Dissecting Apple's Baseband Interface". In: 26th European Symposium on Research in Computer Security (ESORICS) 2021. 2021-10, p. 20.

[60] lcq2. *x86 finds its way into your iPhone*. 2018-09-14. URL: https://lcq2.github.io/x86_iphone/ (visited on 2021-09-17).

[61] Jonathan Levin. *JTool2 - Taking the O out of otool - squared*. URL: http://newosxbook.com/tools/jtool.html (visited on 2021-09-17).

[62] Jonathan Levin. **OS internals, Volume II - Kernel Mode*. 2nd ed. Technologeeks Publishing, 2018-10-24. ISBN: 978-0-9910555-7-9.

[63] Napier Lopez. *Report: Apple's 'Project OGRS' almost turned the iPhone into a walkie-talkie*. TNW | Plugged. 2019-08-26. URL: https://thenextweb.com/news/report-apples-project-ogrs-almost-turned-the-iphone-into-a-walkie-talkie (visited on 2021-09-17).

[64] ARM Ltd. *Cortex-A5*. Arm Developer. URL: https://developer.arm.com/ip-products/processors/cortex-a/cortex-a5 (visited on 2021-09-17).

[65] m4b. *goblin*. URL: https://docs.rs/goblin/0.4.1/goblin/ (visited on 2021-09-17).

[66] Dominik Maier and Andrea Fioraldi. "Fuzzers like LEGO". Remote Chaos Experience, 2020-12-28. URL: https://media.ccc.de/v/rc3-699526-fuzzers_like_lego (visited on 2021-09-17).

[67] Dominik Maier, Lukas Seidel, and Shinjo Park. "BaseSAFE: baseband sanitized fuzzing through emulation". In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '20. New York, NY, USA: Association for Computing Machinery, 2020-07-08, pp. 122–132. DOI: 10.1145/3395351.3399360.

[68] Slava Makkaveev. *Security probe of Qualcomm MSM data services*. Check Point Research. 2021-05-06. URL: https://research.checkpoint.com/2021/security-probe-of-qualcomm-msm/ (visited on 2021-09-17).

[69] Dennis Mantz. *Frizzer*. 2020-08. URL: https://github.com/demantz/frizzer (visited on 2021-09-17).

[70] Microsoft Corporation. *Azure RTOS ThreadX*. 2021-05-08. URL: https://github.com/azure-rtos/threadx (visited on 2021-09-17).

[71] Microsoft Corporation. *Azure RTOS TraceX Documentation*. 2021-08. URL: https://docs.microsoft.com/en-us/azure/rtos/tracex/ (visited on 2021-09-17).

[72] Barton P. Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM* 33.12 (1990-12-01), pp. 32–44. DOI: 10.1145/96267.96279.

[73] MIPI Alliance. *MIPI Parallel Trace Interface (MIPI PTI)*. MIPI. 2011-10. URL: https://www.mipi.org/specifications/pti (visited on 2021-09-17).

[74] MIPI Alliance. *MIPI System Trace Protocol (MIPI STP)*. MIPI. 2016-02. URL: https://www.mipi.org/specifications/stp (visited on 2021-09-17).

[75] MIPI Alliance. *MIPI Trace Wrapper Protocol (MIPI TWP)*. MIPI. 2014-12. URL: https://www.mipi.org/specifications/twp (visited on 2021-09-17).

[76] György Miru. *Path of Least Resistance: Cellular Baseband to Application Processor Escalation on Mediatek Devices*. Comsecuris Blog. 2017-07-28. URL: https://comsecuris.com/blog/posts/path_of_least_resistance/ (visited on 2021-09-17).

[77] Charles Muiruri, Nitay Artenstein, and Anna Dorfman. "The Baseband Basics: Understanding, Debugging and Attacking the Mediatek Communication Processor". OPCDE Kenya 2018. Nairobi, Kenya, 2018-06-27. URL: https://raw.githubusercontent.com/comaeio/OPCDE/master/2018/Kenya/Charles%20Nitay%20Anna%20-%20The%20Baseband%20Basics.pdf (visited on 2021-09-17).

[78] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. "SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale." In: *USENIX Security Symposium*. Vol. 168. San Francisco, CA, 2011.

[79] Collin Mulliner and Charlie Miller. “Fuzzing the Phone in your Phone”. In: *Black Hat USA* 25 (2009), p. 31. URL: https://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf (visited on 2021-09-17).

[80] Anh Quynh Nguyen and Hoang Vu Dang. “Unicorn: Next Generation CPU Emulator Framework”. In: *BlackHat USA*. 2015-08-05. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Nguyen-Unicorn-Next-Generation-CPU-Emulator-Framework.pdf (visited on 2021-09-17).

[81] Ole André V. Ravnås. *Frida - A world-class dynamic instrumentation framework*. URL: https://frida.re/ (visited on 2021-09-17).

[82] PANDA Team. *PANDA.re*. URL: https://panda.re (visited on 2021-09-17).

[83] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. “SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask”. In: *42nd IEEE Symposium on Security and Privacy* (2021-05-25). DOI: 10.1109/SP40001.2021.00012.

[84] PCI-SIG. *PCI Specifications*. 2021. URL: https://pcisig.com/specifications?field_technology_value%5B%5D=express&field_revision_value%5B%5D=1&field_document_type_value%5B%5D=specification&speclib= (visited on 2021-09-17).

[85] radare org. *radare2*. URL: https://www.radare.org/n/index.html (visited on 2021-09-17).

[86] Ole André V. Ravnås. *frida-compile*. 2021-06-21. URL: https://github.com/frida/frida-compile (visited on 2021-09-17).

[87] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, 2018-08. DOI: 10.17487/RFC8446.

[88] Rust Team. *Rust Programming Language*. URL: https://www.rust-lang.org/ (visited on 2021-09-17).

[89] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. “AddressSanitizer: A Fast Address Sanity Checker”. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12). 2012, pp. 309–318.

[90] Victoria Shannon and International Herald Tribune. “The Rise and Fall of the Modem King”. In: *The New York Times* (1999-01-07). URL: https://www.nytimes.com/1999/01/07/news/the-rise-and-fall-of-the-modem-king.html (visited on 2021-09-17).

[91] Natalie Silvanovich. *SMS Simulator for iOS 11.3.1*. 2019-08-07. URL: https://github.com/googleprojectzero/iOS-messaging-tools/tree/master/SmsSimulator (visited on 2021-09-17).

[92] Evgeniy Stepanov and Konstantin Serebryany. “MemorySanitizer: Fast detector of uninitialized memory use in C++”. In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015-02, pp. 46–55. DOI: 10.1109/CGO.2015.7054186.

[93] Telit Communications. *xN930 M.2 EVK User Guide*. 2013-12-29. URL: https://web.archive.org/web/20210730150438/http://www.iot.com.tr/uploads/pdf/Telit_xN930_EVK_User_Guide_r0.pdf (visited on 2021-09-17).

[94] Tensilica, Inc. *Xtensa Instruction Set Architecture (ISA) Reference Manual*. 2010, p. 662. URL: https://0x04.net/~mwk/doc/xtensa.pdf (visited on 2021-09-17).

[95] The iPhone Wiki. *Baseband Device*. 2021-07-18. URL: https://www.theiphonewiki.com/wiki/Baseband_Device (visited on 2021-09-17).

[96] The iPhone Wiki. *Kernelcache*. 2019-06-06. URL: https://www.theiphonewiki.com/wiki/Kernelcache (visited on 2021-09-17).

[97] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. “ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem”. In: 27th USENIX Security Symposium (USENIX Security 18). 2018, pp. 273–290.

[98] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. 1995-05. URL: https://refspecs.linuxbase.org/elf/elf.pdf (visited on 2021-09-17).

[99] Vector 35 Inc. *Binary Ninja*. URL: https://binary.ninja/ (visited on 2021-09-17).

[100] Ralf-Philipp Weinmann. “Baseband attacks: remote exploitation of memory corruptions in cellular protocol stacks”. In: *Proceedings of the 6th USENIX conference on Offensive Technologies*. WOOT’12. USA: USENIX Association, 2012-08-06, p. 2.

[101] Ralf-Philipp Weinmann. “Baseband Exploitation in 2013”. Chaos Communication Congress 30. Hamburg, Germany, 2013-12-27. URL: https://media.ccc.de/v/30C3_-_5618_-_en_-_saal_1_-_201312272145_-_baseband_exploitation_in_2013_-_rpw_esizkur#t=19 (visited on 2021-09-17).

[102] Christos Xenakis and Christoforos Ntantogian. “Attacking the baseband modem of mobile phones to breach the users’ privacy and network security”. In: *2015 7th International Conference on Cyber Conflict: Architectures in Cyberspace*. 2015-05, pp. 231–244. DOI: 10.1109/CYCON.2015.7158480.

[103] Michal Zalewski. *American Fuzzy Lop - Whitepaper*. 2016. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt (visited on 2021-09-17).