École Polytechnique Fédérale de Lausanne

Over-the-Air LTE Protocol Fuzzing

by Marc Egli

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer

Thesis Advisor

Prof. Dr. sc. KAIST Yongdae Kim

External Expert

KAIST EE - SYSEC

N26 201 (CHiPS) - KR-34141 Daejeon

August 31, 2023

Une quête commence toujours par la chance du débutant.

Et s'achève toujours par l'épreuve du conquérant.

— Paul Coelho, L'Alchimiste

To my parents Muriel and Martin, my sister Chantal and

to the memory of my grandparents Annemarie and Heinz Egli.

Acknowledgments

I want to thank my advisors, Dr. Mathias Payer and Dr. Yongdae Kim for their invaluable support, clear guidance, and the many discussions we had that helped me accomplish this work. Their expertise, patience, and constructive feedback have been instrumental in shaping this thesis and enhancing my research skills. I also want to thank Dr. Insu Yun for all the feedback and precious ideas he shared that greatly enhanced this work. Finally, I want to show all my gratitude to the Ph.D. Candidate CheolJun Park who was my mentor during the 6 months of my thesis.

I also want to show my gratitude to all the members of the SYSSEC research group with whom I spent six months. Their assistance with various external matters, administrative tasks, and invaluable assistance in my work made my time in Korea significantly easier and more productive. From my very first day, I felt fully integrated into the group and surrounded by brilliant individuals working on captivating topics. Their support and camaraderie made my 6 months an unforgettable experience.

I am thankful to all my family, my girlfriend Emma, and my friends who supported me during my 6 months abroad but also for all the encouragement they gave me during my years at EPFL and while writing this work.

I also have a special acknowledgment for the CTF teams that I have played with for 2 years: polygl0ts and 0rganizers. During various competitions, I learned many valuable technical skills from a multitude of talented people. I am grateful to know them.

Finally, I would like to show my gratitude again to Dr. Yongdae Kim and all personnel of KAIST who made it possible for me to carry out this research in person in the Republic of Korea.

Daejeon, August 31, 2023 Marc Egli

Abstract

The baseband is a critical component in modern smartphones as it handles all wireless communication such as GPS, Wi-Fi, cellular data, or Bluetooth with other parties. Therefore, memory corruption in baseband firmware presents high risks as it can serve as an initial chain link for a 0-click full-chain exploit, taking over the device's operating system. However, finding such vulnerabilities is challenging because baseband firmware is proprietary and closed source.

Current approaches rely on static or dynamic analysis to verify the security of basebands. However, static analysis and emulation-based fuzzing require a lot of manpower to reverse engineer the firmware binary to understand its internals. Dynamic testing shows excellent results at uncovering semantic bugs using protocol specifications but only tests a subset of properties.

This work shows that over-the-air fuzzing of baseband is a reasonable approach. We present OTAFuzz, an over-the-air fuzzing tool for RRC and NAS-EMM protocols. Our tool relies on grammar-aware mutation strategies and generation-based fuzzing to compensate for the lack of code-coverage feedback due to the black-box nature of the baseband when tested over the air.

We show that over-the-air fuzzing is successful at finding memory vulnerabilities in basebands. While fuzzing 4 different baseband vendors across 5 different devices we uncovered sixteen 1-day RRC vulnerabilities in old Exynos baseband, two 0-day NAS-EMM vulnerabilities in Exynos baseband, and one unconfirmed vulnerability in MediaTek firmware (classification as 1-day or 0-day pending). Additionally, we share an extensive analysis of different oracles to detect baseband crashes.

Contents

Chapter 1

Introduction

The baseband processor (BP) is a crucial component in a modern smartphone is in charge of all radio communication. Vulnerabilities in baseband firmware have a significant impact, as a single vulnerable firmware image can impact millions of devices in the world. Furthermore, a successful baseband exploit can be chained to attack the device's operating system. Attackers can target the application processor (AP), where the device OS runs, through BP as they communicate through IPC. The lack of mitigations [19] and the potential of 0-click exploits [28] makes it an even more interesting attack surface. In early 2023, the Project Zero team from Google reported eighteen vulnerabilities in the Samsung Exynos baseband [16]. The findings were severe, as evidenced by the only protective measure recommended: fully disabling WiFi-calling and Voice-over-LTE awaiting an official patch.

Baseband firmware is proprietary and closed-source by nature. Qualcomm goes even a step further by using its own custom chipset architecture called Hexagon [10]. This fact makes baseband security research harder as it brings multiple challenges to applying usual techniques. Static analysis of baseband firmware, for example, requires a lot of human effort to reverse engineer and figure out the location of specific functions. Dynamic analysis is also significantly harder to apply as the system is black-box which leads to a lack of feedback information about code coverage when applying

fuzzing. It is also more comlicateed to detect crashes.

Previous research was successful at uncovering baseband vulnerabilities but faced various challenges with their approaches. Emulation-based fuzzing has been one of the major areas of research in baseband fuzzing [20] [29] in recent years. With emulation, it is possible to add debugging code and memory sanitization to the firmware. Additionally, code-coverage feedback is available which is crucial to perform mutation-based fuzzing. Emulation is complex and cannot be applied uniformly to all baseband images as it requires first to reverse engineer the firmware and understand the structure of the target firmware binary. Because the internal structure of the firmware varies greatly between vendors previous work on baseband emulation only could re-host basebands from 1 or 2 different vendors.

Static analysis has been proven successful [25] but necessitates a lot of manual work to reverse engineer the firmware image. While part of the reverse engineering can be semi-automated [24] it still requires a deep understanding of the firmware image to apply it. Static analysis also struggles to account for dynamic behaviors or side effects that might arise when running the firmware image. The advantage of static analysis is that it can uncover both semantic and syntactic bugs.

Dynamic testing works of baseband images focus on finding mismatching behaviors between the firmware and the specifications. Previous works [34] [22] [26] [9] use specifications to build specific test cases. They then determine the expected behavior of the baseband for each test by referring to specifications. Then, the test cases are sent to a test device over-the-air and the response is monitored. Many logical bugs in multiple basebands were uncovered by these works. However, the number of test cases is low and they target only a subset of functionalities of a baseband protocol. Furthermore, the majority of vulnerabilities found with this method are semantic bugs rather than memory-related issues.

We present OTAFuzz 1.1, an over-the-air baseband fuzzer targeting RRC and NAS protocols. Fuzzing LTE protocol implementations over-the-air offers the benefit of being able to test any baseband, independent of the vendor, firmware image, or version. This method also allows testing

the firmware running in its native environment and therefore captures all possible side effects that arise during LTE communication. Baseband emulation-based fuzzing research only targets Exynos (Samsung) and MediaTek firmware, but with our approach, it is possible to fuzz modems from other vendors like Qualcomm or HiSilicon. Previous research confirmed their memory bugs in basebands by sending the crash over the air to a test device. This information gave us additional confidence to explore the over-the-air fuzzing approach. Using an over-the-air approach enables also stateful fuzzing because it is possible to prepare the state of the baseband before testing it.

Nevertheless, the over-the-air approach brings multiple challenges. First, the testing speed is several orders of magnitudes smaller than emulation which, even though it can only operate on a restricted set of basebands, achieves a fuzzing speed of hundreds of test cases per second per core [29]. Then, it is also not possible to get any code coverage feedback due to the baseband firmware being black-box. To compensate for the speed and the lack of code-coverage feedback we use generation-based fuzzing and apply grammar-aware mutations to protocol fields. Another challenge is the lack of debugging information available to detect and analyze crashes when fuzzing over the air. Indeed, accessing the baseband is nearly impossible as it is in its own isolated environment from the device's operating system running on the AP. With OTAFuzz, we explore which information sources from the device can be used to detect baseband crashes.

The goal of this work is to explore the capabilities and limitations of over-the-air fuzzing for LTE protocols. We aim to discuss how we built a generation-based fuzzer for RRC and NAS and compare different oracles to detect baseband crashes.

With OTAFuzz, we were able to re-discover vulnerabilities in outdated baseband firmware for Exynos and MediaTek basebands. In total, we found 16 RRC 1-day vulnerabilities in Exynos, 1 RRC vulnerability in MediaTek baseband, and two NAS-EMM 0-day in the latest Exynos firmware image. However, the two 0-day vulnerabilities were already reported but not patched nor assigned a CVE by the time we discovered them with OTAFuzz.

In the upcoming chapters, we present the necessary background information to understand

our work. Then, we present the design of OTAFuzz and key implementation details of the system. Finally, we discuss the results of the experiments we performed to evaluate our system.

With this work we bring the following core contributions:

1. OTAFuzz, an over-the-air RRC and NAS-EMM protocol fuzzer to test their implementations in baseband on any smartphone. The ability to fuzz any baseband without any special device setup or re-hosting makes OTAFuzz a universal fuzzing tool for basebands.

2. A set of effective grammar-aware mutation strategies for RRC and NAS-EMM protocols, specifically designed to uncover memory vulnerabilities in baseband firmware.

3. An analysis of different oracles to detect baseband crashes on a smartphone. Detecting baseband failures while over-the-air fuzzing is a key aspect of OTAFuzz.

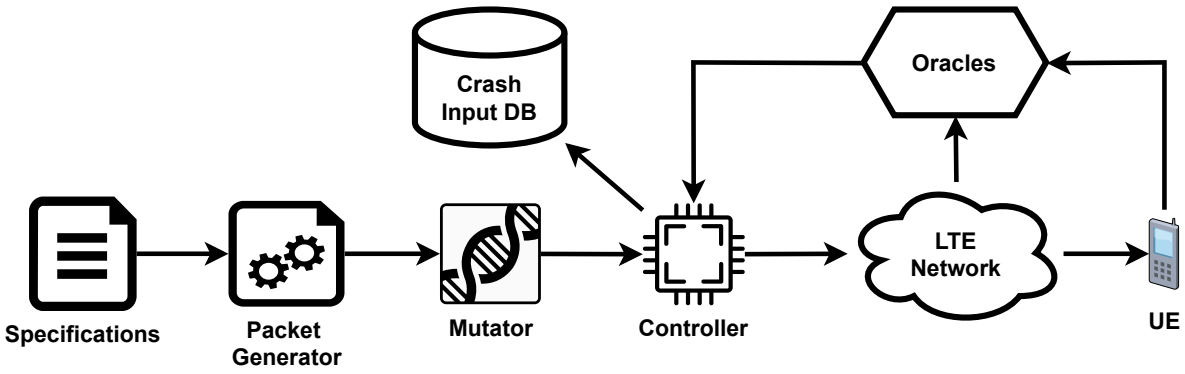

Figure 1.1: OTAFuzz system overview

Chapter 2

Background

In this chapter, we present an outline of fuzzing, the LTE network architecture, and the format of the grammars that describes the protocols we target. First, we explain the fundamental concepts of fuzzing and how fuzzing is used for network protocol implementation testing. Then, we explore the core functionalities of the LTE network and present the protocols we are targeting in this work. Finally, we cover ASN.1 grammar and Tag-Length-Value (TLV) format specifications that define the targeted protocols.

2.1 Fuzzing

Fuzzing is an automatic software testing technique where randomly generated data is passed to a target program. Many testing techniques do not scale well on large-scale software applications. For example, unit testing may not cover all possible scenarios and interactions within the application and requires extensive knowledge of the tested program. On the other end of the spectrum, formal verification becomes impractical as soon as the target application exceeds the magnitude of hundreds of lines of code as the number of constraints that must be verified increases. Fuzzing fills this gap by being able to find vulnerabilities in large-scale software by executing them over and

over using different input values. A fuzzer monitors the target program for crashes or unexpected behavior. This is done by one or more oracles that indicate whether a failure occurred in the target program for a specific input. For each failure, the fuzzer generates a crash report that contains the input to the target that triggered the failure. This allows to reproduction of the crash and to determine its cause and severity. Fuzzing has proven to be a highly effective method in finding vulnerabilities in large software applications such as kernels, web browsers, and compilers [41] [46] [5]. As of August 2023, the largest open-source fuzzing project oss-fuzz uncovered more than 10,000 vulnerabilities in more than 1000 projects [17].

2.1.1 Generational fuzzing

Generation-based fuzzers [14] generate random test cases based on a set of rules. These can be obtained either from the knowledge of the internal structure of the target or its specifications. The input models are used to generate valid inputs which are then mutated to produce diverse test cases. This type of fuzzer differs from the traditional coverage-guided mutation-based fuzzers which rely on coverage feedback. Furthermore, coverage feedback requires instrumentation of the target which is not always possible when doing black-box fuzzing. The advantage of generation-based fuzzing is that it can reach deeper program states within the target program as the generated inputs conform to the target's input model. It does not rely on code coverage to discover new execution paths. For example, in Figure 2.1, a simple target program might only accept an input starting with 50 'A's. Thus, a generation-based fuzzer can be built that will generate test inputs that start with the required prefix. A mutation-based coverage-guided fuzzer will struggle to go past the prefix constraint check. The same code-coverage feedback will be observed if the input contains 49 correct 'A's or only 1.

When doing generation-based fuzzing one needs to obtain the input model for the fuzz target. Input models can be extracted from specifications or formal grammars if the target program's input format is defined by one. When a target program's input model is defined by a formal grammar, it's referred to as grammar fuzzing. If no information is available about the input format, reverse

```
int main() {
    char input[100];
    if (fgets(input, sizeof(input), stdin) != NULL) {
        if (intput[0] == 'A' && ...  input[50] == 'A') {
            // Process rest of the input
        }
        return 0;
    }
    return 0;
}
```

Figure 2.1: Example of a program that reads input and checks if it starts with 50 A's before proceeding

engineering of the binary can be performed to gain a better understanding of the system's internals to define an input model. Another option is to symbolically execute the target program to collect a set of constraints for the input and re-use them for input generation [36].

2.1.2 Network protocol fuzzing

A network protocol sets the rules for how devices should interact to understand each other. Thus, a protocol implementation must conform to the protocol-established standard. Network protocol fuzzing consists of sending random data to such an implementation to uncover vulnerabilities. The main challenge is to explore deep code execution paths that can only be reached with a specific sequence of protocol messages that must conform to the specification. For example, a protocol might first perform an authentication step with a key exchange before starting the actual communication [32]. A protocol fuzzer, to be effective, must be aware of the protocol message structure and the state of the protocol during testing. Due to the stateful nature of protocols, a coverage-guided mutation-based fuzzer is highly unlikely to discover the correct message sequence to perform an authentication procedure for example. Inferring protocol states is an active research topic in the area of protocol fuzzing [11] [6] [42].

2.2 LTE

LTE (Long Term Evolution) is the fourth generation of wireless broadband for mobile device communication. The purpose of LTE is to provide high-speed data transfer, faster than the previous mobile network generations. It also implements encryption and authentication mechanisms to guarantee user privacy and limit unauthorized access to mobile network resources.

The LTE architecture is composed of 3 major entities represented in Figure 2.2. The first entity is the UE (User equipment), a mobile device that has the capacity to connect to the LTE network. The second entity is the E-UTRAN (Evolved UMTS Terrestrial Radio Access Network), all the base stations that serve as links between the UE and the evolved packet core. The last entity of the LTE architecture is the EPC (Evolved Packet Core). The principal entity in the EPC is the MME (Mobility Management Entity). The EPC controls the E-UTRAN and performs user authentication to provide registered users access to the network. It provides EMM (EPS Mobility Management) and ESM (EPS Session Management) functionalities to the connected UEs. EMM is responsible for a UE's mobility within the LTE network. It facilitates flawless cell handover between eNBs. EMM is also in charge of controlling the UE's security by providing an authentication functionality. Once the UE is authenticated, ESM manages different data sessions between the UE and the EPC. The EPC also contains a S-GW (Serving Gateway) that routes user data packets between the E-UTRAN and the P-GW (Packet Data Network Gateway). The P-GW is the gateway to packet data networks (PDN) that can be other networks or the Internet.

The LTE architecture contains a data plane and a control plane. The data plane transports all data communication between the UE and other networks such as the Internet going through the E-UTRAN and the EPC. The control plane establishes and releases data-plane connections, authenticates UEs, and deals with UE mobility management and location updates. In short, the MME in EPC is in charge of the control plane and the S-GW and P-GW handle the data plane connections.

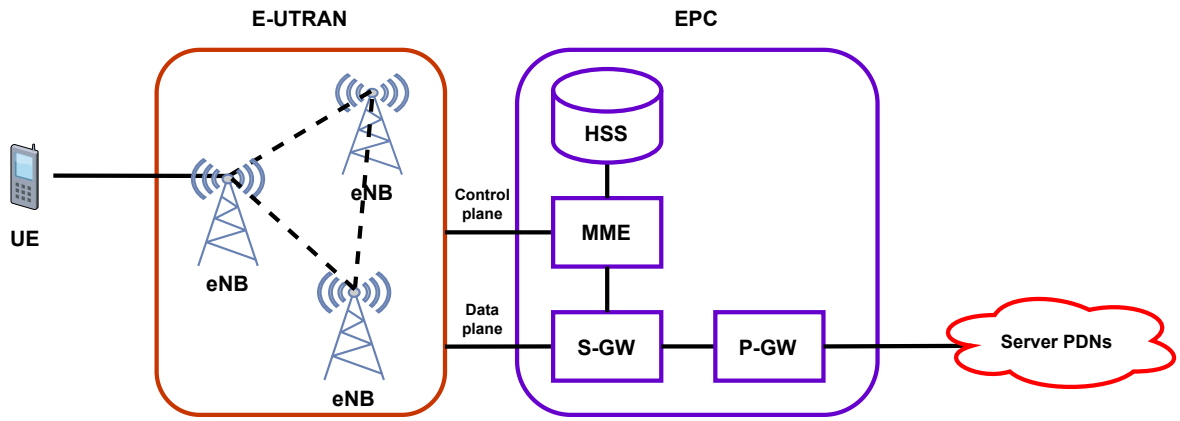

Figure 2.2: Simplified LTE Network Architecture

2.2.1 LTE protocols

All the previously described entities communicate with each other with a plethora of protocols. Some protocols are data plane specific whereas others only handle control plane communication.

Both RRC (Radio Resource Control) and NAS (Non-access stratum) are layer-3 control plane protocols. RRC is used to communicate between the UE and an eNB whereas NAS is used between the MME and the UE. NAS messages are encapsulated in S1-AP messages between the MME and the eNB and are then transported over RRC messages between the eNB and the UE. RRC packets are transmitted over the following layer-2 protocols: PDCP, responsible for header compression, integrity protection, and ciphering. RLC, which handles segmentation and can be run in either acknowledge mode or not, depending on the messages transported. Finally, the MAC protocol takes care of the channel access procedures. MAC is also responsible for logical channel multiplexing over transport channels. A packet in the LTE network is characterized as a downlink if it is Mobile terminated and as an up-link packet if it is emitted from the UE to the LTE network.

2.3 Protocol specification grammars

Network protocol standards cane be defined by various specification formats. The RRC protocol standards are defined using a formal notation like ASN.1. In other cases, the specification format is less formal, like the NAS protocol standards which are defined by Tag-Length-Value table structures.

2.3.1 ASN.1 grammar

ASN.1 is a standardized language utilized for defining protocol specifications. It offers a grammar that can be compiled into various programming languages such as C/C++ or Python. ASN.1 contains four core elements: Modules, Types, Values, and encoding rules. In Figure 2.3.1, a module is presented as an aggregate of definitions that contain types, values, and encoding protocols. ASN.1 defines several data types. The most common are INTEGER, OCTET STRING, SEQUENCE, and CHOICE. A CHOICE type contains multiple continuation fields from which only one can be chosen. The values of INTEGER or OCTET STRING fields can be either bounded or not when defined. Any definition can also be labeled as optional. Furthermore, ASN.1 definitions can also be recursive which means a packet of infinite length can be constructed from it.

The last element that composes an ASN.1 model is the encoding rule. Every module needs to contain information about which encoding rule to use. The standard encoding rule is the Basic Encoding Rule (BER). BER encodes every field to a tag-length-value format and byte-aligns them in the final encoding. RRC uses a variant of BER called the unaligned packet encoding rule (UPER). It is the most compact ASN.1 encoding. UPER encoding does not byte-align encoded fields and does not include the tag in the encoding. In this way, the maximum amount of information can be transferred with the least amount of bytes. Figure 2.4 shows how a sequence of integers is encoded when applying BER and PER encoding rules. The BER encoding yields a larger amount bytes as for each field a tag (red) and a length (blue) are encoded alongside the value (black). With PER, only the length of the sequence of integers and the length of each integer field is encoded as the tags are not

```
MyModule DEFINITIONS ::=
BEGIN
    MyInteger ::= INTEGER(0..255)
    MyBoolean ::= BOOLEAN
    MyString ::= OCTET STRING
    MyChoice ::= CHOICE {
        c1 MyInteger,
        c2 MyString,
        c3 MyChoice,
    }
    MyRecord ::= SEQUENCE {
        field1 MyBoolean,
        field2 MyChoice,
        field3 MyInteger OPTIONAL
    }
END
```

```
int1 MyInteger ::= 42
bool1 MyBoolean ::= TRUE
choice1 MyChoice ::= {c2 'foobar'H}

record1 MyRecord ::=
{field1 bool1,
                        field2 choice1,
                        field3 int1}

record2 MyRecord ::=
{field1 bool1,
                        field2 choice1}
```

Figure 2.3: ASN.1 Type Definitions (left) and Value Assignments (right)

required because the field order is known from the ASN.1 definition. If a field has a fixed length then the length will also not be encoded when using PER.

In LTE, various protocols are defined with ASN.1 definitions such as RRC which is used between UE and Base Station, S1AP (S1 Application Protocol) which provides control plane signaling between the E-UTRAN and the EPC and X2AP (X2 Application Protocol) which provides control plane communication between two eNBs.

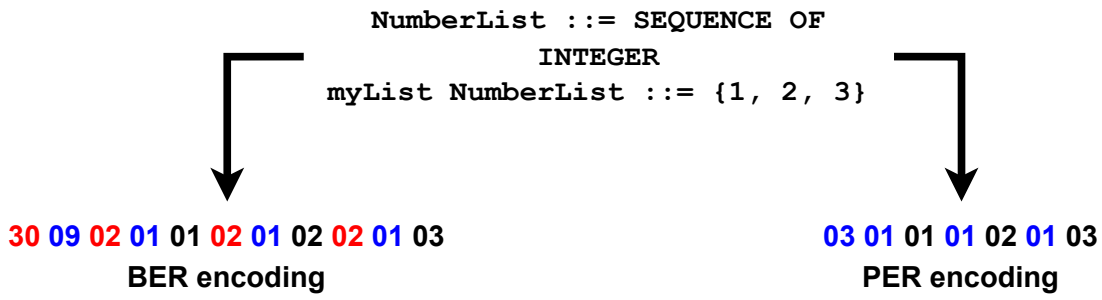

Figure 2.4: Difference between BER and PER encoding for a sequence of integers. Bytes representing a tag are marked in red, those indicating a length are blue and values of fields are black

2.3.2 TLV definitions

The format of NAS messages is described by the TLV-format definition (Tag-Length-Value) in the specifications. In Figure 2.5 we can see the table defining the message structure for an *Attach Reject* message in the specifications. Each entry in the table is a field in the message, also called an information element (*IE*). In the case of NAS-EMM, only the optional IEs have a tag field also referred to as IE Identifier (IEI). The disadvantage is that parsing and converting the specification to a structure suitable for programmatic use is difficult. This is because constraints on message field values are often defined in natural language.

IEI	Information Element	Type/Reference	Presence	Format	Length
	Protocol discriminator	Protocol discriminator 9.2	M	V	1/2
	Security header type	Security header type 9.3.1	M	V	1/2
	Attach reject message identity	Message type 9.8	M	V	1
	EMM cause	EMM cause 9.9.3.9	M	V	1
78	ESM message container	ESM message container 9.9.3.15	O	TLV-E	6-n
5F	T3346 value	GPRS timer 2 9.9.3.16A	O	TLV	3
16	T3402 value	GPRS timer 2 9.9.3.16A	O	TLV	3
A-	Extended EMM cause	Extended EMM cause 9.9.3.26A	O	TV	1

Figure 2.5: TLV Definition for a NAS-EMM Attach Reject message

Chapter 3

Design

3.1 Goals

The main goal of OTAFuzz is to provide an over-the-air fuzzing tool for baseband firmware. It should allow fuzzing of layer-3 downlink LTE baseband protocols. OTAFuzz is also expected to function on any baseband, independent of the manufacturer or phone model. Most design challenges arise from the over the air test environment and the closed source nature of baseband images. First, we want OTAFuzz's fuzzing speed to be bound by the over-the-air rate limit [2]. Then, OTAFuzz should leverage protocol grammars and specifications to generate valid protocol packets and apply grammar-aware mutations to message fields. These mutation rules should be applied to a specific set of protocol fields to uncover potential memory vulnerabilities. By targeting specific message fields we plan to reduce the number of test payloads and increase the potential of discovering crashes. Over the air testing does not permit to monitor the underlying memory of the baseband firmware and add instrumentation. To address the lack of coverage feedback, we aim to define our own coverage metric to help guide our fuzzing. Then, because of the lack of direct feedback information from the baseband we aim to analyze and define a set of universal and vendor-specific oracles to detect baseband crashes. We also expect to be able to fuzz basebands for an infinite amount of

time. Thus, the fuzzer should be able to generate protocol-valid messages with all possible field values randomly. Last, we design OTAFuzz to be modulable and easily extendable to other baseband protocols by implementing distinct generation and mutation steps.

3.2 System overview

Our system aims to provide an over-the-air LTE protocol fuzzing framework. OTAFuzz is made out of multiple modules interacting with each other. The first module consists of packet generation, mutation, and a controller. The packet generation uses protocol specification or already implemented packet encoder/decoder. The mutation pass then applies various mutation strategies to a valid protocol packet. The list of applied mutation strategies can be selected in the fuzzer configuration. The controller is then responsible for sending the mutated packets to the LTE network whenever it requests. It also handles the crashes detected by the oracles. In short, the controller contains the over-the-air fuzzing logic.

The second module of the system is the LTE access network. It is an LTE software implementation modified to send fuzzing payloads to the UE. Whenever it is possible to send a fuzzing payload to the connected UE, a request for a new packet is sent to the controller from the LTE network. Upon receiving the request for a new packet, the controller will send through a TCP socket connection the next packet to test. The payload is then received by the LTE access network which will send it to the connected UE, running the tested baseband firmware. The LTE access network can be configured to operate the fuzzing before or after the UE is authenticated. The logic of the authentication is independent of the fuzzer (generation and mutation) and executed before the fuzzing process begins.

Injecting mutated protocol packets into the LTE network and sending them to the UE is only one side of the fuzzer. To determine if the baseband of the UE has failed, we require one or more reliable oracles. All oracles are connected to the controller, which saves crash logs if one or more oracle is

triggered. When an oracle is triggered, the controller will stop the fuzzing process and log the last 50 packets sent to the UE. Then, the controller will enter a backtracking phase. During backtracking, the last 5 payloads are sent three times again to the UE to determine which is the faulty packet. If a new crash is detected during the backtracking phase, we save the payload as the best candidate for the crash. Moreover, the targeted field and the mutation strategy applied to it are added to a blacklist to avoid fuzzing this particular case again. Only when the backtracking is finished will the controller resume answering requests from the LTE network for new fuzzing payloads.

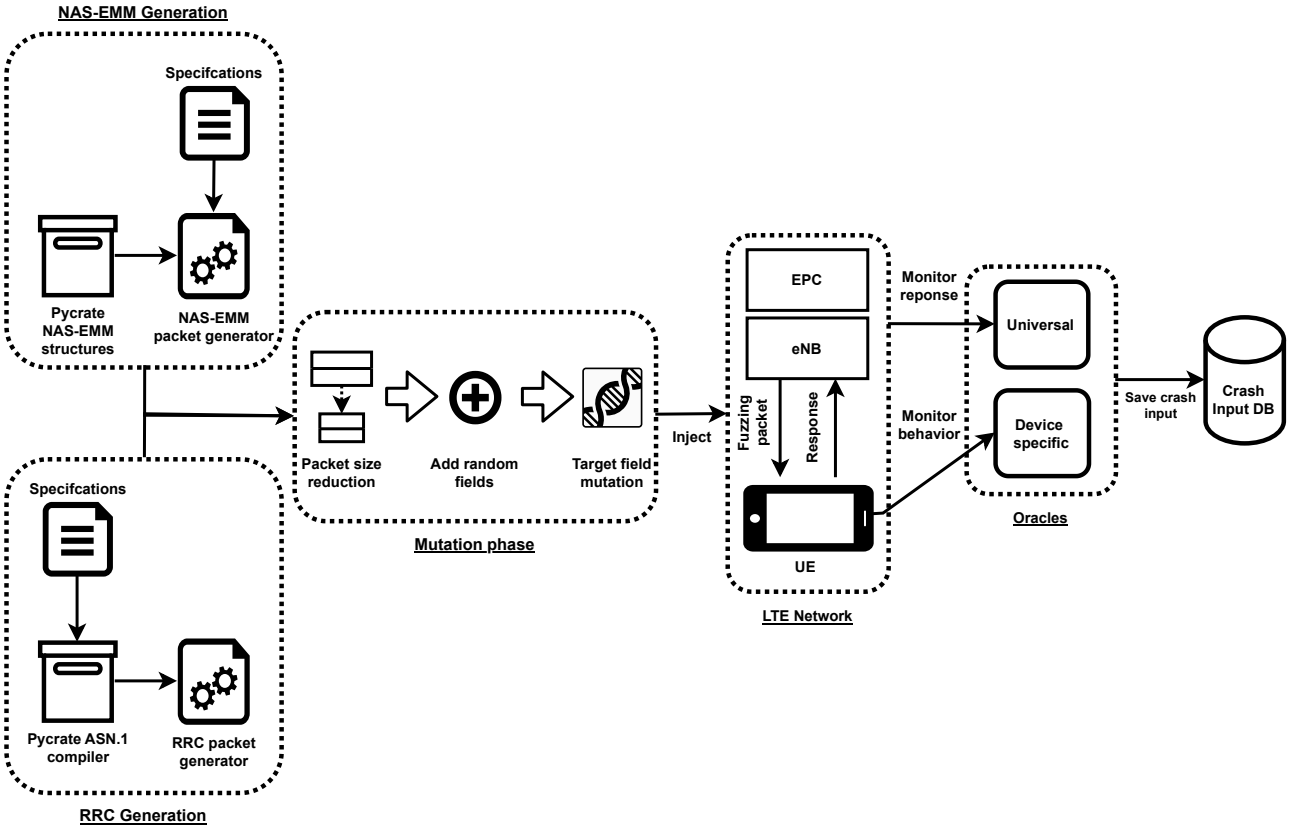

Figure 3.1: OTAFuzz system flow

3.3 Fuzzer

In this section, we describe the different parts of the fuzzer. The task of the fuzzer is to generate valid protocol packets and mutate them by applying a set of mutation strategies to a selection of message fields.

3.3.1 Packet generation

The first step for OTAFuzz is to generate valid packets for the target protocol. A network protocol is defined by specifications, which serve as the basis for its implementation across different systems, facilitating communication by ensuring a shared communication standard. In our case, we use RRC and NAS specifications to generate valid RRC and NAS-EMM packets. RRC specifications are described in the ASN.1 format and NAS-EMM message formats are described in the less practical TLV format. We build the packet generator such that all protocol message fields identified as targets are generated during a fuzzing cycle. Targeting all target fields is key to avoid fuzzing the most common fields over and over. Thus, we keep track of the total protocol target field coverage during every fuzzing cycle. Because we cannot instrument the baseband firmware binary to monitor code coverage, we track the different message protocol fields generated. This follows from the approximation that most of the baseband firmware code for the target protocol is covered if the fuzzer generates all possible message types containing all possible mandatory and optional fields that have valid values. It ensures that most of the handlers in the firmware for the target protocol are all targeted once during a fuzzing cycle. During the generation, we also collect the list of optional protocol fields, if any, and the list of target fields in the generated packets. This information is then used during the mutation phase.

3.3.2 Packet reduction

Before the mutation phase, the size of the generated packet is reduced by the fuzzer as the over-the-air packet size is limited to 2042 bytes for RRC packets and 2037 bytes for NAS packets [2]. The RRC packet size limit exists because RRC packets are sent over PDCP, which has a maximum capacity of 2047 bytes per packet. Additionally, the PDCP header requires 1 byte of space and the MAC address occupies 4 bytes. For NAS-EMM we also have to consider the bytes used by the RRC message encapsulating it.

To perform the packet reduction, the fuzzer removes all non-target optional fields from the packet. However, it can be that there are dependencies between the target field and fields that are removed. Therefore, after reduction, we allow the addition of random fields. This guarantees that field dependencies are covered by the fuzzer and that the packet is small enough to be sent over the air.

The addition of a random field can only add fields that come before the target field in the packet encoding. This is a fuzzing optimization because we expect the parsing to stop when the malformed target field is reached. Either a crash is triggered when the parser reaches the mutated field, or the baseband discards the packet because its format is invalid.

3.3.3 Packet mutation

After generating a valid packet, the fuzzer mutates it by targeting each target field it contains, one by one. We apply a basic mutation strategy called *BASE*. The basic mutation strategy aims at uncovering memory vulnerabilities in fields with a LENGTH + CONTENT format when encoded. The basic strategy mutates the value of the length field and the content to out-of-bound length values and produces mismatches between the length and the length of the content.

RRC strategies The basic mutation strategy is different for every protocol. For example, with the RRC protocol, we target BIT STRING and OCTET STRING fields as they are encoded in a length and content format. Additionally, we implement a basic strategy mutation for INTEGER fields. The value of the integer is mutated to out-of-bound values by still using the assigned amount of bits to the field. For example, if the value is bound between 0 and 18 then the encoding of this field is 5 bits long. This allows us to encode values such as 31 that are out-of-bound but not breaking the encoding. We also use the *TRUNCATE* mutation strategy that will remove a random amount of bytes at the end of a generated packet valid. This strategy ensures that the baseband firmware handles correctly the case where there are no more bytes left to parse but the baseband expects additional data. We decided to not apply bit flip mutations as they would not have much impact on the compact UPER encoding of RRC packets.

NAS strategies For the NAS-EMM protocol, we target all fields in a message. All message fields in EMM messages are either of the Length-Value or Tag-Length-Value format if they are optional or not. Thus, in the basic strategy for NAS, we mutate the length and values for each field. Moreover, we also target specific internal IE fields that implicitly represent a buffer of the form LENGTH + CONTENT.

For NAS-EMM messages, we employ additional mutation strategies to the basic one: *TAG* which mutates the tag values in the field of the TLV format, *APPEND* that appends random bytes to the end of a packet, *FLIP* that flips random bits in a packet and *TRUNCATE* that truncates a NAS message encoding at a random location. We decided to use more mutation strategies for NAS-EMM due to the reduced amount of messages in EMM.

Embedded fields Some target fields in RRC can be embedded inside another field that itself is a mutation target. For example, an OCTET STRING field can contain an encoding that represents a collection of ASN.1 encoded fields that can be mutation targets too. When mutating an embedded field, we adjust the length value of the outer field. We aim at uncovering memory vulnerabilities in the target and not in the outer field. Not fixing the outer length component might lead to missing

vulnerabilities in the embedded field. It is important to notice that we are not missing any crashes of the outer field as it will be considered a target at some point in the fuzzing cycle.

3.3.4 Controller

The controller is responsible for the fuzzing logic of OTAFuzz but also transmits the mutated packets to the LTE network. It also monitors all the oracles available for the targeted UE. When a crash is detected, it will stop the fuzzing, i.e. stop sending mutated packets to the LTE network. Then, the crash is recorded by saving the last 50 packets sent over the air to the UE. Additionally, when a crash is detected the controller can perform backtracking on the last 5 sent packets. During the backtracking procedure, each packet is sent three times in a row to the UE. If a second crash is detected during backtracking it will save the last sent payload as the best candidate for the previously detected crash. We also have the option to blacklist the target field and the mutation strategy used to produce the faulty packet. The blacklist is used to avoid sending again packets to the UE that might crash it and that we already know off. It is an optimization to increase fuzzing speed as performing backtracking and recovering from a crash takes significant additional time due to the over-the-air setup.

3.4 LTE access network

The second part of OTAFuzz is a conform software implementation of a 4G LTE access network. It can accept connections from one or multiple UEs. However, we slightly modified its implementation to inject RRC and NAS packets into the downlink channel from the eNB to the UE. At the location where we want to inject a packet, we set up communication between the LTE software implementation and the controller over TCP. The controller then sends test packets to be injected whenever it receives a request. We also add two procedures to set the state of the baseband to either pre-authentication or post-authentication. In the case of post-authentication, the encryption and the integrity of

the communication are still handled by the LTE implementation. Therefore, in both states, we only have to provide the plaintext packet bytes to be sent to the UE. Additionally, the sending logic within the RRC and NAS stack of the LTE network is modified to enhance crash detection capabilities. First of all, the RRC protocol is modified to only send the next packet when the previous packet is acknowledged by the underlying transport protocol RLC. This modification reduces the transmission rate but enhances the crash detection precision of figuring out which payload crashes the baseband firmware. This modifications also allows the usage of the OOO ACK and Timeout ACK oracles presented in the next section. Similarly, NAS is modified to only send the next message to the UE if a NAS Identity Response message is received from the tested UE. This response is emitted by the UE each time an Identity Request is received. Therefore, we send such an identity request in between every test packet.

3.5 Oracles

We explored a multitude of oracles to help us detect a baseband crash of the UE. We divided the oracles into two categories, universal oracles that work for all different basebands and vendor-specific oracles that are relevant for a limited set of basebands vendors. We defined oracles inspired by previous works and observations we made when the baseband in a device would crash. We also aimed to use diverse feedback sources for our oracles such as direct communication with the baseband, communication through an API, Bluetooth side-channel, or analyzing directly the baseband responses given to the LTE network.

3.5.1 Universal oracles

Oracles that work for any baseband vendor are all LTE protocol-specific, they are universal. They monitor the response given by the UE to the eNB and EPC over the LTE network.

RRC - ACK Timeout The first universal oracle used for RRC verifies that all RLC messages are acknowledged within a given time frame. RLC is the transport protocol that contains RRC messages. If no ACK arrives within 300 ms, the ACK Timeout oracle is triggered. The intuition behind this oracle is that inactivity at the RLC layer (layer 2) is related to UE inactivity and can be a symptom of a crash in the baseband processor. The controller is then informed about the oracle trigger via the same TCP socket sending the fuzz payloads from the fuzzer to the LTE network. The precision of this oracle is enhanced by the modification of the sending logic brought to RRC.

RRC - Out-of-order (OOO) ACK For RRC, we monitor the sequence numbers of the ACKs in the RLC protocol. If an out-of-order ACK is detected, the oracle triggers. An out-of-order ACK means that one previous message was not received correctly by the UE. This follows from the fact that ACKS in RLC are cumulative. An out-of-order ACK can be a symptom of a crashed baseband. The controller is then informed about the oracle trigger via the same TCP socket sending the fuzz payloads from the fuzzer to the LTE network.

NAS - Identity Requests For NAS, we use *Identity Request* messages to monitor the state of the UE NAS layer. After each mutated NAS payload, we send an *Identity Request* message and expect a *Identity Response* from the UE. If no such message arrives after three attempts in a row, the oracle is triggered. The idea behind this oracle is to have a keep-a-live-like message that confirms that the UE can still respond to NAS layer messages. The controller is then informed about the oracle trigger via the same TCP socket sending the fuzz payloads from the fuzzer to the LTE network.

3.5.2 Vendor specific oracles

ADB Log For Android-powered phones it is possible to monitor in real-time the Android debug bridge log and detect *'CP Crashes'* messages, standing for communication processor crashes. In case a crash is detected in the log it is necessary to monitor the log after a crash and only start fuzzing again when the baseband processor has recovered. The recovery of the baseband is indicated by the

[Modem status: ONLINE] log message. Not all firmware vendors log baseband crashes to the ADB log, therefore it can only be used for a subset of baseband vendors.

AT commands On Android-based phones, the AP, where the Android operating system runs, communicates with the BP through a UART serial connection or IPC using shared memory, for example. Most of these communication channels are not accessible without rooting the phone and gaining more understanding by reversing the closed-source firmware. However, we noticed that older devices expose the serial line to communicate with the modem when connected with USB debugging to a computer. While this is a clear security issue, as it exposes the serial channel without requiring any permissions, we use ATCommands (ATC) [21] to verify if the modem is still responsive. ATCommands are one way of controlling and requesting information from the modem. To use this as an oracle we repetitively send requests for the current signal strength to the modem. If the modem does not respond two times in a row, the oracle triggers. This oracle is only available on older phones as non-root access to the modem serial device is not supported in newer phone models for security reasons. Using ATCommands can also be dangerous as some messages can brick the device.

Telephony API Another way of monitoring the state of the baseband is to use the android telephony library [12]. This library can be used to access various information about the baseband such as the roaming status, the current channel number, or the assigned unique network identifier (CDMA NID). This oracle can only be used on Android devices as it issues a built-in Android library. The telephony library is an abstraction to communicate with the baseband and query information from it. The disadvantage is that the additional layer of abstraction above the modem hides most of the baseband crashes and hangs. We use the telephony API to track the signal strength of the baseband and its current state. This oracle is triggered if the signal strength values are not in the normal bounds when the baseband is in the *RUNNING* state. This indicates a reboot of the baseband that can happen after a crash.

Bluetooth monitoring This oracle is inspired by the SMS of Death paper [31] that proposes Bluetooth communication monitoring to detect baseband crashes or hangs. The baseband processor is responsible for all wireless communication. This includes 3G, 4G, and GSM but also Bluetooth. Therefore, a crash or a hang of the BP can possibly interrupt all established Bluetooth connections. Thus, we build a phone application that hosts a Bluetooth server and listens for incoming connections on the RFCOMM channel. Before the fuzzing starts, we connect a client and monitor the state of the Bluetooth connection. As soon as the connection is interrupted, the oracle is triggered. This oracle is device-specific because it depends on how the Real-Time Operating System (RTOS) running on the baseband processor manages the different wireless communication processes. The degree of interaction and isolation between processes can vary between baseband firmware vendors and images, affecting their behavior to crashes in the RRC and NAS processes.

Chapter 4

Implementation

In this chapter, we delve into the crucial implementation aspects of OTAFuzz, focusing on packet generation, payload reduction, and communication between the controller and the LTE network.

4.1 Fuzzer

4.1.1 Packet generation

For both protocols, RRC and NAS, we use Pycrate [33], a Python library that handles various digital formats to generate valid protocol packets. Pycrate provides encoders and decoders for digital formats such as JPEG or GIF but also mobile message formats for LTE protocols such as NAS, GSM, or SCCP.

NAS NAS specifications are written in the TLV format, therefore Pycrate manually implements Python structures to represent all the different NAS-EMM messages and the fields they contain. The Python structures are then used to encode or decode NAS EMM messages. We use these structures to generate random packets that conform to the specifications. However, the field lengths of some

IEs are loosely constrained in Pycrate. For example, the *CipherKeyData* field is embedded in a Type6TLVE container that allows a content length of up to 65536 bytes but the specifications [1] constraint the field length to a maximum of 2291 bytes. Thus, we parse the NAS specification to extract more accurate length constraints for message fields, also called IEs. To extract information from the specifications we use a parser written and used by the authors of Basespec [25]. When generating a valid message, we ensure that it contains all possible optional fields. To sum up, we leverage Pycrate NAS-EMM internals meant for decoding/encoding and parse NAS specifications to build a valid NAS-EMM packet generator.

While implementing NAS-EMM packet generation with Pycrate we uncovered several inaccuracies and issues that were promptly resolved by the maintainer. We want to especially thank Benoit Michau, the principal maintainer of Pycrate, for his help and fast responses whenever we opened issues or had questions.

RRC To generate valid RRC packets, we first compile the latest RRC ASN.1 specification [3] to Python using Pycrate compiler. The compiled ASN.1 specifications is a tree-like structure and is meant to be used for decoding and encoding of RRC packets. However, to generate a valid RRC message, we randomly traverse the compiled RRC specifications using recursion to handle all possible ASN.1 types. We use the constraints on the data of a field that is already included in the specifications to assign valid values to all fields. Additionally, similar to the NAS packet generation, we generate messages that contain all optional fields. The generation step also collects the paths to all optional fields and all target field nodes in the generated packet. Those paths are used to reduce the packet size.

4.1.2 Packet reduction

Before mutating an RRC or NAS packet, we proceed to a pre-processing step where all the optional fields not required when building a valid packet are removed. Only the target and mandatory fields

are kept. This step is necessary as a full packet containing all mandatory and optional fields is bigger than the maximum amount of bytes that can be sent over-the-air to the UE [2]. Reducing the size of a packet also reduces the complexity of it. It then becomes easier to understand the cause of a crash when the faulty payload is identified.

RRC Figure 4.1 represents an RRC packet as a tree where the leaf nodes are fields and parent nodes are fields that contain one or more fields, typically ASN.1 SEQUENCE fields. The target field to mutate is identified by a path in this tree leading to a leaf node. In the same way, we keep a list of tree paths to all optional fields of the packet. We reduce the packet size by removing all optional leaf nodes that are not on the path of the target field. Because of the tree-like structure, we know that removing a parent node leads to the removal of all its child nodes. Therefore there is no necessity to remove all optional fields one by one. We save computation time by reducing the set of paths to remove. We apply a pre-processing step where we identify the smallest subset of nodes that need to be deleted from the tree.

NAS The NAS case is less complex as NAS-EMM messages contain a list of either mandatory or optional fields. Thus, it is easy to remove undesired fields from the message. There is no need to inspect the content of each field recursively as in RRC because the individual message fields do not contain optional fields.

4.2 LTE access network

We run our own LTE Access Network using a modified version of srs_RAN 4G project [44] that includes software applications for UE, eNB, and EPC. In our case, we only use the eNB and the EPC as the UE will be the phone we want to fuzz.

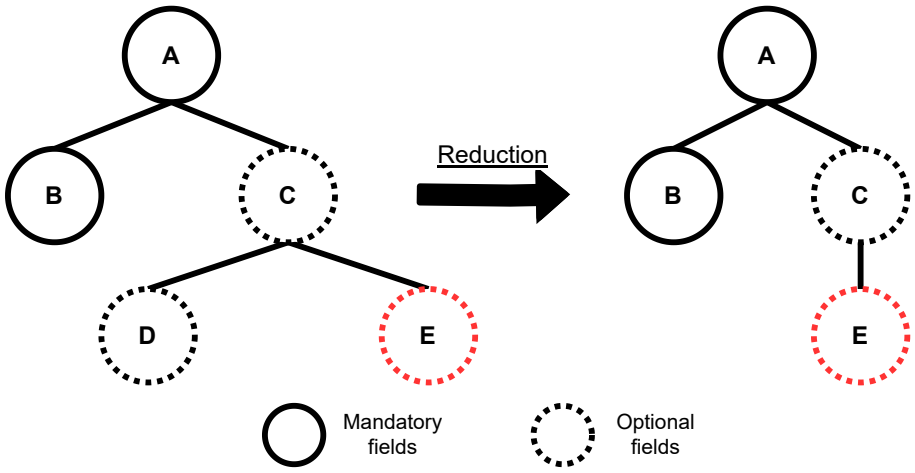

Figure 4.1: RRC packet reduction

4.2.1 Packet injection

RRC To inject RRC packets into the LTE network, a TCP client socket is added to the UE RCC class in SRS. One such class is instantiated for every RRC session between a UE and the eNB. Once the UE connects to the eNB it is ready to receive an RRC message. The TCP client in the eNB application will then send a request to the controller, asking for a new RRC fuzz payload. The controller then answers this request with an RRC packet obtained from the RRC fuzzing module. Once the packet is received, it is sent directly to the UE from the eNB. Because the client socket exists per session, the RRC controller needs to support the necessary logic to allow a new client socket to connect it. This happens when the UE baseband crashes or when the UE spends too much time not being authenticated to the LTE network.

NAS Similarly to RRC, we add a TCP client socket inside the NAS class. However, this class is instantiated only once at the launch of the EPC. Therefore, the NAS controller logic differs from the RRC controller logic as the socket connection for NAS will always be maintained, even during a baseband crash. The NAS connection is also not per-device compared to RRC, but this is not an issue as OTAFuzz is meant to fuzz one device at a time.

4.2.2 NAS - Identity Requests

We describe in detail the packet injection logic at the NAS layer represented in Figure 4.2. First, it is important to keep in mind that NAS messages are transported over RRC packets between the UE and the eNB. This implies that the modifications described earlier in the RRC sending logic also concern the NAS protocol. An RRC packet is only sent if the previous one was acknowledged on the RLC layer. We define the NAS logic to send the next NAS test message only if the previous NAS test message was received (enforced by the RRC layer modification) and if the following-up identity request has been answered with a corresponding identity response by the UE. Every time there is an opportunity to inject a new packet, the logic presented in the below State-Machine is followed. For example, in the case a NAS-EMM test message has been sent to the UE, the next NAS packet sent to the UE will be an identity request. This will happen until either, there is no response 3 times in a row or a response is received. From this, we see that at best half of the messages sent over-the-air to the UE are actual fuzzing messages. The addition of this oracle divides the fuzzing rate by two at best.

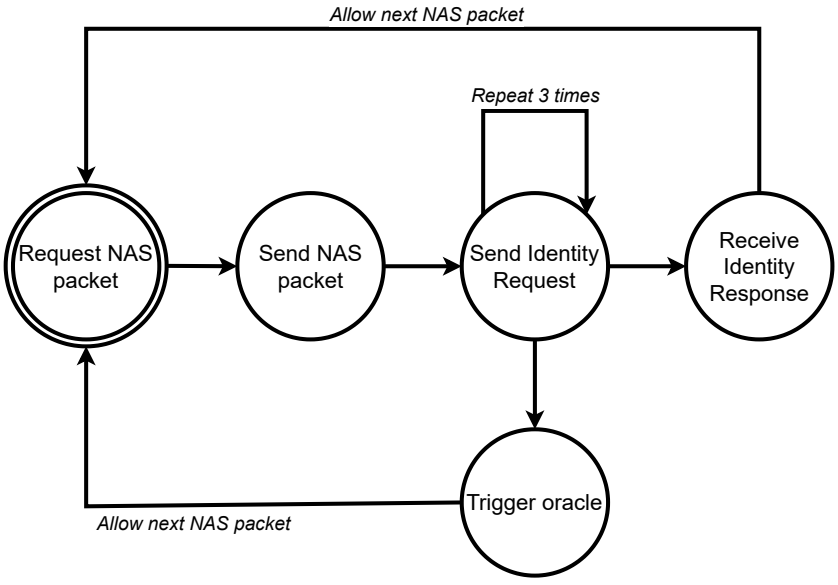

Figure 4.2: NAS packet injection state machine with identity request modification

Chapter 5

Evaluation

DISCLAIMER All experiments carried out as part of this work followed South Korean regulations and laws. We took extra precautions to ensure that only test devices connected to our self-hosted LTE network. No malicious packets were transmitted outside of our controlled environment. The safety and integrity of external devices and networks were always top priorities.

To evaluate OTAFuzz we measure the performance of the generation and mutation steps of both RRC and NAS-EMM fuzzers. Then, we present an analysis of the performance of 7 different oracles we considered for OTAFuzz to detect 2 RRC crashes and 1 NAS crash. After that, we present the results obtained when fuzzing 5 different devices with our RRC fuzzer. We discuss mutation strategy performance, false positive rates and RRC grammar coverage. Finally, we present the limitations we encountered performing all our experiments. With the evaluation of OTAFuzz, we aim to answer the following research questions:

1. **How does OTAFuzz perform compared to standard LTE over-the-air performance?**
2. **Can we select a reliable set of oracles for OTAFuzz to detect baseband crashes?**
3. **Can OTAFuzz find previously discovered vulnerabilities in baseband firmware and new ones?**

5.1 Setup

We describe the experimental setup we conducted all our experiments with. For hardware, we use a desktop with an i7-8700K CPU @ 3.70GHz CPU and 32 GB of RAM. We also use an USRP which can host a functional eNB. The USRP is connected to a manual shield box 5.1 in which we put the UE that should be tested. Additionally, the UE is connected by USB to the desktop to receive ADB commands. On the software side we use OTAFuzz which uses the latest version of Pycrate [33] cloned from git commit *6fd12f67ce44bfa5e8374447f5e4311f349082e6* and a modified version of SRSRAN [44] forked from git commit *d045213fb9cbf98c83c06d7c17197a9dcbfddacf*. For the basebands, we selected 5 different smartphone models from 2 different vendors to cover 4 different baseband vendors. 5.1. Not all baseband images are up to date as this allows us to analyze the performance of OTAFuzz to find old vulnerabilities.

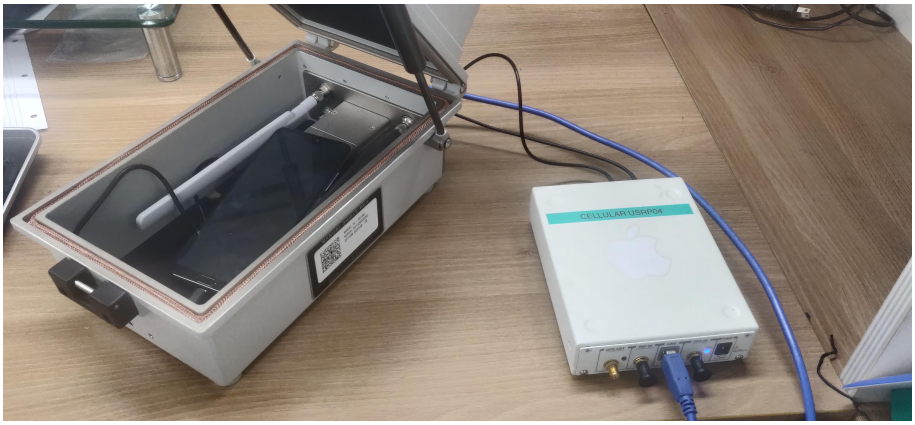

Figure 5.1: Over-the-air fuzzing setup

Phone Model	Vendor	Baseband Vendor	Baseband Version	Last Time Patched
Galaxy S10	Samsung	Exynos	G970NKOU1ASE5	2019-05
Galaxy S21	Samsung	Exynos	G991NKOU4EWE2	2023-05
P20 Pro	Huawai	Hi-Silicon	21C20B369S018C000	2021-12
Galaxy S8	Samsung	Qualcomm	G950U1UEU8DVD4	2022-04
Galaxy A31	Samsung	MediaTek	A315NKOU1DVH1	2022-08

Table 5.1: List of evaluated phones and respective baseband information

5.2 Generation and mutation performance

Over-the-air fuzzing is by nature slower than traditional fuzzing. Therefore, it is important to not slow it down even more. We evaluate the performance of generation and mutation for both fuzzers. The baseline we decide to compare against is the one given in [4], which is 50 RRC packets per second. This baseline also holds for the NAS-EMM fuzzer as RRC packets transport NAS-EMM messages. We validated the correctness of this baseline by observing normal traffic being injected from the LTE network to the eNB, without connecting the fuzzer.

In table 5.2, we present the results of our experiment, where we measured the raw performance of the fuzzers. All mutation strategies were applied during the mutation phase of the fuzzers. We notice that all generation and mutation are faster than 50 packets per second except for the case where we generate RRC packets to mutate OCTET STRINGS, BIT STRINGS, and INTEGERS. This can be explained by the fact that many INTEGER fields are harder to generate as some are only present in rare cases. Furthermore, the amount of mutations generated per INTEGER field is lower (only 3) than for OCTET and BIT STRINGS (between 4 and 20 depending if the field is bound or not). We consider that 49 packets per second are still acceptable as the baseline of 50 represents an ideal case where the connection never drops and the over-the-air connection quality is always perfect. We also observe that RRC has a higher mutation time than NAS. This is because, for each new target field, expensive tree operations are performed in Python to reduce the size of the generated packet. The NAS generation and mutation process is less complex as there are fewer messages, and the structure of the NAS-EMM messages is also simpler. It costs less time to remove optional fields and only keep mandatory and target fields.

We conclude that the performance of both fuzzers is enough to not act as a bottleneck to the over-the-air fuzzing rate.

	Generation	Generation + Mutation	#Payloads	Payloads/sec
RRC - (OCTET STRINGS)	2.25 sec	10.8 sec	1805	167
RRC - (OCTET + BIT STRINGS)	11.3 sec	115.9 sec	11793	115
RRC - (OCTET + BIT STRINGS + INTEGERS)	61.1 sec	700 sec	34369	49
NAS	0.14 sec	2.8 sec	1027	366.8

Table 5.2: Performance comparison of payload generation and mutation speeds for RRC and NAS fuzzer for a single fuzzing cycle with all mutation strategies

5.3 Oracle reliability

We evaluate the performance of every candidate oracle we choose for OTAFuzz. The goal is to determine which oracles are reliable and effective enough to be used during over-the-air fuzzing. To test the effectiveness of the oracles, we need to evaluate them against known crashes on a device that is still vulnerable to them. Therefore, we evaluate all our oracles against a Samsung S10 device running an Exynos baseband firmware with version G970NKOU1ASE5 that was last updated in May 2019. We use 3 vulnerabilities that affect this firmware. The first is an RRC vulnerability found by Firmwire [20] [30]. The second one is another RRC vulnerability that we found in the early stages of the development of OTAFuzz. Finally, we tested them against one of the two NAS-EMM 0-day vulnerabilities we found. The experiment is conducted in the following way: every oracle is tested independently for two scenarios. Either the buggy payload is sent once or multiple times until the UE does not respond. We then record if the oracle detects the baseband crash. If this is the case, we also record how much time past. We compute the time interval from the first time the buggy packet is sent to the UE until the oracle fires for the first time. We repeat this experiment ten times. We will now discuss and interpret the results 5.3 in the rest of this section.

Oracle dependencies First, some oracles have implicit dependencies. For example, the OOO ACK and Timeout ACK oracles are mutually exclusive when sending a buggy packet. There are only three possible scenarios at the RLC layer when this happens:

1. An ACK with the correct sequence number is received.

2. An ACK with an incorrect sequence number is received.

3. No ACK is received.

In the first situation, the Timeout ACK oracle detects a crash. In the second, the OOO ACK that gets triggered, while in the third, neither the OOO ACK nor the Timeout ACK detects a crash. Notably, both oracles cannot activate simultaneously when sending just one packet. This leads to the following observation: In singular scenarios, the combined success rate of both oracles equals to 100% for RRC vulnerabilities during our experiments. We observe that the Timeout ACK oracle detects all crashes when sending multiple packets. In contrast, the OOO oracle misses some. This is because the system waits for an ACK to arrive to send the next packet. If the first packet's ACK times out then the baseband firmware has already crashed preventing any additional ACK from being received. Therefore, while the OOO oracle serves as an indicator for baseband crashes it still can miss some of them. By using both Timeout and OOO ACK oracles we guarantee two things: first, we can detect all baseband crashes within 300 ms, and second, we benefit from the better time-to-trigger speed OOO brings. We also want to point out that the OOO oracle results from an unexpected behavior observed while testing only the S10. Another dependency we uncovered during testing is that sending AT Commands to the device is logged to the ADB log. This fills up the log and can even freeze it. Thus, the AT Commands oracle and the ADB oracle cannot function together.

RRC oracles For the two RRC vulnerabilities, we observed that not all oracles could detect both. The ADB and the Telephony oracle were unable to detect the Firmwire 1-day crash. During the experiment, we noticed that the baseband would not recover on its own when crashing it with the Firmwire bug. To force a recovery, we had to toggle the airplane mode on the device. This is a different behavior compared to the RRC 1-day we found while testing OTAFuzz where the device automatically recovers without any external interaction. From these observations, we conclude that

the Telephony and ADB oracles are only activated upon baseband recovery, rather than at the start of the crash. The high average trigger times observed for them confirm this conclusion even more. The average time-to-trigger of the Timeout ACK oracle is close to 300 ms, which is the duration of the timeout used in the SRS-eNB application. Finally, using the ID_REQ oracle is not possible and does not make sense for RRC fuzzing as the oracle logic resides in the NAS layer, above RRC.

NAS oracles For the NAS 0-day we can see that despite NAS being transported over RRC — which itself is encapsulated in RLC — both the OOO ACK and Timeout ACK oracles do not detect the baseband crash. Thus, we conclude that a failure on the NAS layer will not be immediately visible at the RRC and RLC layers. On another note, the ADB, AT Commands, and Telephony oracles detect the crash every time in all scenarios. However, their average time-to-trigger is deemed impractical for fuzzing over the air. The identity request oracle does not detect the baseband crash in the single packet scenario. This is because the oracle fires only after 3 consecutive failures and that the NAS layer crash is not immediate. Therefore, in the single packet scenario, a valid identity response will be received by the eNB and no other NAS-EMM message will be sent afterward, preventing the oracle from recording a failure of the UE to respond to an identity request message. However, when sending multiple messages it reliably detects all crashes, as an identity request will be issued for each NAS packet sent and will eventually not get any response. The average time-to-trigger is low enough that we consider this oracle usable for over-the-air fuzzing.

Bluetooth oracle The Bluetooth oracle did not detect any baseband crashes during our testing. Unlike other oracles, it depends on the firmware architecture design. The Bluetooth stack in the S10 modem is independent of the GSM stack. Therefore a failure in the RRC or NAS layer does not provoke a full crash of the RTOS running on the BP and the Bluetooth stack remains untouched. However, this oracle should not be considered as a total failure as it is device-specific and we only tested it on a single one.

	RRC Firmwire Bug (SVE-2021-22051)				RRC 1-day				NAS 0-day			
	Single		Multi		Single		Multi		Single		Multi	
	SR	AT	SR	AT	SR	AT	SR	AT	SR	AT	SR	AT
ADB	0%	-	0%	-	90%	413 ms	90%	127 ms	100%	9.6 sec	100%	9.7 sec
OOO ACK	50%	16 ms	80%	17 ms	70%	14.5 ms	70%	52 ms	0%	-	0%	-
Timeout ACK	50%	300 ms	100%	313 ms	30%	300 ms	100%	312 ms	0%	-	0%	-
ID_REQ	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0%	-	100%	148 ms
Bluetooth	0%	-	0%	-	0%	-	0%	-	0%	-	0%	-
AT Commands	100%	2.2 sec	100%	2.2 sec	100%	2.5 sec	100%	2.6 sec	100%	2.1 sec	100%	2.1 sec
Telephony API	0%	-	0%	-	100%	2 sec	100%	3.4 sec	100%	11.5 sec	100%	11.9 sec

Table 5.3: Oracle success rate (SR) and average time-to-trigger (AT) over 10 independent trials for 3 different crashes tried on never patched Exynos baseband firmware running on a Galaxy S10

Airplane mode Every oracle is also tested against the enabling of the device's airplane mode. From this experiment, we notice that the Timeout ACK and Bluetooth oracle register a crash if the airplane mode is activated. The Timeout ACK is triggered because whenever the UE detaches it will not send anymore ACK messages and the Bluetooth oracle fails as the airplane mode deactivates all wireless communication. This terminates the Bluetooth connection which triggers the oracle.

We conclude from the results obtained that the best oracles for the RRC protocol are Timeout and OOO ACK. For NAS the best oracle is ID_REQ. It remains however interesting to use the slower oracles to confirm a baseband crash when analyzing all the crashes found during the fuzzing.

5.4 End-to-end system

In this section, an evaluation of the full running system is presented for RRC pre-authentication fuzzing. We first discuss different aspects of the results obtained from the RRC fuzzing. Then, we discuss the over-the-air challenges and limitations we encountered while performing the experiments.

5.4.1 RRC

We decided to fuzz the RRC protocol targeting OCTET STRINGS and BIT STRINGS fields. We also use all the mutation strategies we defined for RRC, BASIC and TRUNCATE. All devices were fuzzed

over the air for 5 fuzzing cycles using the same seed each time. In 5.4.1, we observe that the number of tested packets highly varies across all devices. This is explained by the fact that the fuzzer will ignore mutation for target fields that were confirmed by backtracking. Therefore, the higher the number of confirmed crashes by backtracking the more packets are blacklisted and the fewer test payloads are sent during all subsequent fuzzing cycles.

To determine whether the crashes found during fuzzing and backtracking are actual crashes we replay them to the devices. For Samsung devices, we set the debug mod level to mid-mode, which will automatically make the device enter recovery mode when a fault in the baseband happens. With this, we can distinguish between actual crashes and packets that have to ability to terminate the connection between the UE and the eNB. For the Huawei phone, we replay candidate packets again and reduce the set of potential buggy payloads again. Then, we manually send each of the remaining candidates over the air and monitor the state of the baseband through the phone GUI.

OTAFuzz uncovered one RRC vulnerability in the A31 and identified 23 actual crashes for the Galaxy S10. 14 were found by the BASIC strategy, and 9 by the TRUNCATE one. By manual inspection, we concluded that there are 15 unique vulnerabilities. This shows the effectiveness of both mutation strategies as the TRUNCATE strategy found a vulnerability that was not discovered by the BASIC strategy.

Coverage analysis In 5.4.3 we observe that the RRC fuzzers covers 30% of the total RRC grammar. We compute this percentage by computing the rate between the number of unique fields present in mutated packets and the total number of unique fields defined in all the RRC ASN.1 definitions. The RRC grammar coverage is slowly growing because only one random field is added to each reduced packet that targets a specific BIT STRING or OCTET STRING. Because we are only allowing the insertion of additional fields that come before the target field in the encoding we will never reach 100% grammar coverage. To increase the grammar coverage growth rate per cycle we can modify the number of additional fields inserted into each reduced packet. Figure 5.4.3 shows that every fuzzing cycle covers all target fields. The time to cover all of them differs as the randomness in the

generation also applies to the type of packets built. Thus, the time to reach full target field coverage is not constant for every cycle. However, on average, the fuzzer can cover all BIT STRING and OCTET STRING fields in about 120 seconds.

False positive rate The false positive rate, even with the backtracking procedure, remains high. We only found one RRC crash for the MediaTek baseband running on the Samsung A31 out of 71 candidates that triggered the oracles twice in a row. The same conclusion can be drawn from results obtained on the Galaxy S8 and the Galaxy S21. The amount of false positives is the result of two factors. First, the Timeout ACK oracle will trigger for every message that terminates the connection between the UE and the LTE network as the UE will not send any ACK back. Second, certain RRC and NAS messages have the capability to terminate the communication between the UE and the eNB or the EPC. For example *RRCConnectionrelease* or *EMMDetachRequest* messages will automatically kill the connection between the UE and the LTE network. Even though we decided to filter out all messages of this kind, we still observe that there exist other RRC messages that pose this problem. After analyzing the false positives we notice that *RRCConnectionReconfiguration* message can in some cases also terminate the connection between the UE and the eNB. It is not reasonable to also ignore all such messages as they hold most of the RRC fields. The combination of the two described factors results in a high false-positive rate. To reduce the amount of false negatives we use a blacklist to avoid fuzzing the same messages that terminate the over-the-air connection.

Mutation strategies success rate We analyze in 5.4.1 the performance of different mutation strategies applied to generated packets when fuzzing the RRC protocol. We notice that the BASE strategy finds more reliable crashes than the TRUNCATE one. This is expected as it mutates specific fields that are encoded as buffers (LENGTH + CONTENT) whereas the TRUNCATE strategy randomly cuts a valid packet in two and keeps the first part. Truncating a message can lead to similar mutations produced by the BASE strategy. For example, a truncation in the middle of a buffer will result in a mismatch between the length and the actual length of the content.

Baseband behaviors after crash While fuzzing the S10, we discovered several 1-day vulnerabilities because the firmware dates from 2019. We also noticed that the majority of vulnerabilities would crash the baseband for 5 to 8 seconds. The baseband then recovered automatically. However, we noticed that for SVE-2021-22051 [20] the baseband would only recover if we manually toggled the airplane mode on the phone. While fuzzing the Galaxy S10 we also discovered a packet that would completely disable the baseband. The only way to recover was to restart the device manually. From this, we conclude that the same baseband can respond in various ways after a baseband crash.

Firmwire 1-day reproduction To develop an effective fuzzer for RRC and NAS, we studied previous vulnerabilities discovered in previous work. We used previous findings to select the mutation strategies OTAFuzz should implement. The main RRC bug we analyzed is **SVE-2021-22051** found by Firmwire [20] in 2021/2022. We observed that our basic mutation strategy, which mutates the length and content of a buffer is not enough to trigger the same bug in the Samsung S10, still vulnerable to it. Indeed, mutating the *lateNoneCriticalExtension* field inside the *RRCConnectionReconfiguration-v8m0-IEs* is not enough. Even though this seems the root cause of the vulnerability. After further investigation, we understood that the presence of the field *dedicatedInfoNASList* was required to trigger the fault in the baseband firmware. From this finding, we made sure that the fuzzer can generate reduced packets that contain the current target field but also other randomly added fields. The structure of the 1-day packet is presented in appendix A.

Phone Model	**#Payloads**	**Total Fuzzing Time**	**#crashes**	**#crashes confirmed with backtracking**
Galaxy S10	6267	43 min	90	40
Galaxy S21	14138	1 hr 10 min	123	23
P20 Pro	42131	2 hrs 48 min	345	19
Galaxy S8	4012	40 min	80	44
Galaxy A31	4254	1 hr 26 min	114	71

Table 5.4: PRE-AKA RRC fuzzing statistics for 5 cycles (seed=19)

Phone Model	#Crashes confirmed with backtracking	% BASIC	% TRUNCATE
Galaxy S10	40	87.1%	12.9%
Galaxy S21	23	68.8 %	31.2%
P20 Pro	19	89.5%	10.5 %
Galaxy S8	44	85.2%	14.8%
Galaxy A31	71	70.3% %	29.7%

Table 5.5: Summary of confirmed crashes using different mutation strategies (BASIC and TRUNCATE) when fuzzing RRC PRE-AKA

5.4.2 NAS

We were not able to complete the full evaluation for the NAS protocol in time. However, we present the major result we found while fuzzing NAS-EMM with OTAFuzz.

Samsung Exynos NAS-EMM 0-days While fuzzing the Samsung Galaxy S21 device we discovered two new vulnerabilities in *EMMDetachAccept* and *EMMAuthReject* messages respectively in early June. From our analysis, we concluded that the root cause was the same and reported them to Samsung. While it was confirmed to be two 0 days, the vulnerabilities were already reported 3 months before our discovery. The vulnerabilities were assigned CVE-2023-36481 [40]. Even though we were not the first to report this issue it shows that OTAFuzz was able to discover unknown vulnerabilities.

5.4.3 Limitations

In the next paragraphs, we present the elements that had varying degrees of impact on our fuzz-testing methodology and describe the measures we took to address them.

First, to fuzz RRC and NAS pre-authentication, the device connects to the LTE network. The authentication procedure is nevertheless not completed by the EPC. Fuzzing messages are sent instead. However, there is a time limit to perform the authentication. If it is not performed in time, the UE will automatically stop the communication. After several failed attempts the device will

no longer connect to our LTE network. The only way we found to make it reconnect is to reset the baseband using airplane mode. The downtime when the device automatically disconnects and reconnects is lower than when it needs a baseband reset. Figure 5.4.3 shows that the downtime is 10 seconds when the UE reconnects on its own compared to around 30 seconds when the baseband needs to be reset. The first four times where the fuzzing rate drops to 0 the device automatically reconnects. The fifth time, the device does not automatically reconnect and needs to be reset using ADB commands.

Second, we notice on 5.4.3 that a crash is recorded when the UE cuts the connection after not having been able to authenticate. Even though a crash is recorded it is confirmed to be a false positive as the backtracking does not find the crash again. Thus, we see that backtracking is an effective method to eliminate false positives.

Third, the over-the-air fuzzing speed is lower than what we expected. We notice in 5.4.3 that we achieve at best a fuzzing rate of 12 packets per second whereas the maximum achievable would be 50 per second [4]. From 5.2 we conclude that the generation and mutation speed is not the bottleneck. After a careful analysis of the timings in OTAFuzz, we noticed that the slowdown comes from the communication between the fuzzer and the LTE network. We also confirmed that the fuzzer always has a packet ready to send to the LTE network, which confirms our results from Section 5.2.

Finally, another issue is that the device does not reconnect to the LTE network even after resetting the baseband with airplane mode. We observed this behavior when fuzzing the Galaxy A31. The only solution was to remove and reinsert the SIM card of the device. Another instance of this problem arose when fuzzing the Galaxy S10 where the baseband would not respond anymore. The only solution was to restart the device during the fuzzing process.

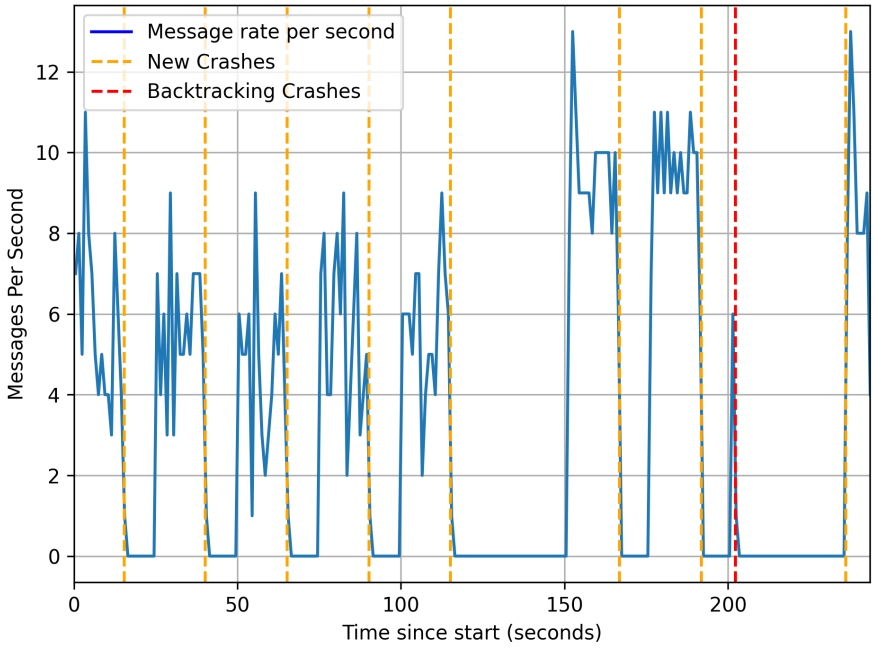

Figure 5.2: Over-the-air Fuzzing Rate when testing P20 PRO RRC Pre-authentication (seed=19, targets=[BIT STRING, OCTET STRING])

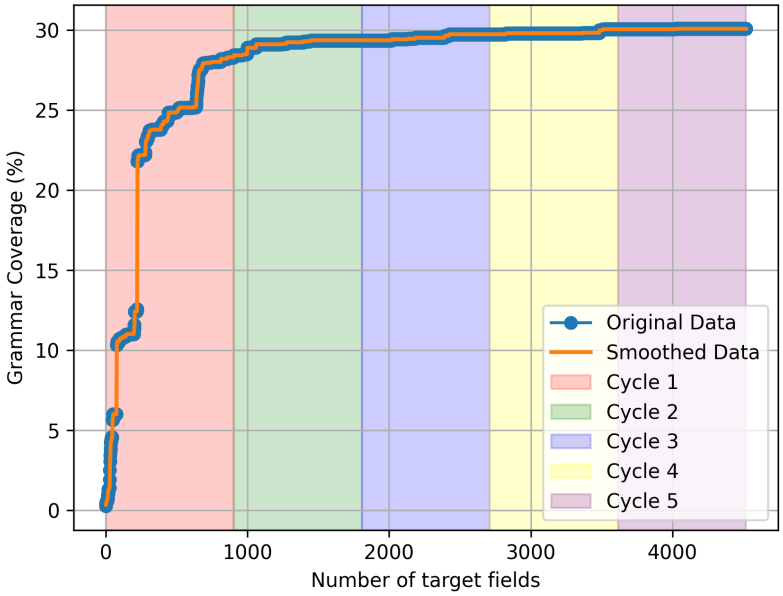

Figure 5.3: Grammar coverage growth for RRC fuzzer over 5 cycles (seed=19, targets=[BIT STRING, OCTET STRING])

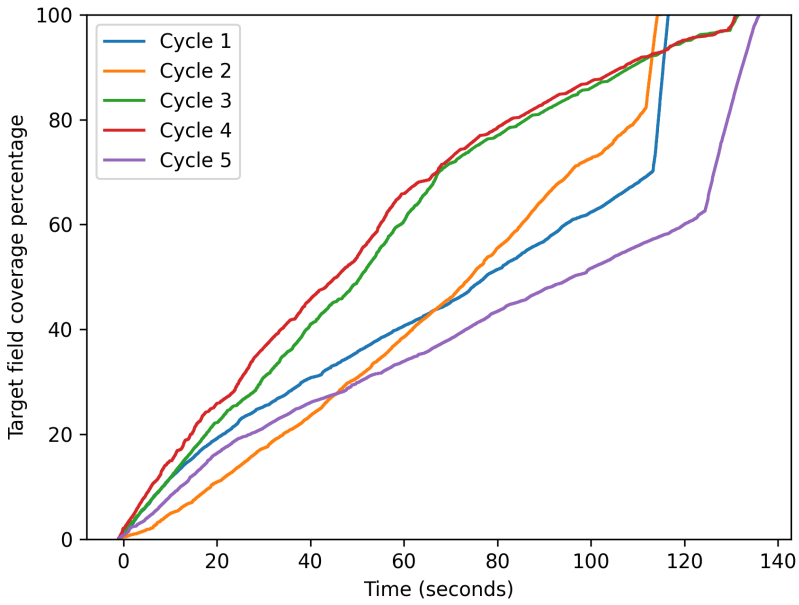

Figure 5.4: Evolution of target field coverage in RRC Fuzzer for 5 cycles over time (seed=19, targets=[BIT STRING, OCTET STRING])

Chapter 6

Related Work

6.1 General purpose fuzzers

State-of-the-art fuzzing is mutation-based grey box fuzzing and relies on binary instrumentation for code-coverage feedback to guide the mutation of the test inputs [15] [45]. While the source code of the target binary might not always be available it is still possible to instrument the target using static binary rewriting [7] [13]. Furthermore, the mutation strategies used by coverage-guided fuzzers do not use any input model definition to apply targeted changes to the testing inputs. They apply random mutation strategies to inputs given in a corpus and then monitor the evolution of code coverage for guidance. OTAFuzz does not have access to any form of coverage feedback. It relies on grammar coverage to ensure that all handlers of interest are tested. Because OTAFuzz has access to the grammar of the fuzzed protocol it applies targeted mutations. The fuzzing speed is also very different as OTAFuzz performs over-the-air fuzzing and is bound by the over-the-air speed limit which is several orders of magnitude smaller than the speed achieved by greybox mutation-based fuzzers.

Generation-based fuzzers rely on input format definition to generate fuzzing inputs for the target. The challenge with generational fuzzing is that the fuzzer needs to be designed to fit the

target. Some fuzzing frameworks such as PEACH [14] allow you to create your generation-based fuzzer by providing a data model. However, creating such a data model for protocols like RRC cannot be done manually and requires a compiler from ASN.1 to the Peach data model. Furthermore, we have less control over the fuzzing logic, the coverage metric and the mutation strategies used by it.

6.2 Protocol fuzzing

Protocol fuzzing is a major research area in fuzzing. Many studies have successfully employed fuzzing to test network protocols [18] [43] [39], yet it remains a challenging topic. This is largely due to the variability of protocols and the unique challenges each one brings. One notable recent project is AFLNet [35] which in addition to code-coverage feedback uses state information obtained by observing the server responses to make progress. It is a mutation-based greybox fuzzer that is seeded with a corpus of previous messages exchanged between the server and the client. Thus, it does not consider protocol specifications and requires instrumentation of the target binary. The disadvantage of mutation-based fuzzing for protocols is that the fuzzer receives information about the general form of protocol messages only in the corpus provided at the beginning. It is unaware of the internal field structure of the individual protocol messages in the corpus.

AspFuzz [27] is a state-aware protocol fuzzer targeted at application layer protocols. They generate protocol messages from specifications and add anomalies. They perform two types of attack strategies: format violation and anomalous order attacks. Format violation attacks are changes of the fields in a message, such as changing the number of fields, setting a field value to an erroneous one, or adding a format string to text fields. Anomalous Order attacks focus on sending messages that are invalid in the current protocol state. Their work is similar to ours as we also use protocol specifications to generate protocol messages and then apply targeted mutations to them. However, they do not use any specific oracles and still manually decide whether an attack was successful or not. Nevertheless, their fuzzer has been proven effective by re-discovering several past vulnerabilities in POP3 and HTTP servers. Similar to our work, they perform grammar-aware mutations on generated

packets.

Berserker [37], similar to OTAFuzz, is a RRC protocol fuzzer. It is a mutation-based but also and generation-based fuzzer. It uses a man-in-the-middle approach where it either applies grammar-aware mutations to an intercepted packet or replaces it with a generated one. With this approach, sitting as man-in-the-middle, it uncovered two new vulnerabilities in srsLTE, an LTE software implementation. Their work only applied on srsLTE and no actual device was tested using their method. Similar to OTAFuzz, it uses the RRC ASN.1 schema to generate valid RRC messages. Instead of mutating valid RRC messages, Berseker mutates the RRC ASN.1 definitions and then generates new RRC messages valid under the mutated ASN.1 definitions.

6.3 Static analysis

BASESPEC [25] compares the reverse-engineered layer-3 message decoder implementation in basebands to the formats extracted from the specification. This comparative analysis is automatic and tries to find non-conforming implementations of NAS message handlers. To perform this comparison it is necessary to reverse engineer most of the baseband which requires a lot of manual work, it is nevertheless a one-time effort. Their approach allowed them to uncover 5 semantic bugs but also 4 memory vulnerabilities. Another notable static analysis paper is BaseComp [24]. It relies on a semi-automated approach to verify the integrity mechanism in NAS protocol implementations in baseband. While human reverse engineering of baseband images was still required they managed to automatically discover the integrity check function. Once this function is identified, symbolic analysis is used to verify the correctness of the integrity mechanism. The authors identified 29 vulnerabilities during their research in Samsung Exynos and MediaTek basebands.

6.4 Baseband emulation

Baseband emulation is the go-to method to efficiently fuzz baseband firmware. Re-hosting the firmware allows for easier debugging as one can hook custom debug code into the firmware to obtain more insight. It also allows the addition of memory sanitization which helps detect memory-related bugs while fuzzing. This is achieved by hooking functions related to memory management. Emulation of baseband is challenging as baseband firmware is proprietary closed-source software and their architecture can vary across different vendors. For example, Qualcomm basebands use a custom ISA called Hexagon [10]. Emulation of baseband necessitates a consequent initial effort to make the baseband run. Once the heavy lifting of reverse engineering and emulation is done, instrumented fuzzing can be used to test the firmware. The advantage of fuzzing a re-hosted baseband compared to over-the-air fuzzing is the absence of ambiguity if the baseband crashed or not.

One notable work in the domain of baseband emulation and fuzzing is Firmwire [20]. Firmwire can emulate 213 unique firmware images by supporting two out of the 5 baseband vendors: Samsung and MediaTek. The emulated firmware is then fuzzed using AFL++ [15] leveraging coverage feedback. In total, 4 new vulnerabilities in LTE-RRC and GSM protocols were discovered and confirmed by sending them over the air to a test device.

BaseSAFE [29], a precursor to Firmwire, supports only MediaTek basebands and focuses its fuzzing efforts on the RRC and NAS protocols. Utilizing the Unicorn emulator, BaseSAFE partially re-hosts MediaTek basebands and uses AFL++ [15] to perform coverage-guided mutation-based fuzzing. The authors achieved the execution of a hundred test cases per second per core with this method. However, unlike Firmwire, the baseband is not initialized from the boot but relies on manual state initialization emulated baseband. The emulation is only partial and fails to replicate interactions between not emulated parts of the firmware. Nevertheless, the authors uncovered multiple memory corruptions in the baseband originating from the same root cause in the NAS EMM parser. LTE and GSM are not the only stacks that have been emulated and tested by previous

work. Frankenstein [38] for example found several vulnerabilities in the Bluetooth stack of Samsung Exynos baseband firmware.

6.5 Over-the-air testing

Over-the-air Dynamic testing uses a more systematic way than fuzzing to verify the correctness of basebands. Several projects [22] [26] [9] carefully analyze the LTE specifications to determine a set of specific test cases. Because the specification describes all the correct behaviors the system should have under any state and condition it is possible to determine for each test if it was successful or not. This approach aims at uncovering semantic bugs in the baseband. This type of bug is typically caused by logic bugs in the specifications or induced by a wrong implementation of them. A notable recent contribution to dynamic testing was brought by DoLTEst [34] which focused on negative testing basebands.

6.6 Over-the-air fuzzing

While over-the-air fuzzing is generally not recommended because of its speed, there are still works that show promising results. Owfuzz [8] tests WiFi protocol implementation of devices over-the-air and was able to uncover 23 flaws which got assigned 8 CVE IDs. They also rely on a random packet generator to produce valid frames and mutate one IE at a time in the packet frame. Similar to NAS, the structure of WiFi IEs is defined using the type-length-value (TLV) format. Over-the-air fuzzing has also already been applied to basebands. T-Fuzz [23] is a fuzzing framework that enables over-the-air NAS fuzzing. It is an extension of the already existing conformance test framework TITAN [47]. Thus, it uses TTCN-3 models as input for its generation-based fuzzer. To detect baseband crashes they monitor the CPU usage, memory consumption, and the logs of the tested device. However, it is unclear from the evaluation which devices have been tested and which exact vulnerabilities were uncovered.

Chapter 7

Conclusion

7.1 Future works

Over-the-air performance The observed over-the-air performance is below the expected standard of 50 packets per second. Even though the over-the-air approach poses stability problems we think that the fuzzing rate can be improved as we observed that the delays come from the communication between the fuzzer and the LTE network.

ASN.1 optional tags In ASN.1 definitions a field may be designated as *OPTIONAL*. Conditional statements can be used in conjunction with this optional tag to specify the circumstances in which the field becomes mandatory, such as the presence of another field. OTAFuzz does not currently consider these dependencies between fields as the conditions are defined by natural language.

Full ASN.1 support OTAFuzz only supports a subset of ASN.1 types during the generation phase as RRC specifications do not make use of all available ASN.1 data types. However, by adding support for these in the generation phase it is possible to generate testing packets for any protocol defined by ASN.1 specifications.

7.2 Conclusion

In conclusion, we build OTAFuzz an over-the-air fuzzer for RRC and NAS-EMM protocol implementations. We overcome the lack of coverage feedback from the baseband by proposing a generative approach coupled with targeted grammar-aware mutations designed to find memory vulnerabilities. Furthermore, we analyze multiple oracles using various feedback sources to detect baseband crashes. We successfully use the most reliable ones to discover baseband vulnerabilities. We tested 5 different basebands from 4 different vendors and discovered 16 1-day vulnerabilities in outdated Samsung Exynos firmware, 2 0-day vulnerabilities in the latest Samsung Exynos firmware, and 1 vulnerability in a one-year-old MediaTek firmware (classification as 1-day or 0-day pending). Our results show that over-the-air fuzzing is a reasonable approach for finding memory corruptions in baseband firmware.

Bibliography

[1] 3GPP. *3GPP TS 24.301 version 17.6.0 Release 17*. https://www.etsi.org/deliver/etsi_ts/124300_124399/124301/17.06.00_60/ts_124301v170600p.pdf. Accessed: 2023-08-28.

[2] 3GPP. *3GPP. TS 36.323 v17.1.0 - Clause 4.3.1*. https://www.etsi.org/deliver/etsi_ts/136300_136399/136323/17.01.00_60/ts_136323v170100p.pdf. Accessed: 2023-08-21.

[3] 3GPP. *3GPP. TS LTE RRC 36.331 v17.1.0*. https://www.etsi.org/deliver/etsi_ts/136300_136399/136331/17.01.00_60/ts_136331v170100p.pdf. Accessed: 2023-08-27.

[4] 3GPP. *3GPP. TS LTE RRC 36.331 v17.1.0 - Clause 11.2*. https://www.etsi.org/deliver/etsi_ts/136300_136399/136331/17.01.00_60/ts_136331v170100p.pdf. Accessed: 2023-08-27.

[5] Cornelius Aschermann, Patrick Jauernig, Tommaso Frassetto, Ahmad-Reza Sadeghi, Thorsten Holz, and Daniel Teuchert. "NAUTILUS: Fishing for Deep Bugs with Grammars." In: *Symposium on Network and Distributed System Security (NDSS)*. NDSS'19, 2019.

[6] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. "Stateful Greybox Fuzzing". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3255–3272. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/ba.

[7] Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer. "ARMore: Pushing Love Back Into Binaries". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA:

USENIX Association, Aug. 2023, pp. 6311–6328. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/di-bartolomeo.

[8] Hongjian Cao, Lin Huang, Shuwei Hu, Shangcheng Shi, and Yujia Liu. "Owfuzz: Discovering Wi-Fi Flaws in Modern Devices through Over-The-Air Fuzzing". In: *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '23. Guildford, United Kingdom: Association for Computing Machinery, 2023, pp. 263–273. ISBN: 9781450398596. DOI: 10.1145/3558482.3590174. URL: https://doi.org/10.1145/3558482.3590174.

[9] Merlin Chlosta, David Rupprecht, Thorsten Holz, and Christina Pöpper. "LTE Security Disabled: Misconfiguration in Commercial Networks". In: *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '19. Miami, Florida: Association for Computing Machinery, 2019, pp. 261–266. ISBN: 9781450367264. DOI: 10.1145/3317549.3324927. URL: https://doi.org/10.1145/3317549.3324927.

[10] Lucian Codrescu. "Qualcomm Hexagon DSP: An architecture optimized for mobile multimedia and communications". In: *2013 IEEE Hot Chips 25 Symposium (HCS)*. 2013, pp. 1–23. DOI: 10.1109/HOTCHIPS.2013.7478317.

[11] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. "Prospex: Protocol Specification Extraction". In: *2009 30th IEEE Symposium on Security and Privacy*. 2009, pp. 110–125. DOI: 10.1109/SP.2009.14.

[12] Android Developers. *Telephony library*. https://developer.android.com/reference/android/provider/Telephony. Accessed: 2023-08-19.

[13] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization". In: *IEEE International Symposium on Security and Privacy*. 2020.

[14] Michael Eddington. "Peach fuzzing platform". In: *Peach Fuzzer* 34 (2011), pp. 32–43.

[15] 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)*. USENIX Association, Aug. 2020.

[16] Google. *Multiple Internet to Baseband Remote Code Execution Vulnerabilities in Exynos Modems*. `https://googleprojectzero.blogspot.com/2023/03/multiple-internet-to-baseband-remote-rce.html`. Accessed: 2023-08-25.

[17] Google. *oss-fuzz*. `https://github.com/google/oss-fuzz/tree/master`. Accessed: 2023-08-22.

[18] Serge Gorbunov and Arnold Rosenbloom. “Autofuzz: Automated network protocol fuzzing framework”. In: *Ijcsns* 10.8 (2010), p. 239.

[19] Marco Grassi and Xingyu Chen. *Over The Air Baseband Exploit: Gaining Remote Code Execution on 5G Smartphones*. `https://keenlab.tencent.com/zh/whitepapers/us-21-Over-The-Air-Baseband-Exploit-Gaining-Remote-Code-Execution-on-5G-Smartphones-wp.pdf`. Accessed: 2023-08-25.

[20] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin R. B. Butler. “FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware”. In: *Symposium on Network and Distributed System Security (NDSS)*. 2022.

[21] *How to talk to the Modem with AT commands*. `https://forum.xda-developers.com/t/a-sgs2-serial-how-to-talk-to-the-modem-with-at-commands.1471241/`. Accessed: 2023-08-07.

[22] Syed Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. “LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE”. In: Jan. 2018. DOI: 10.14722/ndss.2018.23319.

[23] William Johansson, Martin Svensson, Ulf E. Larson, Magnus Almgren, and Vincenzo Gulisano. “T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols”. In:

2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. 2014, pp. 323–332. DOI: 10.1109/ICST.2014.45.

[24] *BaseComp: A Comparative Analysis for Integrity Protection in Cellular Baseband Software (to appear)*. Anaheim, CA, Aug. 2023.

[25] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. "BaseSpec: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols". In: (Feb. 2021).

[26] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. "Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1153–1168. DOI: 10.1109/SP.2019.00038.

[27] Takahisa Kitagawa, Miyuki Hanaoka, and Kenji Kono. "AspFuzz: A state-aware protocol fuzzer based on application-layer protocols." In: *ISCC*. IEEE Computer Society, 2010, pp. 202–208. ISBN: 978-1-4244-7755-5. URL: http://dblp.uni-trier.de/db/conf/iscc/iscc2010.html#KitagawaHK10.

[28] Daniel Komaromy and Lorant Szabo. *How To Tame Your Unicorn: Exploring And Exploiting Zero-Click Remote Interfaces of Huawei Smartphones*. https://i.blackhat.com/USA21/Wednesday-Handouts/US-21-Komaromy-How-To-Tame-Your-Unicorn-wp.pdf. Accessed: 2023-08-25.

[29] Dominik Maier, Lukas Seidel, and Shinjo Park. "BaseSAFE". In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2020. DOI: 10.1145/3395351.3399360. URL: https://doi.org/10.1145%2F3395351.3399360.

[30] Samsung Mobile. *SVE-2021-22051*. https://security.samsungmobile.com/securityUpdate.smsb?year=2021&month=10. Accessed: 2023-08-20.

[31] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. "SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale". In: *20th USENIX Security Symposium (USENIX Security 11)*. San Francisco, CA: USENIX Association, Aug. 2011. URL: https://www.usenix.

org/conference/usenix-security-11/sms-death-analyzing-attacking-mobile-phones-large-scale.

[32] OpenSSL. *OpenSSL States*. https://www.openssl.org/docs/man1.1.1/man3/SSL_in_connect_init.html. Accessed: 2023-08-21.

[33] P1Sec. *Pycrate library*. https://github.com/P1sec/pycrate. Accessed: 2023-08-07.

[34] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. "DoLTEst: In-depth Downlink Negative Testing Framework for LTE Devices". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1325–1342. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/park-cheoljun.

[35] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. "AFLNet: A Greybox Fuzzer for Network Protocols". In: *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*. 2020.

[36] Andreas Pointner. "Mining Attributed Input Grammars and their Applications in Fuzzing". In: *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2023, pp. 493–495. DOI: 10.1109/ICST57152.2023.00059.

[37] Srinath Potnuru and Prajwol Kumar Nakarmi. "Berserker: ASN.1-based Fuzzing of Radio Resource Control Protocol for 4G and 5G". In: *CoRR* abs/2107.01912 (2021). arXiv: 2107.01912. URL: https://arxiv.org/abs/2107.01912.

[38] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. "Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 19–36. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ruge.

[39] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association,

Aug. 2015, pp. 193–206. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[40] Samsung. *CVE-2023-36481*. https://semiconductor.samsung.com/support/quality-support/product-security-updates/. Accessed: 2023-08-27.

[41] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. "KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, pp. 167–182. ISBN: 9781931971409.

[42] Zhan Shu and Guanhua Yan. "IoTInfer: Automated Blackbox Fuzz Testing of IoT Network Protocols Guided by Finite State Machine Inference". In: *IEEE Internet of Things Journal* 9.22 (2022), pp. 22737–22751. DOI: 10.1109/JIOT.2022.3182589.

[43] Congxi Song, Bo Yu, Xu Zhou, and Qiang Yang. "SPFuzz: A Hierarchical Scheduling Framework for Stateful Network Protocol Fuzzing". In: *IEEE Access* 7 (2019), pp. 18490–18499. DOI: 10.1109/ACCESS.2019.2895025.

[44] srsRAN. *srsRAN_4G project*. https://docs.srsran.com/projects/4g/en/latest/. Accessed: 2023-08-07.

[45] Robert Swiecki and F Gröbert. "Honggfuzz". In: *Available online a t: http://code.google.com/p/honggfuzz* (2016).

[46] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. "FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler". In: *Proceedings of the 2023 32th USENIX Security Symposium*.

[47] Artem Yushev, Manuel Schappacher, and Axel Sikora. "Titan TTCN-3 Based Test Framework for Resource Constrained Systems". In: *MATEC Web of Conferences* 75 (Jan. 2016), p. 06005. DOI: 10.1051/matecconf/20167506005.

A Firmwire 1-day packet structure

```
<DL-DCCH-Message>
  <message>
    <c1>
      <rrcConnectionReconfiguration>
        <rrc-TransactionIdentifier>1</rrc-TransactionIdentifier>
        <criticalExtensions>
          <c1>
            <rrcConnectionReconfiguration-r8>
              <dedicatedInfoNASList>
                <DedicatedInfoNAS/>
              </dedicatedInfoNASList>
              <securityConfigHO>
                <handoverType>
                  <intraLTE>
                    <keyChangeIndicator>
                      <false/>
                    </keyChangeIndicator>
                    <nextHopChainingCount>6</nextHopChainingCount>
                  </intraLTE>
                </handoverType>
              </securityConfigHO>
              <nonCriticalExtension>
                <lateNonCriticalExtension>
                  <RRCConnectionReconfiguration-v8m0-IEs>
                    <lateNonCriticalExtension/>
                  </RRCConnectionReconfiguration-v8m0-IEs>
                </lateNonCriticalExtension>
              </nonCriticalExtension>
            </rrcConnectionReconfiguration-r8>
          </c1>
        </criticalExtensions>
      </rrcConnectionReconfiguration>
    </c1>
  </message>
</DL-DCCH-Message>
```