KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

The Rise of the Machines: On the Security of Cellular IoT with Intelligent Fuzzing

Milan Jacqmotte

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,
Ir. Rafael Cavalcanti,
Ir. Ilja Siroš

Academic year 2021 – 2022

Preface

First of all, I want to thank my promotors Bart Preneel and Michael Dieudonné from Keysight for their guidance and support. They gave me the opportunity to work on this thesis about IoT fuzzing which kept my interested at all time.

Moreover, I want to express my appreciation to my three daily supervisors Dave Singelée, Rafael Cavalcanti and Ilja Siroš for the weekly meetings and extensive feedback they provided. Their efforts made me able to achieve this thesis which I couldn't without their unconditionally support.

I want to express my gratitude to Keysight Technologies. Without their help my thesis result wouldn't be as it is now. Especially, I want to give thanks to Rafael Cavalcanti who made time to ship the hardware equipment to Barcelona and helped me unconditionally in configuring the setup and solving all kinds of issues. Also I wish to thank Germán Madueño for sharing his LTE knowledge and experience with me. Besides, my special gratitude goes out to Keysight Barcelona as they offered me an office and testing space when I was there for my Erasmus exchange.

The constructive remarks of Sofie Polin and Abuding Aishajiang during the midterm presentation helped me shaping my final goal wherefore I am very thankful.

I would like to thank my twin brother Thibo as well who supported me in exchanging ideas and discussing the different topics. Although he was also working on his master thesis, he was always helpful, listening and supportive.

And last but not least, I want to speak out my sincerely appreciation to my entire family who has been there all the time with their wholehearted help and support. With their aid I was able to finish my five years of study successfully.

Milan Jacqmotte

Contents

Abstract

The rise of Internet of Things (IoT) machinery in daily life puts more and more risk on the security and privacy of the entire society. Governed by the added-value of connected devices in different domains of the community, like health care, smart homes or Industry 4.0, the prospects are unanimous that IoT devices will even play larger roles in the upcoming future of daily life. However, taking into account that IoT devices often implement complex standards, are embedded systems with scare resources and hence mostly lack proper security mechanisms, the ideal confluence of circumstances are established for a wide range of devastating attacks. Until now, these attacks have been seen already wide-spread. Even worse, the reuse of IoT components in a broad range of products scales the number of affected systems in the ecosystem often to even millions, while patching discovered security vulnerabilities can be very tough and expensive. Therefore, this thesis extends a prior fuzzing framework which tests cellular IoT devices owing their connectivity to the 3GPP's LTE protocol. In particular, the overall fuzzing architecture is upgraded to allow the fuzz-testing to be done more autonomous, dynamic and efficient, while the fuzzer component integrates several techniques to craft packets more intelligently. By sending ill-crafted inputs to the target device, fuzz-testing tries to check as many parts of the implementation as possible to uncover any bugs present. During experimental evaluation of our fuzzing framework, a security vulnerability was found in a commercial modem, leaving it blind for external commands.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

AKA	A Key Agreement
AM	Acknowledged Mode
AP	Access Point
ARQ	Automatic Repeated Request
AS	Access Stratum
BCH	Broadcast Channel
BLSTM	Bi-directional Long Short-Term Memory
CCCH	Common Control Channel
CGF	Coverage-based Grey-box Fuzzing
CN	Core Network
CNN	Convolutional Neural Network
CIG	Code Integrity Guard
COSIC	Computer Security and Industrial Cryptography group
DCCH	Dedicated Control Channel
DCGAN	Deep Convolution Generative Adversarial Network
DDOS	Distributed Denial Of Service
DEP	Data Execution Prevention
DGF	Directed Grey-box Fuzzing
DL	Downlink
DL-SCH	Downlink Shared Channel
DRB	Data Radio Bearer
DRX	Discontinuous Reception
DSE	Dynamic Symbolic Execution
DTCH	Dedicated Traffic Channel
DUT	Device-Under-Test
EA	Evolutionary Algorithm
ECM	EPS Connection Management
ENPSFuzz	Evolutionary Neural Program-Smoothing-based Fuzzer
EPC	Evolved Packet Core

EPS	Evolved Packet System
ETM	Embedded Trace Macrocell
E-UTRAN	Evolved UMTS Terrestrial Radio Access Network
FSA	Finite-State Automation
FSM	Finite-State Machine
GEN	Graph Embedding Network
HSS	Home Subscriber Server
ICS	Industrial Control System
IoT	Internet of Things
LCID	Logic Channel ID
LSTM	Long Short-Term Memory
LTE	Long-Term Evolution
MAB	Multi-Armed Bandit
MAC	Medium Access Control
MBSFN-RS	Multicast-Broadcast Single Frequency Network Reference Signal
MCCH	Multicast Control Channel
MCH	Multicast Channel
MCTS	Monte Carlo Tree Search
MIB	Master Information Block
MITM	Man-In-The-Middle
MMU	Memory Management Unit
MOO	Multi-Objective Optimization
MSRM	Message Sending and Receiving Module
MTCH	Multicast Traffic Channel
MTMGM	Model Training and Message Generation Module
MTNN	Multi-Task Neural Network
NAS	Non-Access Stratum
NBMS	Multimedia Broadcast and Multicast Services
NN	Neural Network
OS	Operating System
PBCH	Physical Broadcast Channel
PCCH	Paging Control Channel
PCFICH	Physical Control Format Indicator Channel
PCH	Paging Channel
PCRF	Policy and Charging Rules Function
PDCCH	Physical Downlink Control Channel
PDCP	Packet Data Convergence Protocol
PDSCH	Physical Downlink Shared Channel
PHICH	Physical Hybrid-ARQ Indicator Channel
PRACH	Physical Random-Access Channel
PSO	Particle Swarm Optimization

PSS	Primary Synchronization Signal
PUCCH	Physical Uplink Control Channel
PUSCH	Physical Uplink Shared Channel
PUT	Platform-Under-Test
RACH	Random Access Channel
RAN	Radio Access Network
RAR	Random Access Response
RB	Resource Block
RLC	Radio Link Controller
RNN	Recurrent Neural Network
ROC	Robust Header Compression Standard
RRC	Radio Resource Control
RUE	Reference UE
SAP	Service Access Point
SDR	Software Defined Radio
S-GW	Serving Gateway
SIB	System information Block
SN	Sequence Number
SRB	Signaling Radio Bearer
SRS	Software Radio System
SSS	Secondary Synchronization Signal
SVS	Static Vulnerability Score
TF	Transport Format
TM	Transport Mode
TTI	Transmission Time Interval
UE	User Equipment
UL	Uplink
UL-SCH	Uplink Shared Channel
UM	Unacknowledged Mode
UTC	Universal Time
VAMAB	Adversarial Multi-Armed Bandit
VM	Virtual Machine
VoIP	Voice over IP

Symbols

S	Finite set of states
S_0	Initial state
Λ	Finite alphabet of input symbols
Θ	Finite alphabet of output symbols
T	State transition function
O	Output function
x	Vector of parameters
$f(\cdot)$	Objective function
$f_p(\cdot)$	Bug function of program p
$f_p^*(\cdot)$	Code coverage function of program p
$\hat{f}_p^*(\cdot)$	Smoothed code coverage function of program p
$R_i(\cdot)$	Equality or inequality constraint
$\mathbb{R}^n$	n-dimensional vector of real numbers
y	Ground truth of branching behavior
y_i	i-th ground truth of branching behavior
ψ	Machine learning model's weight parameters
$\hat{\psi}$	Optimal machine learning model's weight parameters
$L(\cdot)$	Loss function
$h(\cdot)$	Function modeling branching behavior
∇	Differential operator
$\mathbf{G}$	Gradient
e_i	Program edge i

Chapter 1

Introduction

In this chapter an introduction is given to the Internet of Things (IoT), followed by the goal of this thesis, its outline and the contributions made.

1.1 Background

The Internet of Things (IoT) is denoted as the networking technology where multiple heterogeneous embedded devices, also known as IoT devices, are interconnected and provide a wide variety of mostly autonomous tasks. In particular, these devices exchange data with each other using the Internet or any other type of communication network and often differ in their physical implementation (i.e., like software, hardware and sensors) and ability to interact with their environment. For example they can be full-fledged smartwatches or a simpler device for smart weather sensoring. Hence, due to the wide range of value-added services provided by IoT systems they can be currently found in almost every domain ranging from health care, smart cities, Industry 4.0, smart farming, telecommunications and so on. Most relevant to this thesis are the cellular IoT devices that leverage the standards of 3GPP [6] for connectivity with their environment. 3GPP developed several standardized cellular protocols suited for different applications which results in a heterogeneous landscape of IoT platforms. Hence, for cellular IoT systems supporting applications with high throughput connections 3GPP provides LTE [7] and older standards such as 2G and 3G. Power and resource constrained embedded devices, on the other hand, can benefit from NB-IoT [8] and LTE-M [9] implementations which have been designed to exchange throughput for battery efficiency and robustness with respect to coverage.

The total number of IoT devices grew about 8% during 2021 to 12.2 billion active connections despite the world-wide chip shortage and COVID-19 pandemic and thereby exceeding the number of active non-IoT devices such as computers, laptops and smartphones. Statisticians forecast that the trend will start to increase even more so that about 25.4 billion active IoT devices will be reached in 2030 [10]. Whereas IoT Analytics confirms further acceleration of growth in the coming years, their estimate is more optimistic achieving approximate 27 billion active endpoints already by 2025 [11]. However, the exact outcomes are not so important but instead

these numbers indicate the positive sentiment in the IoT market as IoT solutions are revolutionizing industrial and private contexts [11].

Despite the growing interest in the Internet of Things they mainly lack security and privacy features making them vulnerable to different types of attacks. Furthermore, due to the fact that more and more everyday machinery are becoming smart, the attack surface enlarges as they provide seemingly unconditional access from all over the world without the proper embedded security tools. Unsurprisingly, various attacks have already been carried out on IoT devices, e.g., the Mirai attack from 2016 infected around 2.6 million IoT endpoints to launch a distributed denial of service attack (DDOS) [12]. Similarly, Hajime and Reaper are other botnet attacks with dozens of victims in the IoT ecosystem [12]. To this end, Hassija *et al.* summarized the security and privacy issues in different IoT applications and investigated the security threads at different layers of the IoT system [13]. The major challenge in developing more secure IoT devices is that most embedded devices have scarce resources with respect to general-purpose computers. Hence, as pointed out by [14] these tiny embedded networks require efficient, robust and low-consumption cryptography and secure protocols, since common cryptography algorithms are not appropriate for these energy-efficient devices. Another issue in the security domain of IoT devices is that although standards and guidelines exist (e.g., ISO 27001 [15]), the security requirements are mostly described in a high-level due to the heterogeneity inside the IoT landscape [14]. Even worse, it is often difficult to patch a bug to widely deployed IoT devices using a firmware update, when a security problem is discovered, leaving the bug open in the wild. Especially, manufacturers have often poor firmware update mechanisms in place.

Cellular IoT devices follow a detailed protocol standard, like LTE, which defines precisely the set of the specifications and procedures on how the implementation should act. Therefore, flaws inside the standard's design, known as *specification issues*, leave an attack vector for possible exploits, [16]. Weak confidentiality algorithms defined in the standard, as an example, make it possible for unauthorized parties to read protect data and thus violate user privacy. Another problem resulting from an improper part of the specification pertains to LTE standard where the initial communication happens through unauthenticated messages. As a result, a phone can be mislead by a malicious base station due to the implicit trust it puts on the authenticity of the pre-authenticated traffic. Even more sever this vulnerability can be exploited by the malicious network to let the mobile phone reconnect to a downgraded network with less security in place (e.g., 2G or 3G) [16]. However, new releases of the 3GPP standard update the security features continuously to improve security and privacy. Apart from this type of flaw, an *implementation issue* refers to the IoT device having an implementation mistake and thus deviates from the specification [16]. The reason why implementation bugs are introduced inside IoT devices can be divergent. However, it often results from the complexity or any ambiguity inside the standard. For example, Chlosta *et al.* showed that some LTE implementations allowed unauthenticated Internet access or provided a shortcut around re-authentication after sending certain messages [17].

Common in the IoT market is to use inter- or intra vendor components to cut

costs. Vulnerabilities discovered in one IoT device may thus also impact other products resulting in even larger effects [18].

The prevalence of security vulnerabilities inside systems have researches encouraged to develop several techniques to detect bugs inside a target. In fact when a bug can be exploited by an attacker it turns into a security vulnerability which should be patched as quickly as possible to restrict its consequences. However, bug finding is not an easy task for which static and dynamic analysis tools have been proposed [19]. Whereas static tools inspect different components of a software without execution it, dynamic analysis tries to observe bugs when the application is running in its execution environment. Moreover, software vulnerability discovery techniques are evolving from manual reviewing and computer assisted dynamic and static monitoring tools to fully automated methods (like fuzzing) [20]. The drawback of most static and dynamic analysis tools is that they require the source code of the tested program which is of course not always available in IoT systems. In particular, cellular IoT devices can have parts of their implementation integrated in hardware as this increases the efficiency for critical processes. As a result, fuzzing is one of the very few vulnerability discovery techniques to test IoT implementations effectively. Since the inception of fuzzing in 1990 by Miller *et al.* [21], more and more research has been done in this field, as it is an easy to start technique that can automatically find bugs in a variety of target systems. It sends malformed inputs to the target device and tries to observe faulty behavior during execution. However, source code is not necessary but can be advantageous to drive the fuzzing process.

1.2 Thesis goal and structure

The goal of this thesis is to build an intelligent fuzzing engine that extends the wireless fuzzing framework of [18]. Hence, it is requested to construct malformed LTE packets with the help of intelligent techniques such as, e.g., AI or machine learning that try to test the target device efficiently and thoroughly. In summary, this thesis continues the research on intelligent fuzzing to improve the security of IoT devices taking into account their fuzzing challenges. As an example the different fuzzing methodologies developed in literature are researched helping the creation of a smart fuzzing engine. The focus of this thesis mainly lies in upgrading the overall architecture towards dynamism and full automation, and improving the IoT feedback as well as the design of a smart fuzzing engine inside the fuzzer component. Besides, the fuzzing framework is designed to target/hack cellular IoT systems and in particular broadband LTE modems (like the commercial Quectel EC25 LTE modem).

This project is in collaboration with Keysight Technologies and COSIC research group. COSIC (Computer Security and Industrial Cryptography group) is a research institution of the Department of Electrical Engineering at the Catholic University of Leuven leading the evolution in digital security solutions. They drive innovation in five different security domains, namely symmetric key cryptography, public key and protocols, embedded systems and hardware security, privacy and identity management, and finally mobile and wireless security [22]. On the other hand, Keysight

Technologies is an American company manufacturing software, and electronic test and measurement equipment. Examples of their products are power supplies, oscilloscopes, multimeters, wireless analyzers and security related solutions [23]. For more information about Keysight the reader is referred to Appendix A.

The outline of this thesis is composed of an introduction, six additional chapters and a conclusion, which is structured in the following way:

- **Chapter 2 — Preliminaries:** hands the reader the necessary background to understand the rest of the thesis. In particular, it explains the theory of Mealy machines, states fuzzing as an optimization problem, and introduces parts of the LTE protocol. For more information about the performed literature study Appendix C can be consulted.
- **Chapter 3 — Architecture:** discusses the overall fuzzing architecture which is extended from previous work [18]. Hence, given the architectural diagram a light is shed on the individual components that are required to fuzz the IoT device. Moreover, the architectural upgrades are discussed as well.
- **Chapter 4 — Methodology of ENPSFuzz:** details the methodology of the fuzzer block, named ENPSFuzz, that was only briefly presented in Chapter 3. Therefore, the challenges when it comes to protocol fuzzing are handled along with the techniques to make from ENPSFuzz a smart fuzzer. To this end, ENPSFuzz leverages an evolutionary architecture with several functional stages and integrates evolutionary as well as gradient-guided methods.
- **Chapter 5 — DUT's Feedback Mechanisms:** provides the feedback challenges to monitor device behavior in IoT fuzzing with respect to general-purpose systems. Second, the implemented solutions in ENPSFuzz are described.
- **Chapter 6 — Implementation and Experimental Results:** discusses the implementation of the fuzzer used during experimental attacks on an LTE modem.

1.3 Contributions

To end this introductory chapter the most important contributions of this thesis are mentioned:

- A literature study is made on the different types of fuzzers developed in literature (i.e., ranging from stateless program fuzzers to IoT fuzzers and machine learning techniques proposed). Moreover, through out thesis several challenges, confirmed by literature, regarding IoT protocol fuzzing are mentioned and addressed during the design of the smart fuzzing platform.
- Moreover, the fuzzing architecture is extended to allow for autonomous, dynamic and efficient fuzzing of cellular devices by given full control to the fuzzer component.

- The fuzzer component integrates several intelligent fuzzing techniques to solve fuzzing as an optimization problem. Therefore, evolutionary and gradient-guided approaches are combined into the fuzzer architecture.
- A new method is proposed to bridge the gap between advanced stateless grey-box fuzzers with source code availability and black-box IoT device fuzzing. In particular, a reference software is proposed that provides code coverage feedback and a method for differential fuzzing. This approach provides grey-box guidance mechanisms to test hardware implementations of cellular IoT devices.

Chapter 2

Preliminaries

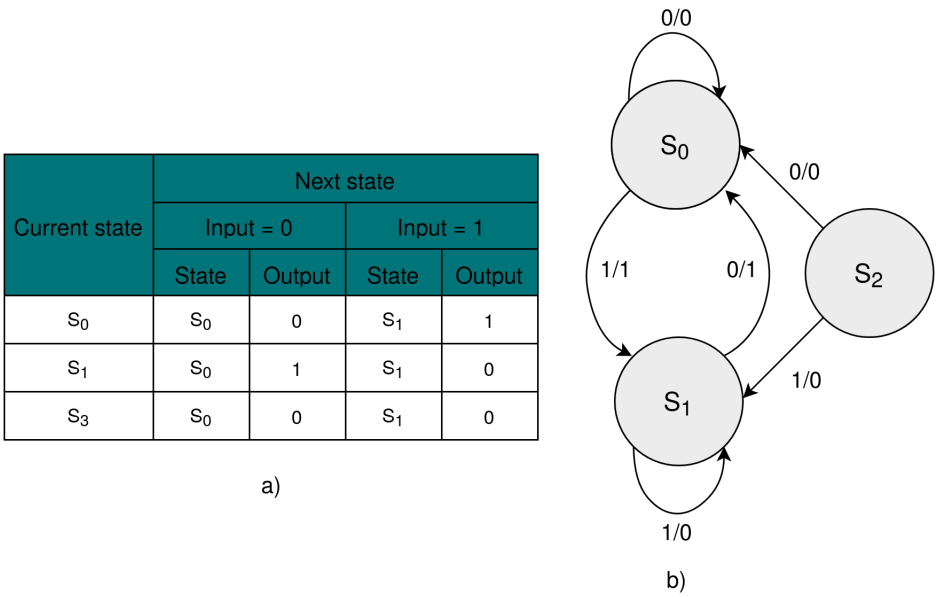
This chapter has as purpose to introduce the reader with the background concepts that are necessary for a good understanding of this thesis. Therefore, the three different sections cover each a fundamental part of the required knowledge. The first section gives some background in Mealy machines that are used in finite-state modeling of protocols. Next, fuzz-testing is formulated as an optimization problem for which two separate techniques have been proposed in research: an evolutionary algorithm and a gradient-guided method using neural program smoothing. Both solutions will be handled separately. To end the chapter, an LTE overview will be given covering its architecture, the protocol stack and how a mobile should communicate with the LTE network. Furthermore, a literature review on fuzzing research can be found in Appendix C.

2.1 Mealy machine

A Mealy machine is a type of finite-state model (FSM) that corresponds to an abstract mathematical diagram with several states and state transitions. In particular, a finite-state model is characterised by a set of states, corresponding transitions and an initial state. Hence, the model starts always from the initial state and transfers to a subsequent state in response to some external event, also known as a state transition. Moreover, finite-state models can be located in only one state at the time and are often used to model communication protocols [24].

The concept of a Mealy machine was invented in 1955 by George H. Mealy and has been used extensively for modeling state diagrams in several applications such as protocol implementations. In fact, Mealy machines are finite-state machines where the output is governed by the current state and the current inputs to the model. On the contrary, Moore machines output a result depending only on the current state. Furthermore, as for each state and each input only one transition is possible, they are also denoted as a deterministic finite-state transducer. More formally, a Mealy machine can be described following a 6-tuple $(S, S_0, \Lambda, \Theta, T, O)$, where [24]:

- S is a finite set of states



Current state	Next state			
	Input = 0		Input = 1	
	State	Output	State	Output
S_0	S_0	0	S_1	1
S_1	S_0	1	S_1	0
S_3	S_0	0	S_1	0

a)

b)

Figure 2.1: An example of a Mealy machine with three states. Part a) denotes the FSM's table, whereas b) shows the actual graph.

- S_0 is the initial state belonging to the set S
- Λ is a finite alphabet of input symbols
- Θ is a finite alphabet of output symbols
- T is the state transition function mapping pairs of an input and a state to the corresponding next state: $\Lambda \times S \mapsto S$
- O is the output function mapping pairs of an input and a state to the corresponding output: $\Lambda \times S \mapsto \Theta$

Figure 2.1 shows a Mealy machine with three states.

2.2 Fuzzing

In this section, fuzz-testing is introduced as an optimization problem for which two different methods exist, that try to solve the underlining problem. The first fuzzing method uses an evolutionary algorithm (EA) to search iteratively for the optimal solution, whilst the second approach implements a gradient-guided optimization technique using neural program smoothing. The ultimate goal of both methods is to find bugs inside the target application.

2.2.1 Fuzz-testing as an optimization problem

The statement of an optimization problem consists of three different elements, as can be seen in Equation 2.1, namely an objective function $f(x)$ and some equality

or inequality constraints $R_i(x)$ that are both conditioned on a vector of parameters x. Hence, the objective in an optimization problem is to find a vector of variables x that minimizes or maximizes an objective function $f(x)$, while at the same time satisfying a set of constraints $R_i(x)$. A mathematical formulation of an optimization problem can be written as:

$$\begin{aligned} &\min_x / \max_x && f(x) \\ &\textit{subjected to} && R_i(x) \leq 0,\ i = 1, ..., k \\ &&& R_i(x) = 0,\ i = k+1, ..., k+l \end{aligned} \tag{2.1}$$

where the vector of variables x belongs to $\mathbb{R}^n$. Applied to fuzzing, the underlining problem can be formulated as an unconstrained optimization problem, where the purpose is to maximize the number of bugs found in the target program during the execution of a specified number of inputs. As can be stated differently, the fuzzer has to supply the test software with test inputs that evaluate the objective function of the program-under-test $f_p(x)$ to one. This means that the execution of the individual test case x triggers a bug or vulnerability inside the software p. Although seeming straightforward, solving the aforementioned objective function for x has to cope with unsatisfying results due to the ill-conditioned function $f_p(x)$. In fact, the function consists mostly of plateaus with locally, at the bug positions, some sharp transitions [25]. This is illustrated in Figure 2.2 with the blue dirac deltas. As a result, literature [25, 26, 27] rephrases the actual problem into the task of obtaining program inputs that maximize the total amount of code exercised (e.g., in terms of edges). As an example, so-called code coverage-based gray-box fuzzers try to find new inputs that trigger new parts of the code. These coverage-based fuzzers make use of an evolutionary approach, which is detailed next subsection. In essence, the updated objective function $f_p^*(x)$ indicates the amount of code activated (e.g., number of triggered edges) by the test input x for the program p. This results from the one-to-one mapping between the series of input bytes and their corresponding code behavior (i.e., code blocks, functions, etc.). It turns out that the reformulated problem can be optimized more efficiently, since more program inputs are characterised by new code coverage behavior than they satisfy the initial bug condition [25]. Figure 2.2 shows the code coverage attained for every input of the program p. This is modeled by the function $f_p^*(x)$. Note the different plateaus present in the objective function.

In general, optimization methodologies, aiming to construct a solution to the optimization problem, start with a primary value for x, which is gradually improved. Several approaches can be distinguished based on the pieces of information they leverage to craft their possible solution. The guidance information can come from several sources: values produced from evaluating the objective function $f(x)$, the imposed constraints $R_i(x)$ and the gradient or higher-order derivatives [25]. What follows is an explanation on the techniques used in fuzz literature for solving the underling fuzzing problem of finding bugs.

2.2.2 Evolutionary algorithm

One popular approach to solve the updated fuzz problem is by means of an evolutionary optimization algorithm, that tries to maximize their fitness function (i.e., coverage information) by refining test inputs iteratively. Most coverage-guided gray-box fuzzers leverage this optimization technique [28, 29, 30]. The process of finding solutions on basis of an evolutionary algorithm starts with an initial seed pool containing one or more seeds. Here, each seed represents an individual input to the test program. Thereafter, the evolutionary optimization method performs several fuzzing loops, called iterations. During each iteration of the fuzzing process, a seed is selected from which subsequent test cases are derived by applying (random) mutations. Next, the new test cases are executed and hence the interesting ones that score well on the fitness function (i.e., uncover new code paths) are retained, and appended to the seed corpus for further manipulation in the near future. This process is repeated until the maximum number of test cases are executed or the predefined time budget is exhausted. Hence, the final population is returned and serves as the obtained solution for the fuzzing problem.

2.2.3 Gradient-guided algorithm with neural program smoothing

When solving the unconstrained fuzzing problem to expose bugs, the evolutionary guidance has been criticized for having difficulties in finding hard-to-trigger bugs and for leading to unsuccessful exploration of new coverage later on in the fuzzing process (i.e., wasting resources at inefficient mutations). Although, they demand less effort and are easier to implement [4, 25]. Hence, as stated by [25] gradient-guided approaches (e.g., gradient descent) can outperform evolutionary algorithms at solving high-dimensional structured optimization problems in diverse fields. In particular, they do take into account the underlining structure by leveraging gradients or other higher-order derivatives. Moreover, they investigated that with this approach the fuzzer can converge more effectively to the solution and hence find bugs. However, as already touched upon before the likelihood and the efficiency of optimization algorithms in converging to the optimal solution depends on the smoothness of the objective function $f(x)$. While the code coverage-based optimization problem improved the function's conditioning, gradient-guided methods for fuzz-testing still suffer from the discontinuous branching behavior (i.e., plateaus and ridges) of real-world programs. Due to these discrete branching irregularities the gradient-guided techniques are not able to calculate the gradients accurately. As a result, these methods will likely become stuck at a certain point in time, as seen in Figure 2.2. As a consequence, the authors of Neuzz [25] and MTFuzz [31] introduced neural program smoothing to overcome this problem. Especially, program smoothing entails the creation of a smooth, meaning in essence differentiable, surrogate function that approximates the discrete code behavior of the tested program with respect to the input bytes. This smoothed function is illustrated in Figure 2.2 as well. For this task, neural networks (NNs) can be powerful in learning the target program's branching behaviors (i.e., using edge coverage information) to deal with the high-dimensional

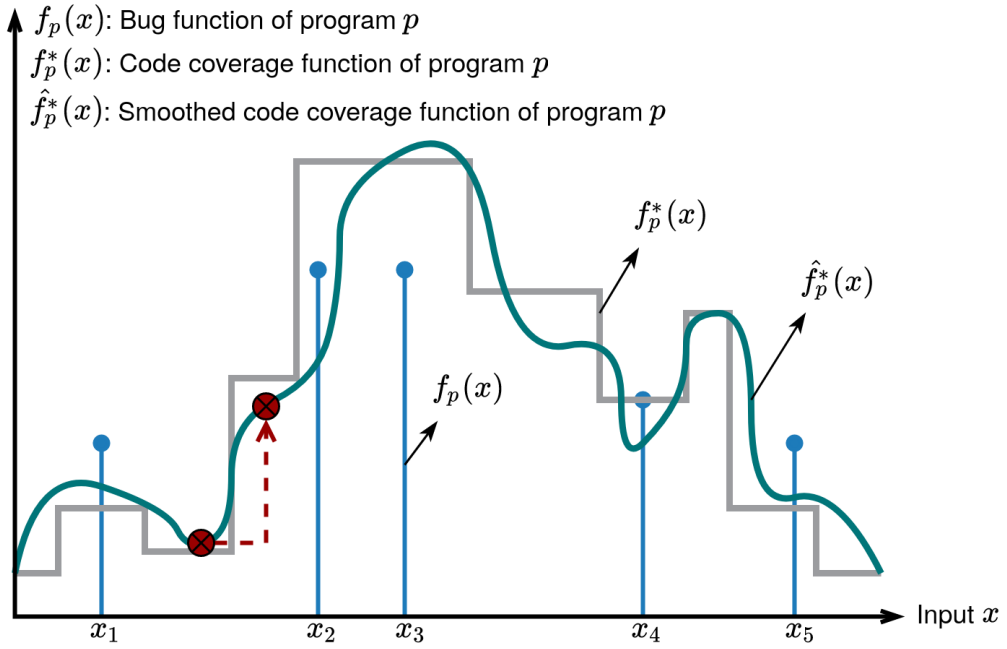


FIGURE 2.2: The three types of objective functions for fuzzing as an optimization problem: (1) the bug function, (2) the code coverage function, and (3) the smoothed code coverage function. The more bugs triggered by an input x_i, the larger its dirac delta height. Note the discontinuities in the first two. The red dotted line indicates a gradient-guided optimization for the smoothed objective function.

optimization problem task. Both MTFuzz [31] and Neuzz [25] utilize program-based smoothing by way of neural network models to make gradient-guided optimization possible. In essence, they designed a neural network that learns the relation between the input bytes and the corresponding activated code. Hence, with this approach the smoothed function can be differentiated to guide the fuzzing process with the help of gradients. The optimization process can, thus, perform better on the updated objective function in finding new coverage. In contrast, prior program smoothing techniques leveraged heavyweight symbolic analysis that could incur considerable performance overhead [25]. The dotted red line from Figure 2.2 illustrates the gradient-guided optimization for an initial input.

2.3 Long-Term Evolution (LTE)

As generally known, Long-Term Evolution (LTE) is a complex framework which has as goal to achieve connectivity between several network nodes. This section tries to make the reader familiar with the most important concepts and abbreviations that exist in the LTE protocol. The first section starts with the LTE architecture, followed by a paragraph about the different protocol layers present in the LTE stack.

Finally, several LTE procedures are discussed that happen between the mobile device and the base station work.

2.3.1 LTE network architecture

In this section the overall system architecture of LTE is introduced to the reader with the relevant details of the functional entities within the LTE framework. As illustrated in Figure 2.3, LTE's high-level architecture is composed of three main elements, namely the core network (Evolved Packet Core, EPC), the Radio Access Network (RAN, also called Evolved UMTS Terrestrial Radio Access Network or E-UTRAN) and the user terminal (User Equipment, UE). Moreover, the Radio Access Network and the Evolved Packet Core together are denoted by 3GPP as the Evolved Packet System (EPS). Despite evolved from the UMTS standards, LTE follows a somewhat different paradigm, visible in its architectural structure, shown in Figure 2.3. In particular, 3GPP simplified the EPS architecture and integrated a fully packet-switched core network for throughput and latency enhancements, and improved mobility between different access networks, such as E-UTRA, other 3GPP systems (e.g., UTRAN) and non-3GPP access technologies (e.g., Wi-Fi and CDMA2000). Furthermore, the LTE architecture is composed of different functional entities where each network node encompasses similar features provided by 4G. Compared to UMTS the reduction in architectural complexity is a result of removing and combining these functional network entities without stating suppositions for the implementation of these global nodes [32].

The EPC provides the LTE radio access with the core (administrative) functionalities within the mobile-broadband network and hence is also denoted as core network (CN). The main tasks for the core network within the LTE architecture is to provide management functions like security control (e.g., identity verification), network access control, mobility management and charging of users, in control-plane, and routing traffic (to the Internet) in user-plane. As mentioned before, the core network realizes this by an all-IP packet-switched topology for both data as well as voice traffic, while older 3GGP standards, such as 3G, still use circuit-switched networks for handling calls. The major benefit of the decoupling between radio access and core network in LTE is that different mobile technologies can be served by one CN. To accomplish its role within the overall structure the EPC is split in the following functional nodes: MME, HSS, S-GW, P-GW and PCRF [32].

The mobility management entity, or MME for short, is the most essential part in the EPC responsible for the control-plane signaling and hence is, for example, concerned with management of connections to the network, network resources, and mobility handling to support tracking, roaming, paging and handovers. Moreover, it regulates all control-plane functions regarding the subscriber (e.g., authentication) and its current session through the Non-Access Stratum (NAS) layer which tunnels control information between the UE and the MME. In contrast the Access Stratum (AS) delivers functionalities between the RAN and the user [32].

Connected to the MME is an administrative database, named Home Subscriber Server (HSS), which acts as an information source about the network's subscribers.

Details necessary for proper operation of LTE and contained in the HSS are related to: mobility management (e.g., location estimates), subscriber authentication (e.g. security keys and identifiers), etc. [32]

The third component in the core network is the Serving Gateway (S-GW). It serves as anchor point between the radio access and core networks where user-plane signaling is passed through. This means that not only the E-UTRAN, but also other 3GPP cellular radio access technologies (e.g., GSM), can connect to the EPC via the S-GW. Besides passing user data between the radio- and core network, this node also functions as a mobility anchor point in user-plane for both inter-eNB handovers and mobility between LTE and other 3GPP technologies, such as GSM or GPRS. Furthermore, billing of subscribers is done based on the gathered information in this component [32].

Next, the Packet Data Network Gateway, P-GW, acts as an interface for routing data between the CN and the Internet where the imposed Quality of Service rules from the PCRF are applied and subscribers' IP address are administered. It also serves as a mobility anchor point between 3GPP and non-3GPP technologies, like CDMA2000. Policy and Charging Rules Function (PCRF) is the last functional node to be descripted in LTE's core network. The purpose of this entity is to charge the subscribers and supervise the Quality of Service rules implemented by the S-GW [32].

After having given a brief overview of the EPC, the Radio Access Network (RAN) of LTE's architecture needs some further analysis. The purpose of the E-UTRAN is to provide terminals or User Equipments (UEs) with cellular communication access to the core network and is therefore made up of several base stations, which are also known as eNodeBs or eNBs for short. In the overall structure the RAN handles in essence all radio-related functions towards its subscribers such as, but not limited to radio-resource management (e.g. access control), routing user- and control-plane data, mobility management, and implementing cellular communication protocols (using e.g. antennas). To cover the geographical area in an efficient way, each eNodeB is assigned one or more regions, called cells, wherein the active located subscribers are managed by the particular base station in charge. To fulfill its functionalities the eNB uses the S1-MME interface with the MME for control-plane communication and connects via the S1-U with S-GW for user-plane packets to be sent or received. Here, the protocols between the mobile terminal and the EPC are denoted as Non-Access Stratum (NAS) protocols, while the protocols between the UE and the eNB are know as Access Stratum (AS). Last, there is a X2 interface interconnecting eNodeBs which has, for example, a role in coordinating handing-over of UEs between cells served by different base stations. Moreover, downlink interference to a particular subscriber receiving signals from two transmission points can be mitigated with the X2 communication [32].

2.3.2 LTE air interface protocol stack

This part discusses the different layers in the LTE air interface between the UE and the eNodeB, starting from the physical layer and working the way up in the protocol

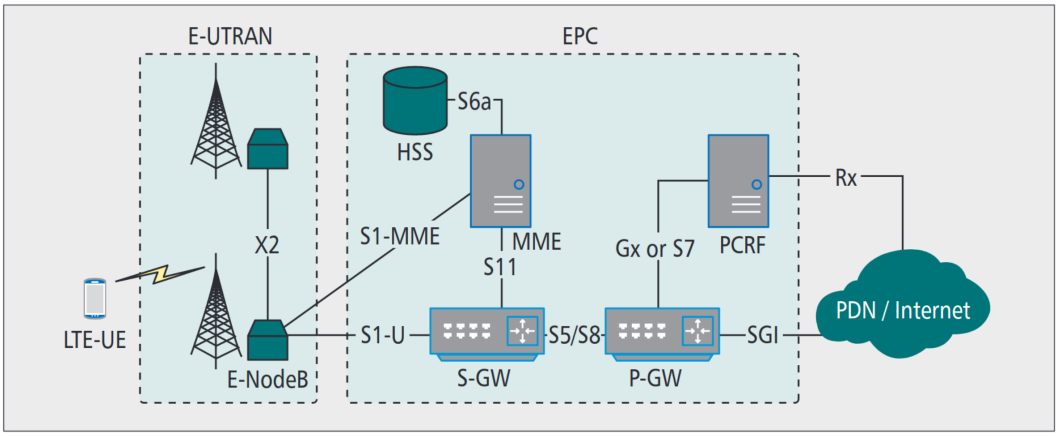


Figure 2.3: The network architecture of LTE [1].

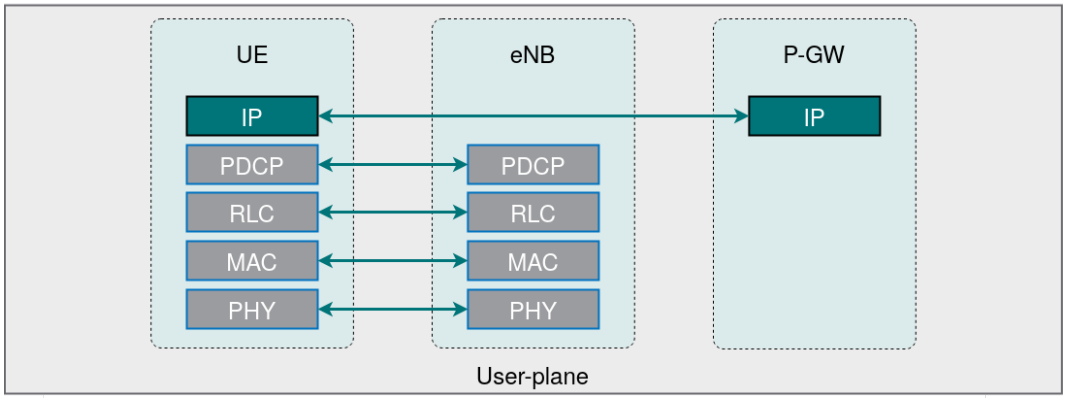


Figure 2.4: The user-plane protocol stack in LTE.

stack. As already mentioned in the paragraph about the system architecture LTE defines a logical separation between user- and control-plane signaling which is also visible in the access network's protocol stack. While Figure 2.4 shows the layers for user-plane, Figure 2.5 illustrates a similar view for control-plane, except for the addition of two extra layers, namely the Radio Resource Control (RRC) and Non Access Stratum (NAS).

The lowest layer of the LTE stack is the physical layer which provides several functionalities to the upper layers such as modulation, coding, error detection and correction, multi-antenna processing, physical-layer hybrid-ARQ processing and assignment of signals to the suitable time-frequency slots. In more detail these services are visible to the MAC layer in form of transport channels where each transport channel define how user- and signaling data are transmitted over the air (e.g., interleaving and encoding properties). As seen in Figure 2.6 the physical layer takes care of the mappings between the physical channels and the transport channels in both uplink and downlink. In contrast with the transport channels a

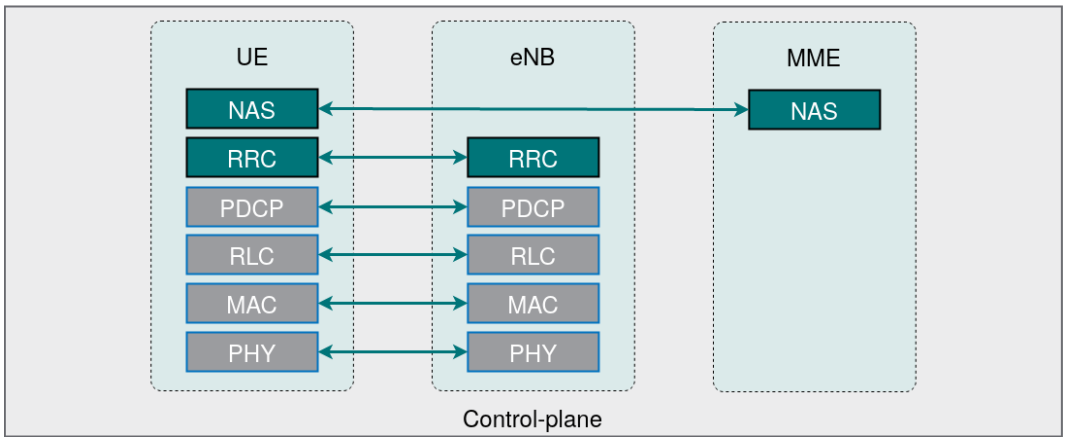


FIGURE 2.5: The control-plane protocol stack in LTE.

physical channel describes the set of frequency-time slots or in other words resource blocks (RB) on which particular data is transmitted in downlink or received in uplink. This means that each physical channel is assigned to several slots of a so called LTE frame structure, given in Figure 2.7. However, not all physical channels have a corresponding transport channel. The functions of these channels without a corresponding transport channel are on the one hand to deliver downlink control information, enabling the device to properly tune itself to receive and decode downlink data streams, and on the other hand to provide uplink control information aimed at informing the hybrid-ARQ protocol and scheduler at the RAN side about the mobile terminals' status. Hence their name: L1/L2 control channels. An enumeration of the physical channels and their respective functions can be consulted in Appendix B.

One layer above, the Medium Access Control (MAC) layer is situated both in the mobile user and in the base station, however, within the latter having some extra functionalities. Similarly to the physical layer providing some low-level services like HARQ feedback to the MAC layer, the MAC layer in turn serves the layers above with several functionalities in the form of logical channels. While a transport channel, as mentioned before, describes in which way and with what properties the signals are transmitted over the air, logical channels instead are subdivided depending on what type of data they convey (e.g., broadcast info, control signaling, etc.). Moreover, at each transport channel data is grouped into transport blocks of whom only one block of variable size can be put on the air interface during every transmission time interval (TTI) by either the eNB or a device. Whenever spatial multiplexing (MIMO) is available, this number increases to at most two transport blocks each TTI. By assigning a transport format (TF) to each transport block the MAC layer can differentiate on how a particular block should be transferred on the cellular interface. This means that a transfer format describes the following characteristics: the size of the block, the antenna-allocation, and the modulation-and-coding scheme [32]. To explain the tasks of the MAC layer in LTE it can be separated in the following functional parts: a HARQ component, a multiplexing/demultiplexing entity, a logical

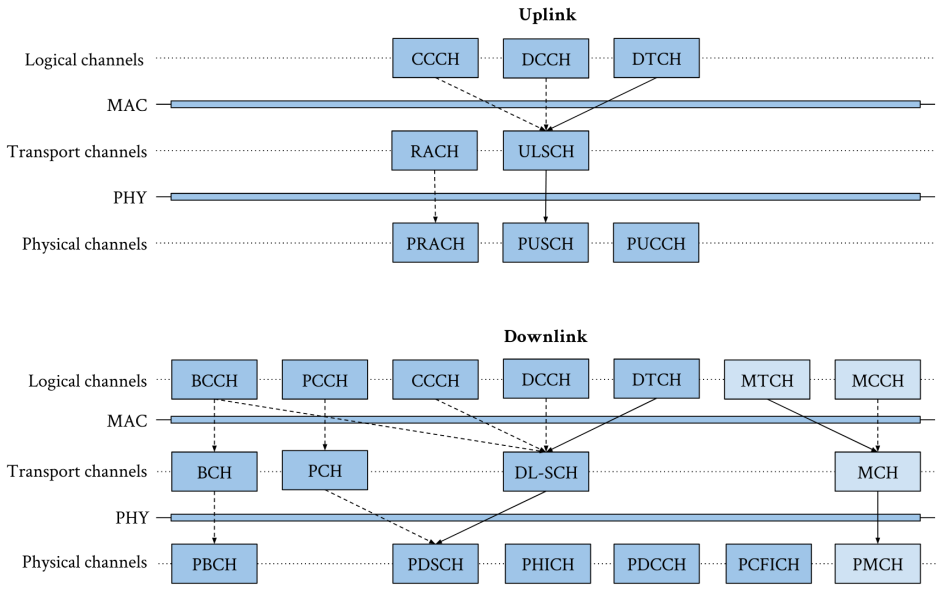


Figure 2.6: Logical, transport and physical channels in up- and downlink with corresponding layers. Light blue boxes are multicast channels. Dashed lines are routes purely for control information. A discussion about the different channels can be found in Appendix B [2].

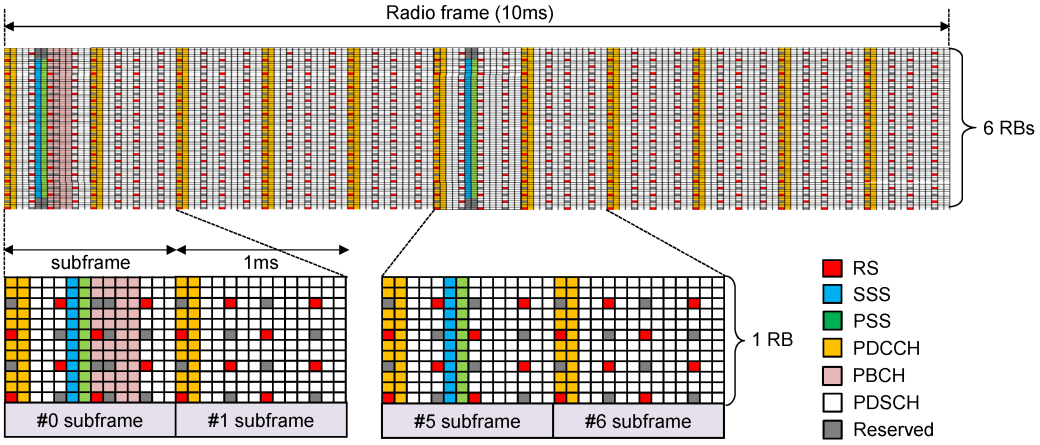


Figure 2.7: Illustration of how different physical channels are mapped to the LTE frame for over-the-air transmission. Here, the network configures the assignment of physical channels to resource blocks [3].

channel prioritization block, a random access control component and a controller (which is not discussed further). First, the HARQ entity has a transmit part that sends transport blocks and waits for the corresponding acknowledgement/negative-acknowledgement to come in. Upon reception of a NACK or after the predefined time-out without response the HARQ process tries to retransmit the block towards the device/base station. The receiving HARQ on the other hand takes care of waiting on, processing and reassembling of the collected data, while sending the corresponding ACK/NACK messages. Since this stop-and-wait operation can prohibit continuous communication of incoming and outgoing packets several parallel HARQ processes can be outstanding at any given moment in time, resulting in a multi-process stop-and-wait HARQ protocol. Second, the multiplexing and demultiplexing component performs the mapping between the different logical channels and the transport channels. Note that data units coming from the upper layer are called PDUs in LTE, whereas data chunks sent to the layer below are denoted as SDUs. In order for the MAC sublayer to convert MAC PDUs to MAC SDUs the multiplexer takes data from different logical channels according to the priority scheduler and transfers it to the correct transport channel, whereas the demultiplexer at the receiver hands SDU packets over to corresponding RLC entities (i.e., the RLC layer is situated below the MAC) in reverse operation. With each of these logical channels a RLC sublayer entity is associated to make a distinction between the priorities of the channels. Furthermore, the MAC sublayer adds a MAC header to all packets at the transmitter to make demultiplexing possible at receiver side. In particular, within this header a subheader is contained for each RLC data unit with its logic channel ID, (LCID) indicating from which logical channel the respective PDU originates, together with the size of the packet in bytes. Third, as already briefly hinted at before, the logical channel prioritization scheduler enforces a priority to all logical channels by deciding on the amount of data that's extracted from each RLC entity queue into a new MAC PDU. Thus, providing a way of quality of service. Last, the random access controller is in charge of handling the random access procedure [32]. However, note that the base station's MAC layer differs with the UE's in the following ways. In uplink the eNodeB reserves radio resources for requesting subscribers by scheduling the transmission of connected subscribers on the PDCCH and hence needs to process the received data from several UEs. In fact, this uplink scheduling of transmitting devices is regulated based on several criteria, such as the bandwidth of the cell, the number of active UEs, the channel-state report from the cellular device's MAC, the (advanced) coding-and-modulation scheme used by the UE, etc. In downlink, on the other hand, the base station needs to communicate with several UEs and send out grants for these uplink resources. When a particular resource blocks is granted to the UE by the scheduler it is allowed to transmit for L subsequent transmissions and receptions, where L is configured by the Radio Resource Control layer, explained below. As a result, the base station's sublayer is more complex [33]. The transport channels can also be found in Appendix B.

The Radio Link Controller (RLC) layer is situated above the medium access control and below the packet data convergence protocol layer. Whereas the communication with the MAC sublayer is done via the standardized logical channels,

whom are distinguished in what type of data they transmit, the RLC uses Service Access Points (SAPs) with its layer above. The tasks of the RLC sublayer are carried out by the different RLC entities, which are controlled by the RRC layer, discussed below. In particular, for each one-to-one mapping between an upper layer radio bearer and a logical channel an entity is made active for transforming PDCP SDUs into PDUs for the MAC sublayer when transmitting, and a reverse entity for the opposite direction. In addition to the active RLC entities in the base station, the corresponding (inverse) entities are also present within the UE. Depending on the transmission mode configured in the RLC entity, it can execute one or more of the following functionalities. First, the RLC protocol segments or concatenates PDCP data into the appropriate size. In fact, the RLC PDU size can be changed dynamically in function of the data rate: for high rates small overheads are incurred in case large RLC PDU packets are dispatched, while on the contrary small payloads are more efficient for low rates. Besides this function, LTE's specification states that the RLC sublayer adds a header to all its PDUs containing information such as segmentation and concatenation information, and a sequence number (SN) used for retransmission and delivery of packets in-sequence to the upper layers. In fact, the retransmission protocol of the RLC layer operates between each pair of receiving-and-sending entities and is in charge of solving errorneous communication like missing PDUs. Whenever the receiver identifies absent SNs within its buffer, the transmit entity is notified and takes the appropriate action (e.g., retransmit) based on the received status report coming from the receiving entity. Note that despite the RLC also having the capabilities to correct faults due to, for example, noise or sudden channel variations, this error-free service is often already provided by the hybrid-ARQ mechanism of the physical and MAC sublayer mentioned before. Another task made available by the layer's entities is to reassemble and put out-of-order MAC PDUs back in the same order as they were initially transmitted, for simplifying PCDP's processing. Similarly, RLC detects and removes duplicate packets by inspecting the received sequence numbers in the headers. The three different modes of operation in which a RLC entity, serving a radio bearer, can be located are [34, 35, 33]:

1. **Transport Mode (TM).** In this mode the RLC sublayer does not provide any of its functionalities, as data is just passed through to the next layer. In particular, at the transmitter the received RRC messages are buffered as RLC PDUs and read by the MAC instance, whilst lower layer data is directly passed on to the RRC upper layer. However, within the LTE framework transport mode is restricted to some logical channels such as the broadcast control channel (BCCH), the PCCH with paging information and the common control channel (CCCH) transporting RRC connection procedure signaling.

2. **Unacknowledged Mode (UM).** Except for the retransmission mechanism UM provides all RLC functionalities and is thus used for error-tolerant but delay-sensitive messaging. Typical examples are real-time applications like video streaming, voice over IP (VoIP) and interactive gaming which do require in order reception and duplicate detection.

3. **Acknowledged Mode (AM).** It is equal to the unacknowledged mode supplemented with data acknowledgements. For this a polling bit is periodically set in the RLC header requesting from the receiver to transfer a control PDU, stating the correctly received data units. Hence, from the RLC status report the transmitting entity knows which of the outstanding PDUs can be discarded from the retransmission buffer and which PDUs should be resent. This mode, thus, suites best error-sensitive but delay-tolerant traffic flows like emails, software downloads, etc.

Moreover, the logical channels are described in Appendix B.

Next, the Packet Data Convergence Protocol (PDCP) sits at the top of the protocol stack in user-plane and is provided with the lower layers' services. The PDCP itself compresses IP headers of user-plane SDUs with the robust header compression standard (ROCH) to reduce the number of bits to be transmitted over the air interface. Furthermore, the sublayer takes care of ciphering outgoing user-plane data, making it impossible for an outsider to sniff on the communication traffic, and protects all control-plane information for integrity. As a result, the receiving side can be sure that control content stems from right source and there has not been tampered with the content itself, while traveling over the air channel. In addition, at the receiver the PDCP executes the inverse ciphering and compression tasks. Another task for the PDCP is when a intra-eNodeB handover of a device takes place between base stations. Here, duplicate handover messages and out-of-order receptions of these packets are mitigated, likewise the PCDP sublayer of the old eNB transfers its undelivered data to the new eNB, while at the same time the UE's PDCP layers retransmits outstanding uplink packets to new base station. In the architecture of PDCP two types of radio bearers exist, namely data radio bearers (DRBS) for user-plane data and signaling radio bearers (SRBs) used for control-plane input. As these radio bearers differ in quality of service, IP packets are mapped to the relevant bearer depending on requirements of the encapsulated data. To make the connection with the lower RLC layer, each bearer has a PDCP entity assigned and subsequently this entity is attached to one or two RLC entities determined by the RLC mode and, whether the bearer is unidirectional or bidirectional. Finally, on top of the PDCP layer in user-plane common networking protocols can be placed (e.g., TCP or IP). However, the LTE standards define for the control-plane two additional control layer protocols, namely radio resource control (RRC) and non access stratum (NAS) [36, 35].

As illustrated in Figure 2.5, control-plane messages from the network are either supplied from the eNodeB by means of the radio resource control protocol or from the most upper layer in the stack, the non access stratum, connecting the UE with the MME inside the core network. The lowest sublayer from these two, called the Radio Resource Control (RRC), accomplishes several control functions and procedures, that are also explained more elaborate in the upcoming paragraph, within the E-UTRAN framework. First of all, from a high-level point of view the RRC is in charge of configuring and coordinating lower layers in their operations and is hence an interesting target for fuzz testing. Second, the control layer above can make use of,

among others, RRC's broadcast service for common control information, its service to inform idle-state UEs and the dedicated control information service. Before a UE can connect to the LTE network, it needs some specific system information. Hence, the role of the eNB's RRC sublayer is to broadcast system information in the different cells on a regular basis. Depending on the current state of the device with respect to the network it listens to distinct data within the received system information. For example, for UEs in RRC_IDLE state the cell selection/reselection parameters and neighboring cell information are important to select the power thresholds, downlink channel bandwidth and other related network configurations. In E-UTRA the system information is divided over two types of logical formats, that is the Master Information Block (MIB) and several System Information Blocks (SIBs):

1. The **master information block (MIB)** contains the most important data needed for a UE to connect to a cell such as the system frame number, PHICH configuration and cell's downlink bandwidth. Moreover, with this block crucial physical layer information is received to decode additional system related messages within the cell. Since the standards define a fixed scheduling for the MIB of 40 ms on the BCCH and next the BCH, all devices know where to look.

2. The **system information block 1 (SIB1)** includes the scheduling of other SIBs and further information with which the UE can asses if it's allowed to connect to the cell. This means that the SIB1 conveys information, among others, related to the operator(s) of the cell, the cell identity and status, and UE requirements (e.g. minimum receiver level). Similarly to the MIB, this system information block follows a fixed scheduling, but with a period of 80 ms and dynamically forwarded on the DL-SCH after the BCCH.

3. The **other system information blocks (e.g., SIB2-SIB20)** carry diverse broadcast information related to neighboring cells for cell reselection, GPS data, coordinated universal time (UTC), earthquake and tsunami notifications, MBMS traffic, interworking of LTE and WLAN, etc. The SIB messages, starting from SIB2, are first mapped to the BCCH and then transmitted through the DL-SCH, where the different SIBs can be distinguished with a special system-information RNTI (SI-RNTI). As already described, the scheduling of these system information blocks is flexible, and indicated in SIB1. With SIB1 and SIB2 together with the master information block enough data is acquired to start the connection procedure.

Additionally, whenever the content of a system information changes, the UEs in RRC_IDLE and RRC_CONNECTED are notified by the RRC sublayer through a paging message. Besides the broadcast functionality, upon request from the MME the radio resource control protocol can send a paging message to the device in RRC_IDLE state in case it should receive downlink data. Another RRC task is to support mobility procedures for its subscribers. Meaning that the serving eNB can instruct the mobile terminal to start, stop and modify physical layer measurements

with as purpose network-controlled UE handovers conducted by the RRC protocol. LTE defines several mobility processes as, for example, cell selection, cell reselection, handover, and redirection. Last, the RRC connection management entity handles one of the most paramount responsibilities that is to say the establishment, modification, maintenance and release of connections between the RAN and its mobile devices. Without a proper connection setup and its preservation it is not possible to have user data and control communication going on, which is the essence of the air interface. The RRC connection establishment procedure, discussed below, creates an RRC context between the UE and the RAN, set ups bearers, configures several parameters and changes, when done successfully, the state of the UE from RRC_IDLE to RRC_CONNECTED. As a result, the connected subscriber is now able to retrieve data from the control channels and the shared data channel, while RRC_IDLE devices only are supposed to scan paging messages and system information blocks. Furthermore, while idle state users don't have associated SRBs or DRBs, connected devices do have them. In particular, as RRC (and NAS) messages are part of the control-plane, they are transported by means of established signaling radio bearers determined by the used channel and state of the connection:

1. **SRB0** serves the transfer of RRC messages in the common control logical channel.

2. **SRB1** is used by RRC messages, piggybacked NAS messages and regular NAS messages that are intended for the dedicated control logical channel ahead of the SRB2 establishment.

3. **SRB2** has a lower priority than the signaling radio bearer 1 and transports NAS messages mapped to the dedicated control logical channel, whilst being created by the E-UTRAN after security activation.

Note that SRBs are always established before the creation of data radio bearers and almost all signaling and data radio bearers (i.e. SRB1, SRB2 and DRBs) are protected against eavesdropping by LTE's ciphering algorithm, although only signaling radio bearer 1 and 2 have integrity protection in place. SRB0, as an exception, can be modified and spied on by a third-person without the receiver nor the sender being aware [37, 35].

On top of the control-plane the Non Access Stratrum (NAS) layer is situated spanning from the top of the mobile device's stack to the MME located in LTE's core network. Hence, it forms a bidirectional communication link between the user equipment and its serving mobile station. In general, the functionalities of the NAS sublayer can be classified into two types of protocols, namely EPS mobility management, EMM for short, and EPS connection management (ECM). The EPS mobility management part occupies itself with tracing, identifying and authentication of the UE, inquiring the user equipment's capabilities and the establishment of the security context (i.e. encryption and integrity) through the aid of for instance an attach, detach, identification, authentication, tracking area update, capabilities enquiry and security mode control procedure. On the contrary, the EPS connection management

entity regulates the connectivity of the UE with the packet data network gateway by, among others, allocating IP addresses when Internet access is requested. This also entails procedures for default and dedicated EPS bearer context activation, EPS bearer updating and its deactivation, as well as supporting primitives for the devices' traffic towards the PDN. Furthermore, the mobility management entity keeps track of each UE's associated ECM and EMM state. In particular, the LTE standard defines the EMM_DEREGISTERED and the EMM_REGISTERED state which signifies the registration of the UE inside MME, whereas the ECM status is expressed by either ECM_IDLE or ECM_CONNECTED relating to the NAS signaling connection with the network. To switch from the ECM_IDLE to ECM_CONNECTED two connectivity characteristics should be met of whom the first one necessitates the setting up of the RRC connection (e.g., through cell selection/reselection). This means for the device to be in RRC_CONNECTED where it should be noted that it can, yet, only communicate control-plane signaling with the network. Thereafter, to facilitate the transition to ECM_CONNECTED state several NAS messages are exchanged until completion of the S1 interface. In ECM_IDLE state, however, no such RRC-S1-MME signaling connection is available, likewise no UE context is present in the radio access network. Next, it is important to know that the EPS mobility management states are partially dependent on the ECM state and influenced by the mobility management mechanism, phrased above. Going from EMM_DEREGISTERED to EMM_REGISTERED is only feasible for user equipment in ECM_CONNECTED, whilst no restrictions apply in the opposite direction (i.e. from EMM_REGISTERED to EMM_DEREGISTERED). As an example of the latter, an implicit detach in ECM_IDLE from the MME triggers to move from EMM_REGISTERED to EMM_DEREGISTERED, although an explicit detach procedure is needed during ECM_CONNECTED. In the EMM_DEREGISTERED state, paging techniques are used as the MME does not have the latest knowledge about the exact location of the UE, and even some of the latest UE context may be saved at both sides to, as an illustration, sidestep a key agreement (AKA) procedure upon a new attach. As a last note, some RRC messages are capable of, and often used for transporting NAS messages as well, also known as piggybacking [38, 37, 39].

2.3.3 Basic LTE procedures

This last section handles the different procedures in LTE that are given to user equipment for accessing and requesting services from the network. Hence, the first part of the chapter handles the basic network procedures from which more elaborate procedures, discussed in the second part, are built up. An example of such a latter procedure is to send traffic over the LTE core network.

Basic network procedures

The basic procedures considered in this part are (1) cell search and selection procedure, (2) random access procedure, (3) additional signaling and finally (4) detach procedure. They all form an essential part in providing services from and to the network. Note

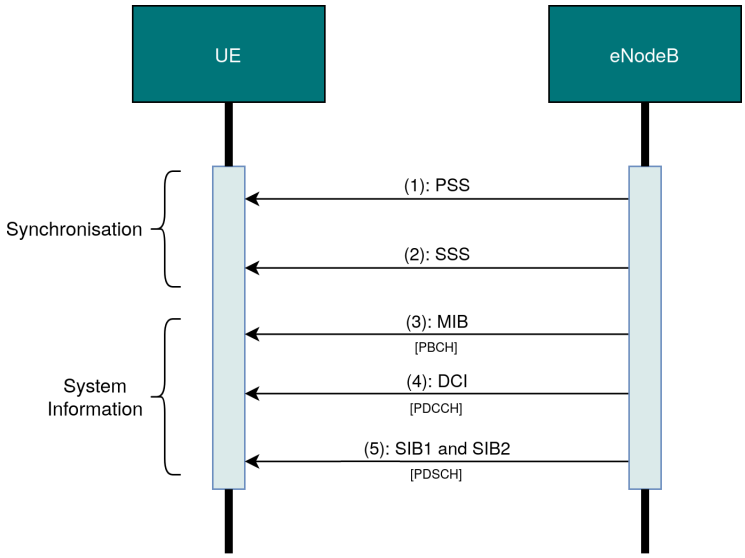


FIGURE 2.8: Sequence diagram of the cell search and selection procedure.

that random access procedure followed by the additional signaling to connect to the LTE network will be denoted as the attach procedure.

Cell search and selection procedure

In this section the purpose and the basic methodology of the cell search and selection procedure are discussed, as can be seen in Figure 2.8. The main goal of cell search is to provide a device with the required information to communicate with an LTE network. However, not only during the initial connection to the network, as would be expected, but also during the mobility procedure the UE is required to constantly scan its neighbouring cells regarding quality measurements. Upon finding superior quality reception in a neighboring cell the user equipment should seek for handover when in RRC_CONNECTED and start cell reselection if RRC_IDLE. The starting point for this procedure is to search for cells in the UE's supported frequency bands. Next, the primary synchronization signal (PSS), followed by the secondary synchronization signal (SSS) broadcasted by the candidate cell give the UE the appropriate network frequency and timing synchronization. These signals equip the connecting device with essential information such as timing and the physical-layer identity of the cell to decode all of 24 bits of the master information block on the PBCH. In turn, with some data contained into the MIB the PDSCH can be parsed which transports several system information blocks. Hence, with the decoded SIB1 and the UE's USIM card the mobile terminal knows if it can camp on the selected cell, otherwise the cell search starts over again. As an example, system information block 1 conveys a set of PLMN IDs that are checked against the network operators and subscription information inside the USIM [38, 40].

Random access procedure

The random access procedure is an elementary method in LTE that establishes a radio-link between the UE and eNodeB. In other words, it has as outcome for the UE to acquire uplink resources on the PUSCH for its traffic, synchronize the radio in uplink, and have assigned a unique identity, also called C-RNTI, if not possessed already. The base station on the contrary can make use of this procedure, whenever it has data to transmit but has lost synchronization with the receiving mobile user. For this purpose the eNB sends a specific message, also called a PDCCH order, wherein information is embodied that specifies the mobile terminal when to initiate the random access procedure. Optionally, the preamble to use may be included as well for the aim of contention-less random access. In general, the random access is utilized in the next situations:

- during initial access (i.e. RRC_IDLE to RRC_CONNECTED), but after synchronization with the access network in downlink via the cell search and selection procedure. Here, the UE still needs to synchronize in uplink.
- after radio-link failure to reestablish the link.
- during the handover protocol for uplink synchronization with the new cell.
- when the mobile terminal is in RRC_CONNECTED, receives or has to transmit data and has an out-of-sync uplink radio.
- if the UE should be located with the positioning methods on the basis of uplink measurements.

The random access procedure is shown in Figure 2.9. For completeness both the physical layer as well as the MAC layer signaling are shown, however, the focus of this discussion is on MAC layer messages. In summary, the physical layer of the eNodeB transmits downlink grants on the PDCCH, telling the mobile terminal where it can find downlink data, and acknowledgements for uplink messages on PHICH [38, 40, 41]. To make the examination of the random access procedure more clear, the messages from the figure will be indicated with *(X)*.

In the first step of the contention-based random access procedure the UE sends one of the 64 possible preambles *(1)*, which are subdivided into two groups, on the allowed resource blocks of the PRACH indicated by SIB2. Here, the expected size of the RRC Connection Request message *(3)* and the measured path loss define the group of the preamble, which is a quantity indication to the eNB for the necessary resources to be reserved. The contention resolution, although, will not be resolved now but later, in case two or more devices send the same preamble at the same moment. Note that there also exists a non-contention-based version which will not be elaborated on further [38, 41].

Subsequently in step 2, if the eNB correctly detects a preamble, it will responds by scheduling a Random Access Response (RAR) *(2)* on the DL-SCH containing, among other things, the index of the received preamble, the timing correction, the

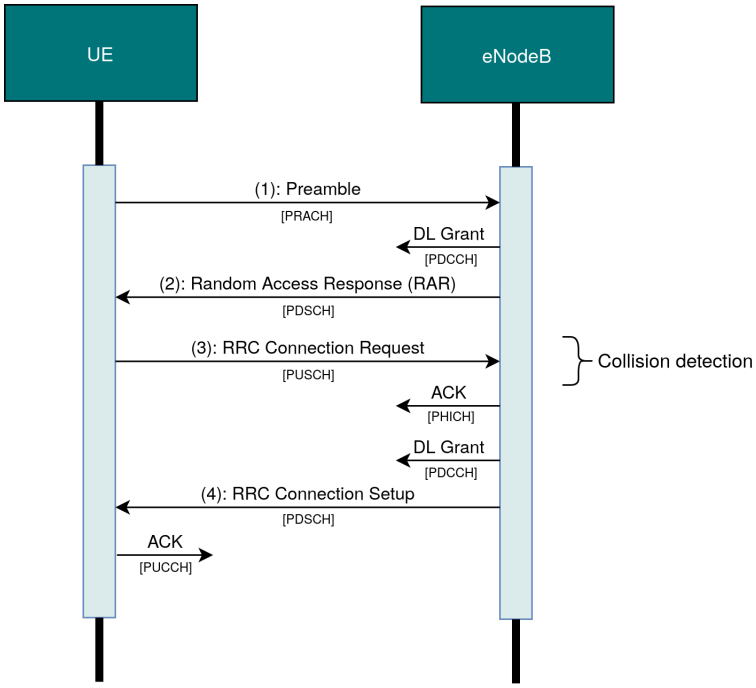


FIGURE 2.9: Sequence diagram of the random access response procedure.

uplink grant and resource allocation for the next message *(3)* on UL-SCH and a temporary ID (TC-RNTI). At this moment, the eNodeB is not yet able to observe collisions due to the invisibility of preamble collisions. Conversely, after having received no RAR for a configurable time-window succeeding three subframes after transmission, the UE restarts the procedure over again. In particular, it selects a new preamble, executes the backoff method and resends the preamble at a possibly increased power-level. Now, in case the maximum number of retransmissions is exceeded, the mobile terminal will stop and report a random access failure to its upper layers [38, 40, 41].

Following the reception of the RAR the device synchronizes its uplink radio with the time correction available in the packet. Next, in the third step of the protocol the RRC Connection Request message *(3)* is put on the scheduled resources of the UL-SCH (SRB0), given in the Random Access Response. An important aspect of this uplink message is the ability for the mobile station to detect collisions and thus to respond only to collision-free RRC Connection Request transmissions. This message includes, for example, information about the reason for the connection establishment and most valuable the device identity that supports the contention-based resolution in the next step. One of the following connection causes can be added in the message; UE initiated signaling; UE initiated data; UE having paging opportunity; call setup; and more. If the device is in RRC_IDLE it uses the core network generated temporal identifier TC-RNTI found in the RAR as ID, while RRC_CONNECTED user equipments include their former assigned C-RNTI as they

are still know by the E-UTRAN cell [38, 40, 41].

In the last step the base station only replies to a collision-free RRC Connection Request with a downlink RRC Connection Setup message *(4)* on the DL-SCH. Hence, whenever the C-RNTI within the downlink grant on the PDCCH corresponds to the random number transmitted by the device in step 3, the UE knows that its random access procedure was successful and thus should decode the received RRC Connection Setup data. If on the contrary the contention resolution timer, set on transmission of the connection request, expires and the mobile terminal didn't receive a contention-resolution message containing its number, the device assumes failure and starts all over from step 1. The difference with the contention process for idle UEs is that not only the temporary C-RNTI in the downlink grant, but also in the RRC Connection Setup itself, have to be equivalent to the device's ID, since the TC-RNTI is not always unique. Moreover, upon success of the random access procedure the TC-RNTI serves form then on as the unique identifier (i.e. C-RNTI) for the UE. Besides the contention resolution, the RRC Connection Request also conveys a lot of information about the configuration of the cell: e.g., physical layer, MAC and RLC layer parameters. This means that the UE is informed for example on the maximum number of retransmission of RLC reports, uplink power-control parameters, the periodicity of measurement reports and so on. Hence, a successful random access procedure transfers the user from RRC_IDLE to RRC_CONNECTED, configures the signaling radio bearer 1 and establishes a radio-link to the E-UTRAN [38, 40, 41].

Additional signaling

The additional signaling is the next basic LTE procedure involving the UE and is the way for user equipment to contact the evolved packet core, in particular the MME. Moreover, it has four main objectives. The outcome of this procedure leads to the UE being registered in the mobility management entity. This means that MME is aware of the mobile terminal's current location and can store relevant UE related information (e.g. supported ciphering algorithm(s)). As a result, the stored states of the UE are updated by the MME to EMM_REGISTERED and ECM_CONNECTED. Furthermore, upon completion the network has configured signaling radio bearer 2 for future NAS signaling messages between the UE and the network, has associated it a new IP address, if necessary, and set up the default EPS bearer towards a default packet data network (e.g. the Internet). In other words, the EPS bearer serves the user equipment with always-on connectivity. The procedure is presented in Figure 2.10 of which the focus will be again on the MAC layer messages and above. It is sufficient to know that the physical layer of the UE needs to ask for uplink scheduling on the PUCCH, whenever it wants to transmit. As a result, the eNodeB will signal on the PDCCH if the resources are granted to the UE. Besides this the mobile station is also required to let each transmitted message be preceded by a downlink grant [38, 42, 41].

First, the UE transmits the RRC Connection Setup Complete message *(1)* on the PUSCH as a response to the RRC Connection Setup when receiving an uplink scheduling grant. This piggybacked RRC message embeds two NAS elements

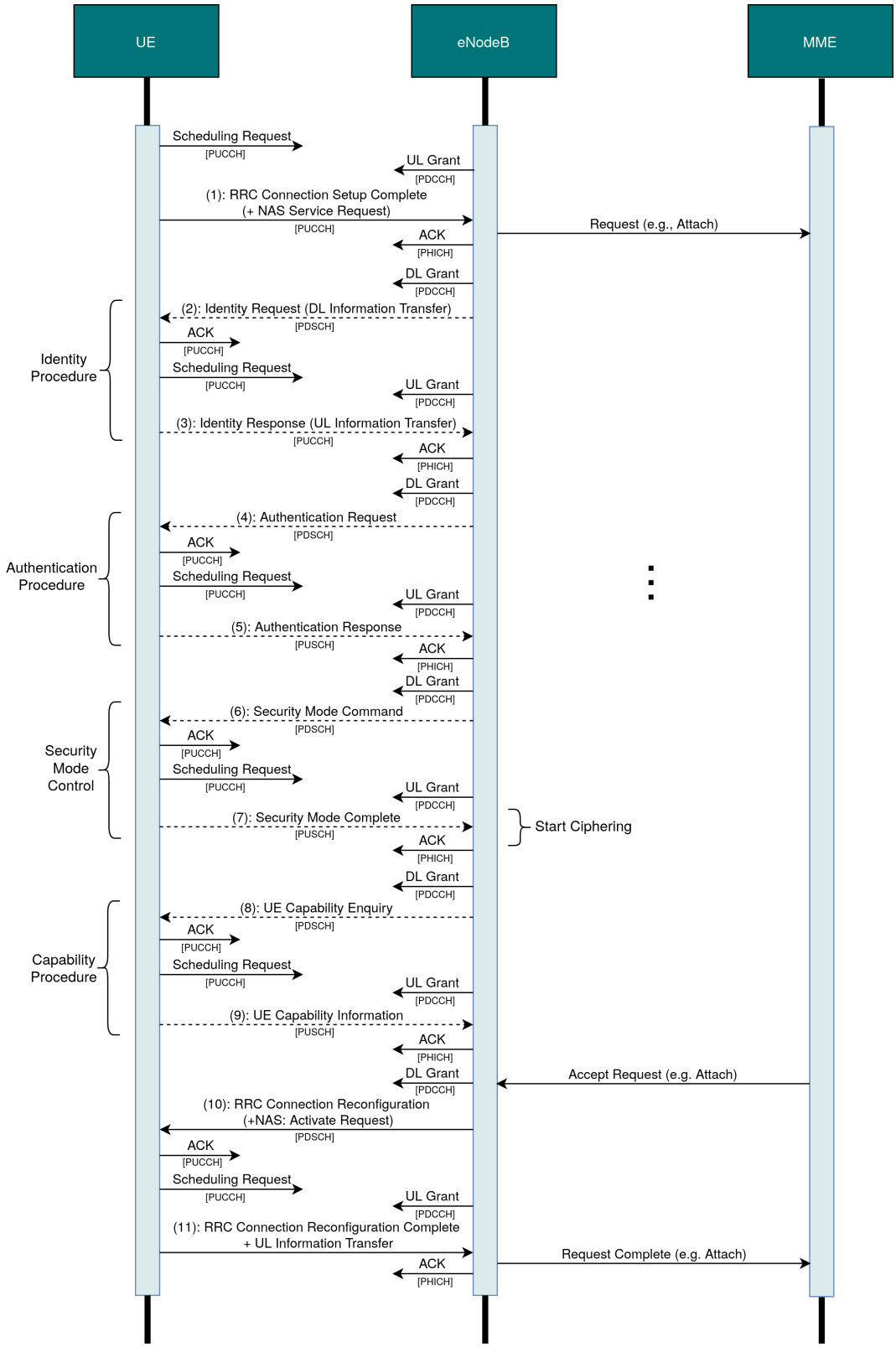


FIGURE 2.10: Sequence diagram of the additional signaling in case of the attach procedure.

(EMM/ECM) namely one EPS mobility management message which can be a Detach Request, a Service Request or a Tracking Area Update Request, and one PDN Connectivity Request for creating a default EPS bearer. Of course, for the attach procedure an attach request is configured. Despite this NAS signaling the message is built up from other data as well, like some protocol configuration options regarding the external network [38, 42, 41].

Subsequently, the additional signaling requires optionally the identification, authentication, security mode control and capability enquiry protocol. If the UE was registered to the network before, the base station and MME still have the required information to skip over these mechanisms. Hence, this improves the latency of consecutive attachments to the network. In the other situation, the MME starts with requesting the user equipment's identity via an EMM Identity Request. For this purpose the so-called DL Information Transfer *(2)* is used which is a type of RRC message solely for transporting EMM and ECM packets. Upon reception the UE answers with an UL Information Transfer *(3)* stating for example its IMSI number. Next, the authentication procedure provides both mutual authentication and key agreement (AKA) between the user and the mobility management entity. It consist of the exchange of Authentication Request *(4)* in downlink and Authentication Response *(5)* in uplink over the air interface. Third, during security mode control the MME begins with transmitting a piggybacked Security Mode Command *(6)* followed by a Security Mode Complete *(7)* meaning that the device and the MME have derived the ciphering and integrity keys and will start using the new security context for further communication. In particular, the Security Mode Command is the first integrity protected message, but not encrypted as the MME starts downlink ciphering after having decoded the Security Mode Complete. The UE on the other hand start its NAS security at the PDCP layer after transmission of the Security Mode Complete messages. Last, with the RRC message UE Capability Enquiry *(8)* the eNodeB asks the device about its access stratum-related capabilities such as the antenna configurations, supported bit rates, bandwidths and supported radio access types. Then, the UE Capability Information *(9)* configures the lower layers of the base station for optimizing the radio-access connection with the UE. However, note that this capability information is stored by the MME so that whenever the mobile changes again from RR_IDLE to RRC_CONNECTED in the future the base station just need to download it from the MME [38, 43, 44, 41].

To end the extra signaling the RRC Connection Reconfiguration *(10)*, the RRC Connection Reconfiguration Complete and an UL Transfer Information Transfer *(11)* are sent over the air interface. The first message not only contains some RRC signaling, but also the embedded EMM Attach Accept (i.e. response to the Attach Request) message together with the Activate Default EPS Bearer Context Request which is a reply to the UE's PDN Connectivity Request previously sent. Now upon detection of this RRC reconfiguration message the mobile modifies its stack parameters as indicated in the RRC packet and sets up the default EPS bearer. First, as an acknowledgement the RRC Connection Reconfiguration Complete is sent over SRB1 and redirected to the MME. Second, the procedure is successfully concluded by piggybacking an EMM Attach complete and an ECM Activate Default

EPS Bearer Context Accept on the UL Information Transfer towards the base station on SRB2. Hence, the UE is now in RRC_CONNNECTED, EMM_REGISTERED and ECM_CONNECTED [38, 42, 41].

Detach procedure

When a device in RRC_CONNECTED and ECM_CONNECTED wants to disconnect from the network (e.g., if switching off), it first has to deregister itself from the network by means of the detach procedure. This is done so by creating an EMM Detach Request (stating if due to switch off) which triggers the EMM to tear down the mobile terminal's EPS bearers. Next, whereas the base station is informed by the MME to delete its UE related context, the mobility management entity itself keeps the required associated data (e.g., security keys, IMSI, GUTI, etc.) for when the mobile wishes to connect in the near future. If the cause for deregistration is not "switch off", the base station instructs the device to clear SRB1, SRB2 and other bearers by putting the RRC Connection Release message on the air interface, otherwise the UE can turn itself off without having to wait for a reply. Another possibility is that the eNodeB itself detaches the mobile by sending RRC Connection Release after a predefined time-out of user inactivity. Mobiles in RRC_IDLE and ECM_IDLE, on the contrary, first have to instantiate the random access procedure followed by the RRC Connection Setup Complete message which should be packed with the Detach Request. From this point on the detach procedure resumes as before. Note that the detach procedure results in the UE being in RRC_IDLE and ECM_IDLE [42, 45].

2.3.4 Elaborate LTE procedures

This section gives an overview of the different procedures in LTE that make use of the basic procedures mentioned above. More specifically, the following are discussed: the LTE connection establishment, the service request and the paging procedure.

LTE connection establishment procedure

The moment the mobile wants to connect to the LTE network the LTE connection establishment procedure is exactly what it needs. After attaching to the core network the UE can e.g. communicate with an HTTP-server or call to another user with the procedures mentioned later on. In order for the user to connect to the LTE network (e.g., when switching on for the first time) it first follows the random access procedure and then the additional signaling with an attach request. To tear down the connection the detach procedure can be used.

Service request procedure

The procedure can be used by the mobile in ECM_IDLE whenever it wants to request services from the LTE network or in other words when it wants to communicate with the outside world. This can be the case either when the UE receives a paging request

from the MME or in case of a UE-originated data call. To make this connectivity possible the service request procedure re-establishes the different signaling and data radio bears as well as the S1 and other requested bearers. Hence, the mobile is transferred back to the EMM/ECM/RRC connected state. Now, to tear down the connection the detach procedure can be used [38, 45].

The exchange of messages is in fact quite similar to the protocols already seen previously. First, the UE starts the random access procedure to create the radio link with the base station. After contention resolution the mobile instantiates a modified additional signaling procedure, where instead of the Attach Request a Service Request message is piggybacked on top of the RRC Connection Setup Complete together with a dedicated bearer request. Finally, after the device has accepted the dedicated EPS bearer with the DL Information Transfer, it is able to use the dedicated bearer for transmission to or reception from the network [38, 45].

Paging procedure

The paging procedure is used when the PDN gateway receives data intended for an user equipment in RRC_IDLE and ECM_IDLE. However, in this case the MME neither has the routing information to the receiving UE nor the exact location and as a result it instructs the involved base stations to send an RRC Paging Request on the PCH. Once the mobile terminal detects the paging message, it executes the service request procedure to gain access to the incoming IP packet [38, 40, 45].

Chapter 3

Architecture

This chapter gives an overview of the architecture that was designed to fuzz the LTE device-under-test (DUT). To start the discussion, the first paragraph introduces the reader to the high-level fuzzing architecture, which is composed of different components. Next, a light is shed on each of these structural elements by describing their functionalities within the fuzzing framework, their implementation and their interaction with other components. However, as this thesis is the follow up of the thesis by J. Hoes [18] several parts of the current architecture are an extension to his work. Therefore, overlapping elements will be touched upon shortly in this section, with as goal to give the reader the necessary tools to understand the subsequent chapters. However, several upgrades have been made in the initial design to come to existing architecture. In the subsequent sections, these upgrades will be discussed in more detail. For more details on the initial architecture designed by J. Hoes the reader is referred to [18].

3.1 High-level overview

Figure 3.1 shows an overview of the high-level architecture used in this thesis to fuzz an IoT user equipment speaking the LTE protocol. As can be seen in the diagram the most important pieces are (1) the fuzzer, (2) the simulated LTE network which helps the fuzzer to craft correct LTE messages, (3) the device-under-test acting as the victim UE and (4) finally the software emulated reference UE. Here, the reference software is a new key idea of the upgraded framework to overcome the main limitations in IoT fuzzing, namely the absence of feedback information. In contrast with the targeted IoT system, the reference UE supplies the fuzzer with relevant information for code coverage and differential fuzzing. Hence, whenever new code is trigger the RUE will record hits of new execute code which is an indication of the inputs' fitness. However, the RUE's coverage does not fully resemble the coverage inside the hardware of the device, but it does capture new characteristics exercised by the inputs inside the tested UE. It serves as an indirect way to measure the span of hardware tested inside the DUT.

Besides the integration of a reference UE, the architecture incorporates several

new interfaces. These communication channels enable the the fuzzer to control the other components and retrieve the necessary data. For example, to retrieve the code coverage of the RUE or to instruct both UEs. Hence, two types of interfaces exist: socket interfaces and shared memory interfaces. Socket interfaces enable the fuzzer to send, among other things, AT commands and is indicated in a purple dotted line in Figure 3.1. On the other hand, the shared memory interfaces are depicted in dotted green. It serves as communication channel for protocol interactions with the RUE and to transfer messages between the eNodeBs and the fuzzer.

The general operation of the architecture to fuzz the DUT goes as follows. In a first step, all components are started up and configured by the fuzzer, after which the fuzzer block starts generating test cases to be handed over to both the eNodeB of the device as the eNodeB of the reference UE. This is done over the shared memory interface indicated with green dotted lines in the upper part of the figure. In turn, the eNBs further process the packets through out their stack and pass the data to their respective front-end device. Hence, in this way the UEs receive (ill-formed) messages assembled by the fuzzer logic. The uplink messages, on the other hand, follow the opposite direction and are observed by the fuzzer as well. In the upgraded setup, the fuzzer also acts as the main controller in charge of several management tasks including, for example, starting and stopping other components, issuing AT commands, and controlling data and connections over network sockets and shared memory. The In the remainder of the chapter each individual architectural block will be investigated and split into their software and hardware pieces. Moreover, more details about the implementation of the fuzzer component can be found in Chapter 4 and Chapter 5.

3.2 Simulated LTE network

The simulated LTE network is an important cornerstone in the overall fuzzing architecture. It helps the fuzzer in producing correct LTE communication sessions with the mobile terminals, in particular for the fuzzed IoT device. As mentioned in [18], the simulated LTE network is implemented using the Software Radio System's (SRS) srsLTE software [46], beneficial for its readability and in addition open-sourced. Hence, the SRS software provides an entire 4G (as well as 5G) software radio suite embodying a simulated user equipment (i.e., srsUE), a software emulated base station, known as srsENB, and an LTE core network in srsEPC [47]. As illustrated in Figure 3.1, all software simulators types from SRS (i.e., srsUE, srsENB and srsEPC) are used to fuzz the system-under-test. In particular, whilst the architecture mimics the 4G's core network with the srsEPC software, each UE has an eNodeB assigned, that serves as the connection point between the mobile terminal at one side, and the fuzzer and core network at the other side. Similarly to [18], both base stations are behaving malicious as a consequence of the presence of the fuzzer, thereby fuzzing the inputs going to the UEs. In the remainder of the thesis, the simulated user equipment will be denominated as reference UE (RUE) for the obvious reason that it functions as an example for the DUT. The architecture integrates srsLTE version

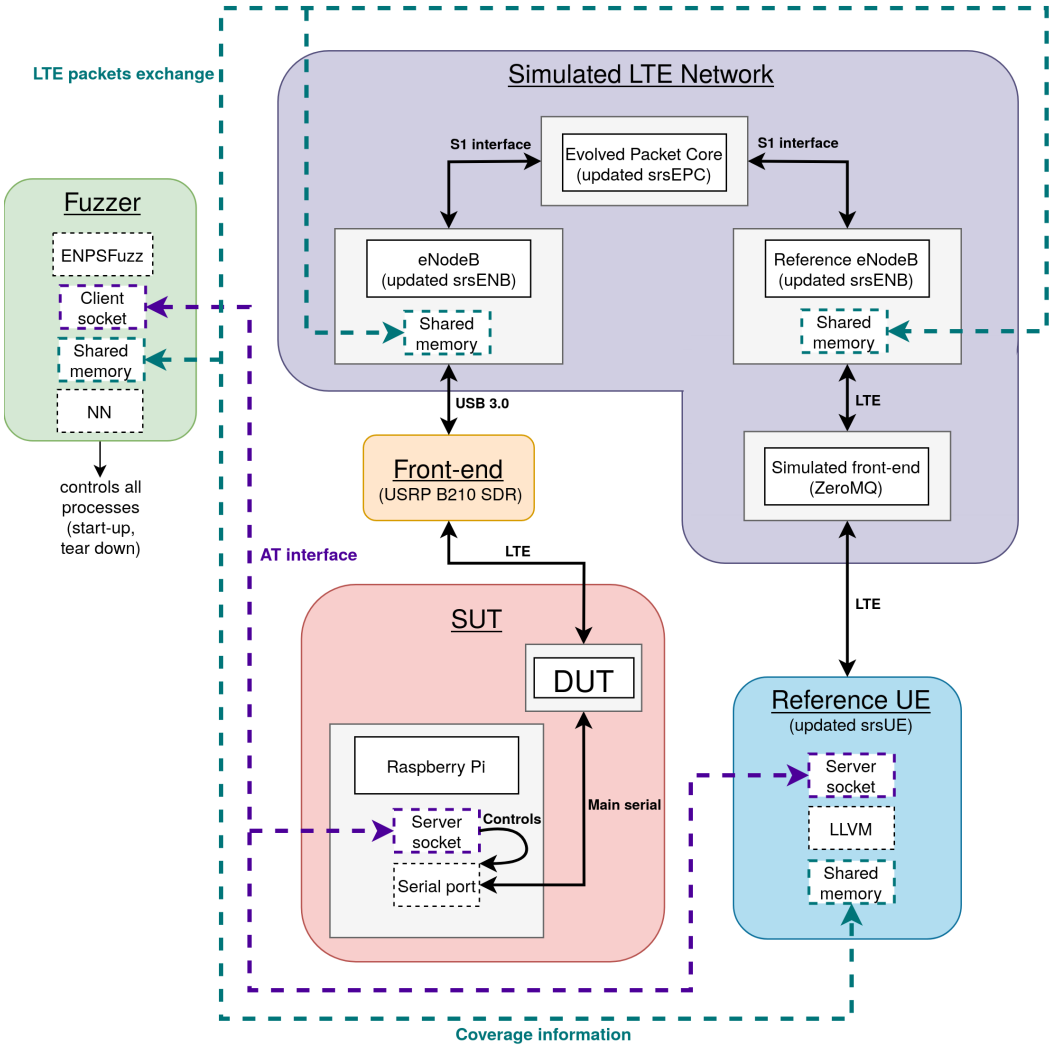


FIGURE 3.1: The overall architecture of the fuzzing framework.

20.04.2 characterised by 3GPP Release 10.

The core network itself didn't require much updating, since the most important interaction in the fuzzing architecture is the link between the fuzzer and the base stations. Only some small changes were made inside the srsEPC such as, for example, enabling the network to always execute the entire attach procedure, mentioned in Chapter 2. In particular, the updated Evolved Packet Core is changed to erase the user data (e.g. UE capabilities and security parameters) from its tables after the mobile terminal detaches from the network. However, as stated in [48], only whenever the necessary user information is present inside the home subscriber server, a UE is able to attach itself to the simulated core network. Accordingly, to make the srsEPC software fully functional, the secret key and the IMSI number found on the IoT's USIM card were added to the EPC's database. In a similar way, a record for the reference UE was added too.

Next, the LTE network simulator is composed of one eNodeB for the device-under-test and one for the RUE. In essence, the srsENBs are modified to send and receive data to/from the fuzzer over shared memory at certain interception points throughout the stack. The updated architecture integrates an extra software emulated UE that communicates with its base station via a virtual radio. This simulated radio is provided by SRS and specified in [49]. In theory, it is possible to serve both UEs with the same base station, however, the srsENB software is not (yet) capable of sending data with a virtual radio to one UE, while serving the other with a physical front-end. Following the specification, the communication with a virtual ZeroMQ radio is only supported between one base station and one user equipment [49]. Moreover, the eNB at the left of Figure 3.1 handles data regarding the system-under-test with a physical software defined radio (SDR). In particular, the USRP B210 from manufacturer Ettus puts the data samples from the base stations' physical layer onto the air.

Linking this radio frequency front-end to the particular eNB only depends on the Ettus' UHD driver, which can be installed from their website and an USB 3.0 connection [18]. Figure 3.2 shows the USRP B210 radio. On the contrary, the rightmost base station from Figure 3.1 serves as an anchor point for the simulated UE by means of a ZMQ virtual radio. To this end the ZeroMQ library has to be installed after which the eNodeB's configuration file needs to be updated to use the virtual radio as front-end instead [49].

3.3 System-under-test

The system-under-test (SUT), shown in Figure 3.3 and Figure 3.4, not only covers the device-under-test, but is in fact composed of several hardware components: (1) a Raspberry Pi 4, (2) a Sixfab LTE hat, (3) a DUT (e.g. the Quectel EC25 modem), (4) a Keysight SIM card and (5) a coaxial cable. First, the role of the hardware components will be described shortly, followed by the presentation of the upgraded communication interfaces in the SUT [18].

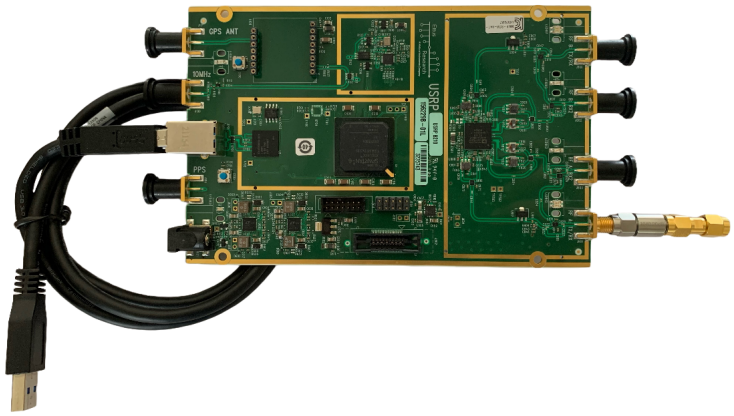


FIGURE 3.2: The USRP B210 physical front-end for the srsENB connected to the DUT for which it converts data samples to over-the-air radio transmissions.

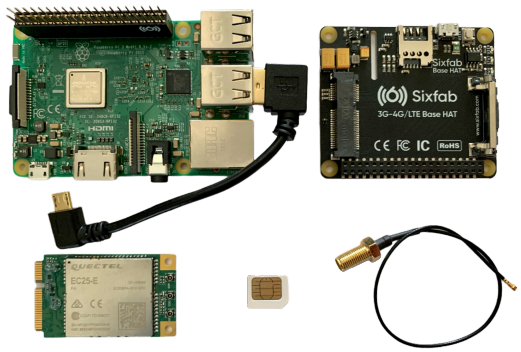


FIGURE 3.3: A picture of a dismantled system-under-test with starting from the upper left corner: a Raspberry Pi 4, Sixfab LTE hat, Quectel EC25 modem (DUT), Keysight USIM card and coaxial cable with u.fl adaptor and regular SMA connector.

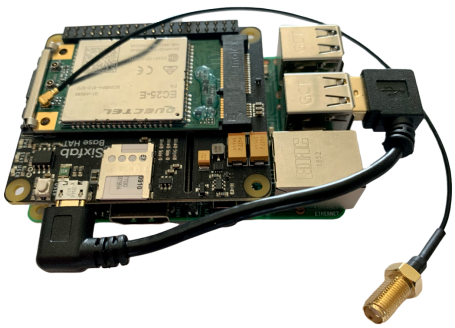


FIGURE 3.4: The assembled setup involving the five hardware components from Figure 3.3.

3.3.1 Setup

The IoT device, for which the fuzzing architecture was built, is a Quectel EC25 LTE modem. The motivation for this choice can be consulted in [18]. Furthermore, the Quectel EC25 is an LTE CAT 4 UE implementing the LTE protocol, as well as older cellular standards. To communicate with the LTE mini PCIe chip, the fabricant has equipped the modem with several interfaces. In particular, there is a PCM interface, a I2C interface, a USB interface, a main serial interface and a USIM interface. However, only the latter three interfaces are used in the fuzzing architecture as described below. In order for the modem to connect to the Raspberry Pi an LTE hat from the company Sixfab has to be mounted on top of the Pi. With this hat not only the Quectel modem can be connected and fuzzed, but also other 3G and 4G mini PCIe chips. This is, for sure, a big advantage of the initial design. Moreover, the hat provides a USIM socket that links the inserted USIM card with the corresponding USIM interface of the mounted modem. In the current system-under-test a USIM card of Keysight is inserted. Besides this functionality, the hat also transfers data from the main serial interface to the Raspberry Pi and enables the Pi to take control over the cellular chip. Lastly, as can be seen in Figure 3.4, an extra black cable forwards signals from the USB interface of the PCIe chip to the USB port of the Raspberry Pi. Next, the last component of the system-under-test is a coaxial cable that connects at one side to the Quectel modem with an u.fl adapter, and at the other side to the USRP front-end of the base station with a regular SMA connector. Instead of sending the signals over the air, this coaxial cable transfers the signals between the UE and the eNB so that less transmission errors occur and hence less retransmission are needed [18].

3.3.2 Interfaces

The AT interface is used to send AT commands to the modem. Hence, an AT command has as purpose to draw the attention of the modem and instruct the modem to do some tasks depending on the type and the actual AT forwarded. In fact, AT is the abbreviation for ATtention (i.e. each command starts with "AT") and is a standardized set of instructions to control and interact with wireless modems as an outsider. Furthermore, the set of AT commands originates from the Hayes command set, named after the developer Dennis Hayes, which was at that time a specific language for communication with smart modems. While incorporating a subset of Hayes' instructions, several extended AT commands have been added to the AT language, which is also standardized by 3GPP [50] for cellular modems. The standard distinguishes four different types of commands, namely test, read, set and execution commands. Whereas a test command checks if the command is available on the modem using AT+<command>=?, a read instruction has the syntax AT+<command>? and returns variables or settings from the modem. On the other hand, the commands belonging to the set command type modify variables or settings inside the modem by sending AT+<command>=<value1,...,valueN>. Lastly, an execute AT command instructs the modem to perform an operation that

is stated in AT+<command>=<param1,...,paramN> syntax [51].

From the Quectel EC25 AT manual [52], it can be seen that the main UART port and the USB ports provide access to the AT interface. As discussed in [18], the main serial interface is used for this purpose as the Sixfab hat gives the Raspberry Pi effortless access to this interface. Moreover, the official document [52] describes a wide range of functionalities offered by the set of AT commands implemented inside the Quectel EC25 LTE modem. The capabilities range from controlling the serial interface, to sending text messages (SMS), establishing calls, configuring audio functionalities, issuing hardware related instructions and controlling the network connectivity. In view of fuzzing most focus is paid on:

- **AT**
- **AT+COPS**
- **AT+CFUN**
- **AT+QPING**

Here, the problem to be tackled is how to make the AT interface available to an outside process and in particular to the fuzzer. To this end, a server process was newly created on the Raspberry Pi which is continuously listening for requests coming from the outside, after being started up by the fuzzer. Hence, the server accepts socket connections where bidirectional data can be sent over. In particular, the Raspberry Pi waits for an AT command and hands it over in a subsequent step to the main serial interface of the device-under-test. With an transmitted AT command towards the server process, the client can ask for the modem's responses at any time. In fact, the created process on the Raspberry Pi monitors the main interface for data from the cellular modem. For example, when the modem could connect to or disconnect from the LTE network with an *AT+COPS* command, it outputs the *OK* message on the main serial interface. Although, upon failure an error message is put on the main interface and transferred to the requesting socket side. At the end of the AT interface's usage the socket server as well as the connection to the AT interface can be shut down properly by issuing the correct stop request. This interface connectivity is a step towards the automatizing of the fuzzing framework.

3.4 Reference UE

This component is the last piece in the architecture that uses the simulated LTE software (i.e. srsUE). Although the reference UE is not a compulsory component to fuzz the SUT, it yields some interesting information to craft inputs more intelligently. As mentioned before, an updated version of the srsUE software fulfills the role of reference UE within the overall framework. Especially, by also sending the fuzzed information to the RUE, the behavior of the simulator can be analyzed by the fuzzer and compared with the actual IoT device under testing. This is also known as *differential fuzzing* which assesses the dissimilarity between several components that only differ in their implementation of the same application. In this way so-called

semantic or *logic bugs* can be exposed that do not result in an explicit crash or hang, but reveal the discrepancies between implementations. The interested reader is referred to Chapter 5 for more information about this topic. Apart from differential testing the reference UE also serves as a source for code coverage information for fuzzed test cases. Hence, whenever new code is trigger in case of, for example, a changed sequence of messages or mutated fields inside a series of inputs the RUE will record hits of new execute code which is an indication of the inputs' fitness. Despite the RUE's coverage does not fully resemble the coverage inside the hardware of the device, it does capture new characteristics exercised by the inputs inside the tested UE. Thus, it can be thought of as an indirect way to measure the span of hardware tested inside the DUT. Hence, following the philosophy of coverage-based fuzzers for (stateless) software programs the more code paths being traversed the higher the likelihood to stumble upon a bug. In this view the reference UE is instrumented using LLVM SanitizerCoverage [53] at predefined parts of srsUE's code. The reason for integrating the LLVM coverage tool in the RUE is due to several advantages it provides over other instrumentation libraries. First, it is an easy to use coverage instrumentation built in from LLVM which alleviates a programmer from developing his/her own time-consuming instrumentation tool (e.g. using LLVM Pass). Second, SanitizerCoverage provides default implementations as well as user-defined callbacks to be inserted at each edge, basic block or function. This means that LLVM adds extra instructions before each instrumentation point (i.e. beginning of a function) at compile-time. Hence, these extra instructions jump either to a default routine specified by LLVM or an overwritten version of these functions defined by the user. As a clarification, the different levels of instrumentation are demonstrated in Figure 3.5 using a short example. Third, this compile-time instrumentation tool makes it even possible to disable instrumentation for fragments of the code without the need for source code modification. For the purpose of collecting code coverage, only the LTE stack of srsUE is compiled with LLVM instrumentation at edge granularity. The choice for inserting instructions at edge-level can be seen in part c) of Figure 3.5. In short, the monitoring basic blocks and function calls are less granular and hence trace less accurately the flow of the program. In more detail, each edge is assigned a unique identifier which is mapped to the corresponding counter inside the bitmap upon executing the edge. The bitmap inside the RUE, or although more precisely named "bytemap", stores an eight bit counter for each edge that is instrumented at compile-time. During the execution of a test cases the overwritten LLVM callbacks manage the different counters inside the bitmap to collect the code coverage, which is used further by the fuzzer process.

3.4.1 Interfaces

Although the reference UE posses the code coverage information, more specifically the bitmap, from the paths traversed by the inputs, the info has still to be transferred to the fuzzer itself. For that reason, shared memory is put in place between the RUE and the fuzzer component. This can be seen by the dark green line at the bottom of Figure 3.1. At the side of the user equipment a thread is constantly checking

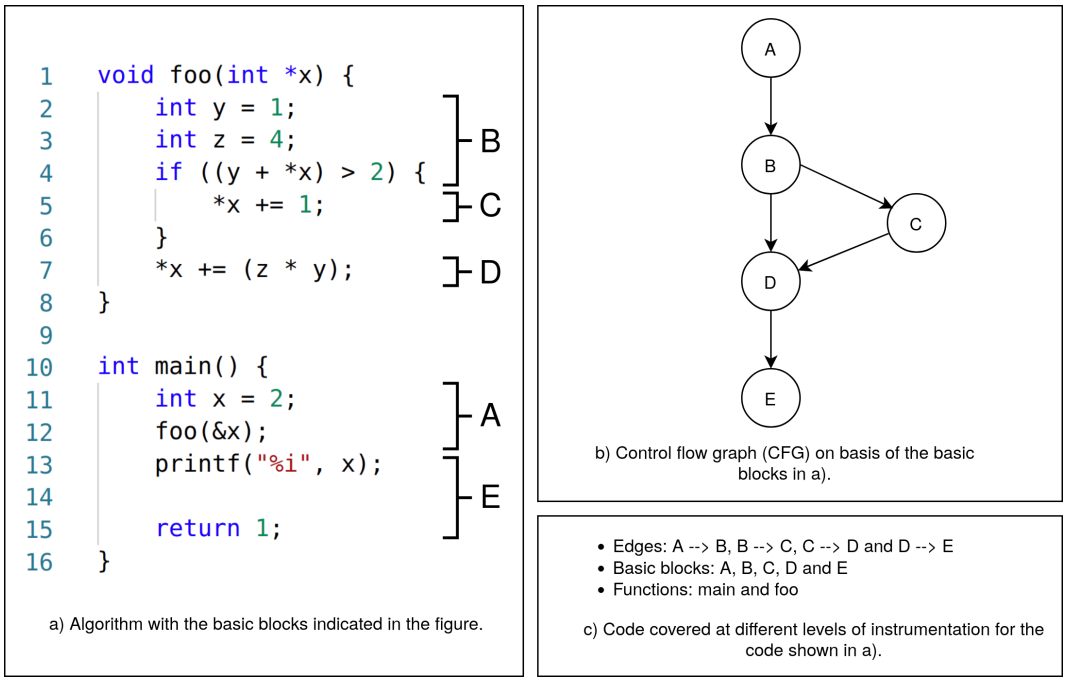


FIGURE 3.5: A short example illustrating the different levels of instrumentation available in LLVM SanitizerCoverage.

on the shared memory for interactions with the fuzzer. The thread expects one of the following commands having each a unique command identity in part b) of Figure 3.6: a code coverage request, a request for resetting the bitmap, or a bitmap size enquiry. With the latter, the reference UE informs the other side about the size of the bitmap in bytes. Thus, it sums the number of eight bit counters inside the bitmap which is equivalent to the number of instrumentation points (e.g. edges or basic blocks) as each point has a unique counter. The advantage of this request is that the receiving party knows how many bytes to allocate if it wants to receive code coverage data from the RUE. The second command, namely the one to ask for coverage, is followed by a transfer protocol, since the bitmap size is in general too large to be transferred over the shared memory at once. Upon reception of this request, the reference UE calculates the number of transmissions depending on the total number of counters and the maximum size on the shared memory reserved for data (i.e. size of the shared memory - three bytes). The shared memory structure to move (parts of) the bitmap to the fuzzer is depicted in part a) of Figure 3.6. Here, the length field indicates the number of bytes, occupied by all fields, out of the total shared memory size used for the current transmission. On the other hand, the data field contains, as the name already suggests, the actual (coverage) data. Besides, the RUE informs the fuzzer by setting the flag field if it should expect an extra transmission, otherwise all counters were successfully sent and the fuzzer can go on with its work. To not overload the fuzzer side and notify the RUE about processing

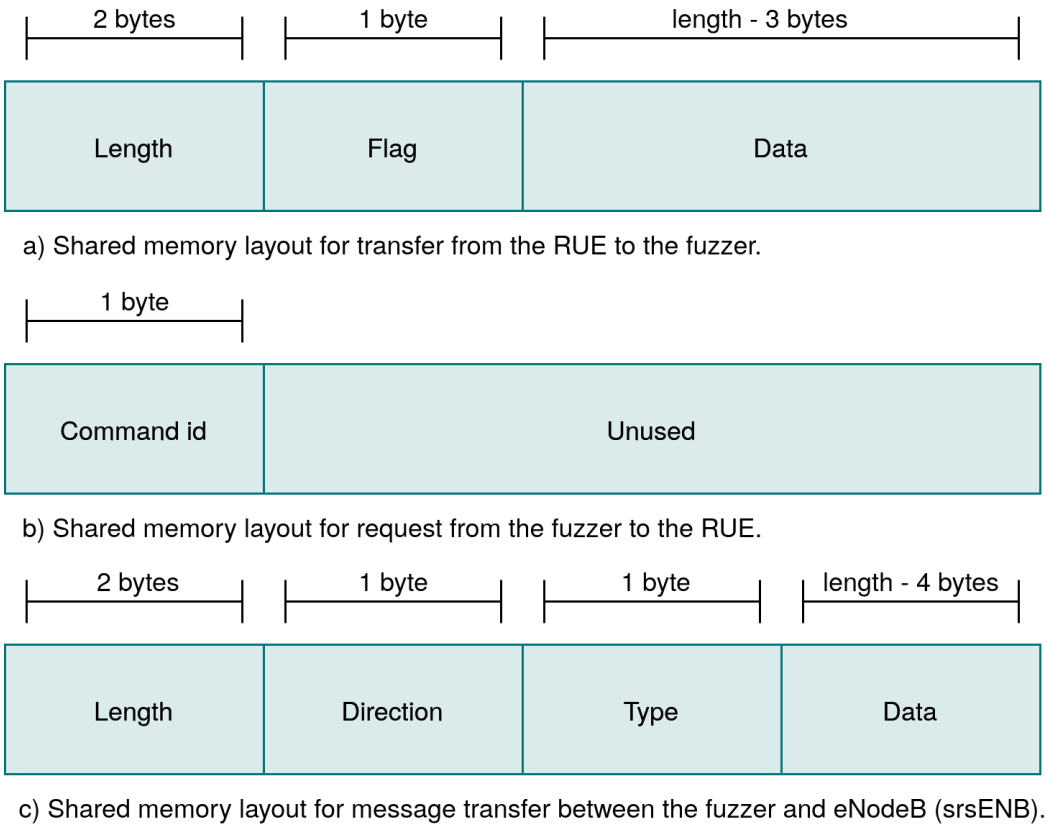


Figure 3.6: The different shared memory layouts used inside the architecture.

problems, each transmission should be acknowledged using the correct command identity in part b) of Figure 3.6. However, when no acknowledgement is observed for a particular data segment the RUE declares failure and aborts the process. Last, the reference UE can be asked to reset the counters in the bitmap for all edges. This is necessary so that the coverage exercised by new inputs have no interference from past test cases. Note that srsUE is an active entity (similarly to the other SRS processes), which is continuously running. This is in contrast to passive programs that are called upon new fuzzed inputs and terminated after execution.

Aside from the shared memory interface, the UE software is also updated with a socket interface. This socket interface is denoted in a purple dotted line in Figure 3.1 of the overall architecture. Similarly to the server thread inside the Raspberry Pi providing AT services to the exterior, this server implements a custom AT interface on top of the user equipment. This means that a third process can query if the RUE is up, connected or disconnected from the simulated LTE network. For example, to see if the UE is connected to the network it is checked if srsUE is situated in the EMM_REGISTERED state. Not only read operations are available but also execution type of commands, as called in AT terminology. Like using *AT+COPS* for managing the connection of a modem, the interface provides the possibility to

instruct the reference UE to connect or disconnect from the network using the attach and detach procedure explained in Chapter 2. However, experience and remarks in [49] showed that a simulated user equipment connected through a virtual radio with a simulated bases station can only attach once to the LTE network. After the detach, both the srsENB as the srsUE have to be restarted in the right order to enable a new connection procedure. While restarting both takes easily several seconds, fuzzing with the reference UE enabled would be problematic, as valuable time would be wasted during this activity. Fortunately, tests showed that precisely resetting the RUE's stack resulted in the same outcome, however, with almost no time elapsed during this operation. For that reason, the socket interface exposes a reset functionality to clear the stack in an efficient way and allow repeatedly new connections to the simulated network with fuzzed inputs. As a matter of completeness the server can be shut down using a specific socket request and acknowledges all received execution requests.

3.5 Fuzzer

This section details the interaction of the fuzzer block in the overall architecture as its methodology is given in next chapter. The ultimate goal of this thesis is to fuzz the device-under-test in an intelligent manner and therefore the fuzzer block is an important component in the overall structure. With the architecture mentioned above, alone, the Quectel EC25 modem can successfully connect to the simulated LTE network, but only due to the innovative method designed in [18] fuzzing is possible. In detail, fuzzing is integrated by intercepting messages in the eNodeB and sending them to the fuzzer via shared memory. Besides the fuzzing tasks done by the fuzz-tester, it also controls the other components from above, inside the architectural structure. Thereby starting up and shutting down the different simulators, and communication channels. What follows is the integration and influence of the fuzzer onto the general platform architecture by means of the different interfaces.

3.5.1 Interfaces

As demonstrated in Figure 3.1 and already mentioned in the discussion of other components, the fuzzer exposes two types of interfaces throughout the framework. The shared memory interfaces are indicated in a dotted green line, whereas the socket interfaces are marked in purple.

Three different shared memory connections can be distinguished, whom can be divided in two types of classes. The first and most principle class for the integration of the fuzzer into the architecture is the interconnection between the fuzzer software and the two simulated base stations. Here, Figure 3.6 c) shows the format of the shared memory for exchanging LTE messages back and forth. Similarly to the other layouts, the length field counts the number of bytes occupied by the different fields, while the direction specifies whether the LTE message inside the data portion is uplink or downlink. Moreover, a one byte field called type informs the other side about the type of message put on the shared memory. This can be, for example, an

RRC Connection Reconfiguration Complete or any other message detailed in Chapter 2. Hence, at the different interception points throughout the stack the eNodeB transfers a valid message to the fuzzer upon which it can respond with a mutated packet [18]. In this way, the LTE messages in the MAC (e.g., the Random Access Response) and RRC layer (and hence the NAS layer too) are captured in uplink and downlink. In particular, the messages from the random access, the additional signaling and the detach procedure can be fuzzed. Note that both simulated base stations use this method to fuzz their corresponding mobile device. The second class was already explained in the reference UE and permits the fuzzer to access and control code coverage at the other side, using the encoding in a) and b) of Figure 3.6. Furthermore, like proposed in [18], the communication channel is built on top of a shared memory module in C designed by another team of Keysight. It provides low latency for inter-process communication. Due to the real-time requirements of the LTE protocol this property is vital.

The communication channel in purple dotted line, on the contrary, enables the fuzzer to control the device-under-test and the reference UE via their AT interface. At different occasions in the fuzzing process the fuzzer needs to be able to instruct a UE. For example, to connect or disconnect it from the LTE network, or to request its current status. Moreover, to guarantee robust operation of the fuzzer each request is either acknowledged by the receiving party or should been replied to (i.e. for a status request) within a predefined time interval. If not, several reattempts are invoked after which a problem is declared. Only with these characteristics the architecture can run robust and autonomous, whilst steering the fuzzing process dynamically. Moreover, for the socket communication ZeroMQ is used, which is also utilized by the SRS software.

Chapter 4

Methodology of ENPSFuzz

This chapter has as goal to to explain the methodology of the fuzzer component inside the LTE fuzzing framework. The grey-box fuzzer is denominated as ENPSFuzz (Evolutionary Neural Program-Smoothing-based Fuzzer) for LTE, since it applies both an evolutionary technique as well as a gradient-guided algorithm using neural program smoothing. First, a high-level architectural overview is given on the design principles of ENPSFuzz. The rest of the chapter zooms in on several important aspects of the fuzz-tester's design.

4.1 System design

In this section, the design principles of the fuzzer block are considered together with the challenges in intelligent fuzzing of stateful protocols compared to stateless programs.

As proposed by [18], the system design of the fuzz-tester should operate as a network protocol fuzzer (i.e., not for emulated IoT software as discussed in Chapter 5). Hence, ENPSFuzz, short hand for Evolutionary Neural-Program-Smoothing-based Fuzzer, is a grey-box fuzzer designed to fuzz the LTE implementation on an IoT device that serves as user equipment. To converge to the solution of the fuzz optimization problem, it applies both an evolutionary technique as well as a gradient-guided algorithm using neural program smoothing inspired by Neuzz [25] and PreFuzz [4].

Before going deeper into the system design of ENPSFuzz, it's worthwhile to note that the ultimate goal of ENPSFuzz is to craft invalid but plausible inputs that pass the initial validity checks. However, fuzz-testing for stateful protocols has to address some extra challenges in contrast with stateless program fuzzers such as AFL [54], e.g., testing a PDF parser. Stateless programs only expect one type of input and don't have an internal state, like communication protocols have. Moreover, current research mostly focuses on fuzzing stateless programs, as can be seen from the literature review in Appendix C. The first complication that arises in stateful fuzzing is fact that communication protocols maintain an internal state. This state is driven by by several types of events, like the succession of received packets. Without

the ability to traverse the state graph of the protocol, the fuzz result will be limited in terms of implementation reached for security testing. Reception of a correct input message, but at the wrong time forces a restart of the protocol. Second, the structure of the individual packets are required to comply to the strict format of the specified protocol. Malformed messages are expected to be discarded early on by the grammar analysis before really testing the DUT. To this end, communication protocols, such as LTE and TCP, follow a layered approach, as discussed in Chapter 2. Here, each layer adds a header to upper layer data, the payload, and passes it on to the next layer. Furthermore, the packets are composed of different fields that can be interpreted using the protocol specification. Hence, the generation of structured inputs as well as smartly mutating packets is of utmost importance for a network protocol fuzzer.

As a result, to integrate protocol awareness into ENPSFuzz several techniques are being used. First, the fuzzer block learns and validates the state graph by sniffing packet traffic between the base stations and the target UE. More information about this topic can be found in Subsection 4.2.1. However, note that the eNB software helps the fuzzer in creating correct LTE messages at the right moments in time. With this aid ENPSFuzz can communicate error-less with the UE. Second, the fuzz-tester incorporates intelligence about the messages' specification using WDissector library, as can be read in Subsection 4.2.2. Moreover, these specifications are used to mutated packets with the 3-level mutators from Section 4.3.

Despite the explained difference in complexity between conventional grey-box fuzzers for stateless programs and network protocol fuzzing, ENPSFuzz has to overcome the extra hurdle of the test program being an embedded IoT device. As stated in Chapter 5, most IoT fuzzing techniques leverage a black-box approach (e.g., Boofuzz [55]) as they only have access to input and output behavior of the DUT. In other words, the lack of feedback mechanisms from the device-under-test itself, like code coverage which is mostly adopted, causes IoT fuzzers to resort to less efficient black-box implementations. Hence, ENPSFuzz tries to bridge the gap between (stateless) code coverage-based grey-box fuzzers and network fuzzing on IoT systems. Therefore, the LTE fuzz-tester integrates a new idea of a reference software (i.e., RUE) that provides several types of feedback. Namely, the RUE's edge coverage is used as an indication for the fitness of an input making it possible to leverage grey-box techniques from literature. In particular, ENPSFuzz integrates a gradient-guided technique using machine learning and an evolutionary approach. Especially, the gradient-guided technique is based on the work of Neuzz [25] and PreFuzz [4]. However, both papers target stateless programs. Besides the code information, ENPSFuzz compares the I/O behavior of the reference UE with the tested target to observe misbehavior. The interested reader is referred to Chapter 5 for more information about IoT feedback and bug oracles.

4.1.1 High-level overview

ENPSFuzz (Neural-Program-Smoothing-based Fuzzer) is a grey-box fuzzer for LTE implementation on mobile devices. To converge to the solution of the fuzz optimiza-

tion problem, it applies both an evolutionary technique as well as a gradient-guided algorithm using neural program smoothing.

The underlining architecture of ENPSFuzz, shown in Figure 4.1, is implemented using an evolutionary approach and composed of several stages. Hence, each stage has its own functionality during the fuzzing process. In more detail, the underlining structure stores a seed corpus where all interesting seeds are awaiting to be mutated in the near future. They are interesting in the sense that some notable feedback features were detected during executing of the seed. This can be, for example, new code coverage attained, exploration of some new state/state transitions or a dissimilarity in comparison with the reference UE. Furthermore, every seed is comprised of a sequence of messages that is received by the device-under-test during one protocol run. For fuzzing the LTE implementation, this is an entire attach and/or detach procedure from the background chapter. Besides containing different packets, the seed also saves some meta data, like the number of mutants derived from it, the attained edge count, the execution time, the freshness and other fuzzing variables together with the identity of the parent seed. Using this parent info the seed pool can be converted into a parent-child relationship tree. Here, children differ with their parent in one or more mutations. Hence, thanks to this evolutionary process of seeds it is possible to have child seeds with more than one viable mutation that are malformed, but still pass the initial syntax checks. The ultimate goal is to explore deeper code paths inside the target. Next, each message inside a sequence of messages, or seed, is wrapped inside a packet instance. This specific structure analyzes the message at hand and provides custom mutators that are designed to target the available fields inside it.

The goal of the architecture in Figure 4.1 is, thus, to support this underlining evolutionary process for different modes of operation (i.e., Section 4.5), which will become more clear shortly. What follows is the description of the system's components. Note that the fuzzing process is divided into a *pre-processing phase* (i.e., I) and a *dynamic analysis phase* (i.e., II). Where the first phase focuses on the preparation and the control function of the fuzzer, the dynamic analysis phase incorporates the actual fuzzing loop. This main fuzzing loop is iterated over and over again until the available time is depleted or the maximum number of test cases are executed. In summary, during one iteration of the dynamic analysis phase a seed is scheduled from which one or more test cases are derived. Next, each test case is executed on the DUT, the feedback mechanisms are monitored and evaluated to update the fuzzer's configuration or report any faulty behavior.

I-A: Control Logic. From Chapter 3 it's clear that ENPSFuzz operates in a much larger framework that needs to be coordinated efficiently. Hence, with aid of the control logic the fuzz-tester takes on the tasks to (1) start up and tear down the LTE simulators (i.e., srsENB and srsEPC), reference UE and neural network process, (2) manage (i.e., start, restart or stop) the connections over both shared memory and sockets to the base stations, reference UE, neural network and the actual system for testing. By use of these communication channels, ENPSFuzz is able to control the different processes during the main loop. It communicates with other components to obtain the necessary information such as, for example, errors, code coverage, and

the status of the user terminals. Due to ENPSFuzz being the main controller, the entire fuzzing framework can run efficiently, autonomously and dynamically during the whole fuzzing procedure.

I-B: Neural Network Module. The neural network module is a standalone component, which is used to model the relation between the reference UE's branching behavior and the different seed inputs. Therefore, it incorporates the necessary logic to generate gradients such as a neural network, and training and data pre-processing techniques. Moreover, the fuzzer can send training commands to the module over socket communication. In ENPSFuzz, the generated gradients from the program-smoothing-based method are used by two gradient-guided algorithms (i.e. to mutate selected seeds) in the input generation stage. Section 4.4 gives more information about the module's implementation.

I-C: Packet Modeling. Packet modeling serves as a way to understand the syntax and layout of the different types of messages during the LTE protocol. The fuzzer inherits the WDissector library from the initial design [18], which is a user-friendly extension to the Wireshark tool. It prevent ENPSFuzz from formulating messages that are easily being rejected due to obvious violations of the message structures. However, ENPSFuzz extends the original design by increasing the field granularity. Moreover, a major benefit of this Wireshark-based approach is the ability of ENPSFuzz to comprehend other Wireshark supported protocols as well. The WDissector C/C++ API works by dissecting the different fields inside a packet as can be seen from Subsection 4.2.2. Hence, WDissector supports protocol aware mutations inside ENPSFuzz.

I-D: State Graph Modeling. The second challenge in stateful fuzzing is the state graph modeling. As discussed in Subsection 4.2.1, the target's state graph is modeled using observed traffic between the DUT and the base station. Based on the theory of Mealy machines and the input and output behavior of the DUT, a finite-state graph is constructed for the protocol implementation of the target device. To this end, ENPSFuzz updates its internal graph by observing the fuzzed protocol runs. Hence, during the main fuzzing new states and state transitions are added using the self-learning capabilities. Furthermore, the fuzzer can also be supplied with a JSON-formatted state model which can be either formulated by the tester, or coming from an older fuzzing campaign. The resulting state model (i.e., .dot file) can be viewed using a graphical description language viewer.

I-E: Target Instrumentation. While IoT instrumentation on the physical device-under-test is not feasible, the Clang compiler is used to instrument the srsUE software by inserting customized LLVM SanitizerCoverage callbacks [53]. Only specific layers of the RUE's stack are instrumented in this way. As detailed in previous chapter, the LLVM method has several advantages and provides the fuzzer on the fly with code coverage guidance during the execution of inputs. More precisely, ENPSFuzz uses edge coverage as metric.

I-F: Seed Corpus. For execution during the main loop of the fuzzer, ENPSFuzz has to be provided with an initial population of seeds. Either the corpus can be supplied by means of a JSON-formatted population file that may be manually constructed, or coming from previously stored fuzz trials. In addition, ENPSFuzz

can be instructed to sniff the entire LTE communication between the device and the eNodeB, and store it as an initial seed. Hence, this captured protocol run will serve as the starting point for the dynamic analysis phase. Whenever all the pre-processing tasks are done, ENPSFuzz is ready to start the fuzzing campaign during the main fuzzing loop.

II-A: Scheduling. The tasks of the scheduler is to select one of the seeds to be mutated, from which new inputs will be derived and send to the LTE modem. Here, it is known that the scheduling stage has to deal with the exploration vs exploitation trade-off, also denoted as multi-armed bandit problem [56, 57]. ENPSFuzz itself provides two different types of schedulers: a random scheduler choosing, trivially, a seed at random and a round robin scheduler which iterates over the seed pool in the same order. Furthermore, the actual scheduler can be chosen freely and, if necessary, the scheduling block can be extended with a new type of scheduler.

II-B: Input Generation. The goal of this stage is to mutate the seed given by the scheduler using different mutators of Section 4.3. An input that results from applying one or more mutations to the scheduled seed is denoted as a *test case*. Moreover, the *energy* assigned to a scheduled seed corresponds to the number of test cases derived from the seed. Inside the mutation block ENPSFuzz defines several mutation processes or *modes of operation*, that leverage diverse techniques:

- **Stress testing.** By sending unchanged messages rapidly after each other stress is put onto the device to see if it is acting abnormally.

- **Insertion mode.** Using this mode the protocol flow is changed by using message-level mutators from Section 4.3. This means that test cases are crafted which are composed of an incorrect series of packets. Hence, the device-under-test is tested to see if it handles abnormal protocol interactions correctly. For example, the target should not accept LTE interactions without the authentication procedure. Note that this is the only mode working with message-level mutators.

- **Sprint mode.** When specific fields inside specific packets should be targeted, this mode of operation supplies the tester with such types of mutation strategies. As an example, all values of a certain given field can be tested with the exhaustive search strategy. Note here that one test case is generated during one iteration of the main fuzzing loop. Hence, several executions of the dynamic analysis phase are needed to test all possible field values.

- **RAND mode.** This evolutionary mode of operation generates one or more test cases from the scheduled seed depending on its fitness score. Furthermore, every aspect of this mode is fully random, e.g., the field to be mutated, the number of stacked mutations and the field-level mutators to be used. Note that the name of this mode comes from its RANDomness during decision making.

- **GRAD mode.** GRAD leverages gradient guidance by means of the program-smoothing-based option provided by the neural network. As the gradient of

the scheduled seed indicates byte positions that have more chance to explore new code coverage, the mode ranks the different bytes in increasing importance and generates a number of test cases for several values of the promising fields. Hence, as the main building block of the mode is the seed's GRADient it is called the GRAD mode.

- **GRAND mode.** Similarly to the deterministic GRAD mode, GRAND makes use of the modeled RUE's branching behavior as well. However, instead of ranking the different bytes, the scheduled seed is divided in a custom number of segments where the probability to mutated one or more fields inside a certain segment depends on its average gradient. The higher the average gradient of the segment, the more likely the segment is selected for mutation. Next, during one iteration of the fuzzing loop one segment is selected based on the calculated fitness-based probability distribution. Finally, several test cases are derived from the seed that differ in the selected segment. To help the exploration of interesting mutations the selected segment is mutated with a nondeterministic process similarly to RAND. Note that GRAND is the contraction of GRAdient and RANDomness.

For more details about the working of the different modes of operation Section 4.5 can be consulted. However, note that it is possible in ENPSFuzz to retrieve test cases from different mutation modes during one fuzzing iteration by putting them all together in the *execution queue*. As the name already indicates, the execution queue holds all newly generated test cases that will be provided as input to device during the next execution stage.

When fuzzing the LTE protocol, it was observed that capturing a protocol run at time x and replaying exact the same seed, without any mutation, at time y resulted in the LTE device refusing to perform the protocol (e.g., to connect to the network). Investigations showed that the exact same message (e.g., the RRC Connection Setup) can change slightly over time. This can be explained due to the fact that the LTE protocol includes some security mechanisms which require, among other things, unique nonces, (temporary) identity numbers, sequence numbers, etc. Hence, just replacing the original message with the corresponding packet from a prior, but unfuzzed input didn't work. To mitigate this issue ENPSFuzz introduces the concept of *patches*. A patch for a certain field indicates the location of the field, the new value of the field and the history of mutators applied to it. Instead of mutating the packet itself, mutators create or update existing patches inside the packet instance. So whenever a real message is observed during the execution stage, the corresponding patches from the mutated packet are applied to the fresh message. This in contrast to the initial idea of replacing observed message as a whole. The first benefit of this approach is that the entire mutation history can be seen for a packet. Second, different UEs can be fuzzed at the same time with the same fuzzed packet (i.e., for the DUT and the reference UE).

II-C: Input Evaluation (execution and monitoring). The input evaluation stage executes the different test cases from the execution queue one by one and

monitors the corresponding feedback mechanisms. In particular, to execute a test case the fuzz-tester uses two independent threads: one for the DUT and one for the reference UE. The thread processes are responsible for the communication with their corresponding user equipment and base station over the provided communication interfaces. This includes sending of AT commands to trigger different procedures, observing the UE's status, and exchanging messages with the eNB. The latter happens over the shared memory and enables to fuzz the protocol interaction. Furthermore, the process fuzzing the RUE also manages the code coverage information. This includes resetting the bitmap upon a new input, and receiving and storing the correct edge counts from the RUE for further use. Moreover, regarding the monitoring tasks the fuzzer stores the activated edge coverage and the input/output behavior for both the target device and the RUE. To this end, the two processes are made so to interact dynamically with their target platform upon, e.g., observing that no responses are send out by the UE. Furthermore, ENPSFuzz stores the I/O behavior for each test case and updates the corresponding FSM with it. Another important aspect is that the input evaluation stage always starts it execution of a test case from a clean state. This means that the user equipments are managed so that they only try to interact with their base station during the execution stage (e.g., to connecting). The clean state is, thus, achieved by disconnecting the targets from the network upon leaving the input evaluation stage. Furthermore, it is imposing that they cannot send new messages afterwards. Note that there are no problems with the real-time requirements of the LTE protocol, since the mutation and other time consuming task are done in a separated stage. Hence, the execution logic only has to deal with the execution of test cases, and recording of their corresponding feedback.

II-D: Configuration Updating. The main purpose of the configuration updating stages it to evaluate all feedback mechanisms detailed in Chapter 5. These mechanism were monitored and stored during the execution in previous stage. In particular, the different bug oracles are investigated to observe any misbehavior of the DUT. This includes the *liveliness check* to see it the AT interface is responding, the *differential check* to investigate differences in I/O behavior, and lastly the *connectivity check* comparing the connectivity between the devices. Apart from the IoT bug oracles, the executed input can be added to the seed corpus. This is the case if either the edge coverage information from the reference UE shows the activation of new edges, or new state/state transitions are exercised. In essence, updating the seed pool is an important step in the evolutionary architecture to increase the fitness of the fuzzing process. It evolves the seeds with viable mutations across several fuzzing iterations.

II-E: Bug Analysis and Reporting. As the name already suggests, the bug analysis and reporting stage logs information about inputs that may have induced some sort of anomaly captured by analyzing the bug oracles. When a faulty behavior is observed, a report is created containing all information used in the configuration updating stage. Hence, the tester can now manually observe the report and trace back the bug. However, as discussed in Chapter 5, the actual problem can be an input prior to current executed test case, e.g., in case of a late crash. Therefore, all uplink, downlink and fuzzer operations are logged through out the different stages in

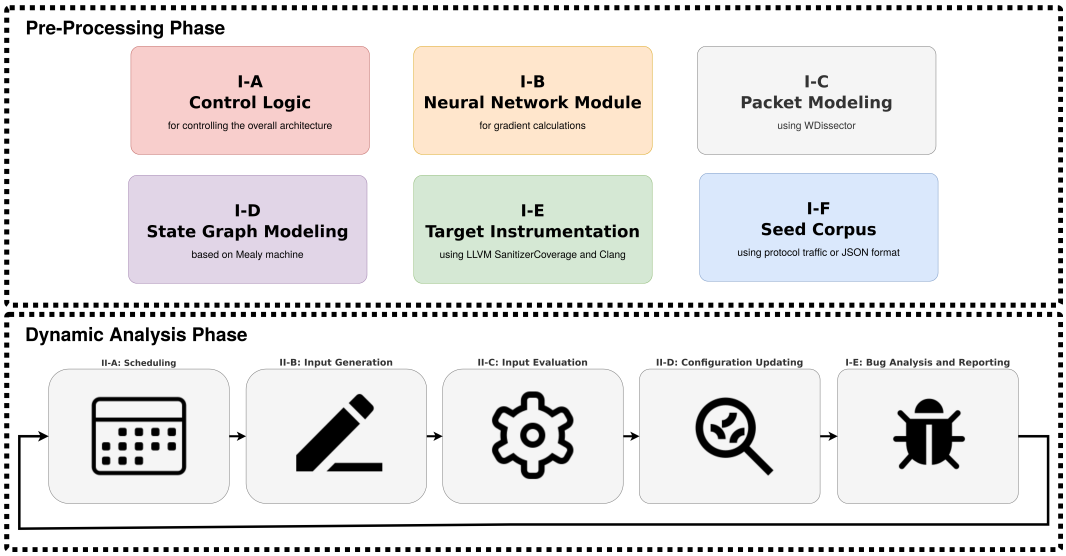


Figure 4.1: The architecture of ENPSFuzz for LTE which is built from different functional stages designed according to an evolutionary approach. The pre-processing phase contains the necessary steps prior to fuzzing, while the dynamic analysis phase is the main fuzzing loop.

ENPSFuzz. Hence, the logs enable further analysis.

4.2 Protocol awareness

In this section, the challenges regarding stateful fuzzing are addressed. In the first part, it is explained how ENPSFuzz constructs a state graph for the LTE implementation of the target device, while observing input and output traffic. Next, the second part deals with the problem on how to supply the fuzzer with knowledge about the messages' structures. However, note that ENPSFuzz is a mutation-based fuzzer as srsENB provides the fuzzer with "protocol intelligence" on correctly generated messages at the right moments.

4.2.1 State graph modeling

For ENPSFuzz the purpose of state graph modeling is to construct a graph which indicates the feasible sequence of inputs to the device-under-test with their corresponding outputs. If an incorrect type of message is received at a certain state, it is likely to cause the system to roll back to the initial state. Hence, with this state automaton the fuzz-tester has an overview of different message sequences the target accepts and replies to. The main usage for this is three-fold: (1) it can be used to generate out-of-order message sequences (see Section 4.5), (2) it supports the recording of new transition/state feedback and (3) provides the tester a graphical way to check the target's I/O behavior manually. Essentially, ENPSFuzz builds a

Mealy-based state graph for the DUT using the observed RRC and NAS traffic, captured at the upper layer's interception points of the eNodeB.

A Mealy machine is a finite-state machine where the output function and transition function depend on both the current state as well as the current input(s). Figure 4.2 illustrates two real but different LTE protocol runs, where both runs combine the attach and detach procedure. Not that the short run is similar to the long run, except for the some extra signaling. As discussed in the background, when the UE is already known to the MME several procedures are not needed in the additional signaling of the attach procedure. The convention in the figure is to show the input triggering the state transition before the "/", while the outputs resulting from the current state and input are mentioned after it. As an example the transition from the idle state reads: when the UE receives the instruction to do an attach request, it responds with a preamble. Moreover, the working of the state model goes as follows. Note that now an input is just one message and not an entire seed. First, the UE has to receive a downlink message or AT command from the fuzzer that triggers one or more outputs from the platform. Then, the state graph transitions from the current state to the next state. This means that the next state is characterised by the current state and the sent input. Hence, different inputs to the same state result in the transition to different subsequent states. Putting this together, ENPSFuzz observes an entire fuzzed protocol run and leverages the downlink and uplink messages to update the model starting from the initial state. It tries to follow the established model, however, when the UE responds to a new input for a particular state, a new transition and corresponding state is created. In other words, every time a seed exposes a new transition and hence state the model is updated (i.e., denoted by a new type of message to the current state triggering an output from the UE). Note that the created model is not a real finite-state machine, but more a manner to visualize the input and output behavior of the DUT.

During the pre-processing phase the state graph either starts from scratch or can be loaded from a JSON file that was, for example, saved in another ENPSFuzz fuzzing campaign. Either way, the fuzzer expands the model on the fly by executing test cases on the test platform in the dynamic analysis phase. Therefore, the self-learning procedure operates dynamically during every execution. Moreover, after hitting the time budget the graph model can be saved and loaded for future campaigns, or analyzed using a graphical visualization tool.

4.2.2 Packet modeling

In the LTE protocol each type of message is characterised by a standardized structure, which complicates the task of the fuzzer in sending mutated messages. Hence, ENPSFuzz integrates the WDissector library as proposed by [18]. As this proprietary Keysight C library relies on the functionalities given by Wireshark, the fuzzer component incorporates knowledge regarding any network protocol supported by the Wireshark tool. The usage of the sniffing tool enables the fuzzer to dissect the different types of packets and to inspect the different protocol layers with their corresponding headers and payloads. In practice, Wireshark links consecutive bits

Figure 4.2: The input and output behavior of the LTE modem based on the theory of Mealy machines. The graph shows two real seeds composed of an attach followed by a detach procedure.

that belong together into a field. Such a field has a specific purpose in a network protocol (e.g., to inform the other side about the available buffer size). On top of the Wireshark library, WDissector provides several services in the C or C++ environment, like easy navigation through the dissected message tree, extracting queried fields from the packet, and setting the protocol to be analysed. Hence, it denotes fields inside a packet using a custom structure of four elements: the length, the offset, the mask and the mask offset [18].

With the interface to the dissector tool available ENPSFuzz can analyze the packets it is provided with. However, in LTE it is common that fields inside the same message which are, although physically separated and have a different overall meaning, still bear the same name (e.g., the nas_eps.spare_bits or the gsm_a.len field inside RRC Connection Reconfiguration message). To resolve this issue, the fuzz-tester characterises fields not only by their name, but in addition are assigned a

unique identity within the duplicate field name. Moreover, ENPSFuzz makes use of this field knowledge in several situations. In particular, when the fuzzer observes a new packet (e.g., a new type of message), it stores the message inside a new packet object and uses its field capabilities to investigate the different fields present. On the other hand, the field-level mutation operators are designed to target specific fields with the help of the dissection tool. Moreover, the mutation stage knows which fields it can mutated in packet, as the packets contain a list of dissected fields.

As an interesting side note, the fields in LTE can be grouped in two broad categories, namely *structure fields* and *value fields*. So whereas bits inside a value field convey specific data that is used during the protocol operation (e.g., the available buffer size), structure fields inform the receiving side about the message layout, thus, making it possible to decode the packet. As an example of a structure field, per.optional_field_bit indicates whether the decoder has to read an optional field, which may have been included in the received message. Furthermore, the LTE specification states that the LTE messages can contain extra redundant bits, known as *spare bits for future usage*, to be used for potential upgrades of the LTE protocol during new releases. Hence, UEs implementing older releases don't have to worry about these bits and can just ignore them. At the same time, newer mobile terminals can make use of these bits to leverage new features introduced in the latest LTE releases. On the contrary, when a certain field is characterised by some *spare values* it indicates that not all values of the field have a meaningful interpretation inside the LTE protocol. Upon observing a spare field value, the UE has to reject it instead of just ignoring it. This is because spare values are undefined by the standard and should thus never been used. As a result, value fields are an interesting vector to target during fuzzing as they show if the device is compliant [18].

4.3 Mutation operators (3-levels)

ENPSFuzz is a mutation-based protocol fuzzer that comes with the mutation operators from Table 4.1. As can be seen, three types of mutators can be distinguished: field-level, byte-level and message-level operators. Whereas field-level mutators change the value of a target field that is dissected using WDissector, byte-level mutators can be used to update an entire byte of a packet. Unlike the latter two, message-level operators change the message order inside a seed. This means that a series of packets can follow an abnormal order due to the insertion of a new packet, the deletion of an existing packet or the replacement of an original message inside the seed variable. Here, the state model can be used to guide message-level mutators and reflect correct and invalid protocol runs that were tried before. Note that the mutation operators indicated by * have a random engine that supports a fully random counterpart. For example, the random counterpart of the "set consecutive zero bits" operator first selects a random start bit inside the given field (e.g., bit number 2 inside the dissected field "1101111"). Then, it draws a random number, while taking into account the field length and the start position, telling the number of consecutive zero bits to be placed starting from the start position (e.g., 4 bits). Hence, for this

Mutator type	Mutation operator	Description
Message-level mutators	Insert packet*	Inserts a well-formed message at the given position inside the message sequence
	Delete packet*	Removes the packet at the given position inside the series of packets
	Replace packet*	Replaces the packet at the given position with another well-formed message
Field-level mutators	Increment*	Increments the value of the dissected field with the given value, if possible
	Decrement*	Decrements the value of the dissected field with the given value, if possible
	Flip bit*	Flips the given bit inside the dissected field, if possible
	Flip bits	Flips a random number of bits inside the dissected field
	Set consecutive zero bits*	Sets a given number of consecutive bits of the dissected field to zero starting from the given position, if possible
	Set consecutive one bits*	Sets a given number of consecutive bits of the dissected field to one starting from the given position, if possible
	Set to value*	Sets the value of the dissected field to the given value, if possible
	Set to minimum	Sets the dissected field to all ones
	Set to maximum	Sets the dissected field to all ones
Byte-level mutators	Increment*	Increments the value of the given byte with the given value, if possible
	Decrement*	Decrements the value of the given byte with the given value, if possible

Table 4.1: ENPSFuzz mutation operators.

example the random result will be "1000011".

4.4 Neural network module (for gradient-guided optimization)

Like proposed by Neuzz, [25] ENPSFuzz integrates a similar neural network to smooth the program behavior of the model UE. However, opposite to Neuzz which is designed to fuzz stateless programs, ENPSFuzz leverage the neural network technique to test the LTE protocol of a physical platform. Hence, by placing a neural network module inside the fuzz architecture gradient-guided mutation strategies can be developed that make use of gradients or higher-order derivatives. In particular, two mutation modes incorporate this optimization, namely one based on the sorted gradient and another one taking into account the real gradient values. However, the program smoothing method can also supports other well-known optimization techniques such as Newton's method and L-BFGS [25]. Note that the study of [4] revealed that different neural networks (i.e., CNN, LSTM and Bi-LSTM) have limited impact on the efficiency of the edge exploration in program-smoothing-bases fuzzers like Neuzz

[25] and MTFuzz [31]. This can be explained due to the fact that the networks are used to model input to edge coverage behavior, instead of trying to predict the interesting bytes that trigger unexplored edges. More advanced neural networks often excel in prediction tasks. Therefore, a feed-forward fully connected NN is used to model the (existing) branching behavior [25].

4.4.1 Training

Inspired by Neuzz [25] and its successor PreFuzz [4] the training of the neural network module is implemented as follows.

The purpose of the training is to learn the machine learning model the relation between a sequence of input bytes and the corresponding control edges exercised (i.e., branching behavior). Specifically, this mapping can be formulated more formal as $h : \{0x00, 0x01, ..., 0xff\}^k \rightarrow \{0, 1\}^l$ showing that the NN expects k-byte inputs and transforms it in an edge bitmap of size l. When the trainable weights parameters of h are given by ψ, the training process tries to obtain $\hat{\psi}$ minimizing the loss function. Hence, denoting the given set of seeds by X and the set of corresponding edge coverage bitmaps by Y forming the training samples (X, Y), the parametric function $h(x, \psi) = y$ requires to the learn the branching behavior based on the ground truth. Therefore, the training task has as purpose to find the model's parameters $\hat{\psi}$ that minimize the cumulative loss between each ground truth $y \in Y$ in the training set and the output of the neural model [25]:

$$\hat{\psi} = \arg\min_{\psi} \sum_{x \in X, y \in Y} L(y, h(x, \psi)) \tag{4.1}$$

which is written in function of the loss function $L(y, h(x, \psi))$. Here, the used distance metric scoring the accuracy of the weight parameters ψ is the binary cross-entropy that calculates the difference between the true edge coverage bitmap and the bitmap produced by the neural network. For this let y_i be the i-th bit in the ground truth's bitmap and $h_i(x, \psi)$ the corresponding bit predicted by the neural network's trained function h using model parameters ψ. Note that a bit inside the bitmap indicates if the edge was triggered or not. Hence, the loss function can be written using the binary cross-entropy as follows [25]:

$$L(y, h(x, \psi)) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \cdot \log\left(h_i(x, \psi)\right) + (1 - y_i) \cdot \log\left(1 - h_i(x, \psi)\right)]. \tag{4.2}$$

To supply the neural network module with the necessary training samples (X, Y), ENPSFuzz sends all executed test cases together with the respective edge coverage information, obtained from the reference UE, to the NN. Different from the original program smoothing-based implementations, an input x for the training process is a concatenation of packets that makes up an entire protocol run (i.e., seed). However, the challenge in modeling the program's branching behavior with neural networks is that protocol runs can be variable in size, whereas feed-forward NNs expect fixed size training data. As a solution, the longest seed in the training set is set as threshold

size. All inputs smaller than the threshold are padded at the end with null bytes. Another dissimilarity with the approach of Neuzz [25] is that instead of incremental learning, which leverages inputs that increase code coverage, ENPSFuzz' module always considers every test cases for (re)training. While this training technique comes with an extra time cost, it makes the machine learning model aware of seed mutations that result in low coverage. For example in LTE, the DUT can reject the connection, when the field controlling the power is set incorrectly. However, the extra cost in training is mitigated by allowing the fuzzer to switch to the evolutionary approach during (re)training sessions, as can be seen in Subsection 4.5.7. Furthermore, as the accuracy of the neural network highly impacts the efficiency of the gradient-guided mutations, the NN module can be instructed by ENPSFuzz to retrain its parameters at any moment in time. These retraining sessions take into account all the currently available data. Upon finishing the training, only the gradients of the seeds inside corpus are update with their new values.

Moreover, the training data is first pre-processed before actually being used, similarly to [25]. The reason for this is that in general only a small selection of the total edges are exercised by the training data, leading to a bias inside the data. An example of such a phenomenon is when certain edge labels are applicable to all test cases. Hence, this label correlation prevents the model from converging to small losses and is denoted in machine learning terminology as *multicollinearity* [25, 58]. Therefore, edges in the training data that are always exercised together are merged into one label following the common machine learning principle of dimensionality reduction. Furthermore, edge labels that have not been triggered inside the training data are also left out. This obvious as the model learns the current branching behavior. For the instrumented stack of srsUE these optimizations reduce the label count from 33,413 to around 4,900. The data pre-processing procedure is repeated for every training session, and therefore clustered edge labels may split again when new fuzzing data indicate a drop in their correlation [25].

4.4.2 Gradient calculation

Gradient calculations enable gradient-guided mutation algorithms to find input positions that have a high likelihood upon modification to trigger new code coverage. In other words, the gradient specifies how much an input byte should be mutated to change one of the output neurons of the final layer from 0 to 1. Therefore, each input byte receives a value between -1 and 1 identifying the consequence of the given input on a certain edge. Hence, the closer to 0 the less important towards new edge exploration.

Let $y = h(x, \psi)$ be the relation trained by the neural network and $y_i = h_i(x, \psi)$ the output of the i-th neuron of the final layer. The latter indicates if the model expects the input sequence x to exercise the particular edge label. Then $\mathbf{G} = \nabla_x h_i(x, \psi) = \frac{\partial y_i}{\partial x}$ describes the gradient $\mathbf{G}$ of the i-th neuron with respect to the input x. From this mathematical expression it can be seen that the dimension of $\mathbf{G}$ equals the length of the input sequence, as the gradient calculates the effect of each input byte on the particular edge output y_i [25].

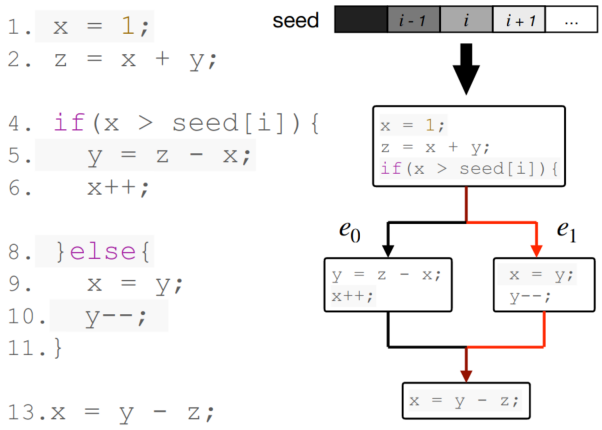


FIGURE 4.3: An example of neural program-smoothing rationale [4].

.

The key aspect in generating gradient vectors for seeds is the process of identifying explored edges that are worthy to be explored and hence to be derived. This will become more clear shortly. In particular, as pointed out by PreFuzz [4] the method of Neuzz [25], which selects explored edges randomly results in the activation of existing branching behavior. Therefore, ENPSFuzz integrates the improved *resource-efficient edge selection mechanism* of [4]. In summary, from the population of final layer edges only the ones, which have a ratio of its explored "sibling" edges over its total number of "sibling" edges lower than a given threshold, are retained as candidates. Intuitively, an edge from which all "sibling" edges are discovered has no capabilities anymore to deliver new code coverage and would be resource inefficient to be used again. As can be seen from Figure 4.3, e_0 and e_1 are "sibling" edges, since they have the same prefix edge. Suppose that the value of the corresponding bits inside the seed matches the access condition of e_0, then this edge will be explored due the one-to-one mapping between the input bytes and the exercised code path. Therefore, mutating the corresponding bits with the gradient technique can explore its "sibling" edge e_1, if changed to the correct access condition. Essential for this method is the run-time edge exploration states telling, for example, for each edge its explored and unexplored "sibling" edges. As PreFuzz [4] compiles its programs with AFL-specific instrumentation tools, they construct the edge exploration states by first decompiling the executable into assembly-level instructions with the aid of the inserted AFL instrumentation flags. Next, they examine the control flow from the parsed instructions automatically. However, as mentioned in Chapter 3 the reference UE is instrumented with LLVM SanitizerCoverage. Therefore, ENPSFuzz updated the parsing and analyzing process to the LLVM implementation, making it possible to use the resource-efficient edge selection mechanism [4].

4.5 Modes of operation

The section called "Modes of operation" considers the different modes that are integrated in the mutation stage of the fuzzer. Especially, each mode of operation has its own strategy to test the platform-under-test (PUT) ranging from subjecting the device to a barrage of non-mutated inputs to gradient-guided techniques. In total six different modes can be distinguished.

4.5.1 Stress testing

As the name already suggests stress testing tries to overflow the target with a quick succession of protocol runs. For this purpose, the fuzz-tester does not have to mutate any field, but has to observe the point at which the UE is connected with as reason to instruct the device to detach again and vice versa. Hence, note that this mode can be accomplished by just one protocol run (i.e., a seed including the attach followed by the detach procedure) and does not require to monitor any code coverage or state feedback during execution. The most critical points in this mode of operation are first to perceive the exact moment when the DUT becomes attached or detached and second to act upon this event as soon as possible. Therefore, the control logic of ENPSFuzz provides a communication channel over sockets to the AT interface of the modem. This can be used to instruct the target what to do, while the interception points in the eNodeB provide the fuzzer with information about the messages being exchanged at any moment in time.

4.5.2 Insertion mode

Test cases that result from the insertion mode differ from other inputs in the execution queue in the way that they have undergone one of the message-level operators from Section 4.3. As already mentioned in the high-level overview, the state model is updated for every executed test case. Hence, thanks to the insertion mode new states and state transition can be discovered as abnormal message sequences are sent to device-under-test. However, sending out-of-order messages to the target is not straightforward. This is because uplink messages coming from the UE first have to travel to the upper layer of the base station, where it can be captured by ENPSFuzz. Just replacing an observed message in the eNodeB with another type of packet, that should be sent instead, doesn't work for the LTE protocol. The major reason is the fact that the srsENB software sends the replaced message through the same channels as the original packet, even if they are not applicable to new version. This is because the base station has no way to observe this channel difference, let alone the change of type. However, this problem can be solved for the upper layers with a bit of effort, but would still induce some other problems when intercepting at lower layers. To solve this issue effectively, the fuzzer informs the RRC layer of the base station, before the execution takes place, about the order of the upper layer packet types that are planned to be sent during the next connection procedure. As a result, whenever the upper layer inside the eNB receives a new uplink message, it generates

the next packet from the received list instead of the usual packet. An interesting benefit of this approach is that the fuzz-tester sees now the correct type of messages at the different interception points. However, when there is no list of message types given, the eNB assumes that the fuzzer wants to follow the normal communication flow. Since the communication to the UE with an out-of-order seed only requires the insertion of a list in comparison with other mutation modes, this mode of operation is called insertion mode.

4.5.3 Sprint mode

Sprint mode, on the other hand, is extended from the general principles in [18] and makes it possible to apply specific mutation strategies. In other words, each existing packet has an associated sprint data structure that provides the necessary tools to target specific fields using one or more mutation plans. The different mutation plans are:

- **Exhaustive bit flip.** This plan flips all the bits inside the field starting, by the first bit and ending with the last one.
- **Exhaustive zero bits.** This strategy starts by putting one zero at the first position of the field, is executed and puts a zero on the subsequent positions until the end is met. Next, the number of adjacent zero bits is increased and put again in a similar way on all possible positions in the field. This is done up to the length of the adjacent zero bits equals the length of the field.
- **Exhaustive full bits.** Similarly to the previous strategy, the plan puts an increasing number of adjacent one bits on all possible positions in the field.
- **Random values.** This plan tests twice as much of random values as the field size.
- **Maximum.** This strategy updates the field to the maximum value of the field.
- **Minimum.** Opposite to the maximum value, this strategy sets the field to all zeros.
- **Exhaustive search.** This plan tries all possible values of the field one by one.

However, note that during sprint mode always the same seed is used and no feedback is taken into account to optimize the fuzzing process (i.e., code coverage). In fact with this mode of operation the tester can tell ENPSFuzz which fields in which packets should be tested using which mutation plan(s). As an example, all values of the field lte-rrc.releaseCause, which is part of the RRC Connection Release message, can be checked using the exhaustive search strategy to find erroneous behavior.

4.5.4 Randomness mutation mode (RAND mode)

Algorithm 1 shows the summary of how the randomness mode generates test cases starting from the scheduled seed. The outline is part of an evolutionary approach that tries to evolve iteratively to the optimal fuzzing solution taking into account the feedback mechanism from Chapter 5. Hence, due to evolutionary architecture, the randomness mode is provided with seeds that score increasingly better on the different fitness functions over time (i.e., code coverage, state exploration, execution time, etc.). Moreover, the randomness mode itself leverages the field-level mutators to explore fitter inputs. For this purpose, the RAND mode is in first instance designed to discover interesting test case by accident, so to speak, using its integrated randomness. Similarly to AFL havoc, this technique helps to uncover interesting mutants by distracting the fuzzer's attention from targeting the same parts of the seed. In other words, the mode can be complementary to the more intelligent gradient-guided mutations, as they may get stuck in the their belief of what is promising to be mutated. Here, the number of test cases to be derived from the selected seed is determined by its calculate fitness score.

As can be seen in Algorithm 1, the key idea of the randomness mode is to derive one or more test cases from the scheduled seed for execution. Here, the number of test cases to be generated is conditioned on the fitness score of the seed and upper bounded by the input value max_{energy}. Therefore, well-performing seeds will have a score that corresponds closely to the maximum number of inputs. In particular, higher scores are given to seeds that (1) execute rapidly with respect to mean, (2) have more than average edge count, (3) are close to the leaf nodes inside the child-parent tree (i.e., incorporate several mutations), (4) have not been fuzzed much and (5) have lead to a lot of new seeds that were retained (i.e., having many child seeds). After the amount of test cases are defined, each new input is generated iteratively and put inside the queue during the first loop. In this loop a random number between one and $max_{stacking}$ is generated uniformly that designates the number of mutations for the new test case. Next, for each stacked mutation a byte offset is drawn randomly, that defines the byte to be mutated with a random selected mutator. In this selected byte, one field is randomly taken and mutated. However, note that only fields that possess at least one bit inside the byte can be selected for mutation. Last, the randomness mutation mode mutates the field using one of its random field-level operators from Section 4.3, given the packet to which the field belongs. Here again, randomness drives the adoption and execution of the exact mutation operator. Finally, all new test inputs are added to the execution queue and returned. Note that every choice has been made at random during this mode of operation, as can be seen from the algorithm.

4.5.5 Pure gradient-guided optimization via neural program smoothing and sorting of gradient (GRAD mode)

The GRAD mode is, an gradient-guided input mutation process via neural program smoothing and sorting of the gradient as illustrated in Algorithm 2. The general

Algorithm 1 Randomness mutation stage

Input: $seed \leftarrow scheduled\ seed$
$max_{energy} \leftarrow max.\ amount\ of\ test\ case\ to\ be\ generated$
$max_{stacking} \leftarrow max.\ new\ mutations\ to\ be\ included\ in\ a\ test\ case$
Output: $queue \leftarrow new\ test\ cases$
Require: $max_{energy} \geq 1$, $max_{stacking} \geq 1$

```
queue ← {}
length ← length(seed)                          ▷ Length of seed in bytes
score ← calculate_score(seed)
iter ← get_energy(score, max_energy)
for i = 1 to iter do
    test_case ← seed
    stacking ← UR1(max_stacking)               ▷ Determine amount of mutations
    for j = 1 to stacking do
        loc ← UR0(length)                      ▷ Choose random byte in seed
        packet ← get_packet(test_case, loc)
        field ← get_random_field(packet, loc)  ▷ Take random field in byte
        random_mutation_field(packet, field)   ▷ Select random field mutator
    end for
    queue.add(test_case)
end for
return queue
```

principle is to leverage so-called "hot" bytes for mutation as they are indicated by the neural network's gradient to have high chance to the change program branching behavior (i.e., exploring "sibling" edges). The process comes in two mutation granularities, namely byte-level (updated from [25]) and field-level, which is determined by the input variable $flag$.

Algorithm 2 presents the outline of the process, where different test cases are derived given an initial seed and the corresponding sorted gradient. First, the byte positions of the input seed are sorted using decreasing criticality. Meaning that the larger the absolute value of the gradient the earlier the corresponding byte appears in the overall ranking. Given the inputs m and n which denote the m-th and n-th place, respectively, within the ranking, the bytes located between these two boundaries including the boundaries itself will be used for mutation (i.e., first loop). Next, for each of these interesting bytes the sign is investigated to generated test cases that differ with the original seed in either the critical byte itself or some field having at least one bit in common with the critical byte. Especially, the sign of the byte's gradient determines the direction of the mutation, namely increasing or decreasing of the field/byte value. With this information the byte-level procedure adds an new test case is to the execution queue for each of the possible byte values larger/smaller than the current value of critical byte. However, the procedure for field-level mutations is a bit more complex. Here, the fields contained in the interesting byte are requested

and the first max_{fields} fields are mutated to generate new test cases, similar to the byte procedure. Hence, each lower or upper field value receives its own test case depending on the sign. Note that for convenience a max_{fields} of value zero goes over all fields in the byte.

4.5.6 Gradient-guided optimization via neural program smoothing and randomness (GRAND mode)

The GRAND mode updates the gradient-guided optimization technique via neural program smoothing of [4], called *probabilistic byte selection mechanism* to protocols and uses its own nondeterministic stage. Also, the major difference with GRAD is the injection of randomness into the mutation procedure which helps for exploration. Note that the GRAD mutation stage is essentially deterministic and always results in the same test cases for a particular seed over several evolutionary fuzzing runs, if its gradient stays fixed.

First, the mutation stage divides the scheduled seed into several equally sized segments, and then assigns a gradient-based probability to each segment. To this end, each segment seg_i receives a fitness score which is calculated as follows [4]:

$$fitness_{seg_i} = \frac{\sum_{j=1}^{seg_i} grad_j}{length(seg_i)} \tag{4.3}$$

where the numerator sums the gradient of all bytes inside the segment and the denominator indicates the length of the segment in bytes. Hence, the segment score $fitness_{seg_i}$ denotes the average gradient for a particular segment of bytes i. Subsequently, Equation 4.4 states the probability that a certain segment is selected for mutation during the GRAND mode. Here, the gradient-based probability distribution can be computed by dividing each segment's score $fitness_{seg_i}$ by the total sum of fitness scores $\sum_{j=1}^{total} fitness_{seg_j}$ [4]:

$$Pseg_i = \frac{fitness_{seg_i}}{\sum_{j=1}^{total} fitness_{seg_j}}. \tag{4.4}$$

Now the probability distribution is available, the randomness stage of Algorithm 1 will be leveraged to generated mutated test cases for the selected segment. The only difference here is that the obtained test cases will have mutations limited to fields situated inside the chosen segment. Again the amount of generated test cases depend on the obtained score of the seed and a random number of mutations can be observed inside one test case. However, note that the segment size can be adapted dynamically, and that variable sized segments that follow the packet layout inside a seed are possible as well.

4.5.7 Integration

As explained in this section, the mutation stage of ENPSFuzz is equipped with different sorts of mutator processes to test the device-under-test. Interestingly, the

different modes of operations, aside from the stress testing, module can be combined during the execution of the mutation stage. This entails that the test cases from the different modes can be put together into one execution queue, which is send in a subsequent step to the input evaluation stage of the fuzzing loop. For example, a scheduled seed can have inputs derived using both the randomness stage as well as the deterministic gradient-based method leveraging sorting of the gradient. As they are different, they can complement each other in finding an optimal solution to the fuzzing problem and so increase the fitness of the seed corpus. Moreover, the program-smoothing-based mutators have the drawback of requiring the neural network to train on existing branch behavior. This results in the fuzzer being idle and waiting for the neural network to be finished. However, with the integration of the other techniques ENPSFuzz can switch to them when observing that no gradient is available for the scheduled seed. This can also be the case for newly saved seeds when the NN has not been retrained yet.

Algorithm 2 GRAD mutation mode

Input: $seed \leftarrow scheduled\ seed$

$g \leftarrow computed\ gradient\ of\ seed$

$m \leftarrow index\ of\ most\ critical\ byte$

$n \leftarrow index\ of\ least\ critical\ byte$

$flag \leftarrow boolean\ determining\ byte\ or\ field\ mutations$

$max_{fields} \leftarrow max.\ fields\ to\ mutate\ in\ byte$

Output: $queue \leftarrow new\ test\ cases$

Require: $1 \leq m \leq n$, $1 \leq n \leq length(seed)$, $max_{fields} \geq 0$

$queue \leftarrow \{\}$
$rank \leftarrow sort(g)$ ▷ Rank byte positions in decreasing criticality
for $i = m$ to n **do** ▷ Loop over the critical bytes
    $loc \leftarrow rank[i]$
    $byte \leftarrow seed[loc]$
    $packet \leftarrow get\_packet(loc)$
    if $flag$ **then** ▷ Do field-level mutations
        $fields \leftarrow get\_fields(packet, loc)$
        if $max_{fields} = 0$ or $> length(fields)$ **then**
            $max_{fields} \leftarrow length(fields)$
        end if
        for $j = 1$ to max_{fields} **do** ▷ Loop over fields
            $field \leftarrow fields[j]$
            if $sign(g[loc]) = 1$ **then**
                for $k = get\_value(field) + 1$ to $max\_value(field)$ **do**
                    $test\_case \leftarrow seed$
                    $to\_value\_mutator(test\_case, field, k)$
                    $queue.add(test\_case)$
                end for
            else
                for $k = get\_value(field) - 1$ to 0 **do**
                    $test\_case \leftarrow seed$
                    $to\_value\_mutator(test\_case, field, k)$
                    $queue.add(test\_case)$
                end for
            end if
        end for
    else ▷ Do byte-level mutations
        if $sign(g[loc]) = 1$ **then**
            for $k = byte + 1$ to 255 **do**
                $test\_case \leftarrow seed$
                $to\_value\_mutator(test\_case, loc, k)$
                $queue.add(test\_case)$
            end for
        else
            for $k = byte - 1$ to 0 **do**
                $test\_case \leftarrow seed$
                $to\_value\_mutator(test\_case, loc, k)$
                $queue.add(test\_case)$
            end for
        end if
    end if
end for
$return\ queue$

Chapter 5

DUT's Feedback Mechanisms

The purpose of this chapter is to clarify the feedback mechanisms implemented for observing faulty behavior exposed by the DUT and in addition to guide the fuzzing process. Whereas fuzzing software on a general-purpose computer entails a variety of mechanism to catch erroneous events (e.g., memory sanitizers) and can provide feedback to be analysed (e.g., code coverage), fuzzers for embedded devices have been proven to be more challenging in practice [5, 59]. Hence, first an introduction is given to IoT fuzzing and their problems, followed by some prior work on IoT fuzzers, whilst the chapter ends with the feedback mechanisms applied in ENPSFuzz.

5.1 Introduction to IoT fuzzing

Before discussing the part about the feedback mechanisms implemented in this thesis, the different types of IoT devices, their possible faults, the main challenges in IoT fuzzing and some solutions are introduced to the reader.

5.1.1 Classification

The most wide-spread definition of embedded devices as they differ from general-purpose machines is that they (1) are composed of dedicated hardware and software to complete a specific task and (2) are designed to communicate with the physical world through sensors and actuators attached to their peripherals. Moreover, embedded devices are often developed with lowest cost and scarce resources for the given function they fulfill. The distinction between an IoT device and a embedded device is not always agreed upon, but for the remainder of this thesis, an IoT device can assumed to be an embedded device equipped with Internet connectivity. Following the classification of Muench *et al.*, as stated in their paper about fuzzing embedded devices, three different types can be distinguished based on the operating system (OS) inside the device itself [5]:

- **Type-I: Devices with general-purpose OSs.** This type of class is characterised by embedded systems that have a general-purpose operating system on board which is, however, slightly customized to the devices' underlining

hardware. In most cases, high-end embedded devices can be classified into this group as they are fabricated with enough resources to support a general-purpose OS in their overall architecture. Well-known examples are Raspian [60] running on top of the Raspberry Pi and the Linux OS kernel (e.g., BusyBox [61]).

- **Type-II: Devices with embedded OSs.** The second category includes systems that require custom operating systems without the necessity for advanced processor features, like a memory management unit (MMU), but the kernel and application code are still separated from each other. The embedded OS is much more tailored to hardware and thereby giving reliability and efficiency benefits to power constrained platforms. Both FreeRTOS [62] and VxWorks [63] are examples of this kind of real-time operating system.

- **Type-III: Devices without OS.** Devices in this class don't have an intervening operating system as they typically are built from a single loop of control, where in addition peripherals trigger interrupts to resolve events from outside the system. This is also termed "monolithic firmware", since system and application code are compiled together. In general, the code running on the hardware of the device can be either entirely custom or written using operating system libraries, providing OS like routines. Nowadays, several types of hardware controllers such as for CD readers, GPS dongles or WiFi cards make use of this monolithic technique.

Investigation of [18] revealed that inside the Quectel EC25 modem version 1.23.1 of the BusyBox OS is running on top of an ARM Cortex A7 processor. In this sense, the LTE modem corresponds to the first class, nonetheless, the LTE protocol is likely not completely executed in software due to the cost and the complexity of LTE. As stated in [18] the part of the modem which is fuzzed with the LTE fuzzer most likely corresponds to a hardware module on chip communication with the operating system over, for example, shared memory. In detail, this means that some processes inside the BusyBox OS take care of controlling the hardware components, such as for example an LTE module. In this way, the Quectel EC25 can also be seen as an IoT device without OS as probably most of the LTE implementation is in hardware and can not utilize the operating system's features/services. Moreover, not much information is available on the Quectel modem from the vendor about the logging tasks of the OS with respect to the hardware [18].

5.1.2 Types of faults

To understand the fact that fuzzing IoT devices has to deal with some extra obstacles when it comes to fault detection, a short introduction is given to the different types of errors that can occur inside a fuzzed process and their difficulty in detection.

In an embedded solution, software and hardware are working together to deliver a specific task during their lifetime as they were designed for. Due the complexity in the design process these hardware and software co-systems are known to have possibly bugs inside their system. Such a *bug* or *error* can cause the device to enter

an unintended state which can result in several types of behavior depending on the type of bug triggered and the security measurements in place. As a result, it is important for the fuzzer to understand the different sorts of bugs as well as the applied security measures inside the device. They determine the behavior of the DUT and the difficulty to detect faults when fuzzing the device. Here, the operating systems on board of the IoT device delivers the majority of services for handling recovery from faulty states. Therefore, the OS gives an indication on the observability of triggered bugs. Whenever a bug leads to an unintended state that can be exploited by an attacker the fault is classified as a *security vulnerability* [5].

The most well-known class of faults are *memory corruptions*. In short, memory corruptions designate several sorts of errors that violate the safety of the memory inside the device by overwriting stored content to faulty values. Depending on the security features a memory error can give rise to an *observable crash* or result in a *silent memory corruption*. An observable crash is characterised by program termination or some kind of routine that performs a recovery procedure through, for example, exception handlers. As already alluded on, this crash is only possible with the support of some extra (OS-based) security mechanisms, like stack canaries. On the other hand, whenever a fault does not directly result in some sort of crash inside the system, the error falls into the category of silent memory corruptions and leaves the device in an unintended state. While observable crashes are favored and supported by most fuzzers, IoT fuzzing deals with devices having no or almost no OS to catch silent memory corruptions. The problem in this setting is that silent memory corruptions are a real thread for the security and safety of the IoT device. One example of a memory corruption is a software stack overflow. Here, data is written to the stack outside the data structure for which it was intended for, and can be exploited by an attacker to inject custom code into the target. Another example is a segmentation fault indicating the occurrence of illegal access to an explicit region in memory. For systems without operating systems this operation mostly happens without consequence, although it can have severe impact on the integrity of an IoT device. on the contrary, embedded systems with an embedded or general-purpose operating system can reveal and anticipate on unauthorized memory access by rebooting or producing an observable crash of the system [5, 59].

Apart from memory bugs another class can be distinguished, which is called *numerical errors*. It consists of faults that come from improper handling of numerical computations, such as typical underflow or overflow errors. However, embedded devices are often resource constrained and don't provide the necessary support for managing numerical errors which can lead, initially, to problems without visible effect. Furthermore, in view of limiting the number resources on the device, the system can also be designed to use supplementary numerical computation libraries. These can, hence, turn computational errors to pure software errors that may as a consequence either slumber in the device or more severely results in some sort of memory corruption [59].

Semantic bugs or *logic bugs*, which were already mentioned in Chapter 3, denote issues in the implementation of an application with respect to the specifications. In other words, this means that the implementation does not comply to the applicable

standard. IoT systems are fabricated for a specific task and should therefore agree with the specifications put forward at the beginning of the design. Hence, misbehavior can lead to some serious security implications. To uncover this type of bugs, the behavior of the device (e.g., inputs and outputs) should be checked against the semantic rules of the specification. In particular, semantic rules define a number of statements the system should respect in order for it to be compliant with the standard. Whenever one of these rules is violated the implementation of the program is denoted as non-compliant, which can result in some unwanted security consequences. In general, following the semantic rules is not an easy process in case of a complex, ambiguous or unclear specification or standard. The LTE protocol, as an example, is standardized by 3GGP using several releases [64].

5.1.3 Main challenges

Muench *et al.* [5] surveyed the main challenges when running fuzzing experiments on embedded devices compared with desktop systems, and formulated three problems. As these challenges were also encountered in this thesis, they will be described shortly in this section.

Problem-I: Fault detection

As covered in Subsection 5.1.2, finding vulnerabilities when an error occurs inside an IoT device is not always straightforward, since most of the embedded systems belong to type-I or type-II regarding their security mechanisms employed. In contrast, general-purpose systems do offer defenses to detect memory corruptions inside running programs and prevent them from occurring with the help of security features like a memory management unit (MMU), Code integrity guard (CIG) and data execution prevention (DEP). Hence, with the aid of the OS abnormal behavior is terminated, which results in an observable crash towards the outside world. Unfortunately, almost none of these hardware-based measurements are present inside IoT devices making it difficult for a fuzzer to detect any faults. As a consequence, the response of the system to a fault can range widely from crashing the embedded device to leaving it unresponsive or showing no effect at all. Because of that Muench *et al.* investigate the different response types upon triggering artificial memory corruptions inside embedded devices for the categories from Subsection 5.1.1. While the results are illustrated in Figure 5.1, the researchers observed the following six responses:

- **Observable crash.** During execution of the device-under-test the system stops running and this event can be visually observed. Only in ideal cases information about the cause of the fault is returned, otherwise a less detailed indication (e.g., a message) is given (this type is marked as opaque in Figure 5.1).
- **Reboot.** The entire systems reboots and restarts its processes. Due to the monolithic software and hardware integration of type-III devices no difference exist between a reboot and a crash. The other types, nonetheless, can have a part of the system crashing, while the rest of the device is functioning correctly.

	expat				mbed TLS			
Platform	**Desktop**	**Type-I**	**Type-II**	**Type-III**	**Desktop**	**Type-I**	**Type-II**	**Type-III**
Format String	✓	✓	✗	✗	✓	✓	✗ (malfunc.)	! (hang)
Stack-based buffer overflow	✓	✓	✓ (opaque)	! (hang)	✓	✓	✓ (opaque)	! (hang)
Heap-based buffer overflow	✓	! (late crash)	✗	✗	✓	! (late crash)	✗	✗
Double Free	✓	✓	✗	✗ (malfunc.)	✓	! (late crash)	✗	✗
Null Pointer Dereference	✓	✓	✓ (reboot)	✗ (malfunc.)	✓	✓	✓ (reboot)	✗

FIGURE 5.1: Discovered behavior of different platforms on several types of memory corruptions [5].

- **Hang.** An input caused the embedded system to ignore new request as it seems to be stuck in an infinite loop.
- **Late crash.** Initially, the memory corruption does not have any impact on the execution of the target, although it manifests after a non-negligible amount of time resulting in a crash (e.g., when sending a specific request).
- **Malfunctioning.** The target system keeps running properly, but generates wrong outputs and faulty results due to the memory corruption.
- **No effect.** The process continues without any observable problem in spite of the fact that the memory is corrupted.

In summary, the graph shows that systems running an OS always terminate in a crash or a late crash making it easier for fuzzers to discover faults, even though type-II and type-III really show the challenge in fault detection for IoT devices [5]. Applied to the Quectel EC25 modem, it can be optimistically believed that since the BusyBox operating system runs inside the modem each bug will expose itself in an observable way. However, as mentioned before the LTE protocol is most likely carried out on hardware thereby not using the security features of the OS. As a result, more sophisticated feedback techniques are needed to monitor and expose faulty behavior [18].

Problem-II: Performance and scalability

Parallel fuzzing of multiple instances of the same software is already been used for desktop systems, e.g., via multi-processing or virtualization. By way of illustration AFL [54] implements a fuzzing mode for (stateless) software programs, where a so-called fork server spawns several child processes at different check-points in the program flow, boosting the performance in terms of speed. In case of IoT or embedded devices parallelization is a major issue due to the fact that several physical copies

of the same system require a considerable amount of resources (e.g., financial) and to a lesser extend pose a demand for infrastructure availability in terms of space and power. In addition to this, between each test case the IoT system needs to be restarted to a clean state leaving no traces from the previous run inside the equipment. For general-purpose devices re-establishing a clean state can be done effortlessly with for example virtual machine snapshots. On the other hand, integrated devices take up in general much more time to reset the state by means of, e.g., a reboot of the entire system. As a result, the performance with regard to executions/s of IoT fuzzers drop considerable compared with their desktop counterparts [5].

Problem-III: Instrumentation

In the early days of fuzzing, black-box fuzzing was the most utilized technique for finding vulnerabilities in target programs until the discovery of the added-value of monitoring the internal state of the tested system in generating new inputs and revealing memory bugs. Since then on fuzzers for general-purpose systems took a more white- and grey-box approach for finding bugs more efficiently and testing the device more thoroughly. Hence, a popular technique nowadays to access information about the state of the device-under-test is to use (a mix of) run-time instrumentation and/or compile-time instrumentation giving code coverage feedback about the executed test cases. The benefit of instrumenting the source code is twofold. On the one hand, with code coverage the exact code path inside the target program can be traced for a specific input. This has lead to the development of clever techniques for enhancing the fitness of new test cases. AFL, as an example, is one of the most widely adopted coverage-based grey-box fuzzers. On the other hand, if source code is available, instrumentation can be used to detect subtle memory corruptions inside the program using the following tools: MemorySanitizer [65], AddressSanitizer [66] and ThreadSanitizer [67]. Embedded systems rarely have source code available for their firmware images or are tough to instrument limiting the feedback information that can be used in the fuzzing process. In fact, the dynamic and static instrumentation tools are closely tailored to the CPU architecture and the operating system, making instrumenting IoT device a difficult endeavor (e.g., for type-II and type-III). In fact it is just this lack of feedback information from the device that restrains the fuzzer from finding bugs in the source code of the SUT. More specifically, with difficulties to obtain the source code or instrument the code itself the fuzzer cannot gather enough information from the DUT and reduces to a less efficient black-box fuzzer, where only responses of the system can be analyzed. So to find weaknesses in an IoT setting without the possibility to obtain source code or instrumentation, it is necessary to use more advanced techniques, like machine learning to overcome the feedback limitations inherent to IoT fuzzing in general. Note that this shortcoming in feedback mechanisms applies both to the process of intelligent fuzzing as well as observing faulty behavior (i.e. fault detection) [5, 59].

5.1.4 Mitigations

In order to mitigate the aforementioned challenges different solutions have been proposed to make, e.g., crashes and errors more observable to the fuzzer, similarly to desktop systems. However, due to the heterogeneous nature in characteristics of these systems, no one fit-all solution exists and depending on the fuzzed platform the most appropriate fix may vary with the device at hand. As stated in [5] and [59] six different options can be discerned with each having their own advantages and disadvantages. What follows is a brief discussion on each of them.

In the optimal case when the source code and a suitable compiler toolchain is available *static instrumentation* can be used to provide run-time information and crash hardness functions (e.g., when the manufacture wants to test his own device). Typically different instrumentation tools are used to collect diverse types of information. Note that source code is almost never open-sourced by IoT manufacturers.

Binary rewriting, on the contrary, instruments the binary firmware image which has to be extracted from the embedded device. However, this is a challenging task that does not always succeed or give the desired results.

Third, *physical re-hosting* as the name suggests is the process where the tester recompiles the source code of a low-end type of device, e.g. type-II or type-III, to a different, but more friendly type of machine such as a Linux PC. Drawbacks of this approach are the source code requisite, and the potential introduction of new and the possible deletion of original bugs during execution of the compiler chain.

Fourth, as shown by Costin *et al.* [68] and Chen *et al.* [69] type-I firmware can be used to extract the application and execute it inside a generic operating system running on a standard emulator, although only if certain conditions are satisfied. Similarly, in [70] the Qemu STM32 tool was designed to emulate, in case the complete hardware specification is available, type-III firmware images. Hence, this technique is also known as *full emulation* or *virtual re-hosting*, since the firmware and device's peripherals are emulated altogether. With this solution performance enhancement is possible due to parallelization, dynamic instrumentation techniques can be used and with the emulator memory information can be collected. The major drawbacks of this technique are the considerable effort that is required and the fact that not every firmware can operate inside an emulator.

Aside from full emulation a more relaxed form of emulation, called *partial emulation*, also transfers systems without OS to devices with, but now forwarding peripheral I/O to the actual physical device. Hence, partial emulation provides almost the same benefits as for full emulation even so when emulation of the peripherals has been unsuccessful or the exact I/O instructions are unknown to the tester. However, the price to pay for the extra flexibility is the reduction in performance and scalability. Avatar [71] and Surrogates [72] demonstrated partial emulation for type-III devices, whereas PROSPECT [73] broadened it to type-I.

Last, some devices are equipped with an advanced hardware debug feature on the chip that can enable a tester to obtain sufficient information to detect faults. This *hardware-supported instrumentation* is often specific to the manufacturer and not available to lower-end IoT systems like type-III due to the extra surface and

hence cost it brings to the platform. In other cases a debug interface may be present but have restricted access to prevent non-vendors from doing system analysis. As a result, the chance of having manufacturer supported tracing capabilities on chip that is, furthermore, ready for use is very unlikely. An example of such an embed hardware tracing interface is ARM's Embedded Trace Macrocell (ETM).

5.2 Prior work

As can be seen from the literature summarized below, IoT fuzzing research has mainly been focusing on black-box techniques as leveraging the internal working of the device is still a challenging endeavor, and so the only viable solution. However, as this results in inferior mutation strategies and hence fuzzing results, some research [74, 75] has been devoted to the process of emulating IoT firmware which allows for more feedback using dynamic analysis tools. Nevertheless, emulation techniques have been proven to contain some shortcomings as well (e.g., the extraction of the firmware, the analysis process, etc.). Despite the aforementioned problems, several research papers have come up with new techniques to improve black-box designs, e.g., with the aid of apps [76, 77] or IoT responses [78].

One of the first fuzzers explicitly designed for IoT devices is IoTFuzzer (2018) [76] by Chen *et al.*. Hence, IoTFuzzer is an automated, black-box framework aiming at finding memory corruption vulnerabilities without leveraging the firmware images of tested IoT systems. They observed that an accompanying app can provide rich information about the IoT platform itself without having to reverse-engineer the protocol specifications. To this end, the device's control plane app is dynamically analyzed, which shows IoTFuzzer the messages to carry out protocol-guided fuzzing, whilst mutating them automatically using the rich information inside the device's accompanying IoT app for probing the firmware. However, whereas the fuzzer tests IoT devices without needing their extracted firmware images, this approach is limited to app-based devices such as smart air coolers.

On the other hand, IoTHunter (2019) [74] is the first grey-box fuzzer, as stated by the authors Yu *et al.*, for IoT devices as it works on extracted firmware to monitor code coverage and state feedback. In particular, the fuzzer incorporates a protocol state scheduler engine on top of Boofuzz [55] that updates the state model dynamically with a multi-stage message generation mechanism and a message mutation fuzzer derived from AFL.

Observing that the generic over-the-air fuzzing has to cope with several limitations such as low execution speed, confined debug abilities and challenges in reproducing results, Ruge *et al.* developed Frankenstein (2020) [75]. It can fuzz Bluetooth implementations on different modems on chip based on an advanced firmware emulation framework. Hence, Frankenstein provides a C programming environment to execute a large portion of the tested IoT firmware which is used to find vulnerabilities with fuzz-testing.

As an improvement of IoTFuzzer, Redini *et al.* proposed an automated black-box fuzzer, called DIANE (2021) [77] leveraging the companion app as well, to target its

corresponding IoT device. They argued that the effectiveness of IoTFuzzer suffers from the fact that there are functions inside the app sanitizing the inputs that are send to the device. Therefore, DIANE combines static and dynamic analysis techniques to differentiate between app logic that converts user interface inputs to internal data structures (i.e., checks that sanitizes the input data), and functions that encode this data for serialization to the network. Thus, the app-based approach targets the latter type of functions to effectively test the IoT device with unconstrained fuzzing data, that would have been discarded otherwise by the app due to its invalid format.

Lastly, Snipuzz (2021) [78] a black-box network fuzzer presented by Feng *et al.* introduces the IoT target's response as a new feedback mechanism to guide the mutation process in detecting vulnerabilities without having access to the firmware. To this end, the authors mention that the device responses are a direct result of the execution path inside the firmware. Hence, the Snipuzz implements a message snippet inference mechanism learning the relationship between input snippets and the corresponding code path executed taking into account the responses produced by the IoT device. Moreover, this mutation methodology enables the fuzzer to infer the hidden grammatical structure of the input without prior knowledge on the syntax.

5.3 ENPSFuzz' approach

In ENPSFuzz two types of feedback mechanism can be distinguished. The first one, also called bug oracles, tells the fuzzer whether any abnormal behavior is noticed when executing a test case. This abnormal behavior can be either a crash, a reboot, a hang, a late crash or a malfunctioning of the LTE stack induced by an underlining memory corruption, a numerical error, or divergent behavior with respect to the reference UE (i.e., signaling semantic or logic bugs). The second type, among which code- and state coverage, have as purpose to guide the fuzzing process by providing interesting information to evaluate test case execution. Note that the feedback information of a test case is monitored and saved during the input evaluation stage, whereas the evaluation of the following feedback mechanisms (e.g. edge coverage) are being done during the configuration updating state as mentioned in the architecture of Chapter 4.

5.3.1 Bug oracles

Liveliness check

Similar to [18], the *liveliness check* is the most simple feedback mechanism that checks if the device-under-test is alive and responding. However, the execution of this command is now sent by the fuzzer over the socket interface towards the system-under-test, where the corresponding process hands it over to the serial interface of the tested modem. Subsequently, the fuzzer waits until it receives an answer from the device. If the modem sends *OK*, ENPSFuzz may safely assume that the DUT and the AT interface are working correctly, and thus they pass the liveliness check. Although, it may be possible that the target program responds with an *OK* message,

it can still fall into a late crash afterwards. While waiting for a longer time could be more likely to catch this behavior, it prevents the fuzzer from executing at high speeds and moreover this late crash would have, nonetheless, been caught after the next execution. On the contrary, when no response is received the fuzz-tester repeats the liveliness check three times after which it declares an error from the target side.

The check is useful in observing the status of the device, but suffers from some restrictions. In the first place, since the modem is a complex system composed of hardware, software and an operating system, the process handling the AT interface is most likely separated from the actual LTE implementation on chip. Hence, when the AT interface has crashed and doesn't respond to the liveliness check, the LTE implementation, although, can be operating properly (as proven in next chapter). The other way around, the AT interface may still communicate correctly after an LTE crash [18]. Therefore, it can be assumed that this check only tests the working of the AT interface.

Differential check

The goal of the *differential check* is to investigate all the messages that are sent by the DUT during one execution and are therefore recorded in the input evaluation stage of ENPSFuzz. Observed from experiments, the target device can behave in three different ways upon reception of a fuzzed packet during the connection establishment and detach procedure. The modem can either continue the protocol run as if a normal message was received, stop the current run and restart the connection establishment procedure over again with a new preamble, or defer completely from transmitting subsequent messages to the base station. However, it is in general challenging to know whether the device acts conforming the specifications when it experiences a fuzzed packet. This is so because the fuzzer is, on the one hand, not aware of the exact details of the standard and, on the other hand, some mutated packets can still follow the standard whereas others don't. For example the, IoT device is expected to accept mutated packets that have an LTE field value changed to another correct value. Hence, the correct behavior of the device is conditioned on the exact situation and therefore often difficult to decide on.

During the execution of a test case both uplink as well as downlink messages are logged for further analysis if needed later on. In other words the two srsENB programs (i.e., for the SUT and for the RUE) send all their messages to the fuzzer over the shared memory interface which are then put in the logging report by the corresponding execution processes of ENPSFuzz. This logging happens during execution and thus in the input evaluation stage of the fuzzer's architecture. Having the required information available for analysis, one option would be to define in advance whether the observed behavior is correct or abnormal. However, defining manually how the device should respond is cumbersome and often not feasible due to the complexity of the LTE standard. Here, the reference UE comes in as the operations of both the device-under-test as well as the reference UE are monitored for leveraging differential fuzzing. So differential fuzzing assesses the dissimilarity between several programs that only differ in their implementation of the same

application. Hence, for this thesis differential fuzzing means that ENPSFuzz sends the same fuzzed protocol runs to the DUT and the reference UE with the goal to observe diverging behavior in the uplink messages the user terminals relay. Note that the better the reference software implementation is, the better abnormal behavior can be exposed. Note that similar faults present in both targets will not be observed by this type of testing. However, this similarity is very unlikely for independent implementations so that deviant functioning can be caught as a sign for several types of bugs.

The benefits of the differential check compared to the limited liveliness check is that it is able to observe if the device stops responding, which can be the consequence of several faults: a (late) crash, a hang or a reboot from the LTE implementation on chip. Nevertheless, other faulty behaviors can be captured as well, such as the device-under-test sending wrong messages due to some internal malfunctioning, by comparing it to the reference UE. Furthermore, in this way the so-called semantic or logic bugs can also be uncovered that do not result in an explicit crash or hang, but reveal the discrepancies between implementations.

Connectivity check

As as an extension to the differential check, the *connectivity check* looks at the connection status of the target UE as well as the reference UE. The check can uncover connectivity problems, meaning for example that the DUT is not able to connect to the network anymore, whereas the RUE does. For this purpose the AT interfaces are consulted during the end of the execution stage, just before the initiation of the detach procedure.

5.3.2 Code coverage

As mentioned in the beginning of the chapter, it is still an open research question on how to develop grey-box fuzzers for IoT devices that can use lightweight instrumentation analysis instead of black-box approaches. With such enhancements in IoT fuzzing, their mutation processes can be made more intelligent, leading to more secure IoT designs. To overcome this challenge, ENPSFuzz leverages edge coverage instrumentation, but without the requirements to extract the firmware or to emulate the IoT device. Instead, the fuzzing framework is designed as a grey-box network protocol fuzzer working in the natural setting of the IoT device, e.g., with the real-time requirements imposed by the protocol. This increases the potential to find vulnerabilities that resemble more to real-life attacks.

The key idea in ENPSFuzz is to solve the instrumentation problem by integrating a reference software program (i.e., the reference UE from the overall architecture), which is given the same fuzzed message inputs and does have source code availability. However, note that giving exactly the same inputs to both would not work out, as protocols do have fields that can be time and/or user dependent. Therefore ENPSFuzz invented the concept of *patches* from the input evaluation stage in Chapter 4. As a result of this new approach, this thesis reduces the gap between the state-of-art grey-

box fuzzers leveraging source code availability for testing mostly stateless software programs (e.g., readelf, jhead or any type of PDF parser), and protocol fuzzing for cellular IoT devices having part of their solution integrated in hardware.

In particular, the edges inside the LTE stack of the reference UE are instrumented with updated LLVM callbacks using the Clang compiler. Thereby, making it possible to record how many times each edges is activated during the execution of a seed at run-time. Despite the availability of the exact counts inside the bitmap, ENPSFuzz doesn't distinguish between one or more activations of an edge, although it would be possible. Similarly to coverage-based fuzzers from literature, the fuzz-tester uses a coverage feedback mechanism to intelligently guide the fuzzing process using the evolutionary and gradient-guided optimization techniques discussed in previous chapter. In short, the evolutionary method uses code coverage feedback from the reference UE to calculate the fitness scores of seeds and to update the seed corpus with new edge triggering test cases. Moreover, the collected coverage information is also used for the neural network model to learn the branching behavior (i.e., neural program smoothing) with respect to the input bytes of the protocol, permitting for gradient-guided mutation strategies. Note that without this new coverage idea it is not possible to port state-of-art techniques to cellular IoT fuzzing, falling back to black-box methods.

Moreover, the upgraded instrumentation part of the framework using the srsUE software can also open new capabilities, e.g., white-box fuzzing, that can be investigated in further research.

5.3.3 State feedback

The last feedback mechanism integrated into ENPSFuzz is state feedback based on the inputs and outputs from the device. Given the Mealy machine constructed during the entire fuzzing process, ENPSFuzz can observe new states or state transitions induced by executed test cases in the input evaluation stage. Hence, upon receiving a notification from the FSM indicating the discovery of new state feedback, the fuzz-tester adds the corresponding test case to the seed corpus. This helps to discover and fuzz more and more states inside the protocol. As an example, an abnormal succession of messages generated by the insertion mode can uncover new states, or field-level mutations can induce new behavior from the DUT (e.g., sending a different type of message).

Chapter 6

Implementation and Experimental Evaluation

This chapter has been left out from the thesis due to security-sensitive information. Contact the supervisors for insight on the full version.

Chapter 7

Conclusion

Since the start of the Internet of Things paradigm, a tremendous growth has been seen in IoT devices, which interconnect large parts of the public and private environment nowadays. However, as IoT devices and in particular cellular IoT systems are rooted deeply in our daily lives their security is of paramount importance to preserve the community from devastating consequences. Unfortunately, IoT systems have been shown to contain lots of security vulnerabilities due to their lack of security mechanism, which have been exploited in numerous attacks. Therefore, this thesis extends the LTE fuzzing architecture initially designed by J. Hoes' thesis [18]. Moreover, it creates a new grey-box fuzzer, called ENPSFuzz (Evolutionary Neural Program-Smoothing-based Fuzzer), transferring the two mostly used state-of-art techniques in literature to cellular IoT fuzzing for protocol implementations. To this end, this thesis introduces the key concept of a reference UE enabling to port advanced stateless software fuzzing techniques to cellular IoT fuzzing on hardware. Experiments showed that the combination of the improved framework together with the intelligent fuzzer could uncover a new bug on the Quectel EC25 LTE modem. The bug enabled the network protocol fuzzer to leave the AT interface of the LTE modem blind for external commands. In other words, it is impossible to control the modem after triggering the bug.

In this final chapter of the thesis a conclusion is given on the main improvements on the overall architecture, the design of the fuzzer and the experimental results. Next, several improvements are proposed which can be carried out in future work.

Extended fuzzing architecture

Similar to the original overall framework [18], the extended architecture accommodates an LTE core network simulator, a simulated eNodeB, a system-under-test, an Ettus USRP B210 SDR and a fuzzer component. However, this thesis upgrades four aspects of the original architecture. First, the fuzzer block replaces the OpenTap controller as main controller in coordinating other elements of the overall architecture. Since the fuzzer entity observes the behavior of the target devices (e.g., through messages or AT commands), it boosts performance to let the fuzz-tester itself react to incoming events, instead of an external controller. In addition, the framework

achieves full automation as the fuzzer can also start and stop other processes (e.g., the simulators). Second, to enable smooth communication between the several architectural blocks, the framework is equipped with socket communication and shared memory interfaces (e.g., to let the fuzzer control the UEs dynamically over their AT interface). Moreover, the interfaces are made robust to deal with connection problems. Third and most important, in this thesis a new technique is designed which integrates a reference UE software that provides code coverage information and is used as a second opinion on the DUT's behavior (i.e., differential testing). A shared memory protocol is put in place that serves as interaction between the fuzzer component and the RUE block for edge coverage data. Last, all messages from the MAC, RRC and NAS layer are captured to fuzz the entire attach and detach procedure, mentioned in Chapter 2.

Fuzzer component: ENPSFuzz

ENPSFuzz starts over from an entirely new design, integrating both intelligent fuzzing logic as well as controller functionalities that were previously performed by Keysight's OpenTap test automation framework [18]. Hence, the fuzz-tester controls the different components of the platform so that it can run fully automated, efficiently and can react dynamically to the behaviors of the DUT and other components. On the contrary, for the fuzzer tasks, ENPSFuzz consists of an evolutionary, but modular architecture composed of different phases and stages. Each stage has its own predefined goal and can be extended or replaced by new methods to improve fuzzing in future work. Governed by the fact that literature focuses on evolutionary algorithms and more recently on gradient-guided techniques using neural program smoothing (see Appendix C for an extensive literature review), ENPSFuzz incorporates and complements both techniques as well as other modes of operation in its mutation stage. Although these approaches were currently only applicable to programs with source code availability, ENPSFuzz is the first smart IoT fuzzer that can leverage these methods. The reason for this is the new proposed idea of a reference software implementation. Besides these intelligent mutation strategies, the fuzzer component comes with 3-level mutators, packet modeling via WDissector, state graph inferring based on the concept of Mealy machines and an interactive neural network module.

To expose faulty behavior from the cellular IoT device, ENPSFuzz leverages on three different checks: the liveliness check, the differential check and the connectivity check. The first check can discover AT interface malfunctioning, whereas the differential and connectivity check serves for detecting hangs, crashes, malfunctioning, semantic bugs, etc. For this purpose, differential testing with the RUE and extensive logging is put in place.

Apart from the bug oracles, code coverage information and state feedback are used to guide the optimization processes in solving the mathematical fuzzing problem and hence to expose bugs.

Future work

To end the conclusion, some final remarks are stated about how future work can extend the current implementation.

First, the effectiveness of the new research approach, in adding a reference software for testing IoT devices with over the air fuzzing, has to be evaluated more thoroughly. To this end, the number of bugs found and the resulting seed corpus (for analyzing the mutations inside the packets) can be compared with other network fuzzers from literature. Another possibility is to replace the hardware device with a similar UE software simulator (e.g., OPNET or OMNeT++ [79]) for which code coverage can be measured. As a result, with this software target the difference in code coverage achieved by ENPSFuzz can be compared with a baseline black-box IoT fuzzer having no indirect coverage feedback. Moreover, studying the improvements of combining the evolutionary algorithm with gradient-guided optimization is also an interesting direction.

Currently, messages are captured at the specific layers throughout out the stack of the base stations, where they are created in case of downlink packets and decoded in the event of uplink messages. Hence, these messages correspond to the RRC (e.g., RRC Connection Setup), NAS (e.g., Attach Accept) and MAC layer (e.g., Random Access Response). In future work, it is possible to intercept and mutate such downlink messages when they are processed in the lower layers, instead of the layer where they are created. In this way, vulnerabilities can be found in the working of lower layers as they add chunks of data to upper layer packets and decode it afterwards. The lower layer headers can be an interesting attack vector. However, note that this gives rise to some extra complexity as these layers are known to split and concatenate upper layer packets for transmission. A possible solution can be to replace one packet inside the seed with multiple lower layer chunks, but this is up for further investigation.

As can be seen from the LTE introduction, other types of LTE messages are present in the currently fuzzed attach and detach procedure, such as scheduling requests, downlink grants and uplink grants. These packets can be included in a seed as well, after interception in the eNodeB stack. Moreover, the cell search and selection messages (e.g., the PSS, SSS, MIB and SIBs) are all essential parts of the LTE protocol that can be caught and included in seeds for fuzzing.

Another research direction is to design a machine learning technique, which can detect hidden anomalous between the target IoT device and the reference UE by means of extracted features from their input and output behaviors. This can then enhance the differential check.

Appendix A

Keysight Technologies

This thesis was developed in collaboration with the company "Keysight Technologies". This appendix will give a glance of Keysight Technologies as a company and an overview of their industrial presence.

Keysight Technologies is a public traded company (NYSE) with headquarters in Santa Rosa, USA. Although they are founded in 2014 as a spin off from Agilent Technologies' electronic measurement equipment branch, their history goes back to 1939 with the foundation of Hewlett-Packard (HP) by William R. Hewlett and David Packard [80, 81].

Keysight delivers advanced design and innovative validation solutions, which help their customers to connect and secure their world. Hence, Keysight's dedication to speed and precision extends to software-driven insights and analytics that bring tomorrow's technology products to market faster across the development lifecycle, in design simulation, prototype validation, automated software testing, manufacturing analysis, and network performance optimization and visibility in enterprises, service providers and cloud environments. The company offers a wide range of products and services like oscilloscopes and analysers, multimeters, software, wireless networks simulators, modular instruments, network security tools and more [80, 81].

Moreover, Keysight serves more than 32.000 customers in more than 100 countries and have around 14.300 employees. Their major global customers are situated across several industries, like automotive, healthcare, communications and industrial ecosystems, energy, aerospace and defense, semiconductor and general electronics markets. 25 out of the top technological companies use Keysight today within their innovation and development processes: Apple, Amazon, Google, Microsoft, etc. To illustrated their scope, the company generated revenues of $4.9B in fiscal year 2021 [80, 81].

In Belgium, Keysight has offices in Ghent and Rotselaar. This thesis was mainly in collaboration with Keysight Laboratories in Rotselaar, who supported the development of this master's thesis extensively. However, due to an Erasmus exchange in the first phase of the thesis, Keysight Technologies Barcelona provided the necessary office space and test environment to help in achieving the final thesis goal. This thesis belongs to the IoT branch of Keysight, illustrating their focus on

security solutions for applications and devices [80, 81].

Appendix B

LTE Channels

This appendix contains the LTE channels starting from physical channels and going up in the stack from the transport channels to the logical channels. Besides the full names, also a short description about the each channel's functions is given.

B.1 Physical channels

The most important physical channels in downlink are [35]:

- The **Physical Downlink Shared Channel (PDSCH)** is a data channel carrying paging information, SIB messages and unicast user data.
- The **Physical Broadcast Channel (PBCH)** is a data channel to which UE have to listen before being able to connect to the network. The Master Information Block (MIB) is the only signaling present on this channel and provides the subscriber with the necessary information to access the network.
- The **Physical Multicast Channel (PMCH)** is a data channel used for transporting Multimedia Broadcast and Multicast Services (MBMS) together with a Multicast-Broadcast Single Frequency Network Reference Signal (MBSFN-RS).
- The **Physical Downlink Control Channel (PDCCH)** signals Downlink Control Information for a particular UE or several UEs such as downlink scheduling needed for PDSCH decoding, uplink power control indicators and uplink tansmission grants for the PUSCH (i.e ACK/NACK).
- The **Physical Hybrid-ARQ Indicator Channel (PHICH)** provides control HARQ acknowledgement information about whether the user should retransmit its transport block(s) (i.e. ACK/NACK).
- The **Physical Control Format Indicator Channel (PCFICH)** is also a control channel telling the UE at the start of a subframe how many OFDM symbols are used for the PDCCH channel, if present.

and in uplink [35]:

- The **Physical Random-Access Channel (PRACH)**, a data channel, carries the data from the random access procedure (RACH) in which the mobile terminal asks the eNB for uplink allocation and gets synchronized.

- The **Physical Uplink Shared Channel (PUSCH)** is used by UE to transmit its uplink data and is therefore the counterpart of the downlink PDSCH.

- The **Physical Uplink Control Channel (PUCCH)** is the only control channel in uplink. It carries HARQ acknowledgements of the UE stating whether the downlink transport block(s) coming from the mobile station was/were decoded properly (i.e. ACK/NACK), downlink channel-state estimates for the eNodeB's scheduler and resource requests for uplink data transfer.

B.2 Transport channels

LTE has defined the subsequent transport channels in downlink [35, 82]:

- The **Broadcast Channel (BCH)** carries the Master Information Block (MIB) which is part of the BCCH signaling and follows a fixed transport format.

- The **Paging Channel (PCH)** is used for broadcasting paging messages coming from the PCCH. When the so-called discontinuous reception (DRX) is activated the device is woken up periodically to send the PCH information saving battery power in the device.

- The **Downlink Shared Channel (DL-SCH)** transports most of the data in downlink. It also supports HARQ, DRX power saving messages, spatial multiplexing, dynamic rate adaptation by changing modulation, coding, power levels, etc.

- The **Multicast Channel (MCH)** can be used to broadcast data in the entire cell of the eNB (e.g., MBMS and MBSFN).

and in uplink [35, 82]:

- The **Random-Access Channel (RACH)** is used for the random access procedure.

- The **Uplink Shared Channel (UL-SCH)** transports most of the data in uplink and can thus be considered as the counterpart of DL-SCH.

B.3 Logical channels

The set of LTE's logical channels are in downlink [35, 82]:

- The **Broadcast Control Channel (BCCH)** is used for broadcasting system information to all UEs inside the cell in form of System Information Blocks (SIBs) and the Master Information Block (MIB). This information is required by a UE to obtain the necessary network configurations before being able to connect to the system.
- The **Paging Control Channel (PCCH)**, as the name already reveals, transports paging information for a device throughout multiple cells, as this channel is used when the exact location of the UE is not known.
- The **Common Control Channel (CCCH)** carries on the one hand control signaling between UEs and eNBs, and on the other hand messages for the RRC connection setup procedure. Note this channel is present both in downlink as well as in uplink.
- The **Dedicated Control Channel (DCCH)** is a point-to-point bidirectional connection between a particular UE and its serving eNodeB where dedicated control information is passed through (e.g., for handover configuration messages). Only after RRC connection this channel is available to the device. Note this channel is present both in downlink as well as in uplink.
- The **Dedicated Traffic Channel (DTCH)** is a dedicated point-to-point bidirectional logical channel carrying user-plane data between a UE and an eNB. Hence, this channel is present both in downlink as well as in uplink.
- The **Multicast Traffic Channel (MTCH)** is a point-to-multipoint channel which is concerned with broadcasting/multicasting of network data towards multiple UEs which have MBMS enabled.
- The **Multicast Control Channel (MCCH)** is the same type of channel as MTCH, but instead provides the necessary control information to UEs that are subscribed to MBSM and want to tune to the multicast traffic channel.

Likewise in downlink, the CCCH, DCCH and DTCH are present in uplink as well [35, 82].

Appendix C

Literature Review

This chapter gives an overview of different fuzzers in literature. Hence, the first section discusses the most common types of fuzzers and their general architecture, followed by the developed machine-learning-based fuzzers and finally the chapter is ended by reviewing research in network protocol fuzzing.

C.1 Fuzzing overview

In this section, the well-known classification of fuzz-testers are discussed, namely, white-, grey-, and black-box. Next, the different stages inside a common fuzzing framework are coined together with some examples.

C.1.1 Fuzzer taxonomy

In short, the category to which a fuzzer belongs depends on the quantity and type of feedback gathered from the internal working of the target program. White-box fuzzers leverage a lot of information, whereas at the other extreme black-box fuzzers work with almost no feedback. Moreover, the most popular fuzzer category, namely grey-box fuzzing, has two diverging research branches: coverage-based grey-box fuzzers and directed grey-box fuzzers.

Black-box fuzzer. Black-box fuzzers denote fuzzers that do not have access to the source code or the internal working of the system-under-test (SUT). In this case, only the input and output behavior of the system is visible to the fuzzer and hence they mostly generate new test cases by applying random mutations following some rules. However, a drawback of this approach is that many inputs are discarded by the SUT as they don't comply with the expected input semantics, decreasing fuzzing throughput. Nonetheless, black-box fuzzers are generally easy applicable to different targets, but have almost no feedback information (e.g., code-coverage) to guide the fuzzing process [83, 84]. Some examples are *Snipuzz* [78] and *PULSAR* [85].

White-box fuzzer. As opposed to black-box, white-box fuzzers require access to the internal logic of the system-under-test such as the source code. Hence, this type of fuzz-testing relies on information acquired through source code analysis by static

or dynamic tools [86]. As an example dynamic symbolic execution (DSE) is a popular instrumentation technique. In addition, these fuzzers are known to generate false positives and incur more overhead than their black-box counterparts due to their heavyweight analysis. However, in some cases the source code of the target is not available making a white-box approach impossible (e.g., when fuzzing IoT devices) [83, 84]. *FuSeBMC* [87] is an example of such a white-box fuzzer for finding bugs in C programs.

Grey-box fuzzer. Similar to white-box fuzzers, grey-box fuzzers are able to leverage some internal working of the target program through lightweight static program analysis and/or dynamic execution data, like code-coverage. Opposite to white-box, they don't work with heavyweight analysis tools. Hence, they achieve the speed benefit of black-box fuzzers as they do not sacrifice a lot of time in program analysis, while having still some lightweight test case guidance as is the case for white-box fuzzers [83], [86]. The most well know grey-box fuzzers are *AFL* [54] and its descendants *AFLFast* [88], *AFLGo* [89] and *AFL++* [90] enabling code-coverage feedback to guide the fuzzing process.

Coverage-based grey-box fuzzer. The key principle in coverage-based grey-box fuzzing (CGF) is the correlation between bug exposure and code coverage as stated by Miller's report: *"A 1% increase in code coverage increases the percentage of bugs found by 0.92%."* [56, 91]. In this type of fuzzing each part of the program's code is treated equally and the goal is to coverage as much as possible of the code space within the available resources. To this end, lightweight instrumentation provides code coverage information at run-time and with this feedback inputs that trigger new parts of the code are kept for further mutation. This evolutionary approach loops continuously to retain a population of inputs that trigger increasingly more code [88]. Moreover, various types of coverage metrics exist such as function, basic block and edge coverage. Examples of coverage-based fuzzers in literature are *AFLFast* [88], *VUzzer* [92], *FairFuzz* [93], *T-Fuzz* [94], *REDQUEEN* [95], *Matryoshka* [96], *TortoiseFuzz* [97], *Zeror* [98], *PathAFL* [99], *MaxAFL* [100], *OTA* [101], *PATA* [102] and *BeDivFuzz* [103]. Note that most of these papers are designed for stateless programs (i.e., programs that expect only one type of input).

Directed grey-box fuzzer. In contrast with CFG, directed grey-box fuzzing (DGF), introduced by Böhme *et al.* in 2017 [89], takes the approach that only a small part of the code of the program contains vulnerabilities. Hence, according to DGF not all code paths are equally interesting and so less effort ("energy") is put into them. Therefore, lightweight compile-time instrumentation computes the distance from the input's execution path to the targeted parts of the code that are assumed by the developer to be bug-prone. In summary, this evolutionary fuzzer retains and mutates seeds that guide the fuzzing process to the bug-prone parts [104]. Examples of research in this field are *AFLGo* [89], *RFUZZ* [105], *Hawkeye* [106], *LOLLY* [107], *TOFU* [108], *RDFuzz* [109], *Ankou* [110], *UAFL* [111], *UAFuzz* [112], *SDHF* [113], *DGF-CFGConstructor* [114], *CAFL* [115], *KCFuzz* [116], *a SAP HANA fuzzer* [117], *Beacon* [118] and *WindRanger* [119].

C.1.2 General fuzzing architecture

In this subsection, the general fuzzing architecture is considered with the different functionalities in each stage of the fuzzing design. In particular, two phases can be distinguished: the pre-processing phase and the dynamic analysis phase. While the pre-processing phase is only done once, the dynamical analysis phase includes the actual fuzzing loop which is iterated over several times until the fuzzing resources are depleted. Note that not all stages are used by every fuzzer, although they comprise the most important functions inside a fuzz-tester.

Pre-processing phase

The pre-processing phase happens at the start of the fuzzing campaign and its primary goal is to enable the fuzzer to have feedback from the device-under-test during execution in the main loop. Therefore, mostly grey-box and white-box fuzzers can make their fuzzing campaign more intelligent and hence efficient. In particular, most fuzzers in literature focus on instrumenting the source code or binary of the target program, which is also called static instrumentation as it is done at compile-time. On the contrary, dynamic instrumentation happens at the execution of an input on the platform-under-test (i.e., in the upcoming input evaluation stage). While static instrumentation on the source code or binary-level incurs less run-time overhead as it is done at compile-time, dynamic instrumentation makes it easier to instrument dynamically linked libraries at run-time [83]. The most prominent feedback technique in literature is code coverage which comes in several types of granularity, e.g., edge coverage used by the grey-box fuzzer *AFL* [54] or more advanced metrics as in *MTFuzz* [120]. However, note that black-box fuzzers cannot use instrumentation as it's only possible whenever source code is available.

Depending on the type of fuzzer some extra target related specifications can be provided to the fuzzer to boost the fuzzing performance. To this end, the fuzzer architecture can incorporate the input and output message syntax, and the rules on how to interact with the target application such as a state machine for protocol fuzzing [121]. However, this can be identified manually [122] or in a more automated way by the fuzzer itself, e.g., with machine learning [121]. Particularly, on the one hand this allows generation-based fuzzers to generate test cases that comply with the specifications of the input messages and hence pass the initial input checks in the SUT. On the other hand, these specifications permit mutation-based fuzzer to smartly mutate parts of the input having a high chance of being accepted by the program-under-test. For protocol fuzzing several attempts have also been made to include finite state machines into the fuzzer logic for leveraging state feedback (e.g., automata learning [123, 124, 125]).

Furthermore, for most mutation-based fuzzers that follow an evolutionary approach, an initial seed corpus has to be provided where the quality and diversity of the initial seeds impact the fuzzing process. In other words, such a population of inputs serves as a starting point for subsequent mutations in the main fuzzing loop. Note that their exists techniques to optimize the initial seed pool to address

the so-called seed selection problem [126] with corpus minimization tools. These methods have as purpose to produce a minimal set of seeds, called a mindset, that e.g., maximizes a coverage metric [83]. A systematic review of seed selection practices and minimization techniques can be found in [127].

Dynamic analysis phase

The dynamic analysis phase entails five stages that are always performed in the same order and in fact encompasses the main fuzzing loop. One execution of this phase is also known as a fuzz iteration. Hence, the fuzzer performs several fuzz iterations until the resources are exhausted (e.g., when there is no time left or the maximum number of inputs are executed). Note that the architecture suited best evolutionary fuzzers as they are most common in literature.

Scheduling. The first stage of the dynamic analysis phase consist of seed prioritization and power scheduling. In fact, the seed scheduler selects for the current iteration the most promising seed from the seed corpus (e.g., based on its history or fitness). Nonetheless, while black-box fuzzers have less information to make this decision, grey-box fuzzers can leverage wealthier feedback such as the obtained code coverage of the seeds. Subsequently, the power scheduler decides how many new inputs are derived from the scheduled seed and ideally more energy should be assigned to promising seeds. Here, the scheduling algorithm is faced with a fundamental trade-off between exploration and exploitation also termed as the fuzz configuration scheduling problem (FCS) by Woo *et al.* [57]. Either the fuzzer can spend time on every seed and acquire more accurate information about them for further decision making, called exploration, or it can pick the most beneficial seeds, known as exploitation [83]. In other words, the problem can also be stated as a multi-armed bandit problem (MAB) where each seed denotes an arm, and the goal is to maximize the expected reward in the long run by balancing between exploration (trying different arms) and exploitation (taking the arm with the highest reward) [57], [56]. Note that for protocol fuzzing some research has also been done to incorporate a state scheduler in the stage [28].

In 2017, Böhme (2017) *et al.* proposed *AFLFast* [88] which models coverage-based grey-box fuzzing as a Markov chain. First, in contrast to *AFL* which always assigns the same energy to a seed independent on the number of times the seed has been chosen, *AFL* assigns energy exponentially dependent on the seed's amount of selection until a certain bound is met. Furthermore, to favor seeds that exercise low-frequency paths over high-frequency paths the power scheduler allocates energy inversely proportional to the density of Markov distribution. Second, while *AFL* schedules seeds in a round robin fashion, the improved *AFL* version benefits seeds that either pass over low-frequency paths or have not been scheduled often. Hence, *AFL* finds the same vulnerabilities, only faster.

Cerebro (2019) [128] by Li *et al.* upgrades the seed scheduling algorithm by integrating a multi-objective optimization model (MOO) bringing together multiple different but sometimes conflicting metrics and a non-dominated sorting based algorithm which calculates the Pareto frontier for an expanding seed pool. Moreover,

the notion of input potential, capturing the complexity of not yet covered code in the neighborhood of the execution path of the input, is introduced to determine the power allocation for a scheduled seed.

Another approach to the excessive energy allocation on high-frequency paths of *AFL* and the inflexible power allocation of *AFLFast* is the adaptive energy-saving grey-box fuzzer *EcoFuzz* (2020) [56] that realizes the same coverage with less test cases. Here, the exploration vs exploitation trade-off is seen a variant of the adversarial multi-armed bandit problem (VAMAB) as, e.g., the number of seeds ("arms") are not constant throughout the fuzzing campaign. A self-transition-based probability estimation method (SPEM) is designed to approximate the reward probability of each seed and consequently the seed with the highest reward probability is taken by the scheduler in the exploitation state. When not all seeds in the queue are fuzzed at least once, *EcoFuzz* transfers to the exploration stage where the energy allocation is done based on an adaptive average-cost-based power schedule (AAPS). This method starts by assigning the average-cost as baseline and increases the energy monotonously.

As reported by authors of *MooFuzz* (2021) [129] the seed selection strategy of *Cerebro* cannot be flexible adjusted to the instantaneous needs of the fuzzing process. Therefore, the authors develop a many-objective optimization seed scheduler taking into account dynamic seed information such as path risk, path frequency and used mutators. During fuzzing the seed pool is dynamically assigned to one of the three different stages: exploration state, search state and assessment state. Particularly, each state has its predefined goals: in the assessment stage the fuzzer explores high-risk program paths, in the search state new paths are being explored and in the assessment state promising seeds are exploited. Moreover, *MooFuzz* uses different feedback mechanisms to select the best seed depending on the current state and finds the most promising seed with a many-objective optimization model constructed with a non-dominated sorting algorithm.

Input Generation. In the input generation stage two distinct types can be observed (1) generation-based fuzzers and (2) mutation-based fuzzers. The mutation-based fuzzer starts from a seed selected by the seed scheduler and creates a number of new test cases depending on the dynamically assigned energy to this particular seed. In more detail, each test case is generated by mutating the initial seed with one of the predefined mutators. The main problem with this type of fuzzers is, however, to find where and how to mutate the current seed. As opposed to generation-based fuzzing, it is mostly used when the input semantics for the SUT are complicated, since it's not necessary for the fuzzer that the syntax is entirely known. In particular, the result of small changes to the well-structured seed has a good chance to be well-formed too without knowing what has been mutated [83]. Besides, knowing the message' grammar or protocol format can help to make more intelligent mutations (e.g., via Wireshark or machine learning).

On the contrary, generation-based fuzzers do have to acquire the input syntax and possibly the protocol rules (not for stateless programs) to be able to pass the initial validation checks of the target device in order to trigger deeper code paths. As mentioned before, inferring the model can be done manually (e.g., via

reverse-engineering) or automatically (e.g., via machine learning) [85, 130]. With this knowledge the fuzzer can generate new test cases conforming the specifications. In practice, this approach is recommended when the inputs of the mutation-based fuzzer are frequently rejected due to the strict syntax imposed by the program [83].

For example, Lyu *et al.* designed *MOPT* (2019) [131] which is a mutation scheduling scheme for mutation-based fuzzers (e.g., for *AFLFast* [88] or *VUzzer* [92]). It uses particle swarm optimization (PSO) to find the optimal mutators for the test program. As explained by the authors the optimization approach tackles the issues that the efficiency of different operators differ widely, is time dependent and varies with the target program. With the particle swarm optimization method the mutators' probabilities are updated dynamically to assure convergence.

Input Evaluation (execution and monitoring). In this step, the generated test cases are executed on the system-under-test, while the bug oracles are monitored and the feedback mechanism saved for further analysis. The bug oracles have as goal to observe different types of anomalies during the execution process such as memory bugs, semantic bugs or crashes/hangs [83, 132]. To this end, different memory sanitizers have been developed to instrument program code (if available) and detect memory vulnerabilities inside the code at run-time: Address Sanitizer (ASan) [133], SoftBound/CETS [134] and Memory Sanitizer (MSan) [135]. Moreover, semantic bugs can be exposed using differential testing which tries to find deviating behavior between similar implementations of the same application [83]. This method has been demonstrated to find implementation bugs in communication protocols as proven by [123] on DTLS' state machine. Other differential fuzzers are *T-Reqs* [136], Duo [137] and *DPIFuzz* [138]. Last, crashes or hangs can be observed by monitoring the response of the SUT on a predefined input.

Note that the feedback information is obtained either from static instrumentation inserted at the pre-processing phase or with the help of dynamic analysis during execution.

Configuration Updating. The difference between white-, black- and grey-box fuzzing is prominent at this part of the fuzzing loop. Whereas black-box fuzzers only inspect the different bug oracles to discover any abnormal behavior triggered by the test cases, white- and grey-box fuzzers effectively update their configurations based on the collected feedback information regarding the internal working of the SUT. This means that besides analyzing the state of the system to detect any faults, the stored data from the execution stage are used to intelligently shape the next fuzzing iteration (e.g., code coverage and state feedback for protocols). In particular, new configurations can be added or removed depending on the outcome of the fuzz run.

A common track in literature is to use evolutionary algorithms that optimize a fitness function to guide the fuzzing process. In detail, a seed pool is maintained where newly created test cases are appended to, when they achieve new or better results for the fitness function [83]. The most well-known coverage-guided grey-box fuzzer leveraging this approach is *AFL* [54] where inputs that trigger non-explored edges are assumed to be promising and retained to be selected later on by the scheduler. However, evolutionary fuzzing algorithms can suffer from the seed explosion problem. Nonetheless, this issue can be mitigated by implementing an efficient scheduler or

applying a corpus minimization tool periodically [139].

For mutation- and generation-based fuzzers that automatically infer the rules, syntax and semantics of the input messages of the SUT the configuration updating step can be used to refine the knowledge of the fuzzer depending on the feedback of the executed test cases. For example, inputs that are well-formed according to the model, but rejected, are interesting to retrain the machine learning model with. As an extension for protocols the finite state machine can be updated like in *AFLNet* [140].

Bug Analysis and Reporting. The last stage of the fuzzing loop is entered when either a bug or fault is exposed or the fuzzing resources are depleted. Hence, any bug observed in the configuration updating stage is analyzed in order to find out the root cause of the problem (e.g., a memory corruption, an implementation inconsistency and so on). Finally, a report is constructed mentioning the triggering test case and additional data such as the faulty behavior observed. The purpose is to make the bug reproducible so that it can be further analyzed in depth afterwards [84]. When there is no time left the fuzzing campaign is terminated, otherwise the next fuzzing iteration is started.

C.2 Machine learning in fuzzing

In this section, the current research of machine learning in the process of fuzzing is summarized. First, a short introduction is given to machine learning followed by the machine learning research in the different fuzzing stages. However, note that literature contains mostly machine learning techniques that are proposed for stateless fuzzers and not for protocols.

C.2.1 Machine learning overview

Machine learning is a technique that trains computer algorithms on experience or sampled data to perform specific tasks without being explicitly programmed to so. Furthermore, machine learning is a specific branch of artificial intelligence (AI) that was first introduced in 1950s by Arthur Samuel working in the field of computer gaming and artificial intelligence at IBM. Currently, machine learning techniques have been integrated in several domains such as image and speech recognition, self-driving cars and fuzzing [54, 141, 142].

The reason for introducing machine learning into fuzzing is to alleviate the current problems in traditional fuzzer and to make the fuzzing process more intelligent. Accordingly, there have been several attempts to incorporate existing machine learning techniques into existing fuzzers to enhance vulnerability discovery in a variety of software programs [143]. Moreover, there are primarily three different types of machine learning used in fuzzing through out literature: supervised learning, unsupervised learning and reinforcement learning. Supervised learning is characterized by algorithms that are trained to classify data into one of the predefined classes. However, creating such a labeled data set for fuzzing may be in most cases challenging and can lead to the problem of an unbalanced data set (e.g. if one class contains buggy

inputs and the other bug-free) [144]. Instead of labeling data, supervised learning methods extract patterns or group similar data without manual intervention. As a result, this technique is preferable when explicit labeling is difficult. For example a trained model can derive hidden patterns in bug triggering test cases [145]. Lastly, reinforcement learning [146, 147, 147] trains an agent to take optimal actions in its environment. More precisely, for each the taking action the agent observes a reward and tries to find the set of actions maximizing the expected reward over time [148], [142]. Besides, a subset of machine learning called deep learning [149, 150, 151] has also retrieved some attention in fuzzing [148].

C.2.2 ML-based fuzzing during the different stages

In this paragraph different fuzzing papers are classified according to the fuzzing stage(s) in which the research paper has developed a machine learning solution for. The different stages correspond to the ones described in Section C.1.2.

Pre-processing. *Learn&Fuzz* (2017) [130], invented by Godefroid *et al.*, is an recurrent-neural-network-based (RNN) fuzzer for PDF parsers. To overcome the complexity of the PDF format specification, the authors propose a neural-network-based algorithm to generate grammar-based inputs conforming the PDF specification. Their proposed generation-based fuzzing algorithm samples the neural-network to construct well-formed PDF objects maximizing the code coverage of the tested Microsoft Edge's parser and subsequently mutates parts of the generated input.

DeepSmith (2018) [149] from Li *et al.* leverages a deep learning model to validate compilers in particular for the OpenCL programming language. While prior state-of-art fuzzers for testing of compilers encompassed a lot of man work, the implemented long short-term memory (LSTM) architecture is fed with open source code and trained to infer a generative model for compiler inputs. As a result, this model is sampled to generate many random high-quality program functions.

Scheduling. *V-Fuzz* (2019) [149] is an evolutionary fuzzing framework combining a neural network (NN) used for prediction which parts of the code are vulnerable and an evolutionary fuzzer that tries to reach those buggy code segments. To this end, a graph embedding network (GEN) evaluates the attributed control flow graph (ACFG) of the binary program to train its prediction capabilities on which part of code is likely to contain bugs. For this the neural network is trained with secure and vulnerable labeled data which makes it capable to learn features and assign vulnerability scores to functions in the software. Second, the evolutionary algorithm of *V-Fuzz* assigns each basic block in the functions a static vulnerability score (SVS) leveraging the predicted vulnerability of the functions. Hence, in the seed selection stage the algorithm chooses seeds that are probable to arrive at a vulnerable part of the program's code. Therefore, the execution path of all inputs are monitored, while their fitness functions are calculated by summing up the SVSs of the activated basic blocks. Both inputs triggering a crash and inputs with high fitness scores are added to the queue of seeds.

Wang *et al.* designed *NeuFuzz* (2019) [145] a grey-box fuzzer extending *PTFuzz* with a deep neural network capable of prediction whether a test case will exercises a

vulnerable path. Inspired by prior research [152] showing that circa 20% of application code contains around 80% of bugs, the authors optimized seed scheduling and energy assignment to target vulnerable paths instead of maximizing code coverage. A deep neural network is trained based on labeled secure and buggy program paths to guide the seed scheduler intelligently in recognizing seeds that most likely exercise vulnerable code paths. By doing so, their LSTM model learns the vulnerability features embedded in the training data and classifies subsequently unseen paths. Moreover, higher mutation energy is assigned by *NeuFuzz* to seeds that are predicted to traverse vulnerable paths, while others are allocated less.

Most of the existing fuzzers leverage fixed heuristics to choose the next seed in the seed scheduling stage. However, while some heuristics work well for fuzzing certain programs, this approach lacks generalization to other programs as no one-size-fits-all rule exists, in general. As a result, *MEUZZ* (2020) [153], a smart seed scheduling based machine learning tool on top of *AFL* and *KLEE*, is designed to extract latent and entangled patterns from past scheduling behavior in a variety of programs. With this, *MEUZZ* can make smart scheduling decisions. Hence, to gain insight in the scheduling process the machine learning model extracts several features (bug-triggering, constraint solving and empirical ones) from the program using lightweight static and dynamic analysis techniques. Moreover, *MEUZZ* automatically labels executed seeds based on the seed's observed performance (i.e., the size of the input descendant tree). Furthermore, the model can be trained continuously on new seed data or offline, using all history data.

While fine-grained coverage metrics allow a fuzzer in most cases to find more bugs and explore deeper program paths, the problem of seed explosion complicates the job of the scheduler [139]. In particular, with fixed fuzzing resources the excessive number of seeds drastically decrease the time spent on each seed and increase the time for the scheduler to examine the seed pool. Therefore, the authors of (*Hier-AFL*) (2021) [139] address this issue by clustering the seeds in the seed pool according to a multi-level coverage metric and designing a reinforcement-learning-based hierarchical scheduler. A multi-level coverage metric forms a tree where each layer measures a different coverage metric (e.g., function, edge and distance coverage) and the higher the layer is situated up in the tree the more coarse-grained the measurement metric is. In addition, a test case uncovering a new function, edge or distance coverage while fuzzing is retained as seed and clustered in the corresponding leaf node of the multi-level metric tree. Moreover, the reinforcement learning-based seed scheduler implements a customized *UCB1 multi-armed bandit (MAB)* algorithm on top of the hierarchical coverage metric, balancing between exploration and exploitation.

Input Generation. *FuzzerGym* (2018) [146] is a machine learning based fuzzer built on top of *libFuzzer* which integrates OpenAI-Gym with LLVM sanitizers for program monitoring. Hence, the architecture uses OpenAI-Gym framework's reinforcement learning algorithms to train the fuzzer to select mutators yielding improvements in code coverage. To this end, an agent is trained to take actions (mutators) while observing the current state with LLVM leading to maximized rewards in terms of code coverage. The reinforcement learning algorithm used for mutator selection is deep double-Q learning since it is able to cope with the problem

of action value overestimation in stochastic domains. Due to the long training time *FuzzerGym* implements "frame-skipping". Hence, instead of learning every test run the agent repeats the same action until the neural network is updated leading to a partially observable Markov chain. Moreover, they also propose to extend the network with a long short-term memory (LSTM) layer enabling the agent to learn dependencies between "frames".

In the paper "Deep Reinforcement Fuzzing" (2018) [147] the authors model fuzzing as a reinforcement learning problem using Markov decision processes. Here, run-time information is used to guide the agent to produce high-quality input mutations and hence optimize the goal to maximize code coverage or minimize the execution time. In this scenario of reinforcement learning the agent interacts with its environment, a given seed, observes different states, the parts of the seed, performs actions, mapping substrings of the seed to probabilistic rewriting rules, and obtains the corresponding reward. Furthermore, a reward is defined as the summation of (1) the reward for the selected action and (2) the reward for the generated test case. The fuzzing algorithm implements deep Q-learning to find the optimal actions leading to high-rewards and by doing so the deep neural network simultaneously predicts the Q-values for all actions (i.e, mutators), given the current state.

According to the authors of *NEUZZ* (2019) [25], a fuzzing framework with neural-program-smoothing-based gradient-guided techniques is able to outperform evolutionary algorithms at finding solutions for high-dimensional optimization problems such as fuzzing. Therefore, to alleviate gradient-guided methods from getting stuck in suboptimal solutions as the fuzzing problem (i.e., the branching behaviours of the program) is highly discontinuous in nature, *NEUZZ* implements a feed-forward fully connected neural smoothing technique which generates a smoothed model of the target program's branching behaviors (i.e., exercised control edges). In other words the neural network learns for each byte of the program's input the corresponding relation to the edges. Using this neural network model each output neuron (i.e., edge) can be derived to an input seed. This results in the gradient telling how likely each byte of the given seed affects a particular control edge. The gradient-guided optimization of *NEUZZ* mutates the bytes of the input having the largest gradients, indicating a great chance of triggering new program behavior in terms of code coverage.

The authors Zong *et al.* of *FuzzGuard* (2020) [144] mention that most directed grey-box fuzzers can not craft test cases that can effectively reach the predefined pieces of code. As an example they concluded from their experiments with *AFLGo* that 91.7% of the time inputs were generated that didn't activate the targeted parts of the code. To tackle this problem they proposed an intelligent directed grey-box fuzzer, called *FuzzGuard* (2020) [144], which is build on top of *AFLGo*. Hence, a supervised convolutional neural network (CNN) was trained to learn the input patterns to predict reachability and filter out unreachable test cases. However, due to the unbalanced training data generated by *AFLGo* (as most test cases are labeled "non-reaching"), *FuzzGuard* integrates a step-forwarding approach. Here, instead of training the deep neural network only on predicting the target code, the model also forecasts if the pre-dominating nodes are reached. In more detail, pre-dominating

nodes are basic blocks which have to be crossed before encountering the target code. Furthermore, the fuzzer incorporates a representative data selection algorithm as the developers noticed that the prediction stage often fails to correctly classify new inputs that arrive at the buggy code through a different, but unseen path. Lastly, the model is updated using incremental learning whenever the false positive rate exceeds a predefined threshold.

While the performance in terms of code coverage of most of the traditional evolutionary fuzzers drop later on in the fuzzing campaign, machine learning-based approaches desire qualitatively and diversified training data for finding relevant mutations. However, most of the tested programs have a high-dimensional and sparse input space impeding the collection of varied training samples to learn the neural network rewarded mutations. Therefore, *MTFuzz* (2020) [120] proposes a machine learning solution to tackle, first, the performance drop by identifying the rare input bytes of the tested program, called "hot-bytes", which are able to enhance the code coverage after mutation. Second, a specific multi-task neural network (MTNN) is designed that is composed of three different stages to overcome the sparsity and diversity problem where current ML fuzzers struggle with. In particular, the neural network is trained to predict several types of coverage (i.e., edge-, approach-sensitive- and the context-sensitive coverage) for a given program input. Hence, the compact embedding in the MTNN acquires knowledge by jointly learning the "hot-bytes" in the sparse input space across the three different tasks (coverage metrics). As a result, the gradient-guided mutation stage focuses on mutating interesting bytes indicated by gradient derived from the compact embedding. Moreover, during the seed selection and incremental learning the new mutated inputs are sampled according to an importance sampling strategy.

The authors of *ATTuzz* (2021) [151] mention that even the state-of-art fuzzers have deteriorating performance over time due to the complex relationship between the test inputs and program coverage. For them the reason is two-fold: (1) they lack the ability to identify the most rewarding seeds, and (2) cannot select and mutate the relevant parts of the seeds. Therefore, *ATTuzz* assigns each basic block of the program a reward based on lightweight analysis (e.g., with the control flow graph) and dynamic fuzzing data (e.g., number of executions of each basic block). Having this information the fuzz-tester identifies the most promising seeds to mutate. Next, *ATTuzz* trains a deep learning model with attention mechanisms using past fuzzing data to predict the likelihood of a particular mutation on a specific byte to cover certain code blocks. However, as no training data is available for the uncovered basic blocks, the goal of the model is to automatically extract features in the input data to reach the least visited pre-dominant blocks hoping to transit to uncovered blocks. In essence, the deep neural network guides *ATTuzz*' algorithm on which mutator to use on which part of the selected seed with the aid of so-called heat maps.

C.3 Network protocol fuzzing

In contrast with stateless programs mentioned above, stateful programs are protocols having an internal state that is driven by previously received messages, or other events. As a result, the input space of the test program can be narrowed down knowing in which state the program currently is, since only state-specific inputs are accepted. Otherwise, the protocol entity doesn't continue, but starts over from the initial state [86, 154]. As mentioned by *AutoFuzz* [155], an automated network protocol fuzzing framework, smart protocol fuzzers have the ability to understand the protocol state machine, and the syntax and field types of the different messages throughout the protocol interaction. The construction of viable sequences of messages is mostly done by means of a tree, like a finite state machine (FSM) or a Markov chain. However, this is an extra hurdle with respect to stateless fuzzing. Along the way, several techniques have been proposed to overcome the problem of understanding the format of the messages, and the succession of packets [86, 154]. In particular, active learning techniques based on Angluin's L^* algorithm refine the protocol format by actively sending queries (more specific messages) to the protocol entity. Next, the constructed hypothesis is then checked for correctness during the validation stage [156, 123, 157]. The major drawback of this method is that it is semi-automated and demands a great deal of time to converge. Especially, active learning requires an expert to develop a mapper that converts real protocol messages to abstract symbols coming from a finite alphabetic set. Moreover, the mapping is also protocol specific and can contain some shortcomings when not carried out carefully [156, 123, 157]. Besides, an automated approach is to infer the protocol state from (source-code) feedback of the system-under-test. In particular, this feedback information can be a measure for the current state in which the target is situated. For example, *TCP-Fuzz* [26] creates a state graph based on a two-dimensional vector of branch coverage activated by a message in the TCP stack. Differently to this, *AFLNet* [140] leverages response codes from the target as an indication of the state, while *StateAFL* [30] dynamically infers protocol state machines by analyzing long-lived memory areas. Nonetheless, still some research is being carried out to find ways on how to integrate these specifications into a fuzzer without having to rely on extensive manual efforts.

In literature two distinct types of stateful protocol fuzzers can be distinguished, namely (1) mutation-based fuzzers (e.g., [28]) and (2) generation-based fuzzers (e.g., [158]). Nonetheless, a fuzzing framework can even incorporate both methods [159]. In fact, a mutation-based fuzzer uses network traffic to capture syntax, semantic and grammar of the implemented protocol without the need for a significant reverse-engineering effort or an automated protocol interference. Moreover, an entire protocol run, also denoted as a sequence-of-messages, takes the task of a seed and will hence be mutated whereas in case of stateless programs only single messages make up the seed pool. On the other hand generation-based fuzzers do have to acquire a protocol model, consisting of the different messages' formats (e.g., fields) and the correct sequences of messages, in order to create test cases that pass the initial validation checks. Early on different protocol fuzzing frameworks were developed requiring protocol knowledge and specifications from experts in advance, while the latest

tendency is to incorporate more advance methods like machine learning techniques for this purpose.

C.3.1 Mutation-based fuzzers

This section summarizes the different mutation-based fuzzers found in literature for protocols in chronological order.

AutoFuzz (2010) [155] is an open-source automated fuzzing framework for network protocols that infers the protocol state machine from captured network interactions using finite state automation (FSA). Moreover, it learns the messages' syntax, e.g., the different fields with the aid of techniques from bio-informatics. Both client and server side messages can be fuzzed using the finite state automation and the provided mutator functions. However, the major drawback of this framework is that it does not reach complete automation.

Tsankov *et al.* developed a lightweight fuzzer for testing security protocols, named *SecFuzz* (2012) [160]. The fuzzer consist of a concrete implementation of either the server or client entity which is necessary to deliver the fuzz-tester with the generation of valid inputs. Subsequently, these sequence-of-messages are first mutated using one of following type of operators: (1) message mutators (e.g., deleting a message from the sequence), (2) field mutators or (3) payload operators, and then sent to the program-under-test. To overcome the problem of creating meaningless messages when altering encrypted fields, *SecFuzz* has been issued access to the encryption keys, the cryptographic algorithms and the random numbers used during the communication. Furthermore, their dynamic memory tool detects any vulnerability triggered by the fuzzer.

In 2019, Chen *et al.* wrote a paper [161] about fuzzing techniques for stateful communication protocols as an extension to *AFL*'s stateless fuzzer. Hence, the framework consists of a state-aware fuzzer with a multi-queue and a multi-state forkserver enabling to reach more code, and deeper code paths. Furthermore, as protocol implementations have different inputs during a communication session governed by several states, a multi-queue stores the different types of messages along with their status information and type specific operands. During the execution of a mutated test case on the instrumented testing program the state engine gathers code coverage, protocol state data and status feedback to decide whether the forkserver should replicate the protocol state, move forward to the next state, or roll-back to the previous state. As a result, the state-aware fuzzer tries to maximize code coverage for each protocol state on the fly.

SGPFuzzer (2020) [28] is an evolutionary grey-box fuzzer for network protocols that leverages both coverage feedback and state feedback to drive the fuzzing process in activating more code coverage paths. Moreover, using their packet filtering and field recognition mechanisms the authors can extract the protocol specification dynamically. Meanwhile, *SGPFuzzer* records the responses from the target during protocol communication, with which the state machine model is updated in case of newly discovered states or state transitions. When a state is scheduled the fuzzer chooses a message sequence from the seed pool that is able to reach that predefined

state and mutates with stacked (i.e., message-level and field-level) operators the last messages that are not needed to reach the scheduled state. Hence, following their evolutionary approach any protocol interaction increasing at least one of the feedback mechanisms is retained as a seed for the corpus.

Developed by Pham *et al.*, *AFLNet* (2020) [140] is built on top of *AFL* and extends the grey-box fuzzer to stateful protocol fuzzing. It starts with a seed pool of recorded series of messages between the server and the client without the need for supplying message grammars to the fuzzer. Moreover, by observing response codes from the server the tool automatically maintains and updates an internal state machine that guides the fuzzing process towards interesting areas in the state space. With different mutation operators variations of the observed server-client interaction are sent to the server side and kept in case they enlarge the code or state space. Whereas the state and seed scheduling approach are similar to *SGPFuzzer*, the mutation operators are too random for effectively generation interesting test cases that conform to the unknown specifications and is, hence, only applicable to servers that output response codes such as HTTP.

The research paper of Liu *et al.* (2021) [162] investigates the performance of three different state selection algorithms when integrated in *AFLNet*, followed by the implementation of their own custom state selection algorithm, named *AFLNetLegion*. This method is based on Legion, which works following the Monte Carlo tree search (MCTS). Their preliminary study shows that the different state scheduling algorithms exhibit similar coverage performance and stability when applied to stateful network protocol fuzzing. However, the authors mention two shortcomings in *AFLNet*'s state scheduler, namely the state machine model based on the various server's response codes is too coarse-grained as the fuzzer doesn't take into account the prefixes. Secondly, the actual method cannot find an efficient balance between exploration and exploitation. As a consequence, the authors update the state machine graph and develop their MCTS variant for exploring and updating the state model repeatedly in an evolutionary fuzzer.

EPF (2021) [29] is an evolutionary, protocol-aware network fuzzing framework which was designed to combine applicability and well-established algorithms for making protocol fuzzing more effective. Particularly, the tool's architecture is composed of a pre-processing stage where the target is instrumented, the packets are modeled with Scapy, the state graph is generated and an initial genetic population is provided as seed corpus. In the dynamic analysis phase, *EPF* works as an evolutionary fuzzer leveraging a genetic algorithm that uses coverage as fitness function, while scheduling packets heuristically with population-based simulated annealing. Moreover, using the integrated Scapy API different protocols can be fuzzed by updating the protocol graph and message dissection algorithm.

Fang *et al.* developed a fuzzing framework called *ICS3Fuzzer* (2021) [163] for finding protocol implementation bugs between supervisory software and field devices. In the pre-processing phase, traces of the proprietary protocol are analyzed to infer the protocol format and obtain the state space, however, the fuzzer is still black-box as instrumentation on the bulky supervisory software would increase the overhead considerably. In particular, for the state graph a state-book is constructed based on

code execution traces and the corresponding inputs. The fuzzing phase, on the other hand, is fully automated and consists of selecting a state, taking a corresponding valuable input from the state-book which is mutated, and executing the test case on the supervisory software.

Inspired by coverage-driven fuzzing techniques for stateless programs that rely on lightweight instrumented feedback and automatically evolve a seed pool over time without any prior knowledge *StateAFL* (2021) [30] implements a state-aware grey-box fuzzer for network servers. At compile-time, lightweight instrumentation inserts probes into the device-under-test for state information by tracking long-lived memory areas and network I/O operations. Hence, during input execution the probed memory snapshot is mapped to a unique protocol state from which *StateAFL* incrementally builds a state machine to guide the fuzzing process. In essence, observing the long-lived memory gives a measure for the current state wherein the server is located. Moreover, its evolutionary approach selects a state from the state machine following its policy (e.g., rareness and history in finding new seeds), selects and mutates a series of messages that can reach the scheduled state and monitors any increase in code and/or state coverage.

In 2021, Zou *et al.* proposed a TCP stack fuzzer for uncovering memory and semantics bugs, named *TCP-Fuzz* (2021) [26] First, to generate valid test cases *TCP-Fuzz* uses a dependency-based mutation strategy where an original input sequence is mutated by a randomly chosen mutator type (e.g., inserting a new syscall) However, the mutated output has to be conform with a predefined list of dependency rules between packets and/or syscalls. Second, the fuzzer introduces a new code coverage metric, called branch transition. In particular, the researches noticed that each TCP message leads to some changes and thus a new state in the TCP stack. Hence, their new code coverage metric is a way to measure the current state of the TCP protocol. The branch transition feedback is characterised by a two-dimensional vector where the first dimension denotes the state and the second the state transition. Whereas the state of the current input is denoted as the vector of all branches (covered or not) in the TCP stack during its execution, its state transition is the difference between the branch coverage vector of the input itself and the previous input. With this dynamic information a state graph can be modeled using, in fact, code coverage information from the TCP stack. Third, *TCP-Fuzz* integrates a differential checker which inputs the same fuzzed message to different TCP stack implementations to find any implementation consistency by comparing the corresponding outputs (also known as a semantic bug).

C.3.2 Generation-based fuzzers

In this section, generation-based protocol fuzzers from literature are reviewed.

In 2006, Banks *et al.* presented *SNOOZE* (2006) [164], a flexible, security-oriented network protocol fuzzer for identifying security flaws. *SNOOZE* allows a tester to input a set of protocol specifications, user-defined fuzzing scenarios and some scenario primitives. Especially, a number of valid sequences of messages can be supplied in form of a state transition diagram, informing the fuzzer about the

different transition rules between states, and the valid messages to be generated in each state. Furthermore, the attack-specific fuzzing primitives allow *SNOOZE* to test the protocol on certain types of vulnerabilities, whereas the user-defined scenarios can be adjusted by the tester to find real-world vulnerabilities.

PULSAR (2015) [85] released in 2015 by Gascon et al. is an open-source black-box fuzzer for stateful proprietary network protocols that automatically reverse engineers the protocol from network traffic including the unknown protocol state machine and the format of the messages. Especially, in the model interference stage a probabilistic approach PRISMA is used to construct a Markov model modeling the the different states and generic format definitions for the messages, called templates. Subsequently, in the test case generation step *PULSAR* automatically generates new test cases from the derived model to evaluate the program for vulnerabilities. Lastly, to increase the coverage in the fuzzed process their proposed fuzzing subgraph (FS) algorithm guides the fuzzer to interesting states in the state model that are, for example, just discovered.

Testing industrial network protocols with fuzzing in order to learn the specific protocol format is time consuming and labor-intensive, and in case of stateful protocols the temporal features make it even worse. As a result, *SeqFuzzer* (2019) [150] from Zhao *et al.* leverages a deep learning approach to learn the target protocol format and the temporal protocol features from captured network interactions, maximizing the learning automation. To this end, the authors propose a sequence-to-sequence encoder-decoder model, where both the encoder and the decoder are composed of an input/output layer, an embedding layer and a three layer long short-term memory (LSTM) network. While the encoder learns the protocol format from the observed network traffic, the decoder generates ("plausible but") falsified test cases used for fuzzing and triggering bugs. This protocol independent fuzzer is able to find packet injection, man-in-the-middle (MITM) and other types of attacks on e.g., the Ethernet for Control Automation Technology (EtherCAT) protocol.

Song *et al.* developed a hierarchical scheduling framework for stateful network protocol fuzzing, *SPFuzz* (2019) [122] which addresses the problem of maintaining states and dependencies of messages. As a result, the tester can manually describe the protocol specifications, protocol state transitions, and inter- and intra-message dependencies of the tested protocol, with the help of *SPFuzz*' implemented language. Its three-level hierarchical fuzzing mutation strategy adopts a head, content and sequence scheduler for finding deeper code paths together with an algorithm that assigns different importance levels to strategies and messages.

*Peach** (2020) [27] is an extension to the generation-based fuzzer *Peach* and is designed to fuzz industrial control system (ICS) protocols. Based on the observation that some chunks of different packets conform to similar construction rules *Peach** breaks successful messages, which achieve new code coverage on the instrumented program, into several parts. Meanwhile, these chunks are used to assist the generation process of new test cases to turn the pointless generation strategy into more efficient construction of new quality inputs for the tested protocol. As a result, the authors expect to explore deep code paths more easily.

In 2021, *ICPFuzzer* (2021) [121] was developed by Lin *et al.* which is a black-box

fuzzing system that automatically generates new test cases to reveal vulnerabilities inside industrial control systems (ICS) and thus enhance its safety. Internally, the fuzzer trains a long short-term memory in learning the protocol features and generating new inputs for the system-under-test. Furthermore, responses coming from the ICS are monitored to adapt the strategy and direct the DUT towards abnormal behavior.

The *BLSTM-DCNNFuzz* framework (2021) [158] designed by Lv *et al.* integrates a bi-directional long short-term memory (BLSTM) in conjunction with a deep convolution generative adversarial network (DCGAN) to find vulnerabilities in a intelligent and automated way. In particular, the overall framework consist of a data pre-processing module (DPM), a model training and message generation module (MTMGM), a message sending and receiving module (MSRM) and a logging module (LM). Each of these module have their own tasks ranging from optimizing the available data for training the models to monitoring any inconsistencies in the industrial control network. In contrast with prior machine learning research pertaining to generation-based fuzzers, this methodology covers both the time dimension as well as the spatial structure dimension of ICSs. Meanwhile, the BLSTM is built up from a forward and backward sub-network learning the different features from the protocol with as goal to train the DCGAN to generate fake but plausible data. Moreover, the developers proposed several metrics to evaluate the effectiveness of their ML-based fuzzing tool, namely test case recognition rate (TCRR), anomalies triggered efficiency (ATE) and diversity of generated data (DGD).

In 2021, Schumilo *et al.* introduced *Nyx-Net* (2021) [165] generalizing the generation-based fuzzer *Nyx* to network fuzzing capable of testing a wide range of targets such as servers, clients and games. *Nyx-Net* is a hypervisor-based snapshot fuzzer where the target runs in a customized virtual machine (VM) and network functionalities are selectively emulated. As a result, the emulation allows the fuzzer, first, to know the precise moments of packet handling within the target, sidestepping the problem to guess the correct timeouts. Second, the emulation allows the fuzz-tester to operate at high networking performance. Moreover, the implementation of incremental snapshots permits *Nyx-Net* to start each new test case from a clean state and allows to skip over different consecutive messages to start fuzzing on later packets in the protocol run (e.g., after the handshake).

Bibliography

[1] A. Bradai, K. Singh, T. Ahmed, and T. Rasheed, "Cellular software defined networking: a framework," *IEEE Communications Magazine*, vol. 53, no. 6, pp. 36–43, 2015.

[2] A. C. Rasmussen and M. Kielgast, "Embedded massive mtc device emulator for lte using software defined radios," Ph.D. dissertation, MA thesis. Denmark: Aalborg University, 17, 2017.

[3] Y.-A. Jung, D. Shin, and Y.-H. You, "A computationally efficient joint cell search and frequency synchronization scheme for lte machine-type communications," *Symmetry*, vol. 11, no. 11, p. 1394, 2019.

[4] M. Wu, L. Jiang, J. Xiang, Y. Zhang, G. Yang, H. Ma, S. Nie, S. Wu, H. Cui, and L. Zhang, "Evaluating and improving neural program-smoothing-based fuzzing," 2022.

[5] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," 01 2018.

[6] 3rd generation partnership project, https://www.3gpp.org/, 2022.

[7] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123854896000011

[8] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128124581000071

[9] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128124581000058

[10] M. Hasan, "State of iot 2022: Number of connected iot devices growing 18% to 14.4 billion globally," 2022. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[11] B. Jovanovic, "Internet of things statistics for 2022 - taking things apart," 2022. [Online]. Available: https://dataprot.net/statistics/iot-statistics/

[12] 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.

[13] V. Hassija, V. Chamola, V. Saxena, D. Jain, P. Goyal, and B. Sikdar, "A survey on iot security: Application areas, security threats, and solution architectures," *IEEE Access*, vol. 7, pp. 82 721–82 743, 2019.

[14] A. Riahi Sfar, E. Natalizio, Y. Challal, and Z. Chtourou, "A roadmap for security challenges in the internet of things," *Digital Communications and Networks*, vol. 4, no. 2, pp. 118–137, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352864817300214

[15] ISO, "Iso/iec 27001:2013 information technology — security techniques — information security management systems — requirements," 2013. [Online]. Available: https://www.iso.org/standard/54534.html

[16] D. Rupprecht, A. Dabrowski, T. Holz, E. Weippl, and C. Pöpper, "On security research towards future mobile network generations," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2518–2542, 2018.

[17] M. Chlosta, D. Rupprecht, and T. Holz, "On the challenges of automata reconstruction in lte networks," in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 164–174. [Online]. Available: https://doi.org/10.1145/3448300.3469133

[18] J. Hoes, "Rise of the machines," Master's thesis, Belgium: KU Leuven, 2021.

[19] T. Ji, Y. Wu, C. Wang, X. Zhang, and Z. Wang, "The coming era of alpha-hacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques," in *2018 IEEE third international conference on data science in cyberspace (DSC)*. IEEE, 2018, pp. 53–60.

[20] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *Proceedings of the 2012 Fourth International Conference on Multimedia Information Networking and Security*, ser. MINES '12. USA: IEEE Computer Society, 2012, p. 152–156. [Online]. Available: https://doi.org/10.1109/MINES.2012.202

[21] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, dec 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[22] COSIC, "About us," 2022. [Online]. Available: https://www.esat.kuleuven.be/cosic/about-us/

[23] Keysight Technologies, 2022. [Online]. Available: https://www.keysight.com/zz/en/home.html

[24] , "Mealy and moore finite-state machine 1. finite-state machine," 2013. [Online]. Available: https://www.academia.edu/34997916/Mealy/_and/_Moore/_Finite-State/_Machine/_1./_Finite-State/_Machine

[25] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 803–817.

[26] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "{TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 489–502.

[27] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, "Ics protocol fuzzing: Coverage guided packet crack and generation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[28] Y. Yu, Z. Chen, S. Gan, and X. Wang, "Sgpfuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations," *IEEE Access*, vol. 8, pp. 198 668–198 678, 2020.

[29] R. Helmke, E. Winter, and M. Rademacher, "Epf: An evolutionary, protocol-aware, and coverage-guided network fuzzing framework," in *2021 18th International Conference on Privacy, Security and Trust (PST)*. IEEE, 2021, pp. 1–7.

[30] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *arXiv preprint arXiv:2110.06253*, 2021.

[31] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, *MTFuzz: Fuzzing with a Multi-Task Neural Network*. New York, NY, USA: Association for Computing Machinery, 2020, p. 737–749. [Online]. Available: https://doi.org/10.1145/3368089.3409723

[32] S. Ahmadi, "Chapter 2 - network architecture," in *LTE-Advanced*, S. Ahmadi, Ed. Academic Press, 2014, pp. 29–119. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124051621000022

[33] C. Cox, *Air Interface Layer 2*, 2014, pp. 171–183.

[34] S. Ahmadi, "Chapter 7 - radio link control functions," in *LTE-Advanced*, S. Ahmadi, Ed. Academic Press, 2014, pp. 311–341. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124051621000071

[35] E. Dahlman, S. Parkvall, and J. Sköld, "Chapter 4 - radio-interface architecture," in *4G LTE-Advanced Pro and The Road to 5G (Third Edition)*, third edition ed., E. Dahlman, S. Parkvall, and J. Sköld, Eds. Academic Press, 2016, pp. 55–74. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128045756000042

[36] S. Ahmadi, "Chapter 6 - packet data convergence protocol functions," in *LTE-Advanced*, S. Ahmadi, Ed. Academic Press, 2014, pp. 289–309. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B978012405162100006X

[37] S. Ahmadi, "Chapter 5 - radio resource control functions," in *LTE-Advanced*, S. Ahmadi, Ed. Academic Press, 2014, pp. 227–287. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124051621000058

[38] S. Ahmadi, "Chapter 4 - system operation and ue states," in *LTE-Advanced*, S. Ahmadi, Ed. Academic Press, 2014, pp. 153–225. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124051621000046

[39] M. Olsson, S. Sultana, S. Rommer, L. Frid, and C. Mulligan, "Chapter 11 - protocols," in *SAE and the Evolved Packet Core*, M. Olsson, S. Sultana, S. Rommer, L. Frid, and C. Mulligan, Eds. Oxford: Academic Press, 2010, pp. 276–342. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123748263000114

[40] E. Dahlman, S. Parkvall, and J. Sköld, "Chapter 11 - access procedures," in *4G LTE-Advanced Pro and The Road to 5G (Third Edition)*, third edition ed., E. Dahlman, S. Parkvall, and J. Sköld, Eds. Academic Press, 2016, pp. 285–308. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B978012804575600011X

[41] 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.

[42] C. Cox, *Power-On and Power-Off Procedures*, 2014, pp. 185–201.

[43] C. Cox, *System Architecture Evolution*, 2014, pp. 21–47.

[44] C. Cox, *Security Procedures*, 2014, pp. 203–213.

[45] C. Cox, *Mobility Management*, 2014, pp. 237–254.

[46] 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," in *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, ser.

WiNTECH '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 25–32. [Online]. Available: https://doi.org/10.1145/2980159.2980163

[47] Software Radio Systems, "srsRAN: Your own mobile network," 2022. [Online]. Available: https://www.srslte.com/

[48] Software Radio Systems, "Getting started," 2022. [Online]. Available: https://docs.srsran.com/en/latest/usermanuals/source/srsepc/source/2_epc_getstarted.html

[49] Software Radio Systems, "srsRAN with ZMQ Virtual Radios," 2022. [Online]. Available: https://docs.srsran.com/en/latest/app_notes/source/zeromq/source/index.html

[50] 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

[51] Elprocus, "At commands tutorial," 2022. [Online]. Available: https://www.elprocus.com/at-commands-tutorial/

[52] I. Z. S. Z. J. G. J. CAI, B. ZHOA 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

[53] The Clang Team, "Sanitizercoverage," 2022. [Online]. Available: https://clang.llvm.org/docs/SanitizerCoverage.html

[54] Google, "AFL," https://github.com/google/AFL, 2022.

[55] J. Pereyda, "boofuzz documentation," 2019.

[56] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "{EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2307–2324.

[57] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 511–522.

[58] A. Garg and K. Tai, "Comparison of statistical and machine learning methods in modelling of data with multicollinearity," *Int. J. of Modelling*, vol. 18, pp. 295 – 312, 01 2013.

[59] 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*, vol. 8, no. 13, pp. 10 390–10 411, 2021.

[60] Raspberry pi os, https://raspberrypi.org/software/, 2022.

[61] Busybox, https://www.busybox.net/about.html, 2022.

[62] FreeRTOS, www.freertos.org, 2022.

[63] VxWorks, https://www.windrive.com/products/vxworks, 2022.

[64] 3rd generation partnership project, https://www.3gpp.org/specifications/67-releases, 2022.

[65] E. Stepanov and K. Serebryany, "Memorysanitizer: Fast detector of uninitialized memory use in c++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, pp. 46–55.

[66] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 28.

[67] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer - data race detection in practice," pp. 62–71, 01 2009.

[68] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 437–448. [Online]. Available: https://doi.org/10.1145/2897845.2897900

[69] D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for linux-based embedded firmware," 01 2016.

[70] A. Beckus, "Qemu with an stm32 microcontroller implementation," 2012. [Online]. Available: http://beckus.github.io/qemu_stm32/

[71] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares," 02 2014.

[72] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling Near-Real-Time dynamic analyses of embedded systems," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher

[73] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: Peripheral proxying supported embedded code testing," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 329–340. [Online]. Available: https://doi.org/10.1145/2590296.2590301

[74] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: Fuzzing iot firmware via multi-stage message generation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2525–2527.

[75] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 19–36.

[76] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[77] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Diane: identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 484–500.

[78] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 337–350.

[79] Z. Albayrak and C. Torun, "Recent lte simulation tools," 05 2016.

[80] Yahoo Finance, "Keysight technologies, inc. (keys)," URL: https://finance.yahoo.com/quote/KEYS/profile/?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAAF_52MQXMaUYf1dDpi7edoakvjewm_-Zd4qdxscUoGDImQIPlRdD-7DfNXFuA9U8e3l45j2tyPohSy8jLLI4HboLkX7-dsKd1RURakODREkNEpt last checked on 2022-06-07.

[81] A. Smith, "Why wedgewood partners is bullish on keysight technologies (keys) stock?" URL: https://www.insidermonkey.com/blog/why-wedgewood-partners-is-bullish-on-keysight-technologies-keys-stock-830433/, last checked on 2022-06-07.

[82] S. Ahmadi, "Chapter 3 - e-utran and epc protocol structure," in *LTE-Advanced*, S. Ahmadi, Ed. Academic Press, 2014, pp. 121–152. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124051621000034

[83] V. J. 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*, vol. 47, no. 11, pp. 2312–2331, 2019.

[84] 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*, 2021.

[85] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 330–347.

[86] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[87] K. M. Alshmrany, R. S. Menezes, M. R. Gadelha, and L. C. Cordeiro, "Fusebmc: A white-box fuzzer for finding security vulnerabilities in c programs (competition contribution)," *Fundamental Approaches to Software Engineering*, vol. 12649, p. 363, 2021.

[88] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[89] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[90] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[91] M. Charlie, "Fuzz by number," in *CanSecWest Conference*, 2008. [Online]. Available: https://www.secwest.net/csw08/csw08-miller.pdf

[92] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[93] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[94] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[95] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[96] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 499–513.

[97] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization." in *NDSS*, 2020.

[98] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 858–870.

[99] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "Pathafl: Path-coverage assisted fuzzing," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 598–609.

[100] Y. Kim and J. Yoon, "Maxafl: Maximizing code coverage with a gradient-based optimization technique," *Electronics*, vol. 10, no. 1, p. 11, 2020.

[101] X. Li, L. Sun, R. Jiang, H. Qu, and Z. Yan, "Ota: An operation-oriented time allocation strategy for greybox fuzzing," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 108–118.

[102] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "Pata: Fuzzing with path aware taint analysis," in *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA*, 2022, pp. 154–170.

[103] H. L. Nguyen and L. Grunske, "Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing," *arXiv preprint arXiv:2202.13114*, 2022.

[104] P. Wang, X. Zhou *et al.*, "Sok: The progress, challenges, and perspectives of directed greybox fuzzing," *arXiv preprint arXiv:2005.11907*, 2020.

[105] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[106] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2095–2108.

[107] H. Liang, Y. Zhang, Y. Yu, Z. Xie, and L. Jiang, "Sequence coverage directed greybox fuzzing," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, 2019, pp. 249–259.

[108] Z. Wang, B. Liblit, and T. Reps, "Tofu: Target-oriented fuzzer," *arXiv preprint arXiv:2004.14375*, 2020.

[109] J. Ye, R. Li, and B. Zhang, "Rdfuzz: Accelerating directed fuzzing with intertwined schedule and optimized mutation," *Mathematical Problems in Engineering*, vol. 2020, 2020.

[110] V. J. Manès, S. Kim, and S. K. Cha, "Ankou: guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1024–1036.

[111] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 999–1010.

[112] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for {Use-After-Free} vulnerabilities," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 47–62.

[113] H. Liang, L. Jiang, L. Ai, and J. Wei, "Sequence directed hybrid fuzzing," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 127–137.

[114] K. Zhu, Y. Lu, H. Huang, L. Yu, and J. Zhao, "Constructing more complete control flow graphs utilizing directed gray-box fuzzing," *Applied Sciences*, vol. 11, no. 3, p. 1351, 2021.

[115] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3559–3576.

[116] S. Wang, X. Jiang, X. Yu, and S. Sun, "Kcfuzz: Directed fuzzing based on keypoint coverage," in *International Conference on Artificial Intelligence and Security*. Springer, 2021, pp. 312–325.

[117] H. Yoo, J. Hong, L. Bader, D. W. Hwang, and S. Hong, "Improving configurability of unit-level continuous fuzzing: An industrial case study with sap hana," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1101–1105.

[118] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *Proceedings-2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[119] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: A directed greybox fuzzer driven by deviation basic blocks," in *2022 IEEE/ACM 44st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022.

[120] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "Mtfuzz: fuzzing with a multi-task neural network," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 737–749.

[121] P.-Y. Lin, C.-W. Tien, T.-C. Huang, and C.-W. Tien, "Icpfuzzer: proprietary communication protocol fuzzing by using machine learning and feedback strategies," *Cybersecurity*, vol. 4, no. 1, pp. 1–15, 2021.

[122] C. Song, B. Yu, X. Zhou, and Q. Yang, "Spfuzz: a hierarchical scheduling framework for stateful network protocol fuzzing," *IEEE Access*, vol. 7, pp. 18 490–18 499, 2019.

[123] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. De Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of {DTLS} implementations using protocol state fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2523–2540.

[124] B. K. Aichernig, E. Muškardin, and A. Pferscher, "Learning-based fuzzing of iot message brokers," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 47–58.

[125] M. Chlosta, D. Rupprecht, and T. Holz, "On the challenges of automata reconstruction in lte networks," in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2021, pp. 164–174.

[126] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 861–875.

[127] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 230–243.

[128] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 533–544.

[129] X. Zhao, H. Qu, W. Lv, S. Li, and J. Xu, "Moofuzz: Many-objective optimization seed schedule for fuzzer," *Mathematics*, vol. 9, no. 3, p. 205, 2021.

[130] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 50–59.

[131] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1949–1966.

[132] M. S. Simpson and R. K. Barua, "Memsafe: ensuring the spatial and temporal memory safety of c at runtime," *Software: Practice and Experience*, vol. 43, no. 1, pp. 93–128, 2013.

[133] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[134] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 245–258.

[135] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 46–55.

[136] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-reqs: Http request smuggling with differential fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1805–1820.

[137] X. Zhang, J. Liu, N. Sun, C. Fang, J. Liu, J. Wang, D. Chai, and Z. Chen, "Duo: Differential fuzzing for deep learning operators," *IEEE Transactions on Reliability*, vol. 70, no. 4, pp. 1671–1685, 2021.

[138] G. S. Reen and C. Rossow, "Dpifuzz: A differential fuzzing framework to detect dpi elusion strategies for quic," in *Annual Computer Security Applications Conference*, 2020, pp. 332–344.

[139] J. Wang, C. Song, and H. Yin, "Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing."

[140] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[141] Datascience Berkeley, "What is machine learning (ml)?" URL: https://ischoolonline.berkeley.edu/blog/what-is-machine-learning/, last checked on 2022-01-21.

[142] IBM Cloud Education, "Machine learning," URL: https://www.ibm.com/cloud/learn/machine-learning, last checked on 2022-01-21.

[143] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzzing based on machine learning techniques," *PloS one*, vol. 15, no. 8, p. e0237749, 2020.

[144] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "{FuzzGuard}: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2255–2269.

[145] Y. Wang, Z. Wu, Q. Wei, and Q. Wang, "Neufuzz: Efficient fuzzing with deep neural network," *IEEE Access*, vol. 7, pp. 36 340–36 352, 2019.

[146] W. Drozd and M. D. Wagner, "Fuzzergym: A competitive framework for fuzzing and learning," *arXiv preprint arXiv:1807.07490*, 2018.

[147] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 116–122.

[148] G. J. Saavedra, K. N. Rodhouse, D. M. Dunlavy, and P. W. Kegelmeyer, "A review of machine learning applications in fuzzing," *arXiv preprint arXiv:1906.11133*, 2019.

[149] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-fuzz: Vulnerability-oriented evolutionary fuzzing," *arXiv preprint arXiv:1901.01142*, 2019.

[150] H. Zhao, Z. Li, H. Wei, J. Shi, and Y. Huang, "Seqfuzzer: An industrial protocol fuzzing framework from a deep learning perspective," in *2019 12th IEEE Conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 59–67.

[151] S. Zhu, J. Wang, J. Sun, J. Yang, X. Lin, L. Zhang, and P. Cheng, "Better pay attention whilst fuzzing," *arXiv preprint arXiv:2112.07143*, 2021.

[152] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 2002, pp. 55–64.

[153] Y. Chen, M. Ahmadi, B. Wang, L. Lu *et al.*, "{MEUZZ}: Smart seed scheduling for hybrid fuzzing," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 77–92.

[154] Z. Hu and Z. Pan, "A systematic review of network protocol fuzzing techniques," in *2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, vol. 4. IEEE, 2021, pp. 1000–1005.

[155] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *IJCSNS*, vol. 10, no. 8, p. 239, 2010.

[156] J. De Ruiter and E. Poll, "Protocol state fuzzing of {TLS} implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 193–206.

[157] S. R. Hussain, I. Karim, A. A. Ishtiaq, O. Chowdhury, and E. Bertino, "Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4g lte cellular devices," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1082–1099.

[158] W. Lv, J. Xiong, J. Shi, Y. Huang, and S. Qin, "A deep convolution generative adversarial networks based fuzzing framework for industry control protocols," *Journal of Intelligent Manufacturing*, vol. 32, no. 2, pp. 441–457, 2021.

[159] S. Potnuru and P. K. Nakarmi, "Berserker: Asn. 1-based fuzzing of radio resource control protocol for 4g and 5g," in *2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, 2021, pp. 295–300.

[160] P. Tsankov, M. T. Dashti, and D. Basin, "Secfuzz: Fuzz-testing security protocols," in *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE, 2012, pp. 1–7.

[161] Y. Chen, T. Lan, and G. Venkataramani, "Exploring effective fuzzing strategies to analyze communication protocols," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2019, pp. 17–23.

[162] D. Liu, V.-T. Pham, G. Ernst, T. Murray, and B. I. Rubinstein, "State selection algorithms and their impact on the performance of stateful network protocol fuzzing," *arXiv preprint arXiv:2112.15498*, 2021.

[163] D. Fang, Z. Song, L. Guan, P. Liu, A. Peng, K. Cheng, Y. Zheng, P. Liu, H. Zhu, and L. Sun, "Ics3fuzzer: A framework for discovering protocol implementation bugs in ics supervisory software by fuzzing," in *Annual Computer Security Applications Conference*, 2021, pp. 849–860.

[164] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: toward a stateful network protocol fuzzer," in *International conference on information security*. Springer, 2006, pp. 343–358.

[165] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 166–180.