KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Rise of the Machines

On the Security of Cellular IoT Devices

Jochem Hoes

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, option ICT
Security and Networks

Thesis supervisors:

Prof. Dr. Ir. Bart Preneel,
MBA. MsC. Michael Dieudonné

Assessors:

Prof. Dr. Ir. Sofie Pollin,
Dr. Ir. Abuding Aishajiang

Mentors:

Dr. Ir. Dave Singelée,
Dr. Ir. Germán Corrales Madueño

Academic year 2020 – 2021

Preface

First and foremost, I would like to thank my promotor Bart Preneel and Keysight's Country General Manager Michael Dieudonné, for allowing me to work on this thesis. It has allowed me to expand my fascination for IoT networks and security in real-life systems.

I want to express my special gratitude to my two daily supervisors Dave Singelée and German Madueno. Their invaluable input has shaped this thesis into a successful closure of my academic career. Without our weekly meetings and their extensive feedback, both the texts and achievements of this thesis would not come close to what I can deliver today.

I must also thank researchers from the ASSET Research Group at Singapore University of Technology and Design, especially Matheus Garbelini, Xingbin Jiang, and Cyril Tan. Their expertise made it possible to take our experiments to the next level.

The involvement of Keysight, in general, brought valuable support for this thesis. They provided space for the platform setup and tools to support its design. Instruments like the base station emulator would not be available without their engagement. I also wish to thank my Keysight colleagues Jan Sirka, Stelle Van Hoven, and Bart Vanhoof for their warm welcome despite complicated Covid regulations.

I am also thankful for my assessors Sofie Polin and Abuding Aishajiang, whose midterm constructive comments helped me achieve the goals of this thesis.

Last but not least, I want to give thanks to my sister Lore for her priceless advice in writing this thesis. Ultimately, without the unconditional support from my whole family, I could have never succeeded these last five years. My everlasting appreciation goes out to them.

Jochem Hoes

Contents

Abstract

With the number of connected embedded systems growing by the day, so does the Internet of Things (IoT). These IoT devices have proven their value in many different sectors, like industrial automation, healthcare, energy, or the automotive market. IoT is here, and it is here to stay. Within this IoT paradigm, the amount of cellular IoT devices is expected to grow the most rapidly. Cellular IoT devices use cellular 3GPP standards for their connectivity and can benefit from the presence of existing infrastructure, leading to an enormous potential for wide-area IoT applications. However, the design of these devices often leaves security vulnerabilities open. Limited resources, the heterogeneity of devices, or the complexity of the specification obstruct the development of secure solutions. A discovered vulnerability, furthermore, impacts many devices. Patching these devices in a deployed environment is often a challenging and costly endeavor. We have constructed a platform that uses network fuzzing for extensive LTE IoT device testing. In fuzzing, we send many malicious messages to a device intending to trigger a vulnerability. Our platform addresses the challenges regarding the fuzzing of complex protocols and the complications of embedded device monitoring. It can be used by system designers to improve the security at design time or by penetration testers to evaluate a product's robustness.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations

AFL	American Fuzzy Lop
ARP	Access Reservation Procedure
CAN	Controller Area Network
eNB	E-NodeB
EPC	Evolved Packet Core
EPS	Evolved Packet System
E-UTRAN	Evolved UMTS Terrestrial Radio Access Network
HSS	Home Subscriber Server
IMSI	Internation Mobile Subsriber Identity
IoT	Internet-of-Things
LPWAN	Low Power Wide Area Network
LTE	Long Term Evolution
MAC	Medium Access Control
MIB	Master Information Block
MME	Mobility Management Entity
NAS	Non Access Stratum
OS	Operating System
PCRF	Policy and Charging Rules Function
PDCP	Packet Data Convergence Protocol
P-GW	Packet Data Network Gateway
RAN	Radio Access Network
RAR	Random Access Response
RB	Resource Block
RLC	Radio Link Controller
RRC	Radio Resource Control
SDR	Software Defined Radio
S-GW	Serving Gateway
SIB	System Information Block
SR	Scheduling Request
SUT	System Under Test
UE	User Equipment
UL	Uplink
USIM	Universal Subscriber Identity Module

Chapter 1

Introduction

In recent years, the Internet of Things (IoT) has been coined as a disruptive technology. The technology is said to transform consumers' everyday lives and increase automation and connectivity in the industry. Simply put, IoT is a networking concept that allows a heterogeneous set of embedded devices to communicate with each other. These devices can range from simple air pollution sensors to complex autonomous car controllers. A wide range of applications can benefit from this paradigm, each with its own specific needs. Because each application has its own set of requirements, IoT is associated with many types of protocols. For applications with devices in close proximity, Bluetooth is often used. For household applications, Wi-Fi or Zigbee are well suited. For larger-scale networks of devices, protocols like LORAWAN, SigFox, or cellular connections are used. This thesis takes a closer look at IoT devices with a cellular connection.

It is estimated that in 2020, the number of connected IoT devices has overtaken the number of connected non-IoT devices (laptops, smartphones, computers). With 11.7 billion IoT devices, there are now more IoT connections deployed than the 10 billion 'classic' connected devices. With an average annual growth rate of 29% between 2013 and 2019, the sector is growing vastly faster than the number of non-IoT devices [1]. Many wild estimations circulate about the future growth towards 2030. For example, Transforma Insights predicts a total number of 24 billion IoT connections by 2030 [2]. With these predictions, we have to make the nuance that the popular 2010 prediction of 50 billion connected devices by 2020 was also a clear overestimation [3]. Nevertheless, the sector's rapid growth indicates the importance of IoT in the industrial and societal future.

Cellular IoT devices are IoT devices that use 3GPP standardized protocols for their connectivity [4]. Common used standards today are the LTE [5], LTE-M [6], and NB-IoT [7] standards. This diversity in standards makes cellular IoT suitable for applications with varying requirements. LTE and other conventional cellular connections (UMTS or GPRS), as used by most phones, are best suited for applications that need a high data throughput. However, many IoT applications have no need for this high throughput. For these use cases, 3GPP has developed the NB-IoT and LTE-M standards. These standards sacrifice the high throughput to get improved coverage, device battery life, and device cost.

In terms of coverage, NB-IoT allows for 164 dB of maximum path loss, and LTE-M allows for 156 dB of pathloss. In a realistic setting, this entails that LTE-M can support

99% of indoor devices, and NB-IoT can even provide service to 95% of devices in deep indoor environments (basements, for example) [8]. The device cost is lowered by using a much smaller bandwidth than in regular LTE, which reduces the complexity of radio frequency hardware design. LTE-M uses only 1.4 MHz of bandwidth compared to 20 MHz for regular LTE. NB-IoT cuts this even further to 200 kHz. Additionally, features that were introduced in LTE for higher throughput, like MIMO, are removed from the LTE-M and NB-IoT specification [9]. These changes result in a throughput of a few hundred kbps to 1 Mbps for the LTE-M standard [10] [11]. The NB-IoT standard achieves a few ten to a hundred kbps of throughput, depending on the network configuration [12].

These cellular IoT connections will play a major role in the expected IoT evolution. Fredrik Jejdling from Ericsson expects the number of cellular IoT connections to grow almost twice as fast as short-range (Wi-Fi, Bluetooth, ...) IoT connections [13]. Cellular IoT has the benefit of partially reusing the same infrastructure as non-IoT cellular connections. This infrastructure has already received many investments over the last 25 years due to the importance of cellular communication to end-users. Most Western and Asian countries already integrated the newer NB-IoT and LTE-M standards in their network [14]. Other benefits of cellular IoT over other LPWAN standards include scalability, interoperability, managed quality of service, and security [15]. Furthermore, the growth of cellular IoT will also be driven by telecommunication providers themselves in search of promising new revenue streams like industrial IoT [16].

Despite the promising improvements IoT gives us, there are also many security concerns about this new paradigm. IoT has become infamous for its lack of security features inside its devices. In 2019 Kaspersky announced they had detected over 100 million attacks on IoT devices [17]. Securing IoT devices is often more challenging than classic end-user devices, like laptops. Ali Abuagoub list some of the challenges in securing IoT devices [18]. The limited resources like memory, energy, and computational power of IoT devices are often a considerable hurdle for a secure design. Common public key cryptography is resource-intensive and not always suited for IoT devices [19]. On top of the limited resources, Abuagoub coins many other challenges like the heterogeneity of devices, scalability of the IoT network, and lack of standardization in the industry. Moreover, when bugs or security flaws are discovered, performing a firmware upgrade to widely deployed IoT devices often is a challenging endeavor [20].

It will come as no surprise that many researchers have found numerous security vulnerabilities in IoT devices. Neshenko et al. classify commonly found vulnerabilities in nine categories [21]. For example, they list the lack of access control in IoT devices, where many devices still function with default credentials. The Mirai malware used this vulnerability to create a botnet of at least 400.000 IoT devices [22]. An example of weak cryptography in IoT devices can be found in [23]. Researchers discovered that Tesla key fobs for the model S only used a 40-bit key. Using a time-memory trade-off table, they were able to impersonate the key fob in a matter of seconds and steal the car. In [24] a team of researchers from KU Leuven and the University of Birmingham proved the lack of secure communications in implantable cardiac defibrillators (ICD). The data transmissions from the ICD's were poorly obfuscated, leading to a risk of severe privacy breaches of patients. Further implementation flaws allowed for DoS attacks, draining the battery of the implants, and replay/spoofing attacks.

When it comes to cellular IoT, the end devices implement a very detailed specification. This specification, like LTE, is often not airtight and can still allow vulnerability exploits [25]. For instance, the very beginning of communication between a device and an LTE base station is still unauthenticated. This traffic is known as pre-authentication traffic. This lack of authentication allows adversaries to deploy fake base stations. It also enables the possibility of a downgrade attack. Here a target device is convinced by a malicious base station to reconnect to another malicious network that uses a less secure standard (like 2G). Due to fewer security features in 2G, it becomes easier for an adversary to attack the device further [26]. Despite these security issues, there is a continuous improvement of the security in 3GPP specifications [25]. Therefore, cellular IoT devices have a relatively strong security foundation compared to the ICDs or car key fobs from above.

On the other hand, the complexity of 3GPP specifications can lead to implementation mistakes, which leave devices vulnerable. As the number of lines of code increases, so do the amount of bugs in that code [27]. Michau and Devine show several implementation vulnerabilities found in LTE modems [28]. For example, they show that some LTE modems still accept unprotected NAS messages even after NAS protection was explicitly enabled. By playing with bit combinations that are not defined in the LTE standard, they could also obfuscate private data from some modems. In 2012, Ralf-Philipp Weinmann showed how attackers could exploit memory corruptions in faulty implementations [29].

The impact of vulnerabilities in IoT devices is often amplified by the supply chain. In the IoT market, it is common practice to reuse components. Starting each device from scratch would be infeasible, as this would explode design costs that could never be recuperated from the number of sold devices. The reuse of components is needed to become profitable in the IoT market. This reuse happens at many different levels. Both software libraries and hardware modules can be reused as they often implement standardized protocols needed in many IoT systems. This reuse can be inter- or intra-vendor. A vendor can internally reuse components from one product in another, but also different vendors can reuse the same components produced by a third party. As an illustration, three vendors of IoT modems: Quectel, Telit, and Thales, all use Qualcomm chips in their modems. This reuse, an inherent trait of the IoT market, magnifies the impact when a vulnerability is discovered. The module in which the vulnerability is found is likely reused in other IoT systems. The widespread consequences of the Ripple 20 vulnerabilities demonstrate this. Early 2020, security researchers of JSOF found 19 zero-day vulnerabilities in a TCP/IP software stack sold by a company named Treck. Over thirty different vendors have been confirmed to be affected by these vulnerabilities [30]. A year later, in April 2021, they announced a new bug they had found in collaboration with Forescout, which again affected possible millions of IoT devices worldwide [31].

Multiple approaches can be taken to avoid security vulnerabilities in a system. A vulnerability often originates from a bug in the device implementation that an adversary can exploit. These bugs can be searched by manually reviewing the application. This is, however, error-prone as programmers are essentially bad at finding bugs. Otherwise, the bug would not exist. Another option is the use of automated verification tools. Frama-c is an example of such a tool that aims to reduce bugs in C source code through static and dynamic analysis [32]. These tools are again limited. They are only as strong as their own programming, do not understand the logic and goal of the code, and can only test

when source code is available. In cellular IoT, parts of the protocol are often implemented in hardware to improve efficiency. We could find much fewer tools that test hardware description languages for bugs through code analysis. A third option is extensive testing. Here a technique called fuzzing comes into play. Fuzzing is the concept of sending lots of malformed or correct inputs to a device under test and monitoring the device's response. Fuzzing has the benefit of being able to test existing systems without access to the source code. It can test devices under the same circumstances as to how they would be deployed in the field.

In this thesis, we continue the research on IoT device security through fuzzing. We will focus on cellular IoT devices which contain broadband LTE modems for their communication. This thesis is the first step in a larger research project on the security of cellular IoT devices. Hence, apart from the security research itself, we will have a strong focus on building a modular platform that can be used in future work. Later, this platform might be extended to work with a broader range of cellular standards like NB-IoT or 5G. In the design, we focus on automating all steps of the fuzzing process while preserving the flexibility to test a heterogeneous set of devices.

An automated, user-friendly platform creates the opportunity to test lots of different devices swiftly. In the future, extensive device validation tests can lead to the adoption of IoT certification. Similar to how energy labels are given to household appliances in Europe, IoT devices can be certified to aid customers in evaluating the quality of a device.

This project is a collaboration between the COSIC research group and Keysight Technologies. COSIC is one of the world-leading security research groups. Their research focuses on symmetric cryptography, hardware security, privacy, or mobile and wireless security. Keysight Technologies creates a wide variety of electronic testing equipment. Their products range from basic oscilloscopes to signal analyzers, cellular base station emulators, or mobile network testing equipment. Their equipment tests devices from the physical layer to the application layer. The interested reader can learn more about Keysight in Appendix A.

The rest of the thesis will adopt the following structure. Chapter 2 will give an overview of the needed background knowledge of this thesis. It provides a compact overview of LTE and the previous research in fuzzing. Chapter 3 will present the overall architecture of our platform. It will discuss the architectural diagram without an in-depth look at the individual components. The following two chapters will take a deep dive into two of our most researched building blocks. Chapter 4 discusses how we tackled the challenges of building a fuzzer for the complex LTE protocol. Chapter 5 explores our researched feedback mechanisms to monitor device behavior. In Chapter 6 we prove the effectiveness of our designed platform through experimental attacks on an IoT modem. These tests allowed us to evaluate the platform's performance and stability, as well as the power of the researched design choices.

Chapter 2

Background

This chapter aims to supply the reader with the necessary background information for a good understanding of this thesis. Two main concepts will be explained in this chapter. In the first section, we give a compact overview of the LTE protocol. We will not give a detailed explanation of the entire protocol, as this is beyond the goal of the thesis. We instead want to explain the concepts and ideas of LTE that are most relevant to this thesis. In the second section, we take a closer look at fuzzing. We discuss the different types of fuzzing, their use in research, and the type of fuzzing we use in this thesis.

2.1 Long Term Evolution (LTE)

As stated above in this section, we take a closer look at the LTE protocol. We start with an overview of the LTE architecture. We will also clarify the many abbreviations often used in LTE. Afterward, we take a closer look at the LTE protocol layers and connection establishment.

2.1.1 LTE Overview

Before delving into the parts of the LTE protocol most relevant to this thesis, we supply the reader with an overview of the basic LTE architecture and its components. Figure 2.1 gives a visual representation of the LTE architecture basics. It can be split into two logical parts: the Radio Access Network (RAN) and the Evolved Packet Core (EPC). The RAN is sometimes also called E-UTRAN (Evolved UMTS Terrestrial Radio Access Network). Together the EPC and RAN form the Evolved Packet System (EPS). There is a logical distinction between the messages in the network that serve as control information and messages that contain end-user data. The control messages are called the control-plane, and the data for end-users is contained in the user-plane. [33]

The **EPC** forms the core of an LTE network. It does not concern itself with wireless communication but provides the administrative processes needed for managing a complex network. Some of these processes include authentication, connection setup, billing, or connectivity to the Internet. Due to the split of radio access and core network functionalities, it is possible to use a single core network to serve older cellular technologies like

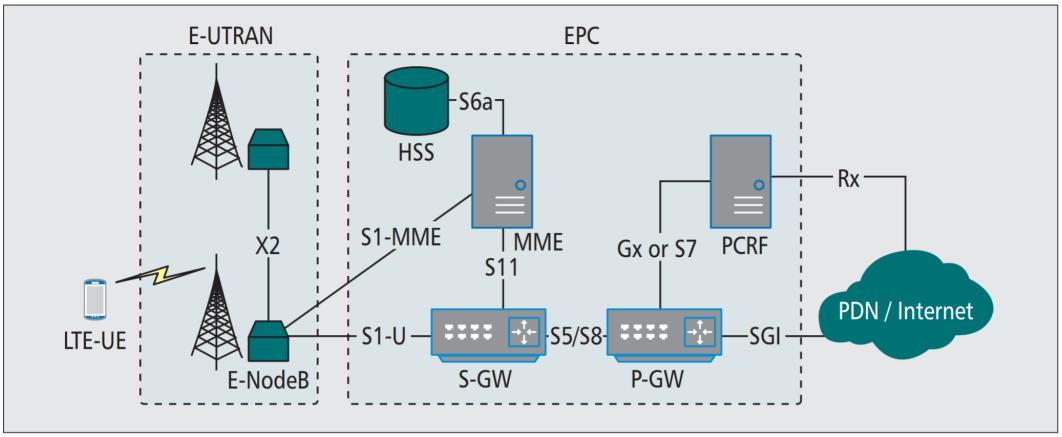

FIGURE 2.1: An overview of the LTE architecture. [34]

3G as well. Interesting about the LTE core network is the use of an all packet-switched domain. Older cellular standards still used circuit-switched connections for calls, but LTE made these connections packet-switched as well. The EPC contains several different nodes with different functionalities, briefly described below. [33]

First, we have the *Mobility Management Entity* (MME). The MME is a cornerstone in the control plane of the network. It is responsible for connection establishment and release, handling state transitions of devices, authenticating users, and more. The MME is connected to the *Home Subscriber Service* (HSS) node. The HSS forms the main database of the network and contains information on the network's subscribers: security keys, identifiers, approximate location estimates, ... Next, we have the *Serving Gateway* (S-GW). It forms the main connection point for the user data from the RAN. The S-GW connects the base stations and the end-users to the core network. It also collects the statistics used for billing subscribers. The S-GW not only connects the LTE RAN but can also form a connection point for older cellular radio access networks. The P-GW, or *Packet Data Network Gateway*, is the connection point between the EPC and the public Internet. Here, IP addresses for the subscribers are managed and Quality of Service rules are enforced. The last block in the EPC is the *Policy and Charging Rules Function* (PCRF). This node is responsible for billing the subscribers and controlling the Quality of Service rules the P-GW will enforce. [33]

With the EPC briefly clarified, we move to the **Radio Access Network**. The RAN handles the wireless communication in the EPS. It consists of the cellular base station, called *eNodeB* (eNB), and the end devices that connect to the LTE network, called the *User Equipment* (UE). The RAN is responsible for all radio-related functionality: medium access control, retransmission protocols, multi-antenna protocols, error coding, and more. The regions which an eNodeB serves are called a cell. Each eNodeB can serve one or multiple of those cells. The cell a UE is located in indicates which eNodeB the UE will try to connect to. The eNodeB connects to the MME for control messages and to the S-GW for user data. Different eNodeBs are also connected to each other with the X2 interface. The purpose of this connection is a smooth *hand-over* of UE's when they travel between cells.

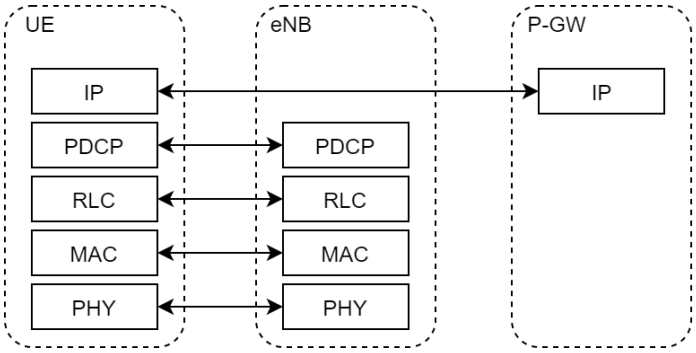

FIGURE 2.2: The LTE user-plane protocol stack.

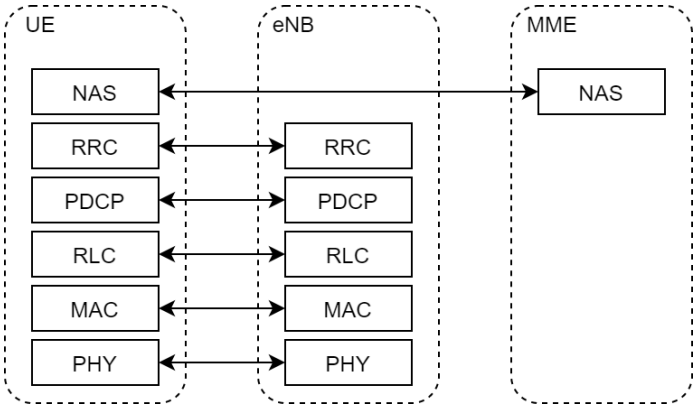

FIGURE 2.3: The LTE control-plane protocol stack.

Additionally, the X2 communication is used by eNodeB's to reduce inter-cell interference. [33]

2.1.2 LTE RAN Protocol Layers

Now that we understand the main nodes of an LTE network, we will discuss the different protocol layers of the RAN bottom up. Figure 2.3 and 2.2 show these layers for the control-plane and the user-plane separately. We see that the control-plane adds two separate layers, which are most important in our research.

The bottom layer is the *physical layer* of the LTE RAN. It is responsible for modulation, error detection and correction, segmentation, interleaving, and channel measurements. It offers functionality to higher layers by exposing transport channels. Figure 2.4 shows the mapping of different channels between the protocol layers. Transport channels are distinguished in the way the data is transferred. These transport channels are mapped to physical channels by the physical layer. Physical channels correspond to certain time and frequency slots in the LTE frame structure. These slots are called resource blocks (RB). An example of a frame structure can be seen in Figure 2.5. We will discuss this no further.

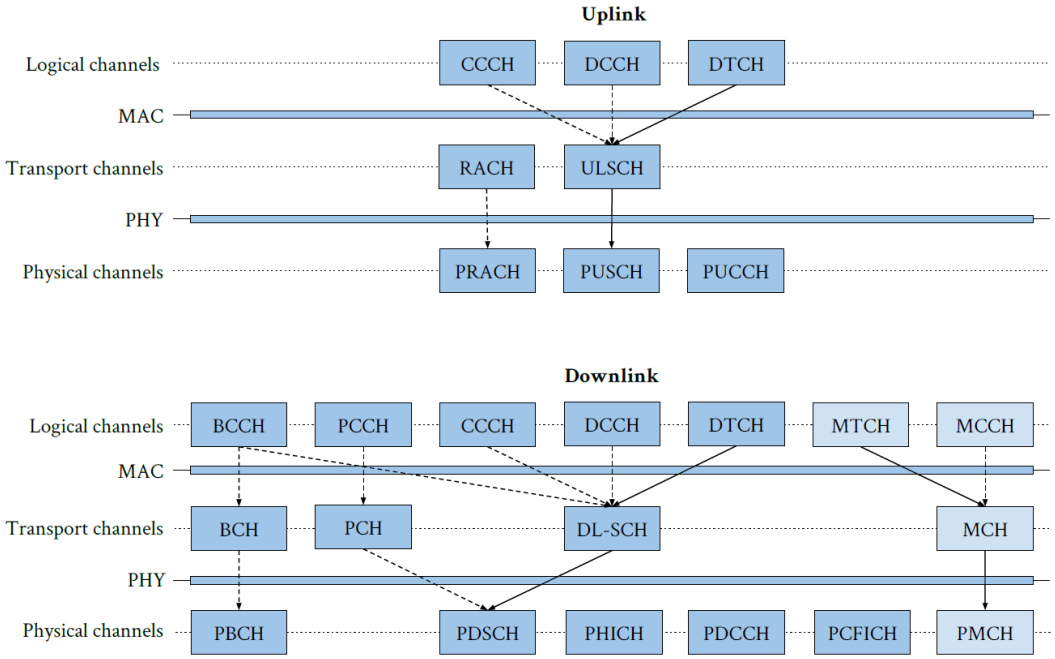

FIGURE 2.4: Logical, transport and physical channels in up- and downlink and how they map between different protocol layers. Light blue boxes are multicast channels. Dashed lines are routes purely for control information. Appendix B can be consulted for the full channel names. [36]

Only the PRACH channel will come up again shortly when discussing the connection establishment procedure. [35]

Above the physical layer sits the *Medium Access Control* (MAC) layer. As the name indicates, its main functionality is arbitration and access control of the shared transmission medium. The MAC layer maps logical channels to the transport channels offered by the physical layer. Logical channels are not distinguished by how data is transmitted but by the type of data transmitted on them. The primary responsibility of the MAC layer is the scheduling and multiplexing of the logical channels to the transport channels. Multiple logical channels need to be mapped to the same transport channels. The MAC layer decides the amount of access each logical channel gets to these transport channels. This also means that the MAC layer applies quality of service rules and prioritizes between logical channels and messages on these channels. Besides multiplexing the logical channels, the MAC layer of the eNodeB also has to schedule the transmissions of the connected UE's. This scheduling is influenced by multiple factors, such as the number of UE's in a cell, the bandwidth of the cell, the capabilities of the UE to support advanced modulation and coding schemes, or the reported radio conditions of the UE. For the last factor, the eNodeB's MAC layer utilizes the channel quality reports sent by the UE's MAC. When a UE receives permission from the scheduler to transmit in a particular resource block, this decision is permanent for N transmissions and receptions. The higher-level Radio Resource Control layer, discussed below, configures the value of N. [37]

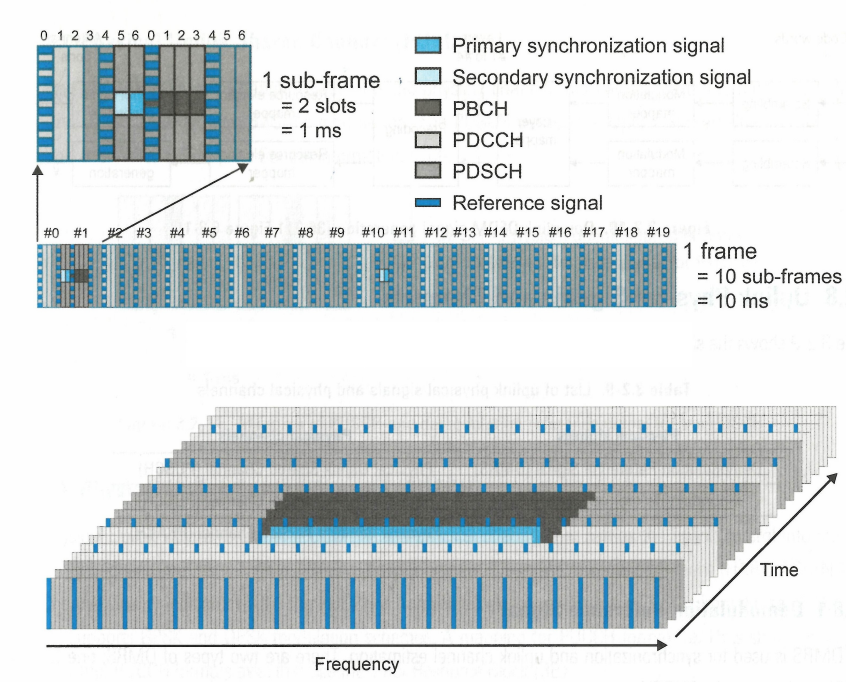

FIGURE 2.5: An example of the assignment of physical channels on top of the LTE frame. Each resource block gets assigned to a specific physical channel based on the network configuration. [35]

The next layer up is the *Radio Link Controller* (RLC) layer. The RLC layer acts as a buffer between the higher layers and the MAC layer. Additionally, the RLC provides extra acknowledgments, concatenation, segmentation, reassembly, and reordering of packets when necessary. The behavior of the RLC is configured by its mode of operation. The RLC has three of these modes of operation: Transparent Mode, Unacknowledged Mode, and Acknowledged Mode. In Transparent Mode, the RLC only serves as a buffer for the MAC layer and no extra functionality is provided. This mode is used for some system information messages. In Unacknowledged Mode, the RLC splits and reassembles packets without acknowledgments. This mode is used for delay-sensitive and error-tolerant traffic, like voice communication. Acknowledged Mode adds the extra acknowledgments for higher-level packets and is used for delay-tolerant but error-sensitive traffic. [36] [37]

At the top of the user-plane protocol stack sits the *Packet Data Convergence Protocol* (PDCP). Its purpose is to perform header compression/decompression, ciphering, and integrity protection. Above the PDCP layer in the user-plane, some regular networking protocols like IP or TCP can be run. For the control-plane, two more LTE-specific layers are present. [37]

The *Radio Resource Control* (RRC) layer performs a management role in coordinating the lower protocol layers. As will be motivated in chapter 4, it is the RRC layer whose messages we will fuzz in this thesis. The RRC layer is responsible for transmitting system-wide information. It, for example, dictates the number of retransmissions on some

physical channels, power thresholds for the UE to switch cells, or the timeouts used for particular parts of the protocol. It does so by creating System Information Blocks (SIB) and a Master Information Block (MIB). The MIB includes the most essential information a UE needs when connecting to the cell, such as the downlink channel bandwidth. It is the first piece of information a UE will look for when connecting to a cell. Therefore, the MIB schedule is fixed in the specification, so a UE knows where to look. The SIB's broadcast additional system information. There exist multiple types of SIB's, each responsible for a type of system information. Unlike the MIB, the SIB's can be scheduled dynamically on the BCCH logical channel. SIB type 1 contains, amongst others, information regarding the scheduling of the other SIB's. With the information from the MIB and SIB1, a UE can decode all the needed system information of the LTE cell. Besides broadcasting system information, the RRC layer is also largely responsible for the connection establishment procedure of a UE. We will discuss this procedure in more detail in the next subsection. The connection establishment moves the UE from the RRC_IDLE state, in which it only listens to system information and paging messages, to the RRC_CONNECTED state. When connected, a UE monitors the control channel and the shared data channel to find information intended for it. [37]

The top layer of the control-plane protocol stack is the *Non Access Stratum* (NAS) layer. This layer contains all functionality that involves direct communication between the UE and the EPC. This includes identification, authentication, security setup, attach procedure for a UE to register itself on the MME, requests for connectivity to the Internet, and more. This layer contains much functionality, but we will only discuss the details present in the connection establishment procedure. We end by noting that some NAS messages can be piggybacked on top of some RRC messages. An example of this will also be given in the connection establishment procedure discussed below. [37]

2.1.3 LTE Connection Establishment

The last part of our LTE overview will focus on the connection establishment. Above, we already explained that this protocol is run whenever a UE wants to switch from RRC_IDLE to RRC_CONNECTED, for e.g. searching a web page or making a call. Note that we will not discuss physical layer messages but only MAC layer messages and above. Figure 2.6 displays the messages sent between UE, eNodeB and MME.

The first step of the connection establishment is the random access procedure. The UE initiates this step after obtaining the system information from the MIB and SIB's. The UE selects a random PRACH preamble from a set of preambles that are configured in the system information. It sends this preamble on the Random Access transport channel (RACH)[1]. Multiple UE's may send the same preamble simultaneously. This contention will later be resolved. [38] [37]

Upon detecting a preamble on the RACH, the eNodeB responds with a Random Access Response[2] (RAR). The RAR contains an assigned resource block where the UE can transmit the third message. If a UE does not receive the RAR after it has sent its preamble on time, it restarts the procedure. If, after a certain maximum amount of retransmissions, still no RAR is received, an outage is declared. Note that the eNodeB

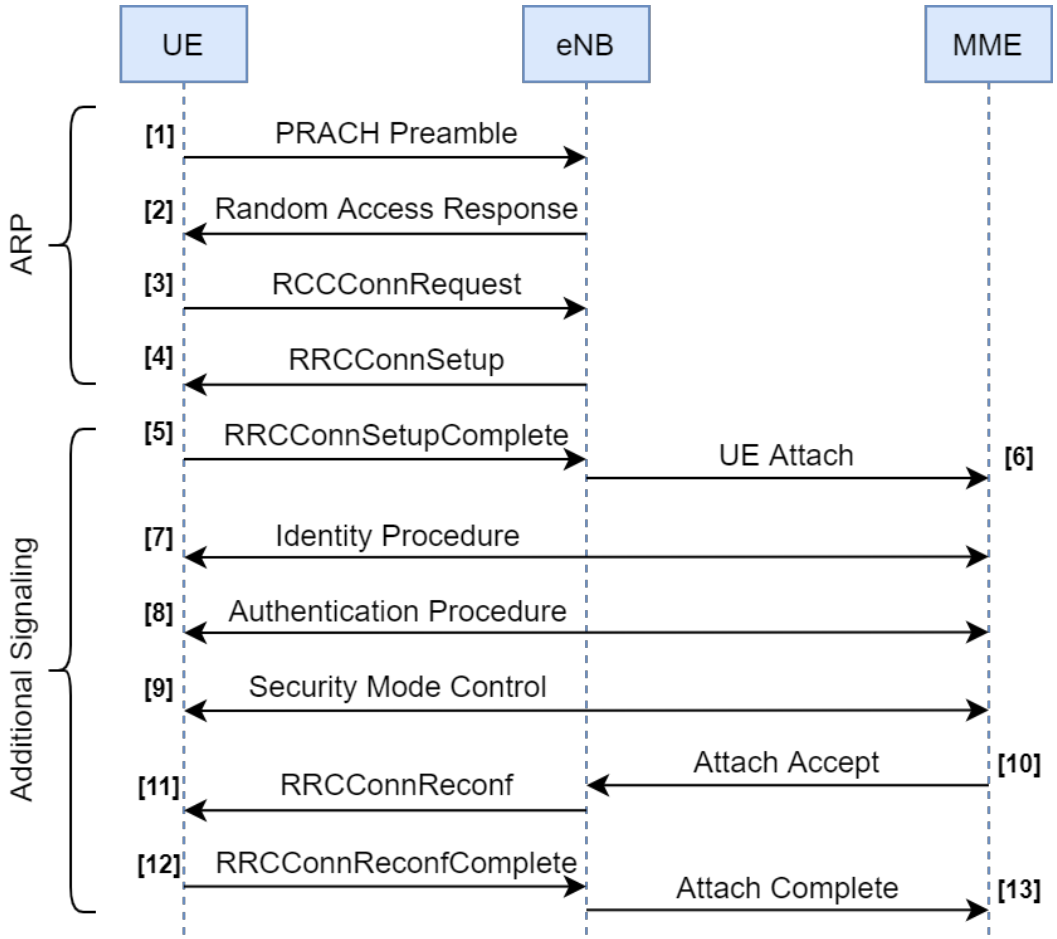

FIGURE 2.6: The LTE connection establishment procedure.

cannot detect collisions in the detected preambles; thus, no contention resolution has been made yet. [38] [37]

After a successful reception of the RAR, the UE transmits the RRC Connection Request message[3]. This message contains, amongst others, the UE's identity and reason for connecting (e.g. voice call or measurement report). An eNodeB can detect collisions in this message and only replies to messages that did not collide. Thus if two UEs chose the same preamble, the contention is now resolved. [38] [37]

The eNodeB responds to the RRC Connection Request with an RRC Connection Setup[4]. The setup message contains a lot of information about the cell configuration. This includes parameters for the RLC layer in up-and downlink, configuration of the UL-SCH transport channel, and physical layer. Examples of the parameters configured with this message are the number of retransmissions of certain reports in the RLC layer, the maximum wait time for reordering RLC packets, the periodicity of power headroom reports, and parameters that control the uplink power of the UE. If there are not enough available resources in the network, the eNodeB will deny the UE's request. [38] [37]

These past four messages are sometimes referred to as the Access Reservation Procedure (ARP). Since there has not been any contact with the EPC (holding the symmetric security keys), none of these messages have integrity protection or encryption. After the ARP, there is some additional signaling between the UE and MME to establish security and quality of service. Note that the fifth message, the RRC Connection Setup Complete[5], can be seen as part of the ARP. The UE sends this message after the RRC Connection Setup message, but it contains more piggybacked NAS information than RRC layer info. Hence it is often also taken as part of the extra signaling. [38] [37]

The next set of messages depend on whether the UE has connected to the network at an earlier time or not. Depending on this, the MME may initiate an identification, authentication, or security mode control procedure. In the identification procedure[7], the MME requests the identity of the UE. These messages can be NAS layer integrity protected or not. The authentication procedure[8] provides mutual authentication between UE and MME. Besides authentication, it also serves as a key agreement between the UE, eNodeB, and MME. Finally, the security mode control[9] enables the security context used for future communications. It tells the UE which ciphering and integrity protection algorithms to use. The MME decides this based on the UE's capabilities given in the piggybacked NAS message inside the RRC Connection Setup Complete. After the security mode setup, ciphering and integrity protection are done at the PDCP layer as well. [38] [37]

Lastly, after setting the security context, there is a reconfiguration with the RRC Connection Reconfiguration[11] and RRC Connection Reconfiguration Complete [12] messages. This procedure is used to modify some more of the parameters in the protocol stack. The difference with the initial RCC setup is that this procedure requires an established security context first. The same procedure is also used on hand-over, but we will not discuss it further. [37]

2.2 Fuzzing Background

One major cornerstone of this thesis is a technique called fuzzing. Fuzzing has been proposed as a way of finding bugs and vulnerabilities in software in 1990. In [39] Barton P. Miller, Louis Fredriksen, and Bryan So used fuzzing to discover bugs in the UNIX operating system. Since then, the technique has evolved massively, gaining intelligence, complexity, and effectiveness. While optimizing the fuzzing process itself is not the focus of this thesis, understanding its principles and core ideas is imperative to understanding this thesis. Hence, this section introduces fuzzing and an overview of the recent developments and possibilities.

First, we explain the idea behind fuzzing and why it is often picked over alternative options. Next, we provide an overview of the different characteristics of a fuzzer, followed by how one can use a fuzzer. We finish with the fuzzing challenges for our purposes, which will later be reiterated when discussing how we tackle them.

2.2.1 Fuzzing Core

Let us start by explaining the idea behind fuzzing. The fuzzing process revolves around sending many specially crafted inputs to a particular system under test (SUT). These

messages aim to induce errors in the system that would not occur in regular operation. These errors can be memory violations, incorrect null handling, infinite loops, a transition into a faulty state, ... The errors can be seen as normal software bugs, but in practical systems can often be transformed into security vulnerabilities. A simple infinite loop error can easily be exploited into a denial of service attack. When this occurs in a critical system, like a digital overpressure sensor in a chemical plant, such an attack can have catastrophic consequences.

Using fuzzing and thereby stepping outside the regular operation of a system, we can uncover these bugs and vulnerabilities, leading to safer and more secure systems. Many recent examples show the effectiveness of fuzzing in current security research. For example, researchers of the Technical University of Darmstadt found nine vulnerabilities in Apple's 'MagicPairing' protocol [40] using different fuzzing techniques. The protocol is built on top of the usual Bluetooth pairing protocol but contained errors that could lead to a crash of the Bluetooth stack or even induce a 100% CPU load. Werquin et al. show the effectiveness of fuzzing to uncover vulnerabilities inside the Controller Area Network (CAN) of automotive vehicles [41]. Two critical security vulnerabilities were uncovered within minutes of starting the fuzzing process. In 2019 researchers of KAIST found 51 vulnerabilities (of which 36 new) in practical LTE network architectures [42].

Solely giving many inputs to the device under test is not enough. We also need to monitor the SUT to check its response to that input. Some fuzzing techniques even use the response of the SUT to craft new inputs for further testing more intelligently. Figure 2.7 shows a diagram of the general fuzzing process. In the first step, we make sure the devices and pieces of software are ready. Then we enter the fuzzing loop. Here we iterate over crafting new inputs, handing them to the SUT, monitoring the SUT, and analyzing its behavior. If needed, this analysis can be fed back to the crafting stage. The component that analyzes if a bug was triggered is sometimes referred to as the bug oracle. When to exit this loop depends on the goal of the fuzzer and the type of fuzzer. For example, a brute-force fuzzer may exit when all possible inputs have been tested. On the other hand, an attacker may want to stop the fuzzing loop whenever one vulnerability is found. After the loop is finished, we can further analyze the fuzzer's findings and examine to root cause of certain anomalous behavior.

Figure 2.8 we took from [44] shows the number of publications in reputable journals regarding fuzzing over time. It proves a healthy interest in the topic, especially since 2010. There are several reasons why fuzzing has gained this popularity. First of all, the concept is relatively simple. Attackers, as well as defenders, can start fuzzing without too much expertise, especially since many open-source options are widely available. Secondly, the testing can be fully automated. Once an automated platform is set up, it can also be used for other similar systems. Tests can be run in the background, while the users' attention can be aimed at finding the root cause of bugs and mitigating them. Alternative methods for bug discovery, like detailed code inspection, are often much more involved and time-consuming. Thirdly, as we have already shown above, fuzzing has proven its effectiveness [45]. While it started as a technique to find software bugs, the concept can be applied beyond pure software testing, as we will show below. These reasons make fuzzing a popular testing technique with a low entry barrier. However, we have to remark that fuzzing gives no guarantees. Even if the fuzzing process does not uncover any bugs,

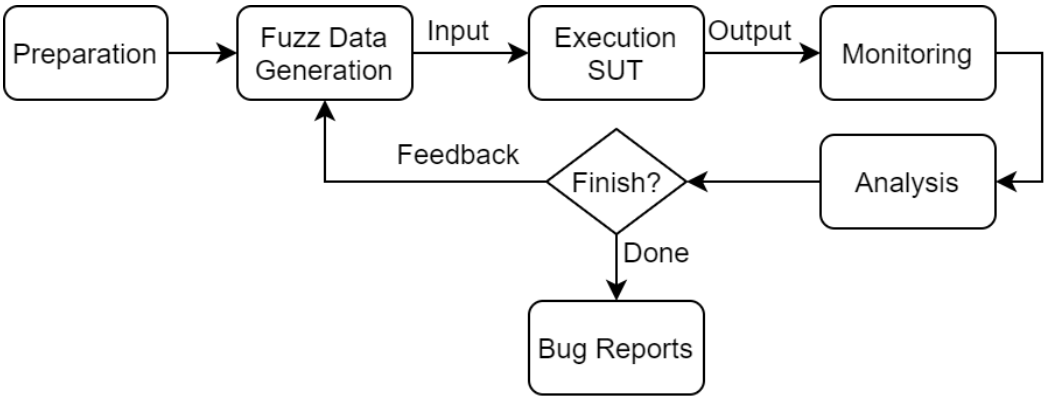

FIGURE 2.7: A graphical representation of the fuzzing process. [43]

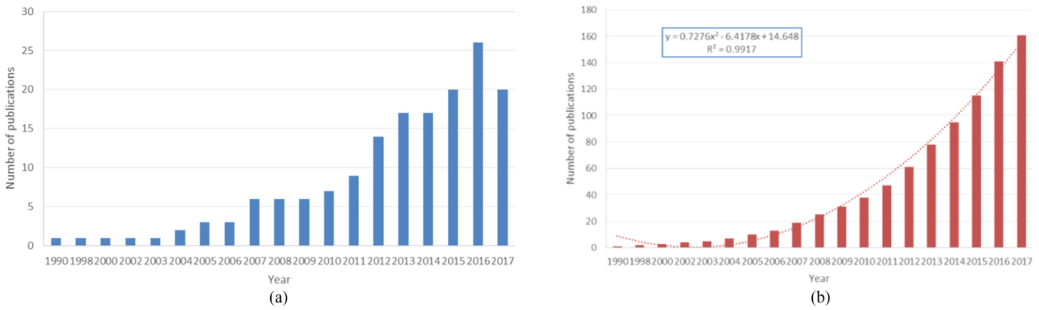

FIGURE 2.8: Fuzzing papers published between January 1, 1990 and June 30, 2017. (a) Number of publications per year. (b) Cumulative number of publications per year. [44]

there is still no guarantee that there are none present in the system.

2.2.2 Fuzzer Characteristics

Years of research into fuzzing have led to various types of different fuzzers. In the current climate, a fuzzer can have many different characteristics and start from a different perspective on the system under test. This section aims to give an overview of possible features and assumptions a fuzzer can hold. Most of this section is based on the seminal work of Manes et al. [45] and Eceiza et al. [43], who have made comprehensive surveys on this topic.

First of all, fuzzers can be categorized based on the prior knowledge the fuzzer has about the system. If a fuzzer does not employ any internal logic of the SUT, it is called a black-box fuzzer. If instead, a fuzzer has full knowledge of the SUT's internals, e.g. the source code, it is denominated as a white-box fuzzer. Since source code is often not accessible when fuzzing third-party software or fuzzing devices, there are also grey-box fuzzers. These do not need the source code but obtain some information about the SUT's internal working via extra instrumentation. Code coverage, the amount of code being executed in a fuzz test, is an example of such instrumentation. Most modern fuzzers are

often grey-box fuzzers. They combine the realistic view of not being able to access the source code with the benefits of the increased effectiveness of white-box fuzzers.

The instrumentation used by grey-box and white-box fuzzers can be obtained statically or dynamically. Static instrumentation often involves analyzing the source code or binary code. It has the benefit that this analysis can be done in a preprocessing stage. Hence, it does not add overhead during the fuzzing loop. Dynamic analysis involves studying the SUT's behavior at runtime. This adds overhead in the fuzzing loop, slowing down the fuzzer's speed, but gives a more complete view of what the SUT is doing during the tests. Of course, a combination of both static and dynamic instrumentation can also be used.

A vital characteristic of a fuzzer is how it constructs the inputs for the SUT. We identify two categories: mutation-based fuzzers and generation-based fuzzers. The first class, mutation-based fuzzers, starts from a set of valid input messages, called a seed. The inputs are then modified by mutations, creating new possible seeds. Common mutations include bit flips, arithmetic additions with a small number, byte permutations, byte deletions, byte insertions, and more. Mutation-based fuzzers have two decisions: which mutation to apply and which mutated inputs should be added to the seed. Appending all mutated inputs to the seed would grow the seed continuously, reducing speeds. The reducing speed is the most considerable drawback of mutation-based fuzzers. Mutation-based fuzzers are best applicable to systems with complex inputs. The fuzzer does not need to learn the sometimes complicated syntax of input messages and can modify specific fields of the input without knowing their meaning. Generation-based fuzzers, on the other hand, do need to learn the syntax of input messages. Starting from a set of rules about the input format and protocol, they generate new inputs from scratch. They are often suited for applications where a mutation in the input creates a high probability of the new input disobeying strict syntax rules. By following the input specification, generation-based fuzzers avoid a large portion of their inputs being rejected. The biggest drawback is that it may prove to be challenging to pour a complex input specification into a set of rules that the fuzzer understands.

A fuzzer's intelligence is defined as the ability of a fuzzer to take the SUT's response into account when generating new inputs. Dumb fuzzers do not consider any feedback from the SUT to generate new inputs. They are often quite simple, fast, and fit best with a black-box approach. In contrast, smart fuzzers use feedback from the SUT to craft new inputs more intelligently. By looking at how a SUT reacts to a previously fuzzed input, smart fuzzers try to learn patterns that can improve the effectiveness of new inputs. Smart fuzzers are often slower than their dumb counterparts, but their crafted inputs are often more effective in finding bugs or vulnerabilities. They are more complex and can again be categorized by the type(s) of feedback they use. Extra complexity arises for smart fuzzers as they need to schedule and prioritize their goals. Woo et al. debate the exploration vs exploitation trade-off [46]. A fuzzer can invest time exploring the SUT and its behavior, thereby increasing the information future fuzz input generations can rely on. It can also decide to invest more time in trying to exploit bugs and vulnerabilities in the SUT with the current information it holds. This trade-off requires a robust scheduling algorithm to find the balance.

2.2.3 Common Fuzzing Techniques

Next, we would like to outline some techniques that can be used to create a fuzzer. These techniques are not real characteristics but are more of a classification of fuzzer and instrumentation design choices found in open-source fuzzers.

The first and oldest technique is random mutations. Here the fuzzer considers no knowledge about the SUT or the input syntax and solely applies random mutations to a seed. While being very simple, this technique also has low performance, with very few inputs finding bugs. The first improvement was the technique of grammar representation. Here we still do not consider any knowledge about the SUT during the fuzzing, but we add grammar rules of the input to the fuzzer. This technique will result in fewer rejected messages as the fuzzed input is more likely to make sense to the SUT. An example of this would be to add the rules of a communication protocol to the fuzzer's input generation when testing a network stack.

These techniques are considered dumb as they do not use any device feedback. The first intelligent technique we discuss is dynamic taint analysis. Here, the fuzzer observes the effect user input has on an SUT's computations and variables. Mapping how the user input affects the SUT's execution can be leveraged to create new inputs that target specific parts of the SUT's code [47]. The drawback of this method is over-tainting and under-tainting. The former means that too many variables are marked as tainted, and analogous, the latter implies that too few are marked as tainted. These problems cut the performance of dynamic taint analysis.

A second popular technique is guided coverage. Being similar to taint analysis, it uses program instrumentation to keep track of the execution path of an SUT. The main goal is to increase code coverage with a minimal number of test cases. Increasing code coverage implies hitting corner cases of the execution, a part of the implementation that is prone to bugs [48]. The challenge here is the need for robust instrumentation that can trace the SUT's execution path.

The most modern propositions revolve around the addition of machine learning in input generation. Machine learning is an active field of research that has found its way into fuzzing already in 2018. Any phase of the fuzzing process we discussed in Figure 2.7 can, in fact, be supported by machine learning. Wang et al. created a review on the use of machine learning in fuzzing and compared the performance of these techniques [49]. We will only elaborate on the use of genetic algorithms in test case generation, as this is the most relevant in this thesis. A genetic algorithm is a search-based strategy for solving intricate optimization problems. The idea is to simulate the evolution of nature. A population of initial test cases is spawned, where each test case is slightly different. Based on a fitness function, the test cases are evaluated. By combining good test cases, eliminating the bad ones, and adding some additional mutations, we hope to create even better results [50] [51]. This approach is iterated to find more promising fuzz inputs continuously. We remark that this is a very simplified view of genetic algorithms, but it outlines the central idea.

Finally, the way fuzzers obtain feedback from the SUT can also classify them. We will not dive deeper into that topic here, as we have a whole chapter dedicated to the device feedback in our fuzzing platform (Chapter 5).

2.2.4 Embedded Device Fuzzing

Fuzzing in the early days was mainly focused on software pieces that run on the same machine as where the fuzzer was run, like the fuzzer for Unix. The fuzzing principles are, however, not restrictive to software. Recent papers have already shown how fuzzing can effectively detect bugs and vulnerabilities in systems of hardware-software co-design [52]. These systems often do not run on the same machine as the fuzzer and can contain functionality in both hardware and software. IoT devices and other embedded systems are perfect examples of this category.

This separation between the system under test and the fuzzer complicates things. Firstly, some of the advanced techniques we discussed above are impossible in this setting. They require instrumentation that cannot be obtained from an embedded device. Additionally, the separation complicates the execution stage of the fuzzing process. When fuzzing software, you can quite easily reboot the software with some input. When the SUT runs on a separate machine, this is no longer straightforward. A different way to give the input to the device and a different way to restart the piece under test is needed. This restart often requires a full reboot of the device, which can take a long time for simple embedded devices. Not only feeding input but also receiving output becomes more difficult. For example, one problem that arises is the distinction between a device crash or just a longer processing time. These types of problems are extensively discussed in chapter 5.

Two options emerge when debating the fuzzing of embedded systems: network fuzzing and emulation. In network fuzzing, we use the device's available interfaces for sending inputs and obtaining outputs. For example, in a Wi-Fi IoT device, Wi-Fi is used to provide the fuzzed inputs to the device. The same can be done for Bluetooth, LTE, LoRa, ... The device itself runs in its intended environment and gets unexpected inputs from the network. On the other hand, in emulation, we no longer run the device but emulate it in software. It involves downloading the firmware from the device, setting up an emulator that understands the machine code, and run that firmware in the emulator. The same source code is hence fuzzed but in a different environment. There are both advantages and disadvantages to both techniques. We will examine them and motivate our choice of using network fuzzing.

Let us start with the advantages of emulation. The most considerable benefit of emulation is removing the black-box view of the device. When emulating a device, the emulator has a clear view of the device's memory usage. The information inferred from this can be used to create advanced instrumentation. The memory accesses show the path in the software being taken and its internal variables. As discussed above, smart fuzzers use this information to improve test case efficacy.

A second significant benefit of using emulation is the increased test speed. In emulation, we run the firmware on much faster hardware than it was running on the original device. Although the firmware translation to the host's machine code adds overhead, the performance difference between a desktop processor, like a modern Intel Core i7, and an embedded device processor, like the ARM Cortex M series, easily makes up for this overhead. A more dominant fuzzing speed enhancement comes from the fact that there is no need to emulate everything. There is no reason why one must emulate a complete software stack. If the fuzzing target is, for instance, a higher layer of the LTE protocol, one

can choose only to emulate that specific layer or even only a part of that layer. You can go as far as target a specific function alone, such as a parser. The difficulty here lies in finding the function you want to target in the firmware blob. It requires intricate reverse engineering to locate the target function(s). Using emulation of baseband firmware, Maier et al. report fuzzing speed up to 15k tests per second [52]. They could isolate parsers from the higher LTE layers and fuzz them at incredible speed.

There are some drawbacks of emulation as well, which serve as benefits for network fuzzing. First of all, emulation requires the setup of an emulator. This emulator needs to understand the machine code of the embedded device to translate it correctly and offer a correct environment for the firmware to run in. Creating such an emulator is not a simple task, but open-source options are available. QEMU, for instance, can understand machine code for ARM, Intel, MIPS, and even more [53]. Hence, the need for an emulator does not form a significant roadblock for emulation.

The roadblock comes in the form of the emulator setup and configuration. Even though Maier et al. created a platform to fuzz baseband firmware, they still admit different target devices need a manual setup, coding, and reverse engineering [52]. This manual tweaking is our biggest argument for using network fuzzing instead. Recalling our goals for the fuzzing platform, we want it to work with a heterogeneous set of devices. Our platform should also be as automated as possible. The manual reverse-engineering required for every new device in emulation would significantly impose this goal. Hence, we sacrifice the test case speed to improve platform flexibility. This flexibility enables future research to test a wide variety of different IoT devices.

Emulation also suffers from the limitation that it cannot easily emulate hardware implementations. Especially for cellular IoT modems, the complex protocol is partially implemented in hardware to improve efficiency. The proprietary hardware implementation cannot be pulled from a device like firmware. Hence, emulation would not allow us to fuzz these parts of the implementation.

Lastly, network fuzzing also comes closer to a possible real-life attack. In a real setting, the attacker has to deal with transmission errors and strict timing requirements. By searching vulnerabilities with network fuzzing, we are sure the exposed vulnerabilities can be exploited in a real setting as well.

However, network fuzzing poses some new challenges in bug detection and fuzzer intelligence, which we need to overcome. We will discuss these challenges in depth in chapter 4 and chapter 5.

Chapter 3

Architecture

In this chapter, we will present how we built a fuzzing architecture. We will elaborate on the different components we used and the connection between them. We start with a high-level diagram of the architecture. Next, we dive into the details of each component, starting with the LTE network simulator. Following, we will describe our choice of the device under test and how we set it up. We conclude with an analysis of how we integrate fuzzing into this system and how we control the testing.

3.1 High-Level Description

Figure 3.1 shows a high-level description of the architecture we try to achieve. The fuzzer component will create the messages that are sent to the system under test, our UE. The messages are given to an LTE network simulator, which in turn sends the fuzzed messages as well as normal messages to the UE using a software defined radio (SDR). The platform is controlled by a test controller instructing the fuzzer, SUT, and network simulator. In the next sections, we zoom into each of these building blocks and translate them to actual hardware or software.

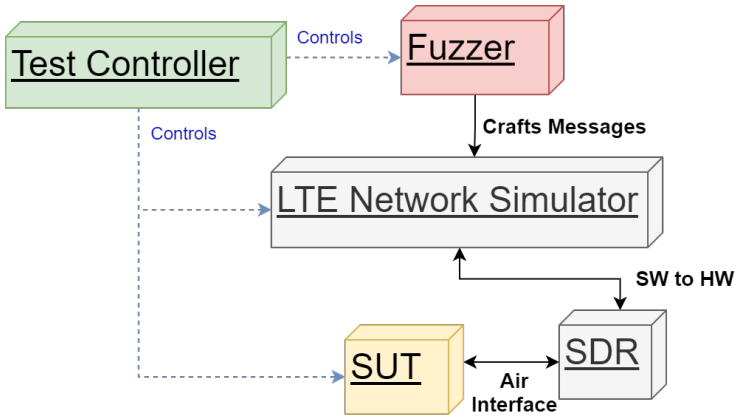

FIGURE 3.1: A high-level view on the platform architecture.

3.2 LTE Network Simulator

Before any device can be analyzed, an LTE base station simulator needs to be created. Due to the complexity of the LTE protocol, creating such a simulator from scratch is a time-consuming task. Luckily there are some open-source options available. Some of the best known options are OpenLTE [54], EURECOM's OpenAirInterface [55] and Software Radio Systems' srsLTE [56]. In this thesis, we opted for srsLTE. The software is very well written, and there is a clear mapping from the specification onto the code. This makes investigating the code simple when changes need to be made. Moreover, there has been previous collaboration between Keysight Labs and SRS, and hence more support for srsLTE when needed.

The srsLTE software, in reality, consists of three separate simulators which only share a common LTE library. These three simulators are the srsUE, srsENB, and srsEPC, which simulate the UE, E-NodeB, and Evolved Packet Core, respectively. We do not need the first one, srsUE, as this role will be filled by the device we want to analyze. We use the srsEPC to simulate the LTE core network, but most of the work focuses on the srsENB. We want to create a malicious base station for the fuzzing and hence focus on the Radio Access Network instead of the core network. Note that we use srsLTE version 20.04.2, which implements 3GPP Release 10.

Before simulating the LTE network with the open-source software, one needs to analyze the LTE protocol sufficiently. We were able to use a base station emulator from Keysight, also called a UXM, to see the protocol in action. The UXM is used to test new LTE-A radio equipment. It can test the device for performance and functionality but is not designed to test device security. Hence, we only used this equipment in the initial phase to comprehend the protocol in a practical setting. From the UXM logs, we retrieved a transcript of the messages sent and received by an LTE base station in regular operation. These proved themselves useful later on when configuring the SRS simulators. We were able to connect a normal phone as well as an IoT device to the UXM.

Since the cellular frequencies are heavily regulated, and to avoid a run-in with law enforcement, we use a wired connection between the UE and the base station. Most LTE modems can be easily modified to use a wired connection by replacing the u.fl antenna with an u.fl adaptor to a coaxial cable. Figure 3.2 shows an example of the u.fl connector on a chip. Figure 3.3 shows the connector we use to connect the UE to the base station. The wired channel has the added benefit of introducing fewer transmission errors than a wireless channel. This entails that we can fuzz the higher layers of the protocol stack without much detail for possible retransmissions due to transmission faults.

The next step was replacing the UXM with the SRS software. The SRS software implements the base station in software but still needs a radio frequency front-end device in order to communicate to the UE. For this front-end we use the USRP B210 (Figure 3.4). This device can operate between 70 MHz and 6 GHz with a bandwidth of 56 MHz [57]. These specifications are enough for our purpose: most European LTE bands are between 900 MHz and 3 GHz [58], and LTE bandwidth is limited to 20 MHz. Installation of the USRP is effortless and only requires the installation of Ettus' UHD driver [59].

With the hardware and drivers ready, we only needed to tweak the SRS configuration slightly. This configuration was minimal. We had to check the local IP addresses, such

FIGURE 3.2: An example of the u.fl connector on a wireless modem. Source: www.quectel.com

FIGURE 3.3: An example of the u.fl adaptor to regular SMA connector. Source: www.amazon.com

FIGURE 3.4: The USRP B210 radio frequency front-end for the SRS simulators. Source: www.ettus.com

that the srsENB and srsEPC could talk to each other, and pick an uplink and downlink frequency of the RAN. To set these frequencies, we must set the 'E-UTRA Absolute Radio Frequency Channel Number' or EARFCN of the network. We set the downlink channel number on 3100 (2655 Mhz) and the uplink channel number on 21100 (2535 Mhz).

Lastly, we made sure to put some of the UE's information in the Home Subscriber Server of srsEPC. This information included, for example, the secret key and IMSI of the USIM card. Afterward, we could fully connect the UE to the LTE network simulator and even ping the public Internet (www.google.com) from the UE.

3.3 UE setup

3.3.1 Hardware

After setting up the LTE network simulator, we moved to selecting and installing our UE. As stated above, we initially worked with two UE's: a regular phone and an IoT LTE modem. The regular phone was used as a reference device during the setup of the LTE network simulators. For the actual device testing, we always used the IoT modem.

There is an abundance of LTE IoT modems available on the market. Large suppliers include MediaTek, Quectel, Thales, or Fibocom. For this thesis, we decided to go for a Quectel LTE modem. The reason for this was two-fold. Firstly, Quectel holds the largest market share in the cellular IoT market [60]. This implies that their products are very widely used. Hence, a vulnerability in one of their devices would impact a large number of deployed systems. Additionally, Quectel likely reuses components of one product in another one to save development costs. Therefore, a vulnerability in one of their products can also affect some of their other products, leading to an even greater impact of the vulnerability. Our second reason for picking the Quectel modem was of a more practical nature. Quectel produces some of its chips in mini PCIe format. This makes it much easier to quickly use the chip as we do not need to create a circuit board to mount the chip ourselves.

More specifically we chose the Quectel EC25 LTE modem (Figure 3.2). This modem is an LTE CAT 4 UE, meaning it can achieve up to 150 Mbps downlink speed and 50 Mbps uplink speed. The modem also supports older cellular standards and even includes GNSS support, although we did not use these features. The PCIe chip on its own is not functional. From the technical specification [61], we can get an idea of how to connect the 52 mini PCIe pins to make the modem functional. Figure 3.5 shows the pinout of the mini PCIe pins. In color, we also indicate the many interfaces exposed by the chip. In orange and red, we can see two serial interfaces. The red pins show the main serial interface, supporting baud rates of 9600bps, 19200bps, 38400bps, 57600bps, 115200bps, and 230400bps. The orange interface is still under development at the time of buying the chip. The blue pins show the USIM interface. Hence our USIM card has to be mounted off the EC25 chip itself. There is also a USB interface, I2C interface, and PCM interface for sending data to the EC25. None of these interfaces were used and we only use the main serial interface and, of course, the USIM interface. The leftover pins are either for connecting a power supply or for control and indicator signals.

Luckily there is no need to solder or own breakout board in order to mount the EC25 mini PCIe. A company called Sixfab produces hats for a Raspberry Pi, which allow mounting an LTE mini PCIe chip (Figure 3.6) [62]. A USIM socket is included on the hat, which is connected to the USIM interface of the mini PCIe. The hat forwards the main serial interface of the EC25 to the Raspberry Pi. It also allows connecting the USB interface with an external cable to a USB port of the Raspberry Pi. Lastly, some additional control is given to the Raspberry Pi. If we drive GPIO26 pin HIGH, we can switch the power of the hat off, which can be useful for forcing a reboot of the modem. Lastly, by pulling GPIO19 HIGH, we can also force the LTE modem in airplane mode. The remaining PCM and I2C interfaces are not connected.

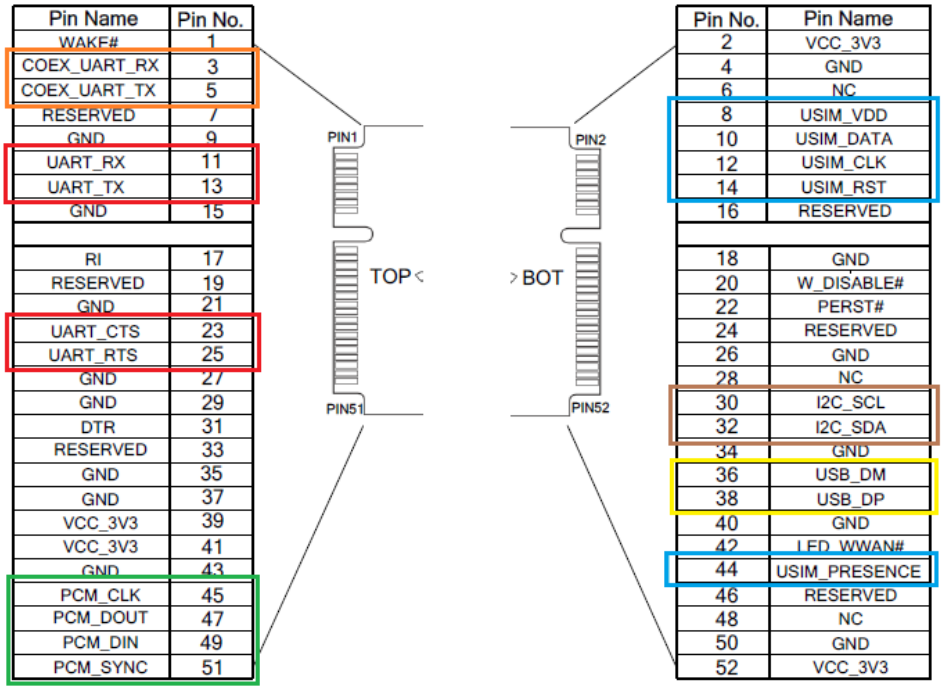

FIGURE 3.5: The pinout configuration of the EC25 mini PCIe pins. [61]

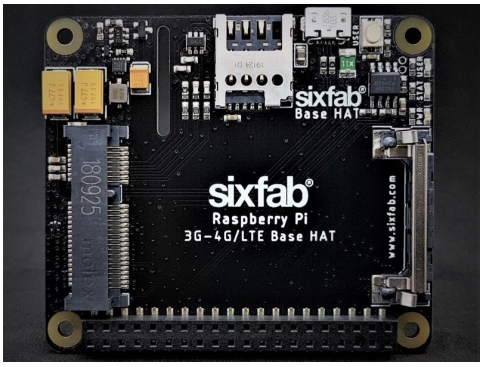

FIGURE 3.6: The Sixfab Raspberry Pi hat for mounting Mini PCIe LTE modems. Source: www.sixfab.com

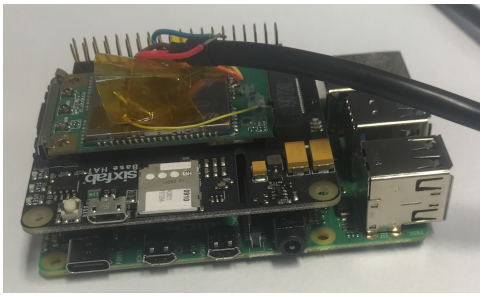

FIGURE 3.7: A picture of the full UE setup. The setup consists of a Raspberry Pi 4, Sixfab LTE hat, Quectel EC25, and Keysight SIM card.

The full hardware setup for our UE hence consists of 4 main components: a Raspberry Pi 4 with the Sixfab LTE hat holding our USIM and EC25 mini PCIe. Figure 3.7 shows a picture of our devices.

3.3.2 Interfacing

We already mentioned the interfaces the chip exposes for control over the selected LTE modem. Due to its easy access from the Raspberry Pi, we opted to use the main serial interface.

Over this serial interface, we can send AT commands to the EC25. AT commands are a common way of talking to wireless modems. 3GPP has created a standardized set of AT commands for wireless modems [63]. The general ideas of the command set originates from the Hayes command set. There are four types of commands: read, write, execute and test commands. The read and write commands read and write parameters on the modem, respectively. The execute commands are used to perform actions with the modem for which no parameters are required or are set up separately in other write commands. Finally, the test commands give information about a specific command itself, somewhat like "-h" on a Linux command. The syntax for a read command is AT+<CMD>?. A write command is given as AT+<CMD>=<val>. An execute command is simply AT+<CMD>. Finally, a test command has the following syntax: AT+<CMD>=?

The AT commands for the Quectel modem allow a user to control the serial interface, get status info, control network connection, control calls, control SMS, control audio, and more. A full overview of possible AT commands can be found in [64]. For our purposes, we mostly use the following commands:

AT Checks if AT interface is present. Simply outputs "OK" if the command is correctly received.

AT+CFUN Allows to set the modem's functionality and reset the system.

AT+COPS Allows to scan for cellular networks, connect to a specific network, or disconnect from a network.

AT+QPING Allows to ping any host using the cellular network as a gateway to the internet.

AT+CMEE Sets error logging verbosity.

On the Raspberry Pi, we use a program called *minicom* to send the AT commands over the serial interface. It is a simple terminal emulator that wraps the serial communication into a text-based send-receive module. It works splendidly to test certain AT commands and manually control the EC25 chip. However, the goal of our platform is to run automated fuzzing tests. For the automated tests, *minicom* does not work well. It expects to be called from an actual terminal window. In our automated tests, we simply want to send a command to the device without the hassle of first having to open a terminal window on the Raspberry Pi. Therefore we also created a small python script that can be called from the command line. The following command simply sends an AT command to the device with our python wrapper and prints the response on standard output:

python interface.py <device> <baudrate> <ATcommand> <Timeout>

A timeout is included to ensure we do not infinitely wait for a device response when the device is offline.

Aside from the main serial interface and the other interfaces exposed on the mini PCIe card, we noticed from the hardware design document that the EC25 chip provides a debug interface as well [65]. From this document, we learned we could access a Linux console and view log outputs. Without any further information from the document to go on, we decided this information could be beneficial for our future tests, especially if we would find a vulnerability and wanted to learn why the device could not handle the specific test case.

Unfortunately, as stated before, the debug serial interface is not exposed to the user via the mini PCI pins. As a result, we had to solder our own serial interface directly to the chip. For this, we picked the TTL-232RG-VREG1V8-WE, which is a 1.8V UART interface from FTDI [66]. From this chip, we soldered the transmit and receive pins as well as the ground pin to the Quectel EC25. Since this cable converts the UART communication to USB, we plugged the USB side into a USB port on the Raspberry Pi.

The last hurdle occurred when trying to access the Linux console of the chip. While our soldering proved solid, we still could not access the Linux files. We still needed to provide the chip with valid credentials. We reached out to Quectel, who provided us with a password for the root user account. Now, we were able to log into the Linux kernel on the EC25.

3.4 Fuzzer

With the setup we currently explained, one can test an LTE network in correct operation. However, for our purposes, we do not want to send standard messages to the UE but modified messages instead. Thus, we need to integrate a fuzzing component into our architecture. For this, we will need two parts: a new component generating the fuzzed messages and a deep dive into srsENB to send those messages to the device. The fuzzer itself is extensively discussed in Chapter 4. Here we explain the integration of this fuzzer in our platform.

3.4.1 srsENB Modification

First, let us discuss how we modified srsENB to send fuzzed messages instead of legitimate messages. Our first step here was identifying where inside the hundreds of thousands of lines of code the messages we wanted to fuzz are constructed and sent to the device. Despite the extremely large codebase, identifying interesting sections was not too difficult due to its clear structure. We quickly decided to focus our efforts on the RRC layer of the E-NodeB. We motivate this choice in Chapter 4.

The bulk of the RRC layer in srsENB is implemented in a single file called `./srsenb/src/stack/rrc/rrc.cc`. We quickly distinguished between the methods that build the SIB messages, RRC Connection Request, RRC Connection Reconfiguration, ... We were again presented with two possibilities: either we modify the building process of the RRC messages directly, or we let srsENB build the messages normally but modify the bytes right before handing the message over to the layer below. Here we went for the second option as this creates a more distinct separation between simulator and fuzzer. Furthermore, this

also gives us much more freedom in creating fuzzed messages. It makes it, for example, possible to create malformed messages and target fields or values that are reserved for future usage.

3.4.2 Interface

Next, we go into more detail on how we establish the communication between the fuzzer and srsENB.

In the beginning, we used the file system to store fuzzed messages. The messages were written to a specific file of the file system. When a UE tried to connect, that file was read in the SRS software, and the scripted message was sent. This method has some disadvantages. Firstly, it is quite slow. Access to the file system goes via the Linux operating system on which the SRS software runs. Secondly, it is difficult to tell from srsENB if the message is fresh. When the device tries to reconnect multiple times before the fuzzer has generated a new message, the SRS software will read the same message from the file multiple times. This could be fixed by including a timestamp in the file or by keeping track of the messages already seen in srsENB. Nevertheless, this requires quite some bookkeeping, adding extra delay. Lastly, this way of communicating between the fuzzer and srsENB makes it difficult for the fuzzer to be responsive. A responsive fuzzer would first read the message that would normally be sent and modify it based on its contents. This concept is difficult to implement via the file system because it would cause two processes to compete over access to the same file. The speed of the file system would also be even more of a bottleneck in this concept. Instead of a single read, we would need two writes and two reads through the file system in the middle of the connection establishment protocol. To not trigger a timeout at the UE, the communication to this responsive fuzzer should be quick.

A better solution came to light when discussing with another team of Keysight. The team had researched the fastest way for two processes to communicate (Figure 3.8). From this, they learned that the fastest way was through shared memory. They used this knowledge to build a small communication module in C, which we were given access to. To overcome all detriments of the file system communication, we used this shared memory channel for communication between the SRS software and the fuzzer.

The idea behind the interface is relatively simple. At startup, a chunk of memory of a given size is allocated as shared memory for the two processes. To block the memory when one process uses it, mutexes are used. Mutexes are flags that indicate if a resource, in our case a chunk of shared memory, is used by another process or thread. If process 2 wants to use the resource, but process 1 holds the mutex, process 2 must wait for process 1 to finish its work and release the mutex. Then process 2 can capture the mutex and start using the resource. Mutexes thus ensure that a resource does not get corrupted by party B when party A has not finished its work.

Due to the simplicity of the shared memory chunk, some logic had to be added in order for it to become a solid communication channel. Since we knew the channel was only ever going to be used by two parties, we kept the protocol on the shared memory very elementary. We use a Type-Length-Value (TLV) encoding of the messages. The first byte indicates the length of the full packet. The second byte indicates the type of

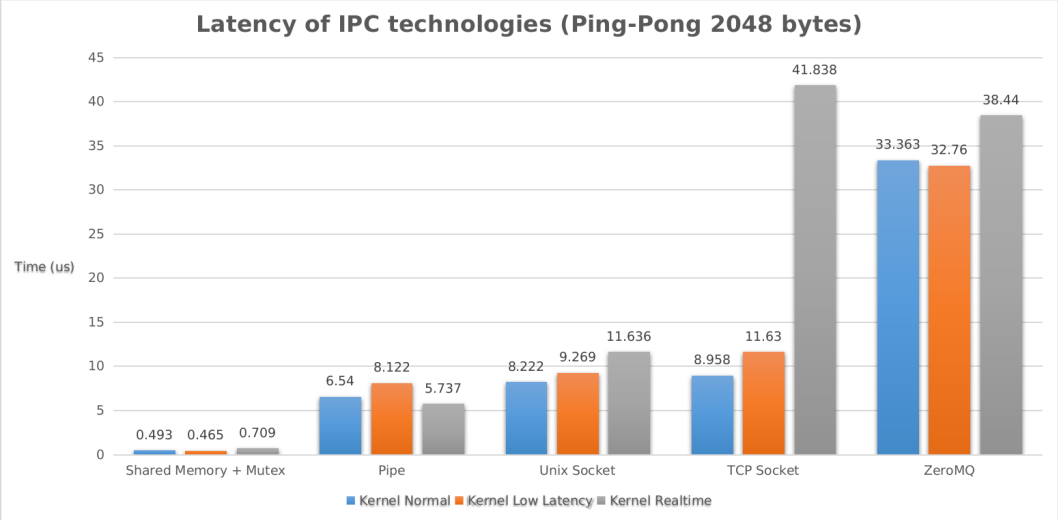

FIGURE 3.8: An investigation by a Keysight team on the latency of different inter process communication techniques.

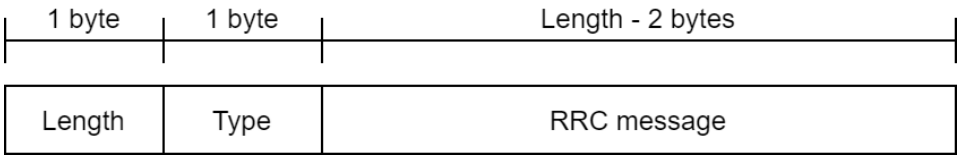

FIGURE 3.9: A depiction of the packets that are exchanged between the fuzzer and the SRS simulator.

the message. For example, 1 means an RRC Connection Setup message and 2 means a Security Mode Command message. The rest of the packet are the bytes of the RRC message itself. A depiction is given in Figure 3.9. This new communication channel allowed us to implement a more responsive fuzzer that generates messages on the fly instead of precomputing them. More details will be discussed in chapter 4.

3.5 Test Controller

This section elaborates on the final component of our architecture: the test controller. The responsibility of the controller is to automate the fuzzing process. It operates as the glue between the other components and ensures the correct flow of operations of each component to create an automated fuzzing platform. More precisely, the test controller will start up the network simulator and fuzzer, run a fuzzing test, collect the test result and run the next fuzzing case.

For our controller, we took a centralized approach. This means that our controller forms a single component that can be run on a completely separate machine. An alternative solution would be a distributed control of operations. There, each component is in

charge of the control of itself and a smooth sequential operation with the other components it communicates with. The advantage of this distributed approach is the autonomy of the individual components. When an error occurs at the centralized controller, the testing will have to be interrupted or entirely stopped, while a distributed controller does not share this risk. A centralized controller, on the other hand, benefits from being a single entity where reports are gathered and test status can be monitored. On top, it is also easier to track bugs and errors in the platform as this method contains fewer "moving parts". Finally, the centralized controller also reduces the risk of increasing complexity. Our platform already consists of many different pieces of code and interfaces. If each of these additionally gets a control function attached to them, we risk losing the structure of our platform and thus a much worse experience for future users. On account of these benefits, we selected a centralized controller.

To implement this controller, we use the open-source OpenTap software, developed by Keysight [67]. OpenTap is a test automation framework. In OpenTap, there are test steps (combining into a test plan), resources, and result listeners. The test step contains the functionality that performs a specific step in the automated test. Resources are, for example, devices under test or measurement equipment. They are opened at the beginning of the test plan and are closed at the end. Resources can be given to test steps to use them in their operation. Result listeners are used for combining test results into a single report. An example of such a result listener is a CSV file or a database.

Out of the box, OpenTap comes with some test steps that mainly concern themselves with test plan control flow: if-else branches, loops, delays, ... These steps are insufficient to run a full test. To extend the functionality, one can either download open-source packages that include new steps and resources or write one's own plugins. We used the latter to get a functioning test controller. These plugins show the benefit of using OpenTap. We can focus on writing the functionality for our test controller without having to worry about a user interface or status updates. The OpenTap software displays an excellent summary of the steps in a test plan, whether or not they succeeded, and even the timing of how long each step took. Figure 3.10 shows a screenshot of a running test plan in OpenTap. We wrote only the functionality of the test steps; the interface, visuals, and controls come included with OpenTap.

The plugins for OpenTap are written in C#. We created a resource for each of the main building blocks of our platform: the LTE network simulator, the UE, and the fuzzer. Since we did not want to restrict the controller to a single machine, we created each of these resources such that they are controlled over SSH. On opening the resources, an SSH session is created (using the open-source SSH.NET library [68]). Later, our test steps can send commands through this SSH session to control the components. We will not delve into the specific test steps we implemented, but the general functionality corresponds to the goals we stated at the beginning of this section: start, run a test case, collect results, and loop. Note that we tried to make the test steps as generic as possible, such that they can be reused on multiple locations in a test plan. For example, rather than creating a step that only disconnects the UE, we made a test step to send any AT command to the UE. The latter can be customized in the test plan to disconnect the UE or do something else.

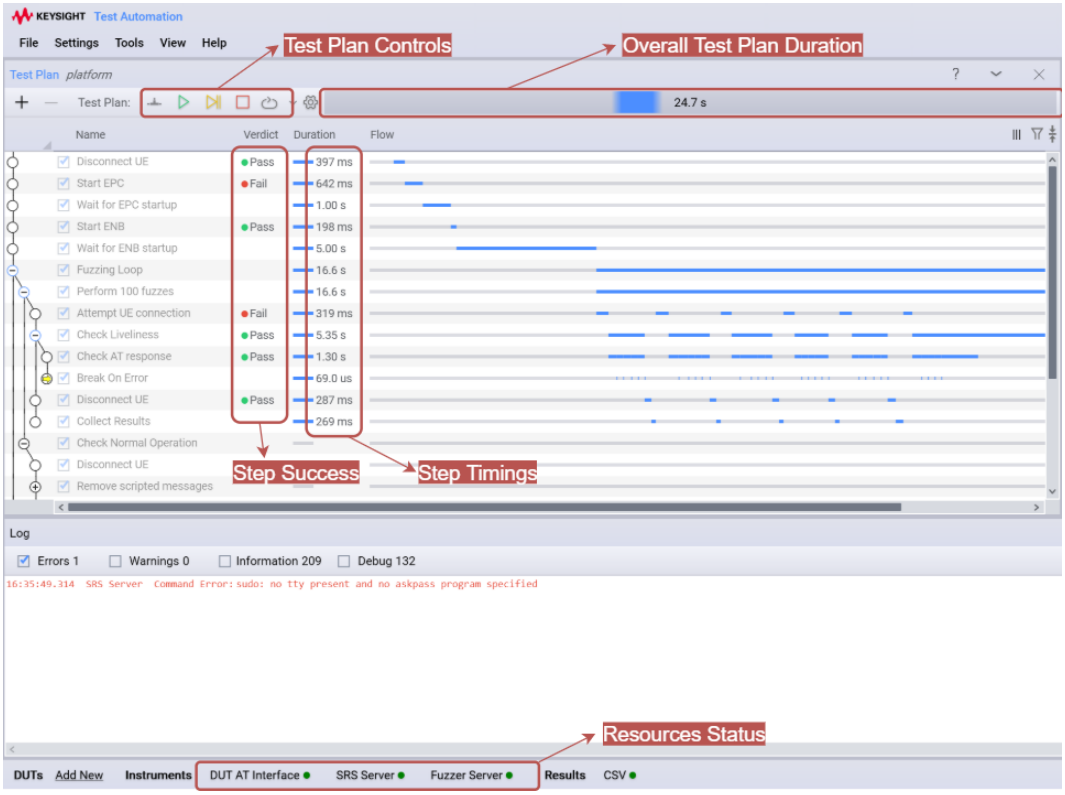

FIGURE 3.10: A screenshot of the OpenTap software while running a test. Some of the features that OpenTap offers are highlighted.

3.6 Complete Architecture Overview

In this section, we reiterate the overall architecture of our platform. Figure 3.11 shows a similar diagram like at the beginning of the chapter, but with the details discussed throughout this chapter.

The figure shows all components of the automated setup and how they interact with each other. The interfaces are displayed in green. The interface in red is the one we are going to fuzz. The blue dotted lines show which components are controlled by the OpenTap controller.

In short, the controller will start all software components and disable automatic connections in the UE. Next, we instruct the UE to connect to our LTE network simulator. In the RRC layer of srsENB, the fuzzer is contacted through the shared memory interface before the message is handed to the lower protocol layers. After the fuzzed message is transmitted to the UE, its behavior is monitored by the test controller. The controller starts the next fuzz test after sufficient feedback has been collected.

Note that the OpenTap controller does not access the minicom terminal connected to the debug interface of the EC25. We will use the debug interface manually to extract logs whenever we have identified a bug. Automating this extraction could, in theory, be done,

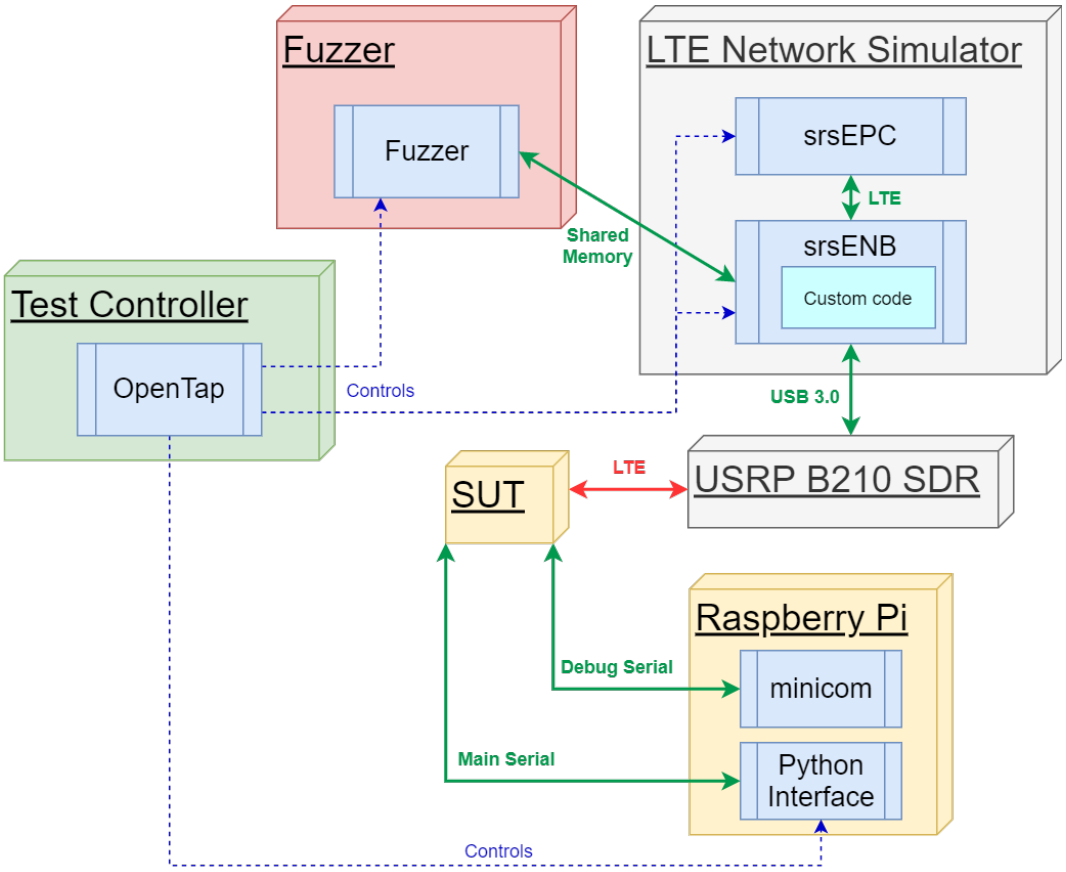

FIGURE 3.11: A diagram displaying our platform's architecture for automated fuzzing.

but it would require much tuning per device because each IoT device exposes different debug information. Since it requires some manual tuning and reverse-engineering to extract the information, it does not fit well in our automated platform.

As a system under test, we chose the Quectel EC25. However, this one could also be replaced by another compatible mini PCIe LTE modem, like the LE910C1 from Telit.

As a final remark, remember that some fuzzers use feedback from the device to create more valuable new fuzzing cases. This feedback is not explicitly shown in the diagram but will be handled by the test controller. In OpenTap, we collect the required feedback and turn this information over to the fuzzer accordingly. The following chapters will discuss the fuzzer component and the device feedback itself in more depth.

Chapter 4

The Fuzzer

In this chapter, we delve into the details of the fuzzer component. First we motivate our choices to target specific messages of the LTE protocol. Secondly, we will describe how we used an open-source off-the-shelf fuzzer in our architecture. We will discuss the obstacles with this decision and why we moved to the self-written WDissector Fuzzer.

4.1 Fuzzing LTE

Before explaining the integration of certain fuzzers in our architecture, we have to explain which messages we want to fuzz and at what layer of the LTE protocol.

In chapter 3 we already mentioned that we concentrate our efforts on the RRC layer of the E-NodeB. We quickly decided to focus our efforts there, because it resides in the upper layers of the LTE protocol. This it makes this layer more comprehensible than the lower layers. The fields set in the RRC layer have a clear connection to the setup of the network. Values in the fields of RRC layer messages affect the whole LTE stack. The RRC layer defines the parameters that set up the entire LTE cell. The layers below, like the MAC layer, concern themselves more with the physical transmission of data (error correction, acknowledgements, scheduling, ...).

The RRC layer also handles the setup of new connections, a moment of unauthenticated data transmissions. From an attacker point of view, unauthenticated traffic is the most promising attack vector. The attacker can modify or forge messages without the UE being able to distinguish them from legitimate messages. These arguments convinced us to focus our efforts on the RRC layer to get the most promising results.

More specifically, we are fuzzing the RRC Connection Setup message. This message is part of pre-authentication traffic. As we learned in chapter 2, a secure PDCP context is only enabled after the Security Mode Command message. The Identity and Authentication procedure messages come before this, but are already NAS layer integrity protected. This leaves a couple of interesting messages for an attacker on the RRC layer who does not have the security keys: the MIB, the SIBs, the RRC Connection Setup, and the RRC headers of the Identity and Authentication messages. We opted to concentrate on the RRC Connection Setup, but the techniques we used to fuzz this message are easily applicable to other messages.

4.2 American Fuzzy Lop

The open-source fuzzer we opted to use was the American Fuzzy Lop (AFL) fuzzer. Michal Zalewski originally developed ALF at Google, but it has now evolved into a well-known open-source fuzzer with many derivatives [69]. Some examples are AFLFast [70], AFLGo [71], AFL++ [72], or Angora [73]. We first give an overview of AFL, followed by a rundown of how we integrated it into our platform. We end with a discussion of the problems with this solution.

4.2.1 AFL Overview

AFL is often seen as a grey-box fuzzer. It does not need all the source code but uses instrumentation to guide the fuzzing process. The instrumentation is used to provide code coverage information to AFL. This instrumentation allows AFL to learn which paths in the code are triggered by specific test cases. This can mark anomalous behavior as well as condense the number of test cases. AFL summarizes different test cases that trigger the same behavior in one. Besides the instrumentation, AFL uses a genetic algorithm to identify bugs and generate new test cases.

The instrumentation AFL relies on can be obtained in two ways. If the source code is available, AFL requires the user to compile this code with a gcc or g++ wrapper, called afl-gcc or afl-g++. These wrappers will insert helper instructions in the code that AFL uses to track the code flow before compiling it with gcc or g++. These compilers are popular C/C++ code compilers, limiting the code that can be fuzzed with AFL. However, the source code is often not readily available to a user. In that case, AFL supports fuzzing through emulation. It uses the QEMU emulator software to run any binaries where the source code is not accessible. We already discussed how emulation provides instrumentation to the fuzzer by observing memory usage. The AFL developers do warn for 2 to 5 times performance drop due to this emulation. [69]

Using the instrumentation, AFL keeps track of tuples of the program's location before and after a branch. It does not memorize the entire path of branches taken but only stores the tuples of these locations. Along with the branch tuple, the number of times it is taken is stored as well. This number is stored as course buckets of 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+ times. New branches or a new bucket for the number of times a branch is taken indicate new behavior of the code. AFL uses this information to generate future test cases. The coarse buckets protect AFL against oversensitivity to the number of times a branch is taken. [69]

The process of how AFL generates new inputs can be summed up as a combination of blind mutations with an evolutionary seed selection. Figure 4.1 shows the idea behind AFL's input generation. AFL starts from a user-supplied seed. This seed can contain multiple valid inputs to the software under test. AFL loads these inputs into a queue and starts the first of multiple rounds. In each round, one test case is pulled from the queue. First, the code's behavior under that test case is monitored, and a table of branches and their respective buckets is created. Afterward, deterministic and non-deterministic mutations are applied to this input, and the code's behavior is monitored. When a mutated input results in a new branch or new bucket for that branch, AFL puts that input at the

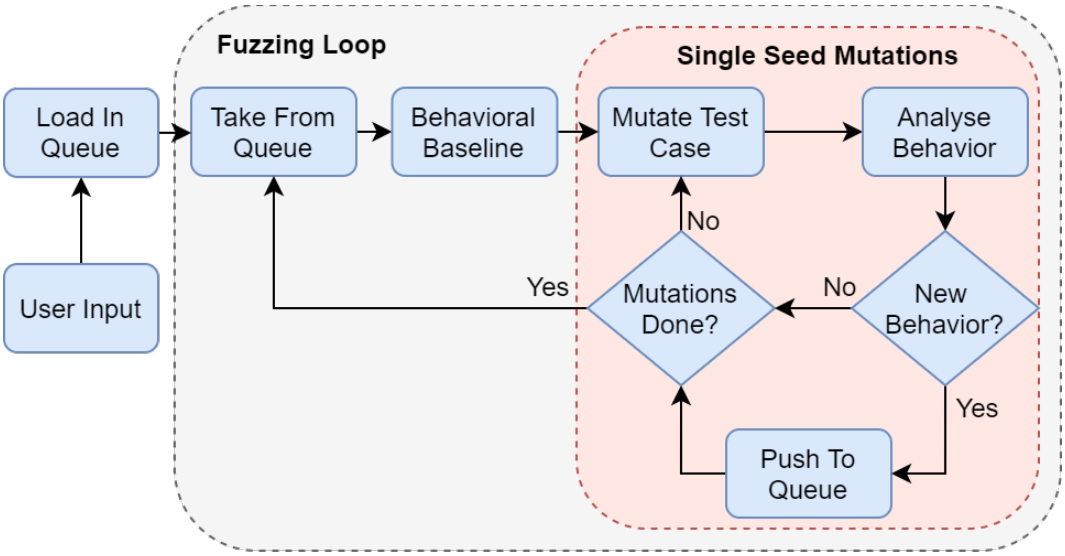

FIGURE 4.1: A simplified flow diagram of the AFL test case generation.

back of the queue. When AFL is done mutating the first input, it pulls a new input from the queue and starts a new round. [69]

The mutations applied to the inputs are a collection strategies that are known to be effective from previous fuzzing research. These include bit flips with various lengths, arithmetic with small integers, insertion of interesting integers (1, 0, INT_MAX, ...), splicing, insertions, deletions, ... The AFL developers call these mutations blind because they do not try to establish the effect of a specific mutation on the code coverage. The mutations themselves are rather dumb, but the intelligence comes from the evolutionary approach of selecting interesting mutation results as a new basis for more mutations. [69]

Additionally, AFL offers features that reduce the set of test cases in the queue. Test cases that have the same effect on the code are culled. This sort of test case minimization can help create a small set of tests that trigger various execution paths. In future releases of the code, this small set can quickly test new code with a diverse set of inputs that are likely to trigger many different executions. Lastly, AFL contains features that guide developers in finding and fixing a bug in the software whenever one is found by the fuzzing process. [69]

4.2.2 AFL Integration

Before we talk about how we integrated AFL into our architecture, we must briefly discuss the practicalities of AFL. Since AFL is built with software fuzzing in mind, it assumes it can easily reboot the system under test. Hence, for every new test case, AFL reboots the software and provides the test case either through the standard command line input or through a file. Thus, if the software expects inputs from another path, a wrapper needs to be built around it.

AFL understands three types of software exits: a return without error, a return with

error, and a crash. AFL also times the software execution, and if it exceeds a configurable timeout, it is killed by AFL itself.

By many design choices, it is clear AFL aims to fuzz software, but we want to use it to fuzz embedded systems via network fuzzing. Accordingly, We need to 'cheat' AFL a bit to make it function in our architecture. The first idea that comes to mind is to fuzz the srsENB software directly. We could modify the SRS code in such a way that it would read a specific file somewhere in the RRC layer before sending a message. However, this concept has some pitfalls. First of all, the startup of srsENB takes several seconds. This would drastically reduce test case speeds and cause much avoidable overhead in the fuzzing process. Secondly, AFL would try to find new paths and bugs in the srsENB code instead of in the UE, which is not our goal. AFL will start to guide the fuzzing to expose bugs in the SRS software, which is not what we are trying to find. Thirdly, srsENB is a large piece of software. This means that there will be many, mostly irrelevant, branches that AFL will track. This increases the memory requirements of AFL and reduces the test case generation speed, creating even more performance loss.

Fuzzing srsENB directly is not a suitable option for the integration. Instead, we decided to write a small C program that serves as a link between AFL and the rest of our architecture. Listing 4.1 shows the outline of this program in a C style format. The general idea of the program is to catch test cases from AFL, send them to the LTE network simulator, and mimic the UE's response to that test case. AFL will now perform its fuzzing on this linking program. Due to its simplicity, reboots of this program are practically instant, and srsENB can keep running in a different process.

The program starts by reading the current test case from the standard input. After some input validation, which we will discuss more in depth in the next paragraph, the test case is sent to srsENB over the shared memory interface. Note that since SRS expects a responsive fuzzer, we first wait to receive a message containing the correct information before putting our test case on the shared memory. After transmission over the shared memory interface, srsENB sends the message to the UE, and the OpenTap controller monitors the device's status. In our linking program, we wait for an alert from OpenTap that the test run is complete. When a test is entirely run, OpenTap informs our linking program of the UE's status. Depending on this status, we terminate our program normally or simulate a crash. The idea is that when the UE has crashed, we give this information to AFL by raising the SIGABRT signal leading to a crash of our linking program.

The input validation step comes from observing some trial runs of AFL on a sample piece of software. When we left AFL running for an hour, we noticed it started to give very long inputs to the sample software. AFL kept on inserting new bytes in the input, leading to test cases of hundreds of bytes. We wanted to avoid these large inputs as they do not make sense as an RRC layer message. At all times, we want to avoid introducing buffer overflows in the srsENB software itself that would lead to a crash of the incorrect component. We try to teach AFL to keep the length of its test case between 5 and 40 bytes. This is sufficient to test many RRC messages without the risk of triggering some bugs in srsENB itself. The linking program quickly rejects any message length outside these bounds.

```
#include<stdio>
#include<signal.h>

int main() {
    // Load test case from stdin
    uint8_t testCase[] = readFromStdin();

    // Validate input
    if (!checkInputValid(testCase))
      return 1;   // Return with error

    // Send test case to SRS
    transmitToSRS(testCase);

    // Wait for controller to signal test done
    while (!UEready())
      wait();

    // Obtain test case result from controller
    bool UEcrashed = getUEstate();

    // Based on how UE handled the test, end program
    if (UEcrashed)
        raise(SIGABRT); // Imitates software crash
    else
        return 0;       // Finish without error
}
```

LISTING 4.1: The outline of the link program between AFL and our architecture.

This concept of integrating AFL with our architecture worked, and we were able to run some long fuzzing tests with over 7k unique RRC Connection Setup messages tested. From this test little results for security research were obtained. We encountered a few reports of crashes, after which the UE still operated fine. We assumed that no real crash had occurred as the device still functioned correctly after these reports. While the results were not practical for security research, the long test did highlight some problems with the usage of AFL as our fuzzer.

4.2.3 AFL Obstacles

After testing our platform with the AFL integration, we identified three obstacles with the current solution. Two problems are of a more fundamental nature, and one is of a more practical nature.

Let's start with the practical impediment. In its intended operation, AFL acts as the controller of the fuzzing process. It generates test cases, executes them, and monitors the SUT. In our setup, we are only interested in the test case generation part of AFL. The other parts are controlled by the OpenTap controller, as it can handle the platform complexity better. This conflict in fuzzing loop control has led to our get-around solution of the linking program. The linking program is fully controlled by AFL and extracts the part of AFL that we want to use. Furthermore, because AFL under regular operation controls the fuzzing, it does not neatly expose status updates that can be imported into OpenTap.

While the practical obstacle causes some inconvenience, the two fundamental problems cause poor fuzzing performance. The first fundamental problem is the fuzzer's intelligence. We essentially do not use most of the intelligence that AFL provides. As we explained before, AFL's intelligence comes from the evolutionary approach to selecting the successive inputs for the seed. This selection is based on the code coverage instrumentation in the software. In our integration, however, the instrumentation sits in the linking program. This tells nothing about the actual execution of the device under test. Hence most of the intelligence of AFL is lost in our use case. We only keep the mutations that AFL executes on a message, which are not intelligently driven. While AFL provides a lot of intelligence in software fuzzing, for embedded device network fuzzing, it boils down to a simple random mutation fuzzer.

The second fundamental problem comes from the complexity of the LTE protocol. While AFL may be able to learn simple data formats, the complexity of LTE protocol messages is way beyond that. This results in many of the messages being very malformed. It is likely that the device thus quickly rejects these messages early on. Subtle, targeted malformations in a message may trigger corner cases in the device firmware, but the messages that AFL generates result from fully random malformations. They are likely less successful in triggering a bug than more targeted malformations that start from some knowledge of the protocol.

In conclusion, AFL is well known and effective fuzzer for software fuzzing. When it comes to embedded device fuzzing, AFL is not well suited. When performing network fuzzing of embedded devices, AFL is limited in understanding the network protocol being used and loses a significant portion of its intelligence due to the lack of accessible instrumentation.

4.3 WDissector Fuzzer

In this section, we discuss an improved design that solves two of AFL's problems: the practical problem of the controller and the lack of protocol awareness in AFL. This design is called the WDissector Fuzzer.

4.3.1 Fuzzer concepts

The WDissector fuzzer is designed around a packet dissector called wdissector. This C library is a part of a proprietary Keysight product, but was made accessible for this thesis. While being proprietary, wdissector also heavily uses the source code of Wireshark [74].

Wireshark is an open-source packet sniffing tool. It can capture network packets and dissect them, meaning decoding the different protocol layers from the bytes. For example, the dissection of an LTE RRC Connection Request packet shows the different MAC headers, RLC headers, the RRC packet, and the meaning of each set of bits is revealed. In essence, packet dissection is the practice of decoding bytes and linking bits with their respective interpretation. A set of bits that are linked to one interpretation is called a field. The wdissector library uses Wireshark's packet dissection and exposes it to other C/C++ code. It adds functionality to easily select a specific protocol, navigate packets, and extract

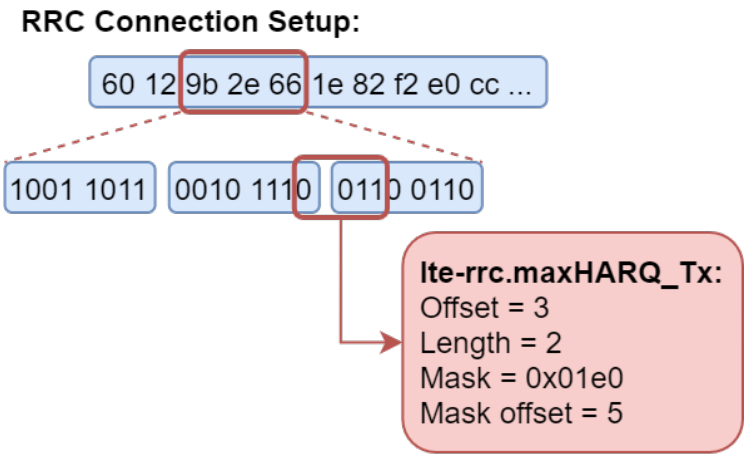

FIGURE 4.2: An example of how fields are identified with wdissector.

specific fields by name. Because it uses the Wireshark code, the naming convention of the protocols and fields follows Wireshark's.

A field of a packet is represented by four variables: the offset, the length, the mask, and the mask offset. Before we explain each of these, note that fields are, in general, not aligned to byte boundaries. Especially in LTE, a field is a collection of bits and often crosses byte boundaries. This makes it a bit more complex to modify fields in C code, where we often work with byte-aligned values. Figure 4.2 shows an example how the four variables uniquely identify a field. The offset indicates the offset of the field's leftmost byte in the packet. The length tells over how many bytes the field spans. The mask contains 1's to indicate which bits of these bytes actually make the field. Finally, the mask offset indicates the amount of 0's before the sequence of 1's.

With wdissector, we are able to target specific fields of an LTE packet. However, we have to add some features to create a complete input generation component. First of all, we added mutation functions that take a field as input and apply some mutation to it. We implemented four mutators: bit flips, zero bits, full bits, and exhaustive search. The bit flips mutator flips one or more bits in the field. The zero and full bits mutators replace parts of the field by zeros or ones, respectively. The exhaustive search mutator iterates over every possible value of a field. We apply these mutations responsive. This means that we do not use a static seed as a base to apply mutations like in AFL, but we apply the mutations to the message that srsENB gives to the fuzzer. In practice, this makes no real difference, as most messages from SRS are the same for every test run.

We did not add any intelligence through instrumentation to our fuzzer. Hence, which mutations to apply and which fields to target could be formulated at compile time. At compile time, we instruct the code to fuzz certain fields and apply certain mutations to those fields. We call this set of planned mutations the *fuzzing strategy*. At the start of the program, the strategy is loaded. In the fuzzing loop, the WDissector Fuzzer proceeds to work through this strategy in a deterministic fashion. When the strategy is exhausted, that is when all planned mutations have been applied to all fields of the strategy, the

fuzzer reverts to random bit flips. These random bit flips do not target specific fields but can occur in any bit of the message. This serves as a fallback option to ensure that we keep testing useful inputs in long-running tests. However, the primary goal is the fuzzing strategy and not these random bit flips. Listing 4.2 shows the simplified execution of our fuzzer.

Lastly, we want to remark that our new fuzzer not only solved two of the problems we had with AFL but it also shows the flexibility of our platform. Through the design of our platform we kept modularity in mind. A high modularity entails we can replace any of the building blocks and optimize them separately. We have proven that we could achieve this modularity using the shared memory interface with srsENB. The design of this new fuzzer took the most of our time, while integration with the platform was seamless. Other open-source or custom-made fuzzers can be integrated into our platform in similar fashion. Some additional code may need to be written to bridge that fuzzer's technicalities to the shared memory interface, but no further actions to the other building blocks are needed. Any fuzzer that respects the communication protocol on the shared memory interface is supported by our platform.

```
int main() {
    // Load strategies
    struct fuzzingStrategy strategy = buildFuzzingStrategy();

    while(!strategy.done) {
        // Get input from SRS
        uint8_t message[SIZE];
        int nbReceived = receiveFromSRS(message, SIZE);

        // Dissect packet
        packet_register_field(strategy.field);
        struct field_info info = packet_dissect(message, nbReceived);

        // Mutate field
        mutate(message, nbReceived, info, strategy.mutator);

        // Return to SRS
        transmitToSRS(message, nbReceived);

        // Update strategy
        updateFuzzingStrategy();
    }

    // Move to random bit flips
    loopRandomBitFlips();
}
```

LISTING 4.2: The outline of our WDissector Fuzzer.

4.3.2 Targeted Fuzzing

With the WDissector Fuzzer, we are able to target specific fields of an LTE packet for fuzzing. In section 4.1 we discussed on which messages we focus in our fuzz testing. Now,

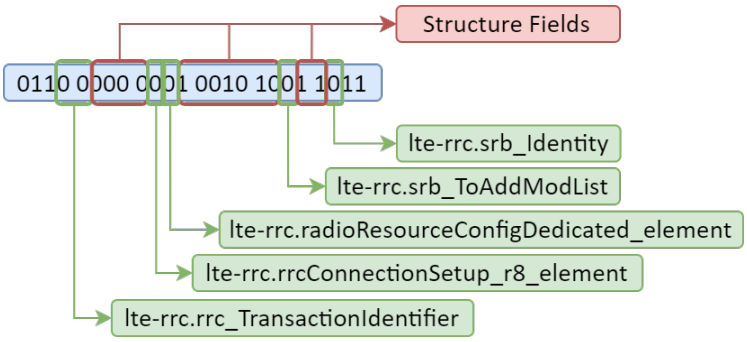

FIGURE 4.3: An example of how many value and structured fields there are in an LTE message. The bits shown are the beginning of an RCC Connection Setup message.

we will discuss the fields of these messages we target. Information about the specific fields were aqcuired using Wireshark and the technical specification of the RRC layer [75].

At first, we target what we can call the **value fields** of the RRC Connection Setup message. These fields contain specific values that configure some parameters of the protocol. An example of such a field is the *p0-UE-PUSCH* field, which sets the power output of the UE in the PUSCH physical channel. Fuzzing these fields tests whether the UE correctly handles all possible values of those fields. It can also test if the UE correctly deals with some illegal values. For example, if the value of a particular parameter cannot exceed 100, but the field consists of 3 bits, then an extra 27 illegal values need to be managed correctly. These fields are easy to target with wdissector because they have unique names in Wireshark.

The value fields make only a portion of an RRC Connection Setup message. We illustrate this in Figure 4.3 There are also bits present that inform a decoder about the structure of the packet itself. For example, which types of optional data structures the message contains or the type of information the packet holds. Fuzzing these **structure fields** often leads to packet malformation. It can be interesting to check if the device correctly rejects these malformations. Targeting the structure fields is more complex. Wireshark does not have unique names for these bits. Most of these fields are mapped to generic names like *per.enum_index*, *per.choice_index*, or *per.optional_field_bit*. This makes it a bit harder in practice to target one specific structure field, but we were able to get around the issue.

Lastly, LTE messages contain some redundancy. There are spare bits left in the specification to allow protocol improvements and expansions. These leftover bits can be used in future releases to add functionality that was not considered earlier. To accommodate the concurrent operation of different releases at once, a UE should ignore the spare bits of his release. Then, newer devices can use the added functionality, while older devices can still operate simultaneously. In practice, however, it can occur that the UE does not handle the use of these spare bits well, causing it to refuse connections it should accept. For example, with the release of LTE Cat-M, Keysight found this problem in the devices of some vendors. With LTE Cat-M, five of the ten spare bits of the MIB are used to carry extra scheduling information. While non Cat-M devices should ignore the usage of these bits,

it turned out some of them refused a connection with the cell altogether. These types of errors are interesting to fuzz for security purposes as well as for future-proofing of devices.

We would like to make a distinction between spare bits that are *reserved for future usage* and *spare values* of some data structures. Both interpretations are called "spare" in the LTE specification. The bits we discussed in the previous paragraph are an example of bits that are spared for future usages. They often sit at the end of a particular data structure. A UE should ignore these bits. On the other hand, spare values should not be ignored. An example of such spare values are bits that indicate the type of message. The specification often defines such fields x bits wide, but only fixes the meaning of a few of the 2^x possible values. The other possible values are then denoted as spares. If we hit these spares, we expect the UE to reject the message, as the content of the message becomes undefined. There are a lot of spare values in the RRC specification, but only very few reserved for future usages bits. We could only find reserved for future usage bits in the MIB, RRC Connection Request, and RRC Connection Reestablishment Request messages. Therefore, our experimentation targets the spare values of the RRC Connection Setup message, but no reserved for future usage bits yet.

4.4 Instrumentation Improvements

We discussed how we introduced protocol awareness into our fuzzer and how we leverage on this to target specific fields of an LTE message. This subsection will briefly discuss an option to solve our last problem with AFL, the lack of intelligence. We did not implement or test this solution but give it as a major suggestion for future work.

The root of our problem is that in the current setup, we have no access to the device's software. Most fuzzer base their instrumentation on some knowledge of the source code, be it the source code itself or the memory accesses during execution. Both are not available in network fuzzing. What is readily available are side channels, and more specifically, power consumption. We suggest the use of power analysis to characterize device behavior. It is well known in the security field that power consumption is closely related to the device's code execution. This has led to many security vulnerabilities where secret keys were extracted from one or more power traces [76].

If we can characterize the UE's behavior using its power consumption, we can use the ideas of AFL to create an intelligent fuzzer. Where AFL looks at the branching behavior of the code, we will define irregular behavior based on power consumption. When the power consumption of a mutated test case differs a lot from the power consumption of the original test case, this mutated test case is added to the queue. While the idea is simple, the challenge will lie in analyzing the power consumption and detecting unusual behavior with the correct sensitivity. The use of machine learning or pattern matching comes to mind to aid in this task, but further research will have to validate this statement.

Although the idea of using power consumption for instrumentation in network fuzzing is new, there already exists some related work. Philip Sperl and Konstantin Böttinger have shown how power consumption can extract features of the code execution and control flow [77]. They can detect when the code transitions into a new basic block (a piece of

code without branches). The execution of a previously not seen basic block is precisely the information that a fuzzer can use in its evolutionary design.

One considerable challenge we can already foresee with this platform extension is the training of the machine learning models. To keep our platform usable for any UE, machine learning training should be as automated as possible as an initial phase of the platform. One can try to obtain a general machine learning model that works regardless of the UE. For example, a model that takes both the baseline power consumption and last measured power consumption as input. Alternatively, one could add a preprocessing phase in the fuzzer, where the model is tweaked to the specific UE under test.

We see the further development of power analysis for fuzzing instrumentation as the most promising direction for future work.

4.5 Summary

In summary, we focus our efforts on fuzzing the RRC Connection Setup message. This message sets up parameters of the full LTE stack and is still unauthenticated in the connection establishment procedure. We integrated two different fuzzers. The off-the-shelf AFL fuzzer was integrated using a linking script. The linking script was needed to overcome the technicalities of AFL, which are aimed at software fuzzing. Fuzzing using AFL showed us its limits regarding protocol awareness and the inability to use code coverage instrumentation. With our custom-made WDissector Fuzzer, we introduced protocol understanding through packet dissection. We could target specific fields of the RRC Connection Setup message. Both structural bits as well as value fields can be fuzzed. To increase the intelligence of this fuzzer, we propose the use of power analysis as instrumentation for code coverage information.

Chapter 5

Device Feedback

This chapter explains the details of the device feedback loop in our architecture. We will discuss the device feedback used as our bug oracle, the way of detecting anomalous behavior of the SUT. First, we explain the different types of embedded devices and which type our SUT falls under. Next, we will elaborate on the challenges of fault detection in embedded systems. After the general information, we will review the different types of feedback we implemented for our bug oracle.

5.1 IoT Device Fuzzing

Before we can discuss the bug oracle we implemented, we first need general information on the type of IoT devices, the type of faults that can occur, and how these faults translate into the device's behavior.

5.1.1 Types of IoT Devices

A detailed definition of an embedded system or an IoT device is still up for debate and outside the scope of this thesis. We consider devices that are designed for a specific task and interact with the physical world through sensors and other peripherals. An IoT device adds Internet connectivity to these devices. We want to distinguish between different embedded devices based on their abstraction layers. Added abstraction layers often translate into more security mechanisms on the device like a Memory Management Unit. This classification was proposed by Muench et al. in their paper on embedded device fuzzing [78].

The first type of IoT device we differentiate are devices that run a general-purpose operating system (OS). Because the OS adds overhead, this category is reserved for high-end devices with plenty of available resources. A well-known example of this category is the popular Raspberry Pi platform. It runs a custom Unix-based, general-purpose OS called Raspberry Pi OS or Raspian [79]. While the OS is general-purpose, it has to be tailored for the target device. For example, functionality to control the Raspberry Pi's General Purpose Input/Output (GPIO) pins must be included in the OS. Another example

is the BusyBox OS, which provides a minimized Linux kernel aimed at embedded devices [80].

The second type of devices still have an OS abstraction, but this OS is much more specific to the hardware platform. These are lightweight operating systems with the goal of improving platform performance when limited processing power is available. Advanced OS features might not be available, but there is still a separation between application and kernel code. These operating systems also focus on dealing with the real-time characteristic of the devices' deployment. An example of such a real-time OS is VxWorks [81].

The last category of IoT devices has no OS abstraction layer. These devices directly execute application code without operating system control or logging. Many low-performance devices fall into this category. Hobbyists often use the ATmega328 [82] or ESP8266 [83] chips for homemade IoT. In industry, devices like car key fobs, fridge controllers, digital thermometers, or remote controls are likely to work without an OS. The firmware for these devices can be called monolithic. While they may contain features of an operating system, these features are compiled together with the application code into a singular firmware blob. Mostly, the execution works with an infinite control loop where external events trigger some code being executed.

Our SUT, the EC25 LTE modem, sits in the first category. Quectel gives little information about the device's software. We were able to obtain some information by digging through the debug command-line interface ourselves and from Osmocom [84]. Our EC25 chip runs version 1.23.1 of the BusyBox OS on an ARM Cortex A7 processor. The LTE protocol itself is likely not fully implemented in software, as this would be infeasible to run on such a low-end device. Some processes on BusyBox control the hardware implementation. They communicate with the hardware using shared memory. In total, roughly 160 MB of memory is available for the OS. Note that while the EC25 chip has an OS, in some sense, we can also categorize it as the last type of device. The part of the chip we are trying to fuzz is not an ordinary process that runs in the OS but likely runs on separate hardware. The OS does offer some logging of the hardware interfaces, but they are difficult to understand and to extract useful information from without any extra information from the vendor.

5.1.2 Fault Types

This section discusses the challenges that arise when fuzzing embedded devices. More specifically, we focus on the type of faults that can occur and the difficulty of detecting these faults.

Bugs in the LTE implementation can cause a variety of faults. A common fault is **memory corruption**. Memory corruptions contain all kinds of errors that lead to the memory of a device containing incorrect values. This can be incorrect values of data items or incorrect addresses of other data items [85]. In software, a stack overflow is a well-known memory corruption that can cause a program to behave out of order. Stack overflows are a vital attack vector for adversaries to execute malicious code on a device [86]. We do not know if the LTE implementation uses a classical stack or heap in our test setup. Nevertheless, similar memory corruptions can, for example, occur on the shared

memory communication channels between the hardware and software. Internal memory used by only the hardware may also get invalid values, leading to incorrect behavior.

Another memory corruption can occur when a piece of code or hardware accesses memory without the required permission. This can lead to the modification of data of another part of the program that will later cause irregular behavior. Many desktop systems have protection in place to prevent this from happening, but these types of defenses are scarcely available in embedded systems [85].

A more specific fault for the LTE protocol is the **transition to an incorrect state**. Complex networking protocols are often implemented as state machines. In such a state machine, the device's state determines what events it expects to occur and how to handle them. The state machine transitions move a device from one state to the next based on the observed events. These events can be the reception of a new message from the other party, a timeout when waiting for a message, a button press of a user, ... Many state transitions are defined for each state to deal with a variety of possible events. An incorrect transition occurs when the transition of the state opposes the specification. This can be caused by, for example, an incorrect implementation of the protocol, a random external event that flips some bits (like an electric pulse in the power supply), or a memory corruption in the variable that stores the state. In the new incorrect state, the device expects completely different events than those it will receive. This can lead to the creation of more faults. A device can also enter an undefined state. The existence of such a state originates from the fact that the number of states is most likely not a power of two. Hence, there are more possible state variable values than there are possible states in the protocol. In an undefined state, it is very unpredictable how a device will act, and a full reset may be required to exit the undefined state.

5.1.3 Device Response to Faults

Desktop systems contain some defenses against memory corruptions. Some examples are the Memory Management Unit (MMU), Data Execution Prevention (DEP), Control Flow Integrity (CFI), or Memory Protection Unit (MPU). The goal of these defenses is to protect the device from performing invalid operations. They are implemented as a combined effort between the desktop hardware and operating system. When a violation is detected, the OS will stop the program that caused the violation, leading to an observable crash. In embedded systems, these features are rarely found. Figure 5.1 shows how few embedded processors support these defenses in their hardware. The lack of hardware protection in embedded systems creates a challenge for detecting faults. When a fault occurs, it will not always result in an observable crash. We have to consider a multi-faceted device response when developing our bug oracle.

Muench et al. researched the different responses of embedded devices when they purposefully introduce memory corruptions [78]. Their goal was not to find bugs in the device but to see how the devices responded when a memory corruption was triggered. They observed six types of responses:

Hardware Family	MPU	MMU	DEP	CFI
ARM 1 to ARM 7	✖	✖	✖	✖
ARM Cortex R	✔	✖	✖	✖
ARM Cortex M	∼ [1]	✖	✔	✖
PIC 10 to PIC 24	✖	✖	✖	✖
Intel MCS-51	✖	✖	✖	✖
Infineon XC88X-I	✖	✖	✖	✖
Infineon XC88X-A	✖	✖	✖	✖

[1] Supported by some micro controllers of the family

FIGURE 5.1: The support of protection mechanisms on common embedded processors. [85]

Observable Crash The execution of the affected process is terminated. In some cases, an additional text message is displayed to help identify the cause of the crash.

Reboot The whole device reboots and starts its processes from scratch. There is no difference between this response and the observable crash for devices without an OS.

Hang The device stops responding to new inputs. It appears to be stuck in some loop.

Late Crash The device appears fine initially but crashes at a later stage.

Malfunction The device keeps responding to new inputs but produces incorrect outputs as a result of the invalid state of the memory.

No Effect The memory corruption has no observable effect.

We ordered these responses from easy to observe to most challenging to observe. Muench et al. also concluded that the devices with an OS either responded with an observable or a late crash for all their tests. This appears to be good news for our test setup, but we will not rely on this fact. As we stated before, we can also categorize our device as one without an OS. It is possible that errors from the hardware implementation do not neatly carry over to the software control. In that case, errors in the hardware will not cause an observable crash visible in the operation system. Therefore we created feedback methods that do not solely rely on an observable crash of a software component.

5.2 Researched Feedback Options

In order to deal with the different possible reactions of a device to a test case, we implemented three feedback mechanisms using supported device interfaces. Each of these mechanisms targets a specific device response or fault type. We also touch on a feedback mechanism that does not use the device's interface but rather its physical response.

5.2.1 Device Interface Feedback

Liveliness Check

The first, most obvious, and simplest feedback option we created is a liveliness check. The goal of the Liveliness Check is to evaluate if the device is still alive and responding. We implemented this check with the AT command serial interface. After sending a fuzzing case to the device, we monitor it by repeatedly sending the *AT* command. We expect a simple *OK* when the device is alive, and the AT interface is operating correctly. The check is performed five times in order to monitor during the complete test. The test messages will have finished sending before the end of the Liveliness Check. We keep monitoring after the messages have been handled to account for late crashes of the device. For each of the five checks, we settled on a timeout of 500 milliseconds. If the device has not responded by then, we conclude a crash and report this.

There is a clear trade-off between feedback accuracy and platform performance, expressed as the number of seconds per test cases. If we monitor longer after the messages have been sent, we are more likely to catch late crashes, but we lose platform performance. Similarly, if we raise our timeout value, we will have fewer false-positive reports, but again at the cost of platform speed. We tend to improve platform performance at the cost of test result precision. This inaccuracy means we have to be more careful when processing our reports. When a test case is reported to crash the device, but the next few test cases report the device is still alive, this could be a false-positive report. If a few test cases after each other report a crash, we possibly have encountered a crash. When deriving which message made the device crash, we have to take preceding messages into account as well due to the existence of late crashes.

The simple AT Liveliness Check also suffers some limitations. The most significant comes from the fact that our device has an operating system. This indicates that the device runs multiple processes in a scheduled fashion. Without extra knowledge on the device's design, there is no certainty, but the process that handles AT commands is probably separated from the LTE implementation and LTE control processes. This means that even if the LTE implementation has crashed, the AT interface will still respond to the simple *AT* command. A better approach could be to use the AT interface to get information directly related to the LTE implementation. Requesting, for example, the LTE network status with *AT+COPS?*, is more linked to the behavior of the actual LTE stack on the device. However, in case of a crash in the LTE stack, we can not be sure how the device responds to the *AT+COPS?* command. It can either give an error or report that the device is not connected to a network. When it gives an error, we can still not be sure of a crash because the error might exactly be the behavior indicated by the LTE specification. We are sending malformed or incorrect messages to the device and thus do not expect the device to be able to connect to our srsLTE network. It will likely report this as an error, which is no indication that the LTE stack crashed or malfunctioned.

Responsiveness Check

To overcome the limitations of the Liveliness Check, we added a responsiveness check in srsENB. This feedback does not work via the AT interface as we described the problems

with that approach. Instead, we now look at the messages the device relays after our fuzzed test message. When a device receives an incorrect RRC Connection Setup message, we can expect four events. We have observed that the device can retransmit the RRC Connection Request message, can restart the connection establishment procedure with a Random Access Request, or can accept the message and respond with an RRC Connection Setup Complete message. Fourthly, the device may stop responding at all. The correct response to a fuzzed message, depends on what information we have modified in the RRC Connection Setup. If we are fuzzing some legal values of a parameter, we expect the device to accept the message. If we are fuzzing the structural bits, with a malformed message as a result, the device should reject the message and retry the connection establishment.

In srsENB, we log the reception of the Random Acces Request, RRC Connection Request, and RRC Connection Setup Complete messages. The controller gathers these logs and adds them to the report. We have not defined the correct outcome in any of the building blocks. In our analysis we have to decide whether the seen combination of uplink messages is anomalous or correct behavior. We could integrate this decision in the test controller itself, but it would require intricate tailoring for each fuzzed message. In our view, by defining the fuzzing strategy, we set the expectation of how the device should respond. In the analysis of the reports, we can then spot if the device behaved as we expected for our fuzzing strategy.

Not receiving a response can indicate a fault, but can also be instigated by a maximum amount of retransmission being exceeded. Multiple faults can cause the device to stop responding: a crash, a reboot, a late crash, or a hang of the LTE stack. A malfunctioning fault can also cause the device to respond with another message than the three we expect. We cannot catch these last faults, as we would need to start logging all possible incoming messages in srsENB. This is impractical. Hence we chose not to expand the Responsiveness Check beyond logging the Random Access Request, RRC Connection Request, and RRC Connection Setup Complete messages.

Normal Operation Check

To increase our chances of exposing a device malfunction, we created a third feedback system called the Normal Operation Check. This check aims at finding a wide range of possible corruptions that damage the integrity of the device. We start from the assumption that a critical corruption will affect the device in correct operation. It may lead to the device refusing to connect to any network, independent of it sending correct or incorrect messages. Our previous two feedback mechanisms will not easily notice this type of corruption as they may expect the device to refuse the connection. The Normal Operation Check verifies if the device still wants to connect when the legitimate messages are being delivered.

Periodically, we will turn off fuzzing in the SRS software and evaluate the device's operation. We check the correct device working in two ways. First, we check the device's connectivity by sending the *AT+COPS?* command. Since we no longer send fuzzed messages, we expect the device to report that it has connected to our SRS network with the correct network identifier. Secondly, we check if the device has access to the internet via our LTE network. We test this with the command: *AT+QPING=1,"216.58.213.228",1,6.* This

command pings an IP address of *google.com* and reports on the delay of the packets. Here we expect the device to reply with six received pong messages. If something is wrong that prevents the device from accessing the Internet, the command will likely end with an error or with no pong messages received.

Checking the normal operation of the device is a slow process. It involves the entire connection establishment and the processing of two AT commands, where the *QPING* command is the slowest one. Performing the Normal Operation Check after each fuzzed message would cost too much performance. Instead, we opted for a periodic approach. After every hundred fuzzed test cases, we perform the Normal Operation Check. Therefore, if this check fails, we still need to investigate which of these hundred earlier messages caused a bug in the device. Here again, we notice the trade-off between bug oracle accuracy and platform performance.

5.2.2 Physical Device Feedback

Eceiza et al. list the physical response as one of the embedded device characteristics that could be leveraged for bug detection [85]. Due to the simplicity of the design of embedded systems compared to the design of general-purpose desktop systems, the physical response is more connected to the performed operations than on desktop systems. Two main attributes of a physical response are the device temperature and the power consumption.

The temperature of any computer system increases as the number of operations per second increases. In desktop systems, advanced cooling techniques are used to mitigate temperature increases. This masks the relation between performed operations and measured temperature. On the other hand, embedded systems mainly rely on passive cooling. This makes temperature readings a good estimation for the load on the device's hardware. However, we will not focus on temperature readings as they suffer from one major drawback: thermal inertia. It takes time for the material to heat up and cool back down. This means that temperature readings offer a delayed estimate of the hardware load. On the other hand, power consumption offers an instantaneous view of the performed operations.

At the end of chapter 4, we discussed how power consumption is related to the executed operations of a computer chip. We can leverage on this information in our bug oracle as well. Device reboots, infinite loops, or hardware resets are likely to cause specific patterns in the device's power consumption. Without fully integrating it into our platforms, we investigated what sort of patterns might emerge from a power trace.

Measurement Setup

In order to measure the device's power consumption, we had to disrupt the power delivery from the Raspberry Pi to the LTE modem. We could choose to measure the total power usage of the Raspberry Pi, LTE hat, and LTE modem, but this would result in a very noisy power trace. Other processes on the Raspberry Pi would obscure the power consumption of the LTE modem. Ideally, we want to measure the power consumption as close to the LTE modem as possible, with nothing more than the modem being powered by our

measurement equipment. Practically, this was too difficult as we would need to cut the tiny wires of the mini PCIe slot and attempt to solder our power supply to these wires. We compromised by powering the LTE hat and LTE modem using our measurement equipment, while the Raspberry Pi remained powered as usual. The components on the LTE hat itself, like the SIM card or voltage conversion module, will consume some power, which creates noise on the power trace, but we accepted this noise to keep our solution practical.

For the measurement equipment, we used the Keysight N6781A power supply [87]. With this power supply, we can fix the supply voltage at 5.0V and measure the current usage by the device. The equipment can record over 500k measurement points with a time accuracy of up to 5.12 microseconds. The current measurements have a 250μA accuracy in the range we use the power supply (between 100mA and 3A). Our power traces are 450k samples long, with the sample rate depending on the number of seconds we needed for our measurement. We should note that these power traces are not collected on the EC25 LTE modem but on the Telit LE910C1-EU modem. This change of device avoided an error in the setup breaking the EC25 chip, which would stop our progress of the fuzzing attacks on the EC25 modem.

When plotting the raw power traces in Matlab, very few features can be discerned. We need to apply some basic filtering before patterns and characteristics become visible. We attempted a simple moving average filter and a built-in Matlab function called *decimate*. The *decimate* function applies a lowpass Chebyshev type I IIR filter of order 8 to the input and reduces the number of sample points by a factor r afterward. Figure 5.2 shows the result of these two filters. From these two filters, we opted to use the *decimate* function for future power traces with a decimation factor of 50.

Scenario Power Traces

With the setup explained in the previous section, we can now discuss the results of our experiments. We measured five scenarios of the Telit modem:

- The modem sits in an idle, disconnected state
- The modem reboots during the power measurement
- The modem connects to a correctly functioning LTE network
- The modem connects, disconnects, and connects again to a correctly functioning LTE network
- The modem attempts to connect to a malicious LTE network sending fuzzed messages.

Before analyzing the power traces, it is advantageous to investigate the electrical characteristics the vendor gives of the modem. From the hardware design specification [88], we learn that the modem in the idle state typically consumes around 15mA at a nominal voltage of 3.8V, or 57mW of power. When connecting in LTE, the device draws a maximum current of 860mA, or 3.27W of power. The device will likely not draw as much power in our setup because the wired channel between the device and base station has much less pathloss than typically seen in real-life LTE use cases.

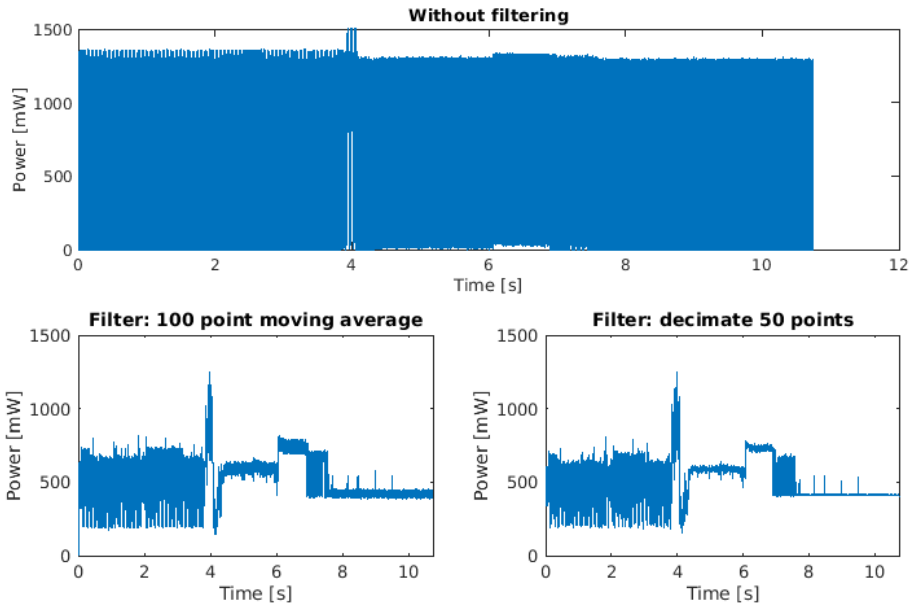

FIGURE 5.2: The raw power trace measurement compared to the filtered traces. Bottom left shows the power trace after applying a moving average filter of 100 points. Bottom right shows the result when using the *decimate* function with a sample reduction of 50.

Figure 5.3 shows a power trace of the Telit modem with disabled network connectivity (*AT+COPS=2*). We measured an average power consumption of 123.81mW during the idle scenario. This extra 66mW of power is likely consumed by the LTE hats' components and SIM card. We could subtract this 66mW from the other power traces to reduce the bias introduced by powering the full LTE hat. However, since we are interested in the observed patterns in the power traces rather than the absolute values, this bias is not relevant.

The other four scenarios were measured ten consecutive times with 450k samples at varying sample speeds. We were surprised how well the ten different power traces resemble each other in all scenarios. Figure 5.4 shows how well the patterns from most traces line up for the reboot scenario. In one instance, the device reacts a few seconds later, and in another, the power consumption before the reboot is a bit higher. All eight other power traces line up very well.

Figure 5.5 shows the average power trace for the four scenarios. We can clearly see some patterns in the device's power consumption, depending on the scenario we put it through (Figure 5.6). It is these types of patterns we could use as a bug oracle. We could establish some patterns of the device based on earlier measurements and then look for these patterns while fuzzing the device. As an illustration, we created a pattern of a reboot by averaging the eight lined up reboot power traces and cutting the trace between 2.63s and 5.72s. Next, we plotted the cross-correlation between this pattern and a random power trace of the scenarios (Figure 5.7). We see that the reboot power trace has a much higher cross-correlation with the pattern than the other power traces. This

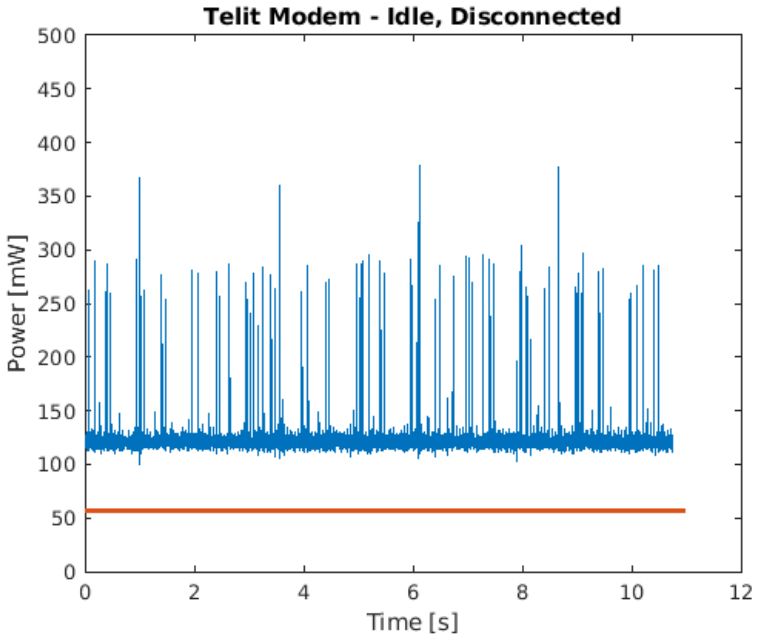

FIGURE 5.3: A power trace of the Telit modem in idle, disconnected state. The power measurement contains 524288 measurement points, sampled at 48.82kHz. The orange line indicates the 57mW power consumption predicted by the technical specification of the device.

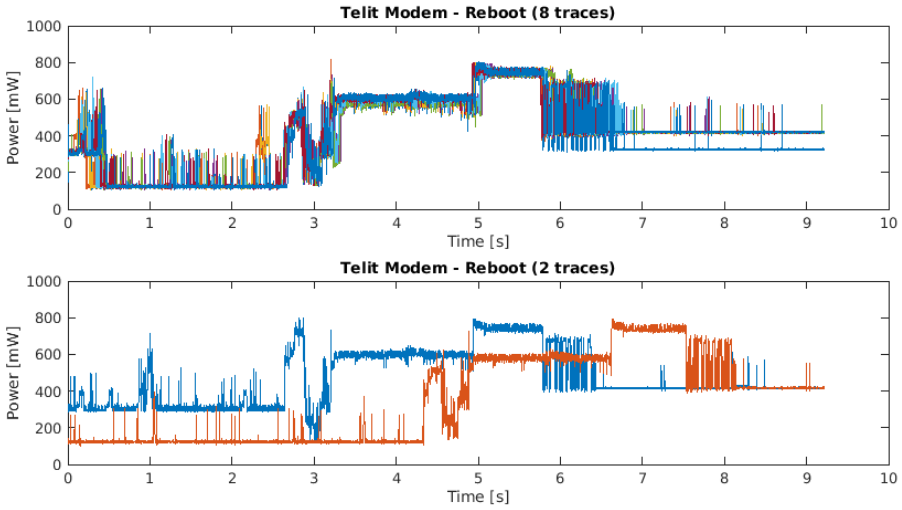

FIGURE 5.4: Different power traces overlayed on top of each other. Each power trace contains 450k samples at 48.82kHz sample rate. A reboot was triggered in the device at 1s. For the overlapping, we increased the decimation factor to 500.

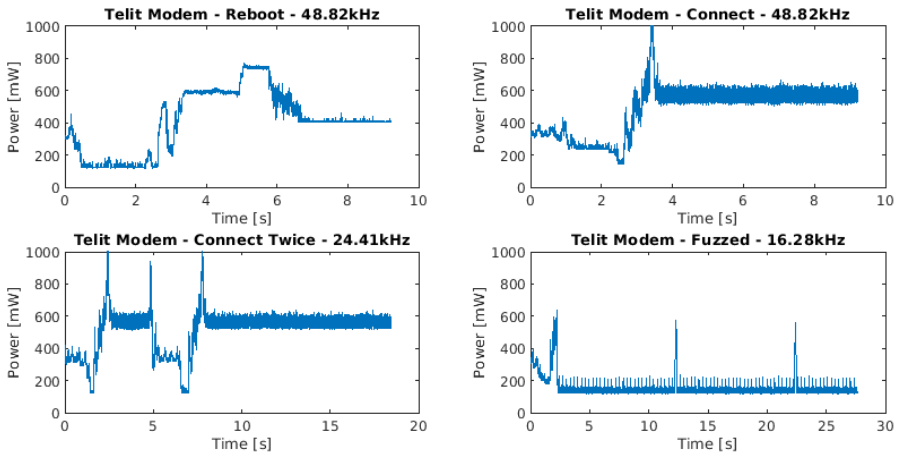

FIGURE 5.5: The average power trace for the four scenarios. We only included the power traces that aligned well. There were eight aligning traces for the top graphs and nine aligning traces for to bottom ones. The sample rate is given in the title of each graph. A decimation factor of 50 was used.

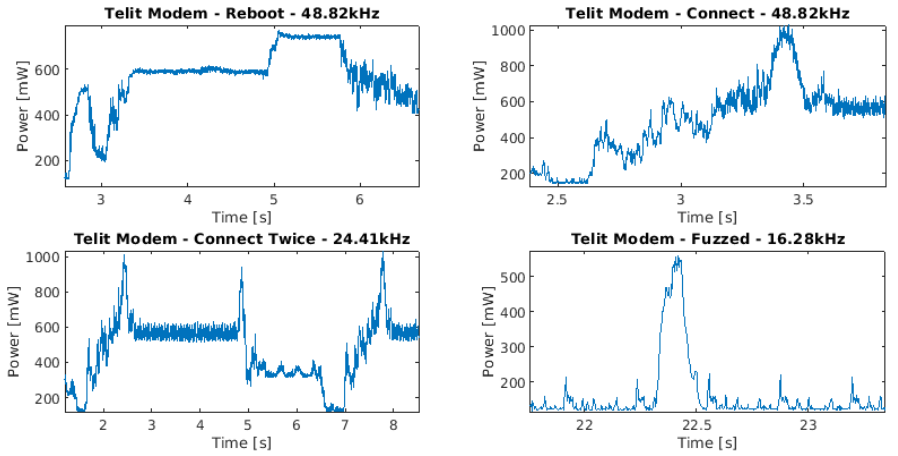

FIGURE 5.6: The same figure as Figure 5.5, but zoomed in on the specific patterns of each scenario.

cross-correlation is one way of finding specific patterns in the power trace that could help improve the bug oracle.

Integration Challenges

We conclude with some challenges that arise when fully integrating the power trace as a bug oracle in the platform. We identify three challenges.

First, the characteristics of the power trace heavily depend on the device's hardware

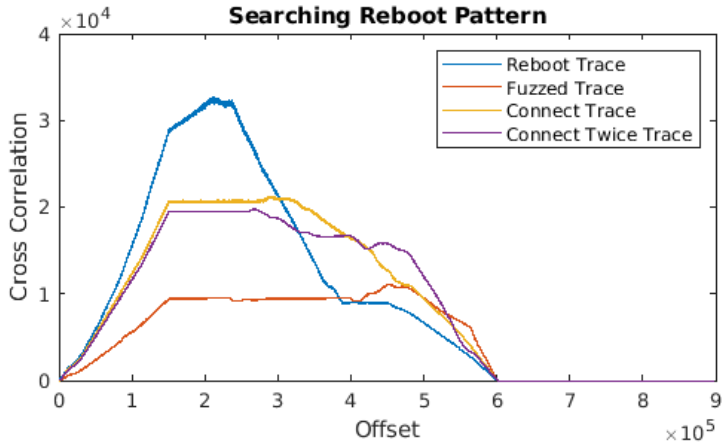

FIGURE 5.7: The cross correlation between the reboot pattern and random power traces of each scenario. The power traces were not filtered through the decimation filter before the cross correlation. The raw samples were used instead.

and software design. This means that the patterns one wants to look for should be created per device. To keep the platform as hands-off as possible, the creation should be automated. It can be implemented as a preprocessing step where specific device behavior is triggered, and a pattern is established. The challenge in automating this preprocessing will come in aligning the different traces to a sufficient degree.

Our second challenge is related to this preprocessing. Establishing patterns for reboots and correct operation is relatively straightforward, but patterns for crashes or hangs are far more complicated. Here we come in a sort of "chicken or the egg" problem. We need the patterns to detect bugs in the device, but we already need to trigger the bugs in order to establish the power trace patterns. One possible solution could be to iterate through the analysis in multiple rounds. Once a few bugs are found with our other bug oracles, we use their power traces to find more bugs in a second iteration and so on. Alternatively, one could create patterns for all types of correct behavior and mark a fuzz test as anomalous when none of these patterns can be found in the power trace.

Our third and last challenge has to do with the platform's performance. In these proof-of-concept tests, we used very detailed power traces with many samples and a high sample rate. These accurate power traces would cause a significant performance drop when integrated into our platform. It takes over seven seconds to transfer the traces we obtained from the N6781A to the test controller. Each power trace is also roughly five megabytes of data, quickly resulting in five to fifty gigabytes of data for the entire fuzzing run. To solve the performance drop, we could use the ELOG or external logging feature of the N6781A. Here, instead of collecting all the data first and delivering it later, the N6781A outputs small chunks of data during the measurement. This could turn our slow transfer of data into a fast feedback mechanism. To reduce the storage requirements, we need to ensure that our pattern matching techniques also work with lower precision measurements using fewer samples and a lower sample frequency.

Name	Goal	How	Limits
Liveliness Check	Observable crashes	AT, AT+COPS?	AT interface too isolated from LTE stack
Responsiveness Check	Observable crashes, hangs, and reboots	Log received messages in SRS	Cannot catch general malfunctions
Normal Operation Check	Catch general malfunctions	AT+COPS?, AT+QPING	Slow
Physical Characteristics	Catch potentially all bugs	N6781A measurements	Preliminary

TABLE 5.1: A short overview on the different bug detection mechanisms.

5.3 Bug Oracle Overview

Table 5.1 presents an overview of the four bug oracles we examined in this thesis. The table marks the goal as well as the limitations for each technique. One significant facet that is missing in this table is the effectiveness of a technique to find bugs. Because we worked with a commercial, real-life IoT device, we had no control to introduce bugs ourselves in order to evaluate our feedback mechanisms. Meunch et al. started from a software piece that purposely contains memory bugs. We did not run our own code on the device, meaning we could never be sure when or if a bug in the device was triggered. Therefore, our feedback systems are designed from logic and the knowledge we have of the device rather than from experimentation. This makes a formal evaluation of the effectiveness of our feedback mechanism impossible. Future research can shed some light on this topic by running custom code containing some bugs on a cellular modem, where it is known how to trigger bugs on the device.

Chapter 6

Experimental Validation

This chapter has been left out by the authors because of responsible disclosure. To get access to the confidential chapter, please contact the authors at dave.singelee@esat.kuleuven.be

Chapter 7

Conclusion

The tremendous growth in the IoT and cellular IoT market signifies the importance of secure IoT systems. IoT systems have had their fair share of exposed security vulnerabilities, leading to infamous names like the "Interconnection of Threats" [89]. In this thesis, we have shown how to build an automated platform that can test the security of LTE implementations of IoT devices. The platform is designed in a modular fashion where components can be replaced as long as they respect the platform's interfaces. By sending malicious packets to the device under test, a technique called network fuzzing, we can test the robustness of LTE modems.

In these final sections of the thesis, we look back at the design of our platform and results, and look forward to the remaining challenges and future work that can be carried out to improve it.

Platform Architecture

Our modular platform consists of an LTE network simulator, a system under test (SUT), a fuzzer, and a test controller. The LTE network simulator is obtained using the open-source simulator called srsLTE. Using the Ettus USRP B210 SDR, we connect the simulator to our SUT. We chose to attack the Quectel EC25 LTE modem. This device was mounted on top of a Raspberry Pi 4 using the LTE hat from Sixfab. The serial AT interface to the EC25 chip was used to configure the device in our fuzzing tests.

We integrated two different fuzzers in our platform: an off-the-shelf, popular fuzzer called American Fuzzy Lop (AFL) and our custom-made WDissector Fuzzer. We encountered two hurdles when using AFL as a tool in our network fuzzing platform: its lack of protocol awareness and the absence of accessible device instrumentation. Our custom-made WDissector Fuzzer solved the first of these hurdles, paving the way for targetted fuzzing attacks. Through packet dissection, we could target distinct fields of the RRC layer messages.

The typical traits of embedded IoT systems, like limited process monitoring or memory management, impede the process of discovering bugs. We have incorporated three checks in our platform aimed at recognizing different anomalous behavior. A Liveliness Check intends to catch observable crashes and reboots, the Responsiveness Check logs incoming uplink messages in order to spot device hangs, and the Normal Operation Check - our

most general and slowest check - attempts to detect a variety of malfunctions in the device.

The Keysight open-source OpenTap test automation framework was employed to control all components of our platform and ensure smooth operation during the fuzz testing.

Experimental Results

This section has been left out by the authors because of responsible disclosure. To get access to the confidential chapter, please contact the authors at dave.singelee@esat.kuleuven.be

Future work

We conclude with some remarks on the used techniques and possible future work to improve the platform.

In future research, we would be hesitant to recommend using the Raspberry Pi as the host platform for the device under test. In our experience, the Raspberry can act finicky due, primarily when powered via the GPIO pins. On one occasion, we burned through one of our Raspberry Pi OS SD cards, causing some data loss. We recommend building a more robust, **custom breakout board** for multiple chip formats.

Another architectural modification, aimed at improving platform performance, could be to **duplicate the setup**. Using the USRP B210's second channel, we can connect a second LTE modem and effectively double our platform's performance.

We already alluded to the use of **power consumption analysis** to enhance our platform. Power consumption is highly correlated to a device's operations. By detecting irregularities in the power consumption, we can discover test cases that trigger new parts of the LTE implementation and create an intelligent protocol-aware fuzzer. Additionally, we also investigated how power trace analysis can be leveraged as a bug oracle. In Chapter 5 we showed how certain device-behavior scenarios leave a specific footprint in the device's power usage. Integrating these measurements in our platform can help detect test cases that trigger bugs or inversely rule out test cases that are handled correctly.

We can improve the fuzzer further by taking a more **active approach to generating messages**. Instead of mutating only the data fields present in the message we receive from srsENB, we could replace whole data structures in that message. LTE supports much more information than what the SRS software uses, and this enhanced test case generation would lead to a more thorough test set.

Lastly, We also would like to point out that we focused only on one message of the pre-authentication traffic. Other messages like the MIB or SIBs are a promising target for attacks as well. They come with some extra challenges because it is less apparent when the device will decode them, but fuzzing them can uncover more vulnerabilities. Due to time constraints, we did not fuzz these messages and kept our attention to the RRC Connection Setup. Nevertheless, **fuzzing the SIBs and MIB** is an essential aspect of further work on the platform and device attacks.

Appendix A

Industrial Context

This thesis is executed in collaboration with Keysight Technologies. This appendix gives an overview of Keysight as a company as an illustration of the industrial context of this thesis.

Keysight Technologies is a publicly-traded company headquartered in Santa Rosa, USA. Their global reach expands over 150 worldwide locations, 13900 employees, and customers in more than 100 countries. They have their own ASIC fabrication centers and microelectronic packaging facilities. 29 of the Top 30 Tech companies use Keysight products in their development process. R&D centers in 15 different countries have produced over 3500 patents. In Belgium, Keysight holds branches in Rotselaar and Ghent. This thesis was performed in the Rotselaar office. During the thesis, there was a close collaboration with the IoT research group of Keysight Aalborg, Denmark. Additionally, collaboration with the Keysight office in Singapore and the ASSET Research Group at Singapore University of Technology and Design (SUTD) supported the development of the WDissector Fuzzer.

Keysight technologies originates from the Hewlett-Packard (HP) company. In 1939, HP was founded with its products focusing on electronic measurement equipment. Over the years, HP became omnipresent in the technology market with a product portfolio ranging from the original measurement equipment to printers or desktop computers. In 1999, the measurement equipment branch op HP spun off to form Agilent Technologies. By then, measurement was far beyond electronic measurements. Agilent still is a significant player in chemical analysis equipment, clinical testing equipment, and more. In 2014, the electronic measurement equipment branch spun off again to form Keysight Technologies.

Keysight creates products for various markets: wireless communications, automotive industry, networking and cloud, aerospace and defense, and IoT. This project falls under their IoT product portfolio. Within IoT, they develop products for layer 1 through layer 7 of the communicating stack. From electrical measurement equipment like signal analyzers to full-blown base station emulators for cellular devices. They develop both hardware and software solutions to aid the customers in their product design over the entire stack. Keysight acquired Ixia Application and Threat Intelligence Research Center in 2017, indicating their focus on device and application security.

Appendix B

LTE Channels

This appendix lists the full name of the LTE channels. This is not relevant for the understanding of the thesis, but the interested reader can learn more about these channels from their full name.

Physical Channels

PRACH	Physical Random Access Channel
PUSCH	Physical Uplink Shared Channel
PUCCH	Physical Uplink Control Channel
PBCH	Physical Broadcast Channel
PDSCH	Physical Downlink Shared Channel
PHICH	Physical Hybrid Automatic Repeat Request Indicator Channel
PDCCH	Physical Downlink Control Channel
PCFICH	Physical Control Format Indicator Channel
PMCH	Physical Multicast Physical control format indicator channel

Transport Channels

RACH	Random Access Channel
UL-SCH	Uplink Shared Channel
BCH	Broadcast Channel
PCH	Paging Channel
DL-SCH	Downlink Shared Channel
MCH	Multicast Channel

Logical Channels

CCCH	Common Control Channel
DCCH	Dedicated Control Channel
DTCH	Dedicated Traffic Channel
BCCH	Broadcast Control Channel
PCCH	Paging Control Channel
MTCH	Multicast Traffic Channel
MCCH	Multicast Control Channel

Bibliography

[1] K. L. Lueth. (2020). “State of the iot 2020: 12 billion iot connections, surpassing non-iot for the first time”, [Online]. Available: https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time (visited on 02/05/2021).

[2] M. Hatton. (2020). “The iot in 2030: Which applications account for the biggest chunk of the $1.5 trillion opportunity?”, [Online]. Available: https://transformainsights.com/blog/iot-24-billion-connected-things-15-trillion (visited on 02/15/2021).

[3] D. Evans, “The internet of things: How the next evolution of the internet is changing everything”, Cisco Internet Business Solutions Group, Cisco Systems, Tech. Rep., 2011. [Online]. Available: https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

[4] *3rd generation partnership project*, https://www.3gpp.org/, 2021.

[5] E. Dahlman, S. Parkvall, and J. Sköld, “Chapter 1 - background of lte”, in *4G LTE/LTE-Advanced for Mobile Broadband*, E. Dahlman, S. Parkvall, and J. Sköld, Eds., Oxford: Academic Press, 2011, pp. 1–13, ISBN: 978-0-12-385489-6. DOI: https://doi.org/10.1016/B978-0-12-385489-6.00001-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123854896000011.

[6] O. Liberg, M. Sundberg, Y.-P. E. Wang, J. Bergman, and J. Sachs, “Chapter 5 - lte-m”, in *Cellular Internet of Things*, O. Liberg, M. Sundberg, Y.-P. E. Wang, J. Bergman, and J. Sachs, Eds., Academic Press, 2018, pp. 135–197, ISBN: 978-0-12-812458-1. DOI: https://doi.org/10.1016/B978-0-12-812458-1.00005-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128124581000058.

[7] O. Liberg, M. Sundberg, Y.-P. E. Wang, J. Bergman, and J. Sachs, “Chapter 7 - nb-iot”, in *Cellular Internet of Things*, O. Liberg, M. Sundberg, Y.-P. E. Wang, J. Bergman, and J. Sachs, Eds., Academic Press, 2018, pp. 217–296, ISBN: 978-0-12-812458-1. DOI: https://doi.org/10.1016/B978-0-12-812458-1.00007-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128124581000071.

[8] M. Lauridsen, I. Z. Kovacs, P. Mogensen, M. Sorensen, and S. Holst, "Coverage and capacity analysis of lte-m and nb-iot in a rural area", in *2016 IEEE 84th Vehicular Technology Conference (VTC-Fall)*, 2016, pp. 1–5. DOI: 10.1109/VTCFall.2016.7880946.

[9] N. Networks, "Lte-m – optimizing lte for the internet of things, white paper", Nokia Solutions and Networks Oy, Tech. Rep., 2015. [Online]. Available: https://lafibre.info/images/4g/201508_nokia_lte-m_white_paper.pdf.

[10] E. U. Ogbodo, D. G. Dorrell, and A. M. Abu-Mahfouz, "Performance measurements of communication access technologies and improved cognitive radio model for smart grid communication", eng, *Transactions on emerging telecommunications technologies*, vol. 30, no. 10, n/a, 2019, ISSN: 2161-3915.

[11] B. E. Benhiba, A. A. Madi, and A. Addaim, "Comparative study of the various new cellular iot technologies", in *2018 International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS)*, 2018, pp. 1–4. DOI: 10.1109/ICECOCS.2018.8610508.

[12] L. Feltrin, M. Condoluci, T. Mahmoodi, M. Dohler, and R. Verdone, "Nb-iot: Performance estimation and optimal configuration", eng, in *24th European Wireless 2018 "Wireless Futures in the Era of Network Programmability", EW 2018*, 2018, pp. 50–55, ISBN: 9783800745609.

[13] F. Jejdling, "Ericsson mobility report", Ericsson, Tech. Rep., Nov. 2020. [Online]. Available: https://www.ericsson.com/4adc87/assets/local/mobility-report/documents/2020/november-2020-ericsson-mobility-report.pdf.

[14] GSMA. (Nov. 2020). "Mobile iot deployment map", [Online]. Available: https://www.gsma.com/iot/deployment-map/ (visited on 02/22/2021).

[15] Qualcomm, "Leading the lte iot evolution to connect the massive internet of things", Qualcomm Technologies, Inc, Tech. Rep., Jul. 2017. [Online]. Available: https://www.qualcomm.com/media/documents/files/whitepaper-leading-the-lte-iot-evolution-to-connect-the-massive-internet-of-things.pdf.

[16] T. Malevitis, K. Yalowitz, B. Berg, and E. Gonzales, "The internet of things for telcos: Build from core to win", Accenture Strategy, Tech. Rep., 2019. [Online]. Available: https://www.accenture.com/_acnmedia/PDF-97/Accenture-The-Internet-of-Things-for-Telcos.pdf.

[17] Kaspersky. (Oct. 2019). "Iot under fire: Kaspersky detects more than 100 million attacks on smart devices in h1 2019", [Online]. Available: https://www.kaspersky.com/about/press-releases/2019_iot-under-fire-kaspersky-detects-more-than-100-million-attacks-on-smart-devices-in-h1-2019 (visited on 03/09/2021).

[18] A. M. A. Abuagoub, "Iot security evolution: Challenges and countermeasures review", English, *International Journal of Communication Networks and Information Security*, vol. 11, no. 3, pp. 342–351, Dec. 2019, Copyright - Copyright Kohat University of Science and Technology (KUST) Dec 2019; Last updated - 2020-02-27. [Online]. Available: https://search-proquest-com.kuleuven.ezproxy.kuleuven.be/scholarly-journals/iot-security-evolution-challenges-countermeasures/docview/2354296086/se-2?accountid=17215.

[19] L. Shuai, H. Xu, L. Miao, and X. Zhou, "A group-based ntru-like public-key cryptosystem for iot", *IEEE Access*, vol. 7, pp. 75 732–75 740, 2019. DOI: 10.1109/ACCESS.2019.2920860.

[20] P. Pingale, K. Amrutkar, and S. Kulkarni, "Design aspects for upgrading firmware of a resource constrained device in the field", eng, in *2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, IEEE, 2016, pp. 903–907, ISBN: 9781509007745.

[21] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, "Demystifying iot security: An exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations", *IEEE Communications Surveys & Tutorials*, vol. 21, no. 3, pp. 2702–2733, 2019. DOI: 10.1109/COMST.2019.2910750.

[22] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "Ddos in the iot: Mirai and other botnets", *Computer*, vol. 50, no. 7, pp. 80–84, 2017. DOI: 10.1109/MC.2017.201.

[23] L. Wouters, E. Marin, T. Ashur, B. Gierlichs, and B. Preneel, "Fast, furious and insecure: Passive keyless entry and start systems in modern supercars", Aug. 2019.

[24] E. Marín Fàbregas, D. Singelée, F. Garcia, T. Chothia, R. Willems, and B. Preneel, "On the (in)security of the latest generation implantable cardiac defibrillators and how to secure them", eng, Schwab, Stephen, vol. 5-9-December-2016, ACM, 2016, pp. 226–236, ISBN: 9781450347716. [Online]. Available: $$Uhttps://doi.org/10.1145/2991079.

[25] D. Rupprecht, A. Dabrowski, T. Holz, E. Weippl, and C. Pöpper, "On security research towards future mobile network generations", *IEEE Communications Surveys & Tutorials*, vol. PP, Oct. 2017. DOI: 10.1109/COMST.2018.2820728.

[26] L. Huang, "Lte redirection attack - forcing targeted lte cellphone into unsafe network", May 2016.

[27] M. Lipow, "Number of faults per line of code", *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 437–439, 1982. DOI: 10.1109/TSE.1982.235579.

[28] B. Michau and C. Devine, "How to not break lte crypto", Jul. 2016.

[29] R.-P. Weinmann, "Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks", Aug. 2012, pp. 2–2.

[30] K. Moshe, S. Ariel, O. Shlomi, D. Alon, Z. Andrey, and S. Yuli. (Jul. 2020). "Ripple20", [Online]. Available: https://www.jsof-tech.com/disclosures/ripple20/ (visited on 05/27/2021).

[31] d. S. Daniel, D. Stanislav, A. Amine, W. Jos, O. Shlomi, and K. Moshe. (Apr. 2021). "Name:wreck, breaking and fixing dns implementations", [Online]. Available: https://www.forescout.com/company/resources/namewreck-breaking-and-fixing-dns-implementations/ (visited on 05/27/2021).

[32] N. Kosmatov and J. Signoles, "Frama-c, a collaborative framework for c code verification: Tutorial synopsis", in *Runtime Verification*, Y. Falcone and C. Sánchez, Eds., Cham: Springer International Publishing, 2016, pp. 92–115, ISBN: 978-3-319-46982-9.

[33] E. Dahlman, S. Parkvall, and J. Sköld, "Chapter 4 - radio-interface architecture", in *4G LTE-Advanced Pro and The Road to 5G (Third Edition)*, E. Dahlman, S. Parkvall, and J. Sköld, Eds., Third Edition, Academic Press, 2016, pp. 55–74, ISBN: 978-0-12-804575-6. DOI: https://doi.org/10.1016/B978-0-12-804575-6.00004-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128045756000042.

[34] A. Bradai, T. Rasheed, T. Ahmed, and K. Singh, "Cellular software defined network – a framework", *IEEE Communications Magazine*, vol. 53, Jun. 2015. DOI: 10.1109/MCOM.2015.7120043.

[35] M. Yokoyama, B. Ying, R. Yonezawa, M. Stambaugh, J.-P. Gregoire, M. Leung, and M. Rumney, "Physical layer", in *LTE and the Evolution to 4G Wireless: Design and Measurement Challenges*. 2013, pp. 91–157. DOI: 10.1002/9781118799475.ch3.

[36] A. C. Rasmussen and M. R. Kielgast, "Embedded massive mtc device emulator for lte using software defined radios", Ph.D. dissertation, 2017.

[37] S. Singh, S. Charlton, V. Ratnakar, D. Sabharwal, P. Goldsack, and N. Das, "Upper layer signaling", in *LTE and the Evolution to 4G Wireless: Design and Measurement Challenges*. 2013, pp. 159–194. DOI: 10.1002/9781118799475.ch4.

[38] G. C. Madueño, J. J. Nielsen, D. M. Kim, N. K. Pratas, Č. Stefanović, and P. Popovski, "Assessment of lte wireless access for monitoring of energy distribution in the smart grid", *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 3, pp. 675–688, 2016. DOI: 10.1109/JSAC.2016.2525639.

[39] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities", *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279. [Online]. Available: https://doi.org/10.1145/96267.96279.

[40] D. Heinze, J. Classen, and F. Rohrbach, "Magicpairing: Apple's take on securing bluetooth peripherals", eng, in *Proceedings of the 13th ACM Conference on security and privacy in wireless and mobile networks*, ser. WiSec '20, ACM, 2020, pp. 111–121, ISBN: 9781450380065.

[41] T. Werquin, M. Hubrechtsen, A. Thangarajan, F. Piessens, and J. T. Muehlberg, "Automated fuzzing of automotive control units", eng, 2021.

[42] H. Kim, J. Lee, L. Eunkyu, and Y. Kim, "Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane", in *Proceedings of the IEEE Symposium on Security & Privacy (SP)*, IEEE, May 2019.

[43] M. Eceiza, J. L. Flores, and M. Iturbe, "Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems", *IEEE Internet of Things Journal*, pp. 1–1, 2021. DOI: 10.1109/JIOT.2021.3056179.

[44] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art", *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018. DOI: 10.1109/TR.2018.2834476.

[45] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey", *IEEE Transactions on Software Engineering*, pp. 1–1, 2019. DOI: 10.1109/TSE.2019.2946563.

[46] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing", ser. CCS '13, Berlin, Germany: Association for Computing Machinery, 2013, pp. 511–522, ISBN: 9781450324779. DOI: 10.1145/2508859.2516736. [Online]. Available: https://doi.org/10.1145/2508859.2516736.

[47] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)", in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[48] M. Buss, D. Brand, V. Sreedhar, and S. A. Edwards, "A novel analysis space for pointer analysis and its application for bug finding", *Science of Computer Programming*, vol. 75, no. 11, pp. 921–942, 2010, Special Section on the Programming Languages Track at the 23rd ACM Symposium on Applied Computing, ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2009.08.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642309001208.

[49] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzzing based on machine learning techniques", eng, *PloS one*, vol. 15, no. 8, e0237749–e0237749, 2020, ISSN: 1932-6203.

[50] X. Zhou and B. Wu, "Web application vulnerability fuzzing based on improved genetic algorithm", in *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, vol. 1, 2020, pp. 977–981. DOI: 10.1109/ITNEC48623.2020.9084765.

[51] K. Deb, "An introduction to genetic algorithms", eng, *Sadhana (Bangalore)*, vol. 24, no. 4, pp. 293–315, 1999, ISSN: 0256-2499.

[52] D. Maier, L. Seidel, and S. Park, "Basesafe: Baseband sanitized fuzzing through emulation", in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '20, Linz, Austria: Association for Computing Machinery, 2020, pp. 122–132, ISBN: 9781450380065. DOI: 10.1145/3395351.3399360. [Online]. Available: https://doi.org/10.1145/3395351.3399360.

[53] *Qemu*, https://gitlab.com/qemu-project/qemu, 2021.

[54] *Openlte*, https://github.com/mgp25/OpenLTE, 2021.

[55] N. Nikaein, R. Knopp, F. Kaltenberger, L. Gauthier, C. Bonnet, D. Nussbaum, and R. Ghaddab, "Demo: Openairinterface: An open lte network in a pc", in *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '14, Maui, Hawaii, USA: Association for Computing Machinery, 2014, pp. 305–308, ISBN: 9781450327831. DOI: 10.1145/2639108.2641745. [Online]. Available: https://doi.org/10.1145/2639108.2641745.

[56] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, and D. J. Leith, "Srslte: An open-source platform for lte evolution and experimentation", eng, 2016.

[57] E. Research. (2021). "Usrp b210 sdr kit", [Online]. Available: https://www.ettus.com/all-products/ub210-kit/ (visited on 03/21/2021).

[58] M. Arena. (Jul. 2019). "What lte bands are used in europe?", [Online]. Available: https://mobilityarena.com/what-lte-bands-are-used-in-europe/ (visited on 03/21/2021).

[59] E. Research. (2021). "Usrp hardware driver (uhd™) software", [Online]. Available: https://github.com/EttusResearch/uhd (visited on 03/21/2021).

[60] S. Mandal. (Feb. 2020). "Quectel widens gap with competition in global cellular iot module market during covid-hit q2 2020", [Online]. Available: https://www.counterpointresearch.com/quectel-widens-gap-with-competition-in-global-cellular-iot-module-market-during-q2-2020/ (visited on 03/22/2021).

[61] M. ZHOU, F. WANG, L. LIU, N. LIU, W. WANG, and E. SHAN, "Ec25 mini pcie hardware design", Quectel, Tech. Rep., Apr. 2019. [Online]. Available: https://sixfab.com/wp-content/uploads/2021/02/Quectel_EC25_Mini_PCIe_Hardware_Design_V2.3.pdf.

[62] Sixfab. (Mar. 2021). "Raspberry pi 3g/4g & lte base hat", [Online]. Available: https://sixfab.com/product/raspberry-pi-base-hat-3g-4g-lte-minipcie-cards/ (visited on 03/28/2021).

[63] 3GPP, " Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; AT command set for User Equipment (UE)", 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 127.007, Mar. 2016, Version 13.3.0. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/127000_127099/127007/13.03.00_60/ts_127007v130300p.pdf.

[64] J. CAI, B. ZHOA, I. ZHANG, S. ZHU, J. GENG, and A. TANG, "Ec25 & ec21 at commands manual", Quectel, Tech. Rep., Sep. 2018. [Online]. Available: https://www.quectel.com/download/quectel_ec25ec21_at_commands_manual_v1-3/.

[65] W. WU, L. LIU, F. WANG, M. ZHANG, A. ZHANG, and K. ZHANG, "Ec25 hardware design", Quectel, Tech. Rep., Apr. 2018. [Online]. Available: https://www.quectel.com/download/quectel_ec25_hardware_design_v2-3/.

[66] *Ttl-232rg-vreg1v8-we*, https://ftdichip.com/products/ttl-232rg-vreg1v8-we/, 2021.

[67] *Opentap*, https://www.opentap.io/, 2021.

[68] *Ssh.net*, https://github.com/sshnet/SSH.NET, 2021.

[69] *Technical details*, https://github.com/google/AFL/blob/master/docs/technical_details.txt, 2021.

[70] *Aflfast*, https://github.com/mboehme/aflfast, 2021.

[71] *Aflgo*, https://github.com/aflgo/aflgo, 2021.

[72] *American fuzzy lop plus plus*, https://github.com/AFLplusplus/AFLplusplus, 2021.

[73] *Angora*, https://github.com/AngoraFuzzer/Angora, 2021.

[74] *Wireshark. go deep.* https://www.wireshark.org/, 2021.

[75] 3GPP, "LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification", 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 36.331, Jul. 2018, Version 10.22.0. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/136300_136399/136331/10.22.00_60/ts_136331v102200p.pdf.

[76] J. Balasch, B. Gierlichs, O. Reparaz, and I. Verbauwhede, "Dpa, bitslicing and masking at 1 ghz", eng, in *Cryptographic Hardware and Embedded Systems – CHES 2015*, ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 599–619, ISBN: 3662483238.

[77] P. Sperl and K. Böttinger, "Side-channel aware fuzzing", in *Computer Security – ESORICS 2019*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds., Cham: Springer International Publishing, 2019, pp. 259–278, ISBN: 978-3-030-29959-0.

[78] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices", Jan. 2018. DOI: 10.14722/ndss.2018.23176.

[79] *Raspberry pi os*, https://www.raspberrypi.org/software/, 2021.

[80] *Busybox: The swiss army knife of embedded linux*, https://busybox.net/about.html, 2021.

[81] *Vxworks*, https://www.windriver.com/products/vxworks, 2021.

[82] *Atmega328*, https://www.microchip.com/wwwproducts/en/ATmega328, 2021.

[83] *Esp8266*, https://www.espressif.com/en/products/socs/esp8266, 2021.

[84] J.-P. Lang, *Open source mobile communications*, https://projects.osmocom.org/, 2021.

[85] M. Eceiza, J. L. Flores, and M. Iturbe, "Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems", *IEEE Internet of Things Journal*, pp. 1–1, 2021. DOI: 10.1109/JIOT.2021.3056179.

[86] A. One, "Smashing the stack for fun and profit", *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[87] *N6781a two-quadrant source / measure unit*, https://www.keysight.com/be/en/support/N6781A/2-quadrant-smu-battery-drain-analysis-20v-1a-6v-3a-20w.html, 2021.

[88] Telit, "Le910cx hw user guide", Telit, Tech. Rep., Jul. 2020. [Online]. Available: https://y1cj3stn5fbwhv73k0ipk1eg-wpengine.netdna-ssl.com/wp-content/uploads/2020/11/Telit_LE910Cx_Hardware_Design_Guide_r26.pdf.

[89] J. Nagamalai, "Internet of things (iot) as interconnection of threats (iot)", in. Apr. 2016, pp. 3–21, ISBN: 978-1-4987-2318-3. DOI: 10.1201/b19516-3.