TECHNISCHE
UNIVERSITÄT
DARMSTADT

MODIFICATION OF LTE FIRMWARES ON SMARTPHONES

CARSTEN GERALD BRUNS

Master Thesis

May 1, 2017

Secure Mobile Networking Lab
Department of Computer Science

Modification of LTE Firmwares on Smartphones
Master Thesis
SEEMOO-MSC-0093

Submitted by Carsten Gerald Bruns
Date of submission: May 1, 2017

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisor: Matthias Schulz

Technische Universität Darmstadt
Department of Computer Science
Secure Mobile Networking Lab

ABSTRACT

Every mobile phone contains a modem subsystem responsible for communication with mobile networks. In contrast to the well known main system of smartphones, running for example an Android operating system, the modem hardware details and its software are secrets of the manufacturer, leaving the modem as a black box to us. Hence, this work analyzes recent *Qualcomm* modems supporting the latest deployed communication standard LTE. We then use the gained knowledge to implement a patching framework allowing easy modification of the modem's firmware binary in a high level programming language. To demonstrate its usability, we realize applications ranging from debugging tools up to LTE MAC layer sniffing, security key extraction and access to channel estimates of the physical layer. These also show that malicious code in the modem subsystem imposes a severe and realistic threat. Furthermore, this work opens the modem as a research platform for recent mobile network technologies, removing the need for expensive special equipment in many research projects.

ZUSAMMENFASSUNG

Jedes Mobiltelefon enthält ein Modem-Teilsystem, welches für die Kommunikation mit Mobilfunknetzen verantwortlich ist. Im Unterschied zum gut verstandenen Hauptsystem von Smartphones, auf welchem zum Beispiel ein Android Betriebssystem läuft, sind die Hardware-Details und die Software des Modems Geheimnisse des Herstellers, weshalb das Modem für uns eine Black Box darstellt. Daher analysiert diese Arbeit neuere *Qualcomm* Modems, die den neuesten eingesetzten Kommunikationsstandard LTE unterstützen. Wir nutzen die dadurch erlangten Kenntnisse dann um ein Patching Framework zu implementieren, das Modifikationen der Firmware-Binärdatei auf einfache Weise in einer höheren Programmiersprache erlaubt. Um dessen Nutzbarkeit zu demonstrieren, realisieren wir Anwendungen von DebuggingTools bis hin zu LTE MAC Layer Sniffing, Security Schlüssel Extrahierung und Zugriff auf Kanalschätzungen des Physical Layers. Diese zeigen auch, dass bösartiger Code im Modem-Teilsystem eine ernste und realistische Bedrohung darstellt. Zusätzlich öffnet diese Arbeit das Modem als Forschungsplattform für neueste Mobilfunktechnologien und beseitigt die Notwendigkeit für teures Spezialequipment in vielen Forschungsprojekten.

CONTENTS

I INTRODUCTION 1

1 MOTIVATION 3
- 1.1 Mobile communication modems 4
- 1.2 Aim of this thesis 5
- 1.3 Basebands in smartphones 5
  - 1.3.1 Market Share 6

2 RELATED WORK 7
- 2.1 Baseband reverse engineering and modification 7
  - 2.1.1 Contrast to this work 8
- 2.2 Internal data collection 9

3 BACKGROUND: LTE BASICS 11
- 3.1 LTE overview 11
- 3.2 PHY layer 13
  - 3.2.1 OFDM 13
  - 3.2.2 LTE resource grid 14
  - 3.2.3 Channel fading 16
- 3.3 MAC layer 17
  - 3.3.1 Random access procedure 18
- 3.4 Security 19

4 TARGET HARDWARE 21

II ANALYSIS OF QUALCOMM MODEMS 23

5 SYSTEM ARCHITECTURE 25
- 5.1 Overall system 25
- 5.2 Modem architecture 25
- 5.3 Digital baseband 27

6 HARDWARE COMPONENTS 29
- 6.1 Hexagon Processor 29
  - 6.1.1 Versions 30
  - 6.1.2 Instruction packets 30
  - 6.1.3 Constant extenders 31
  - 6.1.4 Hardware Threads 32
  - 6.1.5 Other specialities 33
- 6.2 Other related components 33
  - 6.2.1 Access Protection Units 33
  - 6.2.2 Tightly-coupled memory 35

7 MODEM SOFTWARE 37
- 7.1 Firmware loading 37
- 7.2 Firmware authentication 38
  - 7.2.1 Issues 40
- 7.3 Operating System 40

7.3.1 Tasks 41
7.3.2 Memory management 42
7.4 Security features 44
7.5 Communication with HLOS 44
7.5.1 Protocol stack 45
7.5.2 QMI 45
8 REVERSE ENGINEERING THE MODEM 47
8.1 String analysis 48

III MODIFICATION OF FIRMWARES 51
9 PATCHING FRAMEWORK 53
9.1 Firmware image format 54
9.2 Patching process 55
9.2.1 Extracting the base firmware 57
9.2.2 Firmware wrapper generation 58
9.2.3 Preparation: fw_org functions generation 60
9.2.4 Compiling 62
9.2.5 Linking 62
9.2.6 Patching 63
9.2.7 Image generation 65
9.3 Porting to new targets 66
9.3.1 Implementation 67
9.3.2 Limitations 68
9.3.3 Matching performance 69
10 IMPLEMENTED PROJECTS 71
10.1 Information exchange 71
10.1.1 Using the QMI protocol 72
10.1.2 Kernel module 73
10.1.3 User space implementation 75
10.2 Framework usage example 75
10.3 Modem memory access 76
10.3.1 Security aspects 77
10.3.2 Outlook: Debugger 77
10.4 LTE MAC Layer frame sniffing 78
10.4.1 Limitations 79
10.4.2 Choice of the target layer 81
10.4.3 Sniffing scheme 81
10.4.4 Implementation 82
10.4.5 RACH preamble 85
10.5 LTE cryptography key extraction 85
10.6 LTE channel estimation 86
10.6.1 Implementation 88
10.6.2 Output data 89
11 PROBLEM: FIRMWARE AUTHENTICATION 91
11.1 Configuring protection units 91
11.2 Fuse bits 92

11.2.1 Locking access to fuse bits 93
11.2.2 Changing the root-of-trust 94

IV DISCUSSION AND CONCLUSIONS 95
12 DISCUSSION 97
12.1 Considered scenario 97
12.1.1 Extending the scenario 98
12.2 Target devices 99
12.2.1 Bypassing the firmware authentication 100
12.3 Applications 101
12.3.1 Malicious applications 101
12.3.2 Modifications wanted by the user 102
12.4 Outlook 104
13 CONCLUSIONS 107

V APPENDIX 109
A APPENDIX 111
A.1 Task startup sequence 111
A.2 Occurred problems 114
A.2.1 Hexagon issues 114
A.2.2 Fastboot with the Asus PadFone Infinity 2 . . . 116

BIBLIOGRAPHY 117

LIST OF FIGURES

Figure 1 Baseband market share in first half of 2016 . . 6
Figure 2 LTE protocol stack layers 12
Figure 3 LTE downlink resource grid for 1.4 MHz, two transmit antennas 15
Figure 4 LTE RACH procedure 19
Figure 5 LTE key derivation scheme 20
Figure 6 Snapdragon 800 system overview 26
Figure 7 Modem architecture overview 27
Figure 8 Hexagon execution units 31
Figure 9 Modem firmware loading process 38
Figure 10 Overview of the patching process of the framework 56
Figure 11 Percentage of automatically detected functions 69
Figure 12 Information exchange with patch code in the modem 72
Figure 13 LTE MAC sniffing capture in Wireshark 80
Figure 14 LTE MAC sniffing setup 82
Figure 15 Channel estimation plots 90

LIST OF TABLES

Table 1 Overview of related work 8
Table 2 Channels in LTE 13
Table 3 LTE bandwidth options 14
Table 4 LTE frame structure 15
Table 5 LTE cryptography algorithms 20
Table 6 MSM8974 (Snapdragon 800) specifications . . 21
Table 7 Target devices specifications 22
Table 8 QDSP6 versions 30
Table 9 Execution of hardware threads in the QDSP6 pipeline 32
Table 10 Enforcing of rules by MPUs in case of overlap 35
Table 11 List of active tasks 43
Table 12 Inter Process Communication mechanisms . . 43
Table 13 Protocol layering for communication between Modem and HLOS 45
Table 14 Files in a modem firmware image 54
Table 15 Files used by the patching framework 57

Table 16 Automated porting results 69
Table 17 Implemented projects 71
Table 18 Message structure to tunnel messages through QMI ping service 73
Table 19 MAC uplink frame generation tasks 83

LISTINGS

Listing 1 Example strings found in modem firmware . . 48
Listing 2 Firmware function declaration example 59
Listing 3 Example of a generated *fw_org* function 62
Listing 4 Examples of function annotations 64
Listing 5 Captured example snprintf output messages . 76
Listing 6 LTE key extraction output messages 87
Listing 7 Startup sequence of tasks and corresponding thread creations 111
Listing 8 Script to start *fastboot* on the Asus PadFone Infinity 2 116

ACRONYMS

ADC	Analog-to-Digital Converter
API	Application Programming Interface
APU	Area Protection Unit
ARM	Advanced RISC Machines
ASCII	American Standard Code for Information Interchange
BSR	Buffer Status Report
CSI	Channel State Information
DAC	Digital-to-Analog Converter
DMS	Device Management Service
DMT	Dynamic Multi-Threading
DSM	Data Services Memory
DSP	Digital Signal Processor
EEA	EPS Encryption Algorithm
EIA	EPS Integrity Algorithm
ELF	Executable and Linkable Format
eNodeB	Evolved Node B
EPS	Evolved Packet System
FAT16	File Allocation Table 16
FDD	Frequency Division Duplexing
FFT	Fast Fourier Transform
GPS	Global Positioning System
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HARQ	Hybrid ARQ
HLOS	High Level Operating System
HMAC	Keyed-Hash Message Authentication Code

IFFT Inverse Fast Fourier Transform
IMT Interleaved Multi-Threading
IP Internet Protocol
IPC Inter-Process Communication
IQ Inphase and Quadrature

JSON JavaScript Object Notation

LTE Long Term Evolution

MAC Message Authentication Code
MAC Medium Access Control
MBA Modem Boot Authenticator
MIB Master Information Block
MIMO Multiple Input Multiple Output
MME Mobility Management Entity
MPU Memory Protection Unit

NAS Network Access Service
NAS Non-Access Stratum

OEM Original Equipment Manufacturer
OFDM Orthogonal Frequency-Division Multiplexing
OS Operating System

PBL Primary Boot Loader
PDCP Packet Data Convergence Control
PHR Power Headroom Report
PHY Physical Layer
PPP Point-to-Point Protocol

QAM Quadrature Amplitude Modulation
QMI Qualcomm MSM Interface
QoS Quality of Service
QPSK Quadrature Phase-Shift Keying
QuRT Qualcomm Real Time operating system

RACH Random Access Channel

RAR	Random Access Response
RB	Resource Block
RF	Radio Frequency
RG	Resource Group
RISC	Reduced Instruction Set Computer
RLC	Radio Link Control
RNTI	Radio Network Temporary Identifier
ROM	Read-Only Memory
RPU	Register Protection Unit
RRC	Radio Resource Control
RTOS	Real Time Operating System
SDK	Software Development Kit
SDR	Software Defined Radio
SDU	Service Data Unit
SIB	System Information Block
SIM	Subscriber Identity Module
SMD	Shared Memory Driver
SMEM	Shared Memory
SMS	Short Message Service
SoC	System on Chip
SR	Scheduling Request
TCM	Tightly-Coupled Memory
TLB	Translation Lookaside Buffer
TLV	Type-Length-Value
UDP	User Datagram Protocol
UE	User Equipment
UMTS	Universal Mobile Telecommunications System
VLIW	Very Large Instruction Word
VoLTE	Voice over LTE
WCDMA	Wideband Code Division Multiple Access
WLAN	Wireless Local Area Network
XPU	Access Protection Unit

Part I

INTRODUCTION

In the first part, we give an introduction into the topic and motivate this thesis. After presenting some general background, the aims of this work are defined and explained. We show why we focus on *Qualcomm* hardware only. The second chapter of this part summarizes previous work related to this thesis. Background knowledge about the LTE communication standard closes this part.

1 MOTIVATION

Today, almost every person carries a mobile phone or even a smartphone in the pocket throughout the whole day. These devices contain a large amount of various sensing capabilities, ranging from a microphone over a Global Positioning System (GPS) receiver to cameras and many other sensors. In addition, users often store many login credentials and other sensitive data on their phone. Paired with rich communication facilities, such as Wireless Local Area Network (WLAN), Bluetooth and mobile communication networks, they build a valuable target for attackers to spy on users and to collect data.

Most attack scenarios today focus on the main operating system, High Level Operating System (HLOS), running on the phone: *Google Android* (86.8% of market share), *Apple iOS* (12.5%), *Microsoft Windows Phone* (0.3%) to name the ones with the most important shares (others: 0.4%) in the third quarter of 2016 [18]. However, a modern smartphone consists of many subsystems responsible for different subtasks of the whole system and each of them might contain a processor running some kind of instruction code. The main processor running the HLOS is just one of them, even though the most important one as it coordinates and controls the other subsystems like audio processing or WLAN.

A modern smartphone consists of many subsystems, each realizing a subset of the overall functionality.

These subsystems are often forgotten but an attack mounted directly against one of them would be able to act hidden from the user and withstand common countermeasures as only the internals of the subsystem are modified and not its external appearance to the other parts of the system. In case of a communication device, the attackers code could intercept all of the user's communication and forward it to the attacker. On the other hand, depending on the overall system architecture, the subsystems usually have limited access to other subsystems (especially memory containing sensitive data, we show that in detail later), minimizing the capabilities a bit but still presenting a severe danger.

Besides security considerations, access to these subsystems and knowledge of how they work is crucial to use smartphones as cheap research platforms. They can be used to implement new concepts, for example a new communication standard or to analyze the effects of modifications to existing protocols. Also, access to internal functions, measured values or states would allow various research usages, like analyzing how a system adapts to certain external conditions or getting measurement values in a test setup. These are just some applications, in general it is desired to use the already existing system as

a base instead of having to rebuild everything from scratch or using expensive special equipment.

Unfortunately, from this point of view, most subsystems are presented as closed black boxes to end users. Only the company building the subsystem, or sometimes also the company designing the smartphone integrating them, has access to internals and, in case it contains programmable components, the code running on it or configuring it. This code is also called firmware.

The modem is presented as a black box to us, we do not know how it works internally and cannot modify it.

Of course, our wish for an open modem which we can modify, contradicts the security aspect discussed before: an attacker should not be able to modify the system to insert malicious functionality, for this, it is desired to check the authenticity of the subsystem's code.

One of these subsystems is the mobile communication modem that we discuss in the next section and which is the focus of this thesis.

1.1 MOBILE COMMUNICATION MODEMS

The mobile communication modem is responsible for handling communication of the phone with a base station in a cellular network. It enables data communications of the phone, transmitting and receiving voice and allows other services like Short Message Service (SMS). All lower level details, for example how the signals are physically transmitted over the air, are hidden from the rest of the system. The external interface of the modem exposes only the final user data and a small amount of control commands. Chapter 5 shows why the modem is often called *baseband*.

Handling processing tasks of the modem in a separate subsystem helps facilitating system design and, more important, allows meeting the deadlines in the processing of time critical tasks (real time system), which is needed often in communication systems, for example the system has to send at an exactly defined time. In addition to that, the processing system in the modem is optimized for the calculations occurring, leading to energy efficient operation.

Furthermore, the code of the modem is subject to strong regulations and needs to be certified by a responsible authority in order to allow operation of the device in some countries. By the separation into different subsystems, companies can change the main operating system without a new complex certification of the mobile communication component.

Modems usually implement a collection of cellular network standards, depending on the target market and development time. Since this work focus on recent modems, we only consider Long Term Evolution (LTE) capable modems. In addition to this, GPS is implemented together with the modem hardware in some cases.

It gets obvious that the modem is a quite interesting target for an attacker. All user communication passes through it, including data

and phone calls. With GPS being integrated, also precise location information is available. Besides, the interest in a smartphone modem as a basis for research on mobile communication networks is easy to see since all the communication standards are implemented here, mostly in software as we show in the analysis part.

1.2 AIM OF THIS THESIS

This thesis has two main aims:

- Gain a better understanding of basebands and their internal architecture
- Modify code running in the baseband:
  - See how well it is protected against attacks
  - Use the hardware for research

This thesis takes a look into the modem, analyzing its internals and opening it for modifications.

The first aim is a prerequisite for the second one. Without knowing anything about the system, modifying the code is impossible. The security aspect also relates closely to how the system is designed and, for an analysis in this direction, we need to know the internals of the architecture. However, it is also interesting on its own to know how the baseband works internally, for example to discover design flaws, finding improvement possibilities, as basis for other work on basebands or out of curiosity. We address this aim in Part ii.

The second aim was discussed extensively in the motivation before. We want to see how well the baseband is protected against attacks, make it possible to demonstrate attacks on deployed devices and open it for modifications in the context of research. This could, for example, simply be accessing the baseband's internal data and logging it for network analysis. More advanced scenarios are testing new algorithms and adding features to the modem. Part iii targets this aim and also presents example modifications.

1.3 BASEBANDS IN SMARTPHONES

As integration in smartphones increases to reduce costs and enable smaller devices, a single chip integrates many functions. Therefore, the baseband is usually bundled in one chip with the main processor and only radio frequency analog processing is done externally. Since this chip includes many important system functions, we refer to it as System on Chip (SoC). Smartphones with separate baseband chips barely exist anymore and because of this they are not considered here. Feature phones usually do not include LTE capabilities, even though this might change in the future as *Qualcomm* recently released a SoC with LTE intended for use in feature phones [29].

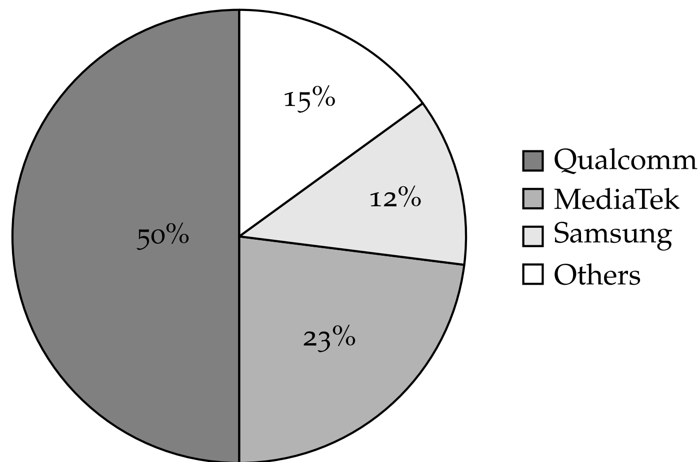


Figure 1: Baseband market share in first half of 2016, numbers from [21]

As a result, the manufacturer of the baseband is identical to the one of the main processor in the phone. This leads to only few different devices (manufacturers) of basebands present in the market.

1.3.1 *Market Share*

Only a few companies develop the modems found in smartphones.

Figure 1 shows the market share of basebands in the first half of 2016, including those not capable of LTE. We can see that only a few companies hold significant shares, namely *Qualcomm*, *MediaTek* and *Samsung*. If only LTE capable basebands are considered, *Qualcomm* gained an even higher share of 54% [21] and as *MediaTek* is mainly active in China, for Europe *Qualcomm's* leading position is even stronger.

This makes their devices an interesting target for attacks but also for research purposes as many already purchased devices can be used. As a consequence, we focus this work on basebands by *Qualcomm* solely.

2 RELATED WORK

We can group related work to this thesis in two categories. The first class analyzes modems and firmware modifications and is therefore related to the method we use to achieve the aims in this thesis. Work in the second group, on the other hand, is related to one of our key target applications, the collection of baseband internal data, approaching this goal by other methods.

2.1 BASEBAND REVERSE ENGINEERING AND MODIFICATION

Previous works in this direction pursued three different intentions:

- Gaining knowledge about the system
- Modifications or attacks having direct access to the phone
- Security aspects concerning scenarios with over-the-air vulnerabilities, in which an attacker has no direct access to the phone

In 2010, Weinmann gave a first insight into smartphone basebands with "All your baseband are belong to us" [39]. His work focused on the Global System for Mobile Communications (GSM) software stack running in the modem, where he discovered many security issues. These included classical memory corruptions and buffer overflows, due to unchecked length fields from GSM messages (his work focused on layer 3 of the protocol stack). He assumed this to be due to the code basis of the software stack originating in the 1990s when security threats were less ubiquitous and network elements could be considered trusted. However, hardware capable of running a GSM base station got much cheaper and is now also affordable to attackers, allowing to exploit the security issues. Two years later, in his paper "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks" [40], he describes the found vulnerabilities in detail and presents exploits to gain remote code execution.

Previous works analyzed various basebands, often revealing severe security issues.

In between, in 2011, Delugré [11] analyzed the operating system running on a Universal Mobile Telecommunications System (UMTS) stick with a *Qualcomm* baseband. Modifying this device was possible since the authenticity of the firmware was not checked and it contained diagnostic commands to read and write memory at arbitrary locations. He then also presented a functional debugger for the device.

Miras worked on unlocking carrier locks in smartphones. His talk [24] goes far beyond these main topics and discusses suitable reverse-

engineering and patching techniques for his target system, an *Apple iPhone 3G*.

With his next talk on basebands in 2013 [41], Weinmann published a good starting point for this work. He talked about the evolvements in basebands from *Qualcomm*, especially the new *Hexagon* Digital Signal Processor (DSP) used instead of a classical Advanced RISC Machines (ARM) core, which is the topic in Section 6.1. He also mentioned the risk that the baseband had a master role on the data bus in older chipsets and therefore could access all other peripherals. After mentioning security mitigations done by *Qualcomm* (stack canaries, non-executable stack and heap, kernel/user-mode separation, ...), he shows that he could still find many classical security issues, like stack overflows, in the basebands.

Recently, Qualcomm started to include attack mitigation techniques into their basebands.

The talk [13] shows research on another platform and targets *Samsung's Shannon* baseband. In this talk, the authors break down the firmware binary format, have a look at the running operating system and software stack, as well as search for vulnerabilities and exploit them. In addition, they give a detailed look at their reverse engineering process.

Table 1 gives an overview over the related work discussed.

2.1.1 *Contrast to this work*

The reader might now ask why another work in this direction is necessary after these. This is for several reasons:

Table 1: Overview of related work

Year	Title	Author	Target platform	Covered Topics
2010	All your baseband are belong to us [39]	R.P. Weinmann	Infineon/Intel Qualcomm (ARM)	over-the-air attacks (GSM)
2011	Reverse engineering a Qualcomm baseband [11]	G. Delugré	Qualcomm (ARM USB modem)	operating system, own debugger
2011	The baseband playground [24]	L. Miras	Apple iPhone 3G (Infineon baseband)	firmware patching, reverse-engineering
2012	Baseband Attacks: remote exploitation of memory corruptions in cellular protocol stacks [40]	R.P. Weinmann	Infineon/Intel Qualcomm (ARM)	over-the-air attacks (GSM)
2013	Baseband exploitation in 2013: Hexagon challenges [41]	R.P. Weinmann	Qualcomm (Hexagon based)	evolvement of basebands
2016	Breaking Band - reverse engineering and exploiting the shannon baseband [13]	N. Golde D. Komaromy	Samsung Shannon	reverse-engineering, over-the-air attacks

- The work is outdated: basebands changed significantly in the last years.
- None of the previous publications presented a general overview over the modem, including its hardware and software.
- They were considered with specific aspects rather than a general overview.
- Different scenarios were considered, e.g. over-the-air attacks, in contrast to our assumption of full access to the phone and firmware modifications done by "legitimate" users.
- We present a patching framework which allows to easily apply patches to firmwares.

In addition, we implement applications that are possible with the gained knowledge and developed framework.

2.2 INTERNAL DATA COLLECTION

There are also previous works with the intention of obtaining internal data from the modem. All the projects known to the author do not modify the baseband but instead leverage a diagnostic interface offered by the unmodified baseband firmware (available from the Android system under */dev/diag*).

Other projects to collect modem internal data exist. As they all use an existing diagnostic interface, they are limited to the data available there.

Qualcomm itself developed a tool called *QXDM* [30]. It runs on a PC connected to the phone, rather than directly on the phone, therefore, its use is limited to cases where an external computer is available and not feasible for data collection in everyday scenarios with users carrying the phone around. Since it is introduced directly by the manufacturer, it can make use of a large set of data options, possibly all that is available on the diagnostic interface. In the case of LTE, this includes layer 3 messages, measurements of the physical layer, current state of protocol state machines and much more. Please refer to the documentation of the tool for an extensive list.

With *Azenqos' AZQ Android*, a commercial tool making use of the interface exists [4]. It runs a user space application directly on the phone, together with a modified kernel. It offers similar functionality to *Qualcomm's QXDM*. Measurements, decoded layer 3 messages and other data are shown in real time to the user. It is also capable to log data and it provides analysis tools for common scenarios.

The *MobileInsight* project is slightly different [23]. Here, researchers try to make the data of the diagnostic interface available for further research in other projects. It therefore includes a documented Application Programming Interface (API), is extensible and can be used for research without any charges. Its functionality is limited to the message types currently implemented, again mainly layer 3 messages and some physical layer measurements are supported.

We can summarize that the three tools use the same data source and collect similar kinds of data. The main difference is where the tools collect the data and how they provide it to their users. This also leads to a common problem of all these: they are limited to the data offered by the diagnostic interface, if the desired data is not available at the interface, there is no way to get it from here. For example, this is the case for Channel State Information (CSI), which we extract in Section 10.6. This limitation is weaker for *Qualcomm's QXDM* as they have access to the firmware and can implement the desired messages. However, for some sensitive data like encryption keys *Qualcomm* might not be interested to expose them externally and with that introduce new security threads.

Firmware modifications are rather complex but allow to retrieve all internal data from the modem.

By gaining access to the modem's firmware, the approach presented here has the potential to extract all internal data used by the baseband. On the other hand, we have to find and reverse-engineer the relevant code section of the firmware, in order to know how the manufacturer implemented the target feature and how we can read the desired value. As a result, we suggest to first use the diagnostic interface for data collection and only start modifying the firmware for cases which cannot be handled by that.

Other projects intend to collect data of LTE networks with Software Defined Radio (SDR) implementations, for example in [8] the authors decode the control channel of LTE to monitor resource allocations of all users in a cell, also allowing to keep track of the load of the network. As these are based on specific hardware rather than on commercial off-the-shelf smartphones, they are targeting quite different use cases than this thesis and cannot satisfy the aim of research on cheap devices.

Please also note that data collection is only one aim of this thesis. All the presented projects are not able to modify the modem code. Therefore, they can only listen but not actively modify data, for example to see how a network reacts to manipulated values. For the same reason, they are not suitable for extending the modem with new features.

3 BACKGROUND: LTE BASICS

The objective of this chapter is to provide a minimum background knowledge of the LTE technology to understand this thesis, if LTE is already known this chapter can be skipped. It is far out of scope of this work to provide a full summary of the LTE standard. Therefore, we discuss only important parts of the technology and explain only few details which are used later in this work. For more details, we advise the reader to have a look at the LTE specifications or one of the numerous books and online explanations available (for example [34]).

3.1 LTE OVERVIEW

Figure 2 summarizes the protocol layering used in LTE. We primarily focus on the Physical Layer (PHY) and the Medium Access Control (MAC). However, as we can see in the figure, higher layers interact with these layers, mainly to be able to configure parameters of the lower layers by higher level messages. As a result, a basic knowledge of these layers is also required.

The main tasks of the different layers are (based on [44]):

LTE uses a layered protocol stack to break down the complex system into smaller parts.

PHYSICAL LAYER (PHY) modulation for air interface, power control, cell search, measurements (for example channel estimation)

MEDIUM ACCESS CONTROL (MAC) (de-)multiplexing from logical channels to transport channels, (priority based) scheduling of logical channel data, retransmission and error correction with Hybrid ARQ (HARQ)

RADIO LINK CONTROL (RLC) concatenation and segmentation of SDUs to fit in frames, duplicate detection, error correction with ARQ

PACKET DATA CONVERGENCE CONTROL (PDCP) in-order packet delivery (user and RRC control data), ciphering for user and control data, integrity protection for control data, Internet Protocol (IP) header compression

RADIO RESOURCE CONTROL (RRC) broadcast system information for NAS, paging, connection management, security (configuration, establishment, key management), maintenance of radio bearers

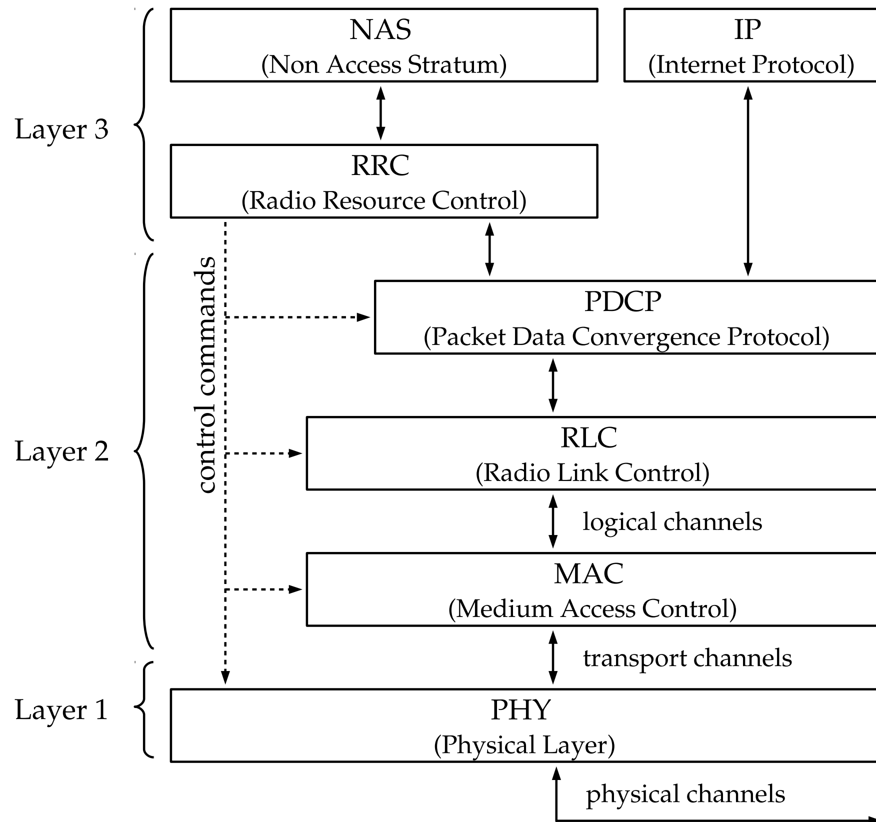


Figure 2: LTE protocol stack layers (adapted from: [44])

Non-Access Stratum (NAS) support of mobility, authentication, session management: establish and maintain IP connection for moving devices

Communication between the layers is organized in channels classifying the data and determining how it should be processed.

In this description, we already used the concept of logical, transport and physical channels between the layers. Logical channels in the interface between the RLC and MAC layers are used to distinguish between various kinds of traffic. They differentiate control and data traffic as well as whether it is dedicated for a single user or shared by multiple users. Transport channels in the communication of MAC and PHY layers on the other hand define how the data should be transmitted. At last, physical channels relate to the representation of the data in radio transmission and define which physical resources are used. For example, for the initial random access procedure (explained later in Section 3.3) dedicated physical transmission resources are reserved. Table 2 describes the available channels. For the transport and physical channels, the downlink channels (base station to User Equipment (UE) communication) are listed first, followed by the uplink channels (UE to base station). The "Maps to" column indicates over which lower level channel the data is transmitted.

3.2 PHY LAYER

In this section, we present the basics of the LTE PHY layer related to this work. LTE uses a different modulation scheme in the uplink than in the downlink. However, for this thesis mainly the downlink structure is relevant and we therefore focus on this direction here. Also we solely consider Frequency Division Duplexing (FDD) LTE networks.

3.2.1 *Orthogonal Frequency-Division Multiplexing (OFDM)*

The basis for all LTE communication is the Orthogonal Frequency-Division Multiplexing (OFDM) modulation. This technique splits the available carrier bandwidth into subcarriers. Each of these subcarriers is then modulated individually with data mapped to Quadrature Amplitude Modulation (QAM) or Quadrature Phase-Shift Keying (QPSK) symbols.

OFDM is the basis for the LTE downlink, dividing the available bandwidth into independent subcarriers.

Table 2: Channels in LTE (based on [35])

	Acronym	Channel Name	Usage	Maps to
Logical	BCCH	Broadcast Control Channel	Broadcast of System Information Blocks (SIB) and Master Information Block (MIB)	BCH/DL-SCH
Logical	PCCH	Paging Control Channel	Paging of UEs	PCH
Logical	CCCH	Common Control Channel	Control information for multiple UEs	DL-SCH
Logical	DCCH	Dedicated Control Channel	Control information for a particular UE	DL-SCH
Logical	DTCH	Dedicated Traffic Channel	Traffic (user-plane) for a particular UE	DL-SCH
Logical	MCCH	Multicast Control Channel	Multicast control information	MCH/DL-SCH
Logical	MTCH	Multicast Traffic Channel	Multicast data (user-plane)	MCH/DL-SCH
Transport	BCH	Broadcast Channel	Broadcast of MIB	PBCH
Transport	PCH	Paging Channel	Paging of UEs	PDSCH
Transport	DL-SCH	Downlink Shared Channel	Data transfer, SIB	PDSCH
Transport	MCH	Multicast Channel	Configuration of multicast transmissions	PMCH
Transport	RACH	Random Access Channel	Initial access to network through RACH	PRACH
Transport	UL-SCH	Uplink Shared Channel	Data transfer	PUSCH
Physical	PBCH	Physical Broadcast Channel	Broadcast of MIB	
Physical	PDCCH	Physical Downlink Control Ch.	Control channel: scheduling, UL grant	
Physical	PDSCH	Physical Downlink Shared Ch.	Data transfer, SIB, paging	
Physical	PMCH	Physical Multicast Channel	Multicast	
Physical	PHICH	Physical Hybrid ARQ Indicator	HARQ ACK/NACK status	
Physical	PCFICH	Physical Control Format Indicator	Control channels metadata	
Physical	PUCCH	Physical Uplink Control Ch.	Signaling (scheduling requests, HARQ)	
Physical	PUSCH	Physical Uplink shared Ch.	Data transfer	
Physical	PRACH	Physical Random Access Ch.	Random access	

With the subcarrier values, we define the desired components of the signal at the subcarrier's frequencies. Therefore, the subcarrier values are equal to the representation of the desired signal in the frequency domain. As a result, the OFDM modulation can be implemented by calculating an Inverse Fast Fourier Transform (IFFT) on the subcarrier values to get the time domain data to sent. Similarly, the receiver can calculate a Fast Fourier Transform (FFT) on the received time data to find back the modulated subcarrier symbols.

Please note that LTE additionally applies scrambling to the user data stream, before mapping it to symbols for the subcarriers, to deal with burst errors. If Multiple Input Multiple Output (MIMO) is active, also some precoding is applied to the symbols and a mapping to the different antenna ports is performed.

3.2.2 *LTE resource grid*

The OFDM modulation divides the available bandwidth into subcarriers. Table 3 lists the bandwidth options specified for LTE, together with the used number of subcarriers and resource blocks (which we define later). In recent LTE releases, a network can combine multiple of these carriers in a single cell (carrier aggregation, up to 5 carriers from LTE Release 10 onwards), thus increasing the amount of available resources.

OFDM together with a fixed structure in time divide the available transmission resources into a regular grid.

In the time domain, LTE uses a regular structure with fixed durations of each element. The biggest unit is a frame. Each frame consist of ten subframes which have two slots each. These again contain seven OFDM symbols, in case the network is configured for extended cyclic prefix only six symbols fit into one slot. Table 4 summarizes this structure.

By this means, the available resources for transmission are divided in frequency (OFDM subcarriers) and time (slots in frames). Each slot and subcarrier pair describes a distinct resource which can be used to send data. LTE leverages this to assign these blocks for different purposes: the physical channels introduced before, synchronization signals and reference signals which we explain in Section 3.2.3. The

Table 3: LTE bandwidth options

Bandwidth	Available subcarriers	Resource blocks
1.4 MHz	72	6
3 MHz	180	15
5 MHz	300	25
10 MHz	600	50
15 MHz	900	75
20 MHz	1200	100

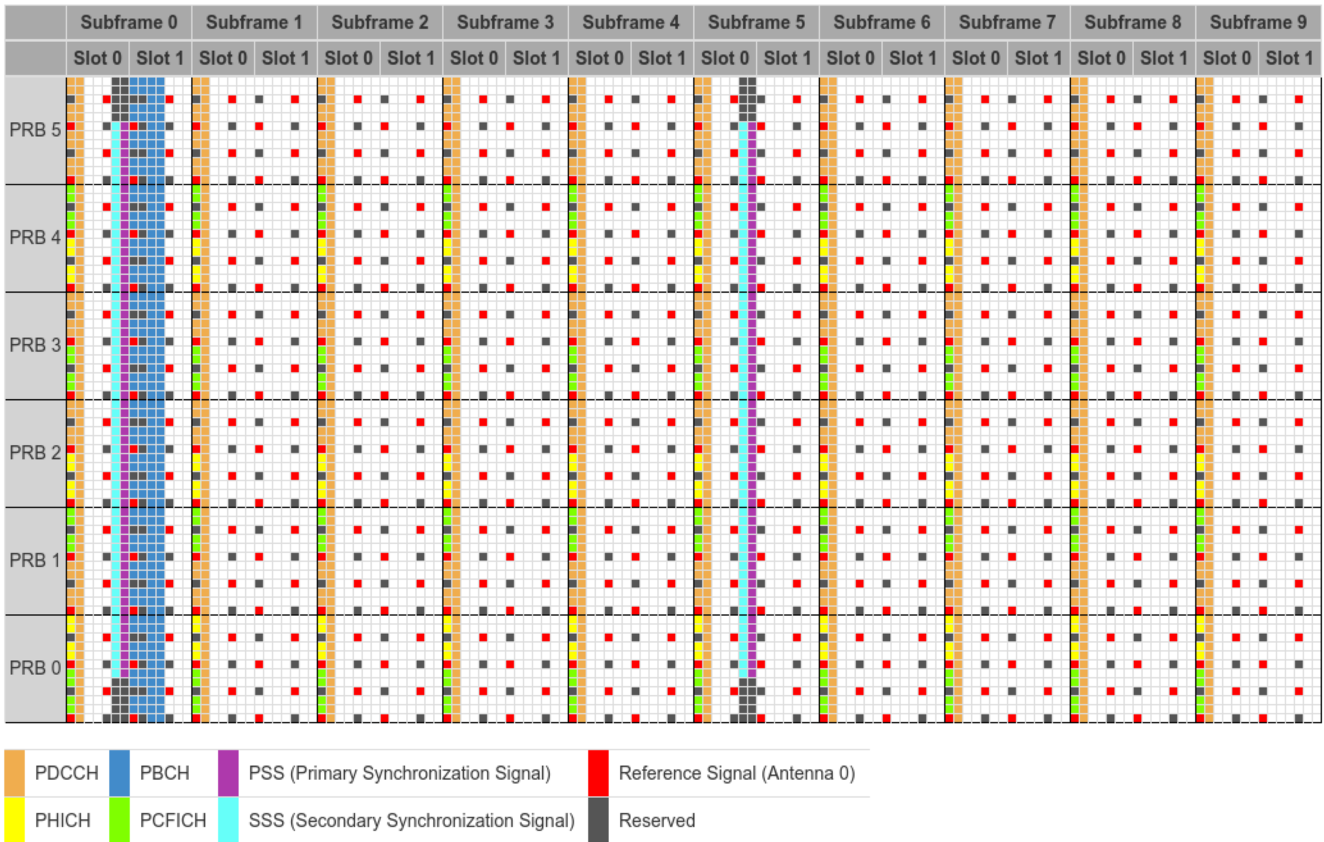


Figure 3: LTE downlink resource grid for 1.4 MHz, two transmit antennas (based on [12])

blocks of the PDSCH channel are further assigned to different users. To reduce signaling overhead, a Resource Block (RB) groups 12 adjacent subcarriers and one slot in time and distribution to different users is done in this granularity. Figure 3 illustrates the resource structure in a grid of time and frequency, showing an example of the static channel allocations for a 1.4 MHz LTE FDD network with two sending antennas (small bandwidth chosen for better overview). White blocks are available for PDSCH channel allocations.

The resource grid allows flexible and fast allocations depending on the current demands of active users.

With this resource allocation scheme, LTE is flexible in sharing the available bandwidth between users, depending on their traffic demands and signal quality, to increase the total throughput over the network while keeping fairness between the users.

Table 4: LTE frame structure

Unit	Duration	Consists of
Frame	10 ms	10 Subframes
Subframe	1 ms	2 Slots
Slot	0.5 ms	6/7 OFDM symbols

3.2.3 *Channel fading*

The physical communication channel between the UE and the base station, also called Evolved Node B (eNodeB), imposes attenuation and phase change to the signal. For typical LTE channels, this is not a constant factor but the channel's influence on the signal highly depends on the frequency, for example, caused by multipath propagation. In addition, due to mobility of the users and changes in the environment, the channel strongly varies over time. These effects are called frequency selective fading. At a fixed point in time, the channel H in frequency domain can be described as the (complex) quotient between the received signal Y and the transmitted symbol X:

$$H(f) = \frac{Y(f)}{X(f)}$$

For a small enough range of f (bandwidth), we can assume H to be constant (only flat fading over time). In particular, this is the case for an OFDM subcarrier of LTE with a bandwidth of 15 kHz. As a result, the channels influence can be described by a single complex value for each subcarrier, called $H_s(k)$ in the following with k being the index of the subcarrier.

Typical LTE channels experience strong frequency selective fading. For a single subcarrier, the fading is flat enough to be characterized by only a single complex coefficient.

If these are known to the UE, it can compensate the channel's influence, by reversing the channel equation in the discrete case, to get back the transmitted symbol from the received signal (Y_s and X_s are the discrete equivalents to Y and X):

$$X_s(k) = \frac{Y_s(k)}{H_s(k)}$$

In addition to using the channel characteristic for decoding, the UE can decide which subcarriers have the best signal strength and feed this information back to the base station. This uses the data to optimize the allocation of RBs, trying to use the currently best subcarriers for each user.

3.2.3.1 *Multiple Input Multiple Output (MIMO)*

LTE uses multiple sending and receiving antennas to increase the data rate or signal quality depending on the selected mode and current channel conditions, called MIMO. The propagation properties between each sending antenna and receiving antenna might be different. Therefore, a different channel exists between each of these pairs, which can be described in the form explained before. This leads to a set of channel descriptions, alternatively represented by a channel matrix containing the same information.

For spatial multiplexing, the sending antennas transmit different signals. If now at least as many receiving antennas as sending antennas are used, both sending streams can be recovered by the received

signals and their differences, since the contribution of each sending stream to each signal is known by the channel matrix. However, if the channel coefficients for the receiving antennas are too similar, it is no longer possible to separate different transmit streams as all receiving antennas report the same signal. In this case, transmit diversity, which sends the same signal on both sending antennas, can be used instead to improve signal quality but no longer profit from increased data rates.

3.2.3.2 *Channel estimation*

Until now, we simply assumed that the UE knows the channel characteristic $H_s(k)$. In this section, we now explain how this is achieved. The resource grid in Figure 3 shows that resources in a regular pattern in frequency are occupied by reference signals (red). Fixed symbols, known to the receiver by the LTE specification, are transmitted here, also called pilot symbols. The UE is able to compute the channel coefficients at the reference symbols by their definition equation, as it knows the received signal as well as the transmitted signal. For the subcarriers in between the known pilots, the measured channel characteristic can be interpolated or simply the closest neighbor can be taken, leading to an estimation of the whole channel at the time of the used reference signals.

Regular transmission of pilot symbols allows estimation of the channels between different antenna pairs.

Since the channel is fading over time, the estimation is only valid for a short period of time and has to be repeated regularly to always have a recent representation of the channel conditions. Because of that, the reference signals are also repeated in time leading to a regular grid of resources used for channel estimation signals.

In the MIMO case, all channels between antenna pairs need to be estimated individually. To allow this, only one transmit antenna sends a pilot signal at a resource while the other antennas remain quiet on this block. A receiver can then determine all channels from this transmit antenna to its receiving antennas like in the single antenna case, since the pilot signal is not disturbed by other transmissions. The resource grid in Figure 3 is for a base station with two transmit antennas but only the view of one antenna. The marked reference signal blocks contain the reference signals of the first antenna, the ones of the second antenna are transmitted in a similar grid which gets visible by reserved resources in the perspective of the first antenna.

3.3 MAC LAYER

The LTE MAC layer is located between the RLC and the PHY layers. As a consequence, all messages from the higher layers pass through it, including user data as well as configuration messages. In Section 3.4, we show that encryption is applied in the PDCP layer. Therefore,

data seen in the MAC layer from higher layers than itself might be encrypted, depending on the current UE configuration.

The MAC layer is responsible for multiplexing logical channels to transport channels in the downlink and demultiplexing the transport channels in the uplink. It takes care of scheduling the transmission of content from logical channels, based on the priority of the data. For this, it also controls the uplink scheduling of the physical layer, for example, by requesting it to send a Scheduling Request (SR) to the network in order to get more uplink resources allocated.

A mechanism called HARQ realizes automatic retransmission and error correction. Additionally, the MAC layer generates reports to inform the network about the current amount of data in the UEs buffers (Buffer Status Report (BSR)) and the currently used transmission power compared to the available power (Power Headroom Report (PHR)). In the next section, we explain the MAC mechanism to get a first uplink grant for data transmission.

3.3.1 *Random access procedure*

When a UE wants first to connect to a network, for example after it was turned on, it does not have the needed uplink resources allocated to transmit data to the base station and register in the network. Similarly, no allocations are available when the UE is in idle state because it did not send data in a long time. To solve this, LTE includes a random access procedure.

The random access procedure allows devices to request initial grants for data transmissions when they have no resources assigned.

Devices without an active uplink grant are allowed to transmit a special preamble message on the RACH/PRACH channel, which we will call *Msg1* from now on. The UE can choose randomly between 64 available preambles, each of them orthogonal to the others such that the base station can receive multiple preambles at the same time. Note that conflicts can happen when two UEs choose the same preamble at the same time. This is resolved later in the random access process.

If the base station successfully received this message, it will reply with a Random Access Response (RAR) on the DL-SCH channel, *Msg2*. The target device of this message is addressed by a RA-RNTI (Radio Network Temporary Identifier) value which is derived from the timeslot in which the preamble was sent. The message includes data for each sent preamble at this time: a temporary C-RNTI value which will be used to identify the UE in the next steps, a timing advance value to compensate for the signal propagation delay between the device and the base station as well as an uplink resource grant to allow communication on the UL-SCH channel.

The UE reacts with an RRC connection request message (*Msg3*) including the assigned C-RNTI value to identify itself. It gives the reason for connection establishment together with a new identity value,

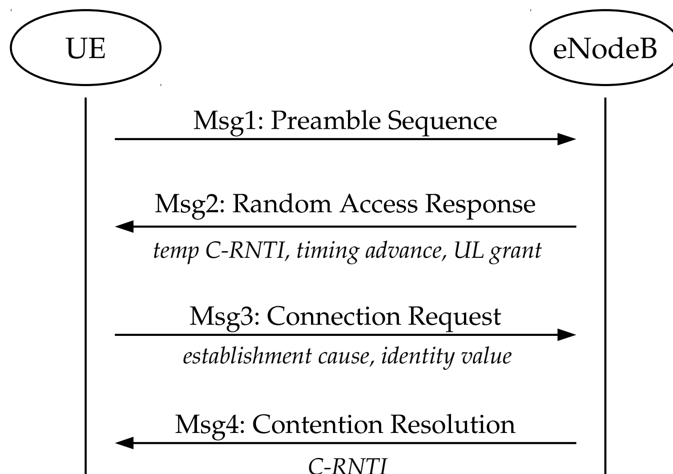


Figure 4: LTE RACH procedure (adapted from: [5])

which can be either chosen randomly or reused from a previous connection.

This is required to resolve contention between two devices. Let us consider that two devices chose the same preamble at the same time in the first step. Until this moment, the base station and UEs did not notice that a conflict happened. Now, the base station will at most successfully receive one of the Msg3 due to interference of the signals and reply only to this one with the last message of the procedure (*Msg4*). As this includes the identity chosen by the UE, all other devices know that their RACH procedure failed and will try again later, only the addressed device will continue communication with a new C-RNTI value assigned in this message.

As transmissions in this procedure are random, conflicts can happen and need to be resolved.

Figure 4 depicts the whole message exchange process. Note that a slightly different contention free version of this procedure exists which is used, for example, in handover cases. Here, the preamble to select is preassigned to the UEs, hence no conflicts with identically chosen Msg1 can occur.

3.4 SECURITY

LTE implements security functionality in the PDCP and NAS layers. The PDCP layer is responsible for security between a UE and a base station (eNodeB). It provides confidential message delivery through encryption and integrity protection by adding Message Authentication Codes (MACs) for data from the RRC layer in the control plane. For user data, meaning IP traffic, it only provides confidentiality and no integrity protection. Different keys are used for the two planes.

LTE includes encryption and integrity protection for control and user data.

The NAS layer in addition ensures integrity and confidentiality of its messages independent of the PDCP layer. It provides this security between a UE and the Mobility Management Entity (MME), which is

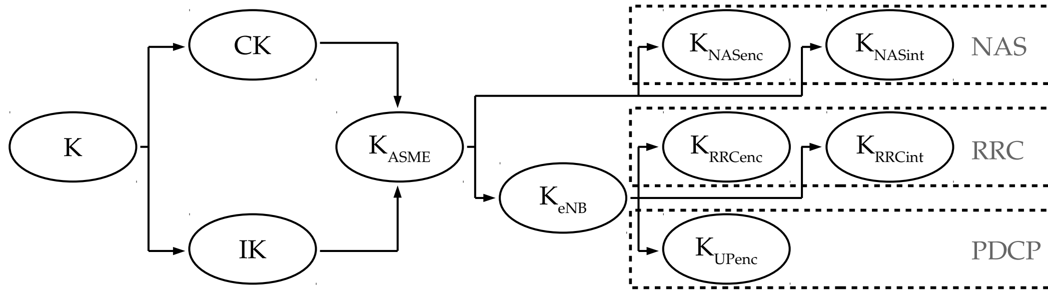


Figure 5: LTE key derivation scheme (adapted from: [9]). Keys for cryptographic operations are derived from a pre-shared master key (K).

a component deeper in the LTE system architecture and the target of NAS communication.

To allow these, LTE provides mutual authentication of the UE and network, as well as mechanisms for key negotiation. Figure 5 depicts the key derivation scheme. Keys are derived from a pre-shared master key (K), stored in the Subscriber Identity Module (SIM) and at the network provider. This is used to generate a cipher (CK) and an integrity key (IK). From these two, the local master key (K_{ASME}) is derived which allows generation of NAS keys (K_{NASenc}, K_{NASint}) and a base key for the current eNodeB (K_{eNB}). The keys for PDCP operation (K_{RRCenc}, K_{RRCint}, K_{UPenc}) are then generated from this. Note that in the figure we assigned K_{RRCenc}, K_{RRCint} to the RRC layer as the PDCP layer uses them for messages of this layer.

A range of algorithms is supported, including "null" algorithms for emergency connections.

LTE supports different types of encryption algorithms and integrity algorithms, called EPS Encryption Algorithm (EEA) respectively EPS Integrity Algorithm (EIA). Table 5 lists the standardized options. Which algorithm to use is determined by RRC signaling and the used cryptography keys depend on this selection. Four values form the inputs to cryptographic operations, additionally to the key and the actual message: count (sequence number), bearer ID, direction (uplink or downlink) and the message length. Note that also null cipher and integrity algorithms are defined. These apply no encryption and produce a MAC of only zeros. Their purpose is to allow emergency connections even when no SIM is available and thereby no keys can be derived, a network should never be configured to use them as algorithms for usual connections.

Table 5: LTE cryptography algorithms

Algorithm	**Based on**
EEA0 / EIA0	none / null algorithms
EEA1 / EIA1	SNOW 3G
EEA2 / EIA2	AES
EEA3 / EIA3	ZUC

4 TARGET HARDWARE

We first intended to carry out our research on the *Google Nexus 5* smartphone. However, as the baseband's firmware of this device is authenticated, which is the topic of Section 7.2, it is not possible to run modified firmwares on the modem of this phone, without exploiting vulnerabilities of the security mechanisms.

As a consequence, we decided to shift towards the *Asus PadFone Infinity 2 (A86)* , since *Asus* does not enable signature checks in some phones (details in Section 7.2.1) and this model uses the exactly same SoC as *Google's* Nexus device, a *Snapdragon 800* or in *Qualcomm's* naming scheme an *MSM*8974. Table 6 gives the main features of the SoC. Further specifications of the two smartphones are also similar, as Table 7 shows.

We use an Asus PadFone Infinity 2 as primary target in this thesis. Most results are also applicable to other devices.

Even though we largely focus this thesis on the used target, many of the results and, in particular, the developed patching framework can also be applied to other phones, possibly based on other SoCs. This is caused by very similar designs of recent *Qualcomm* modems and simplified by all baseband firmwares originating from the same source code base. We implemented scripts assisting in porting to other phones, or just other firmware versions for the same phone. They are presented in Section 9.3.

Please note that, at least until a vulnerability is found to bypass the authentication, modifications are only possible on devices with

Table 6: MSM8974 (Snapdragon 800) specifications (source: [31])

CPU	4x Qualcomm Krait 400
L1 (CPU) Cache	2x16 kB (instructions + data) per Core
L2 (CPU) Cache	2048 kB per Core
GPU	Qualcomm Adreno 330
Technology	28 nm
DSP	Hexagon audio/sensor/user DSP
Modem	*LTE Category 4*
	2x2 MIMO
	DL: 150 Mbps, 64QAM, 2x10 MHz (CA)
	UL: 50 Mbps, 16QAM, 1x20 MHz
	Hexagon (QDSP6v5) based
TCM (modem)	256 kB
Further features	QuickCharge, Bluetooth, 802.11n/ac,
	image processor (camera), video core,
	NFC, USB 3.0, GPS/GLONASS, ...

disabled authentication or on devices which use insecure debug keys for signing. This is the biggest limitation currently, decreasing the length of the list of potential target devices significantly. We discuss this issue in Chapter 11.

Table 7: Target devices specifications (sources: [3, 22])

	Asus PadFone Infinity 2 (A86)	**Google Nexus 5**
SoC	MSM8974	MSM8974
OS (newest)	Android 5.0	Android 6.0.1
RAM	2 GB	2 GB
Storage	16/32 GB	16/32 GB
Screen	5 " (1920x1080)	4.95 " (1920x1080)
Release date	October 2013	October 2013

Part II

ANALYSIS OF QUALCOMM MODEMS

We take a look at the internals of *Qualcomm* basebands in this part. After a first focus on the hardware architecture, its integration in the overall smartphone system and a detailed view on the used processor, we review the modem's software in detail.

5 SYSTEM ARCHITECTURE

To understand the function of the modem, in particular the digital baseband part of it, in the following sections we present the architecture, starting with a coarse overview up to a detailed view of the baseband internals.

5.1 OVERALL SYSTEM

Since all recent *Qualcomm* chips have a similar structure, at least regarding the modem, we have an exemplary look at the *Snapdragon 800 (MSM8974)* chip. Figure 6 shows the SoCs architecture, with the modem in the lower right corner. This view demonstrates the connection between the modem and the main processor: shared memory. The modem has direct access to the system's main memory. To communicate with the HLOS, it places data in a common section of this memory. Communication in the other direction is done in the same fashion. As a consequence, large amounts of data can be passed easily between the two entities. We give more details on this process, especially on the protocols used for data exchange, in Section 7.5.

In modern smartphones, the modem is part of the main SoC. It communicates with other parts of the system through shared memory.

A problem of this direct access to the main memory of the modem is security. The modem could in theory access all user data, for example variables of running applications or data from the Android kernel. Therefore, a malicious modem would lead to a complete system compromise. Section 6.2.1 presents the mechanism used to prevent this.

Note that communication through shared memory is only used in case of a modem integrated in a SoC. In case of a discrete modem chip or a USB stick modem, some kind of high-speed serial link, for example USB, replaces this link.

5.2 MODEM ARCHITECTURE

Figure 7 depicts the modem system.We obtained it by simplifying various smartphone schematics and it is only a coarse overview, omitting details like amplifiers or baseband filters.

Since mobile communication networks are defined on a wide range of frequency bands, the system includes multiple antennas, each optimized for a certain range of frequencies. Also, the system needs multiple antennas on the same frequency band to support MIMO transmission modes. These are connected to switches which connect them to filters for the different frequency bands supported by the antenna. After passing through the filters, received signals are fed to

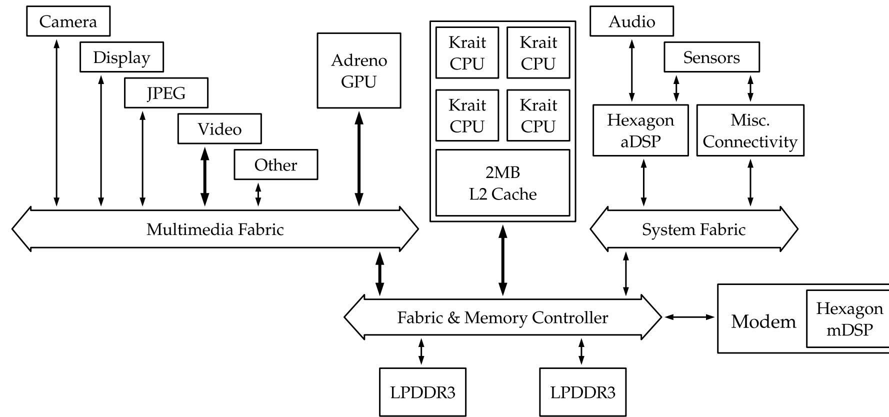


Figure 6: Snapdragon 800 system overview (adapted from: [10])

an input port of the transceiver chip. Analogously, signals to transmit take the inverse way, coming from an output port of this chip. These ports are internally connected over a switch to the receiver (or transmitter) circuitry. Note that the number of antennas and filters shown in the diagram is chosen arbitrarily and will be different in a real system. To overcome the limitation of the maximal possible supported bands by the number of ports the transceiver chips offers, system designers can add additional switches in between. However, this introduces some signal attenuation.

The modem consists of an analog part responsible for converting between RF and baseband signals and a digital part for processing of the baseband signals, with algorithms implemented in software.

The transceiver converts the Radio Frequency (RF) signal down to the baseband with Inphase and Quadrature (IQ) components. For the transmitting case, it converts up from the baseband IQ signal to RF. Now this signal contains only frequencies lower than the used maximum channel bandwidth and can be sampled using an Analog-to-Digital Converter (ADC) in the digital baseband block. All further processing is done on the digital data using a processor. For the transmit path, this system generates the signal and it is then converted to an analog signal for the transceiver by a Digital-to-Analog Converter (DAC). Since this is a straightforward transceiver system found in common literature, we do not discuss it in further detail here.

Note that only the digital processing (so the baseband block) is part of the SoC, all analog processing is done by external components. This is also the reason why it is common to refer to the modem subsystem as baseband: the digital system we are talking about is only the part of the modem responsible for processing and generating the baseband signals. In the rest of this thesis, we are only concerned with this part of the modem.

In addition to the signal paths, the transceiver and the switches need control signals. These are also generated by the baseband block. On its other side, this block interfaces with the rest of the SoCs system as seen in the previous chapter.

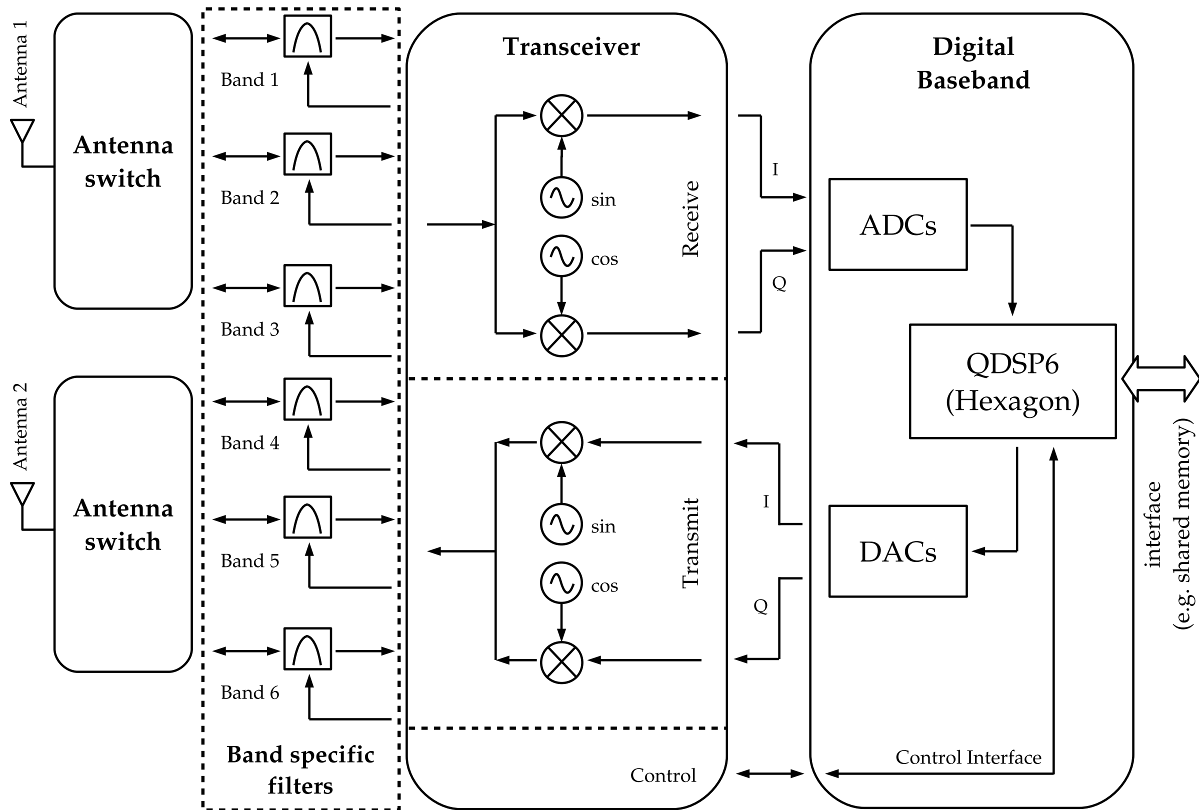


Figure 7: Modem architecture overview

5.3 DIGITAL BASEBAND

As described before, the digital baseband block in Figure 7 handles decoding of the sampled signal and generates a stream of samples for transmitting. This means it demodulates the signal and decodes its meaning according to the implemented communication technology, handles commands, generates responses and so on. For transmitting, it encodes its messages and modulates them according to the communication standard. In other words, it implements the protocol stack of this technology or, more generally speaking, all stacks of the technologies it supports.

Baseband signals are processed in the digital domain, on a DSP also running the higher layers of the communication technologies.

Since these stacks are extensive, a processor running some code is used. In case of the *Qualcomm* modems considered here, a special DSP developed by the company itself is integrated. It is called *QDSP6 (Hexagon)* and we review it in detail in Section 6.1. All code, including control intensive protocol parts, runs on this DSP in addition to the actual signal processing. Older basebands, or those from other vendors, might use an additional general purpose processor (often an ARM core) or a combination of this and a DSP.

6 HARDWARE COMPONENTS

In this chapter, we review the hardware components used in the baseband. After describing the included processor with its specialities, we look at access protection units and a dedicated on-chip memory.

6.1 HEXAGON PROCESSOR

Qualcomm uses a self-developed processor called *QDSP6 (Hexagon)* for code execution in their modems. This processor is a DSP and therefore optimized for the calculations occurring in signal processing, for example digital filters or the calculation of an FFT. It can perform this kind of operations faster and more energy efficient than a general purpose processor.

A second look at Figure 6 reveals that the same processor type is re-used for audio and sensor data processing. Actually, *Qualcomm* released a Software Development Kit (SDK) [28] to enable developers to create their own code running on the DSP for processing of sensor data or using it as a general hardware accelerator. Even though this SDK targets another DSP instance in the SoC, its tools can be used for the processor of the baseband. As a consequence, a full toolchain with compiler, assembler, linker and other relevant utilities for processing binary code files, including a disassembler, are freely available. Note that, to deploy own code on the basebands DSP, additional steps not present in the SDK are necessary, for example to pack the final firmware binary into the correct image format. We target these steps in Chapter 9.

Qualcomm released documentation and an SDK for the used Hexagon DSP. Also plugins for IDA exist.

Qualcomm's release also includes the documentation of the utilities and, most important, the processor core [33]. Unfortunately, this documentation is incomplete and does not contain all instructions, for example the ones solely intended to be used by the Operating System (OS) are not documented. However, in the process of reverse-engineering, we also see these instructions and have to understand their behavior. For the *IDA Pro* reverse-engineering tool from *Hex-Rays* at least two *Hexagon* processor module plugins exist, both of them still containing some problems. The one available at [17] was found more stable for code analysis and the Python disassembler of the one at [25] is used by the patching framework in Chapter 9.

After an overview of the different versions of the *Hexagon* DSP, in the remaining sections of this chapter, we show the details of the processors architecture. If not explicitly denoted otherwise, the information is gathered from [10, 33].

6.1.1 *Versions*

The *QDSP6 (Hexagon)* family is the latest revision of *Qualcomm's* DSPs. It comes in different versions, released starting from 2006. Table 8 summarizes the main features added with each of the versions. This information is collected from the programmer's reference manuals [32, 33] and from Weinmann's talk [41] as these documents are not longer available in the Internet for all older processor versions.

The smartphones used in this work contain a *MSM*8974 SoC which uses a *QDSP6V55* for the baseband.

6.1.2 *Instruction packets*

Hexagon is a VLIW architecture. Thus, the processor's instructions are grouped into packets which execute in parallel.

The *Hexagon* DSP has four execution units which accept data. Each of them gets a separate instruction from the system's code memory, meaning the action they perform is explicitly encoded in the firmware by the compiler. Therefore, instructions for the *Hexagon* processor are grouped into packets which it executes in parallel in the same processing cycle. Due to large amount of instruction data needed for that (up to four 32 bit words for a single processing cycle), this concept is called Very Large Instruction Word (VLIW). Figure 8 shows the execution units.

Table 8: QDSP6 versions

Version	Release	New features
v1	2006	first release, mainly as a multimedia DSP
v2	2007	lower power consumption higher efficiency for control code improved support for modem applications
v3	2009	new instructions (pause, vector instruction, undocumented ones)
v4	2010	virtualization instructions support for SDR debug & trace enhanced
v5	2012	dynamic multi-threading (DMT) floating point support enhanced data cache prefetch
v55	2013	cycle count registers (cycles since last reset) new vector instruction
v60	2016	dual-cluster micro-architecture simultaneous multi-threading (SMT)
v61	2016	new instructions stack security features packet counting

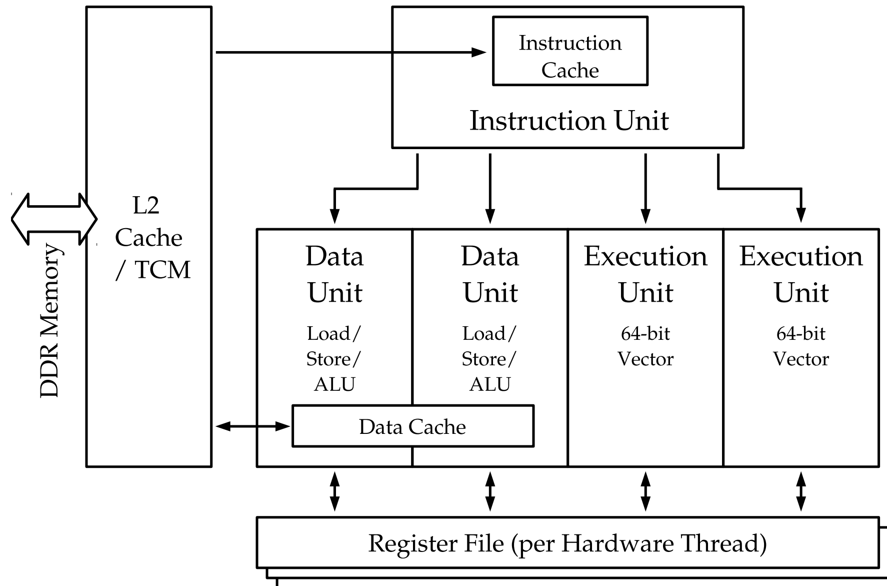


Figure 8: Hexagon execution units (adapted from: [10])

We can see that the units are not equal but target different types of instructions (here called data and execution units). Thus, a packet cannot contain four times the same instruction but the instruction composition has to follow the abilities of the available execution units. For the exact constraints and possible combinations, please refer to *Qualcomm's* programmer's reference manual for the target DSP version, for example [33]. In the normal case of writing code in a higher level language (for example in C), it is the compiler's task to group instructions into packets. However, when reverse-engineering or patching existing code, we have to analyze packets by hand and need to generate valid ones.

Heterogeneous execution units introduce constraints on the instructions in one packet.

Note that the instruction encoding only contains flags to indicate whether or not another instruction follows in the same packet but not to indicate the start of a packet. These are only known implicitly. This can lead to problems in reverse-engineering and patching the code, when the start of a function is detected in the middle of a packet or a jump into a packet is generated. Luckily, it is rather easy to deal with this but existing tools developed for other platforms might have problems.

6.1.3 *Constant extenders*

Instructions can usually only include immediate values smaller than a whole word, as they are only one word long and need space to encode the instruction itself. Since jumps or computations often need constants of the full 32 bits, this is a strong limitation and leads to many instructions just to load a constant value. To overcome this, *Hexagon* supports *constant extenders (immext)*. These precede an in-

Constant extenders allow efficient encoding of large immediate values.

struction and carry the remaining bits of a constant value needed there. As a consequence, the processor can load constants with the size of a full word directly and jumps over the whole address space are possible (memory addresses are also 32 bit long) in a single instruction. The disadvantage of these extenders is that they also consume an instruction slot and therefore at least one execution unit has to be left unused within the same packet. As in many cases the compiler will not find an instruction for all units in a processing cycle, this reduces performance only slightly, likely less than the advantage of the faster constant loading.

6.1.4 *Hardware Threads*

Qualcomm uses hardware threads to simplify the architecture. For the programmer, the DSP looks similar to a multi-core system.

The QDSP6 features multi-thread support in hardware. Version 4, for example, includes three hardware threads. Instructions from these are executed by Interleaved Multi-Threading (IMT) in a round-robin fashion, always in the same order. For the programmer, the different threads look like independent processor cores running their own codes. If more software threads are used than hardware threads are available, they are multiplexed on these virtual cores. On the other hand, when less threads are used only a fraction of the total DSPs performance can be achieved.

The number of hardware threads matches the number of execution pipeline stages. With this trick, whenever a new instruction is fetched, all previous instructions of the same thread are finished completely as shown in Table 9. Each row represents a pipeline stage and each column an execution cycle, given by the thread for which a new instruction is fetched. The entries represent first the thread and then the instruction number in this thread. Note that “Start” in the table means not the first processing cycle but the start of the observation, being at a time when each thread was already active once. At each point in time, each pipeline step works on an instruction of another thread which means no dependencies exist between them.

As a consequence, *Qualcomm* can largely simplify the internals of the processor: no pipeline stalls can occur and no result forwarding, branch-prediction or other techniques for performant pipelined pro-

Table 9: Execution of hardware threads in the QDSP6 pipeline

		Instruction fetch				
		Start	Thread 1	Thread 2	Thread 3	Thread 1
Stage	1	T3 - I1	T1 - I2	T2 - I2	T3 - I2	T1 - I3
	2	T2 - I1	T3 - I1	T1 - I2	T2 - I2	T3 - I2
	3	T1 - I1	T2 - I1	T3 - I1	T1 - I2	T2 - I2

cessing are needed. Nevertheless, other issues are introduced, for example a separate register file is needed for each thread.

Starting from version 5, the *Hexagon* DSP supports an additional mode called Dynamic Multi-Threading (DMT). With this mechanism, hardware threads can be removed from the scheduler, for example because they are not used, on cache misses or when waiting for interrupts. The scheduler can then skip these threads and instead issue a packet of the next thread. Note that a new packet for a thread can still only be issued when the previous one finished completly. However, some instructions do not need the full pipeline depth but can finish in two cycles. Thus, with DMT and at least one inactive thread, a new packet can be scheduled earlier after a packet with only such fast instructions and as a result the overall performance improves slightly for some applications. [14]

6.1.5 *Other specialities*

The *Hexagon* DSP includes a few more specialities. *Compound* and *Duplex* instructions encode multiple instructions into a single word to reduce code size. Hardware loops realize loops with less overhead than software only loops (up to one level nesting). Since a detailed discussion of all these is not the topic of this thesis, please refer to the reference manual for more details on these and other additional features.

6.2 Other related components

Besides the DSP, the baseband contains ADCs to sample the received analog signals, respectively DACs to generate baseband signals to transmit. Additional hardware components, presented below, complete the system.

6.2.1 *Access Protection Units*

Qualcomm includes Access Protection Units (XPUs) into their chips to implement access rights depending on the requesting system component. They are placed between the peripheral components and the interconnect bus. Thus, each peripheral unit has its own XPU. Our target SoC *MSM8974*, for example, has at least 86 XPUs included, maybe more which we did not discover. We gained the knowledge about the units presented here from the experiments in Chapter 11.

XPUs ensure that certain addresses in the SoC can only be accessed by authorized subsystems.

After the initial configuration, such a XPU unit compares every access to it against a set of rules and blocks if the access is not permitted. In addition, an interrupt can be triggered on right violations, allowing to take further measures, from reporting or logging the incident up to restarting the complete system.

Rules define individual permissions for each system component.

These rules, also referred to as Resource Group (RG) in *Qualcomm's* strings in the firmware, use two mechanism to define rights. First, permissions (read/write) are defined for components of the SoC, for example the application processor or the modem DSP. Second, rights are given to contexts (roles) of the running code, here are *non-secure*, *secure* (*TrustZone*) or *modem* possible. Only when a permission is given by both criteria, the access is granted. Each rule has an owner who is allowed to change the rule later, which is also one of the roles of the accessing code. No one else can modify, deactivate or read the rule after it was first given to an owner. As a consequence, even *TrustZone* code, for example, cannot simply deactivate rules of the modem, although it configured them at startup but then passed the ownership to the modem.

Most XPUs allow to specify individual permissions in rules for each component in the SoC (up to 32) for the first mechanism. However, some use a simplified hardware which is only capable to enforce different permissions for one component and all the remaining ones together.

In addition to this, different types of XPUs exist:

Register Protection Unit (RPU) These are used to protect registers of security critical peripherals. Each rule applies permissions to a predefined set of registers. An example is the protection of accesses to cryptography peripherals, such that another subsystem cannot tamper with the operation of the unit.

Area Protection Unit (APU) The second type is similar to the first one but is intended for larger address areas, for example to protect the Read-Only Memory (ROM) containing the Primary Boot Loader (PBL) of the modem, instead of only single registers.

Memory Protection Unit (MPU) Rules of the last type do not protect predefined regions as the other two. Instead, each rule contains a start and an end address of the region it should be applied to. As their name suggests, they are used for memories, especially the main RAM. By this means, they allow to isolate the memory regions of the different subsystems in the shared main memory and, for example, accesses from the HLOS to modem internal data is prevented by hardware.

An MPU can isolate a subsystem's memory region from other subsystems in the shared main memory.

Bootloaders configure the XPUs during the startup phases of subsystems. In our focus case of the modem subsystem, this is the Modem Boot Authenticator (MBA) for the memory regions of the loaded firmware. Thereby, the modem's memory can no longer be read or modified by the HLOS after it was authenticated, we discuss details of this process in Section 7.1. Obviously, shared memory regions are left accessible from the HLOS.

Note that, in the case of MPUs, rules can overlap, meaning that two or more rules can be defined for a memory region. This is taken into account by the units and, depending on the owners of the rules, they implement appropriate behavior, as listed in Table 10 for two rules and the possibilities of their ownerships. Especially, if the modem and another party configure rules for a region, it gets locked for all accesses, even when both rules would give full permissions to all components. Each accessed address is checked individually, thus, for only partly overlapping rules, this logic only applies in the actual overlapping regions of the rules.

On all tested phones, violations of the modem memory permissions from the Android kernel lead to no visible extra measures from the baseband system, except from the access attempt being blocked by the XPU. However, the event could still be logged internally.

6.2.2 *Tightly-coupled memory*

Qualcomm's SoC includes an additional memory in its baseband part, called Tightly-Coupled Memory (TCM). It originates from the *ARM* world and is, for example, described in [1]. As the name suggests, it is a fast memory coupled closely to the processing core and accesses to it can be handled in a significant shorter time than to the main memory. This makes it similar to processor caches and its purpose is indeed the same: speed up the access to frequently used code and data. The main difference between the two is that a cache automatically decides which memory locations it stores and, as a result, is transparent to the running code, that means the operating system. For TCM on the other hand, the software can explicitly determine which memory addresses should reside in it, leading to higher predictability of its behavior and memory access times.

TCM is a fast memory similar to a cache but more predictable since it can be controlled by the programmer.

The instruction and data elements that are placed in the TCM are controlled by the linker, it places the desired parts in special sections, for example the *memcpy* function is located here. The OS of *Qualcomm* then loads these sections into the TCM, when desired. Some sections stay in the fast memory during the whole runtime while others are loaded only when needed, depending on the currently used mobile technology (GSM, UMTS, LTE, ...). The remaining space of the TCM memory is used for dynamic allocations, this means for demands oc-

Table 10: Enforcing of rules by MPUs in case of overlap

	Non-Secure	**Secure**	**Modem**
Modem	Locked for all	Locked for all	Only last rule
Secure	Only secure rule	Both rules	
Non-Secure	Only last rule		

curring during runtime which cannot be foreseen statically at linking time.

The TCM operation is close to a cache for the user code running on top, only differing by the OS operations. Especially in the context of this work, we can see it as a transparent cache without influence on, for example, the code patching operations we perform in Part iii, where we apply changes to the baseband's firmware. All modifications are independent of this mechanism as the TCM content is loaded during runtime from the usual firmware data, including our modifications.

However, it is still of importance to understand it and know its way of working as we will see references to the TCM during reverse engineering, for example, in strings found in the firmware.

7 MODEM SOFTWARE

The code running on the DSP and how it is loaded is the focus of this chapter. We also include the security aspect of the code authentication.

7.1 FIRMWARE LOADING

In order to bring-up the modem, its firmware has to be placed in the system's memory, the HLOS does this. It is a three stage loading process as Figure 9 illustrates it and explained in the following. Numbers in brackets refer to the step numbers in the figure.

To start the modem, first all related clock signals and voltage regulators are enabled, the modem restart signal removed. The modem now runs a Primary Boot Loader (PBL) which resides in ROM inside the chip and is not changeable after manufacturing. It also includes a digital certificate which is used to verify authenticity of the next execution stages.

Firmware loading is a three stage process, with each step authenticating the next one, building a chain-of-trust.

In the next step, the Android operating system loads the second stage bootloader which is called Modem Boot Authenticator (MBA), and tells the PBL where it located the MBA. The PBL then checks the authenticity of the data by verifying a signature, contained in the binary, with its certificate, we describe details on this verification in Section 7.2. When this check succeeds, the PBL will pass control to the loaded MBA. All code executed from now on resides in the system's main memory.

As a next step, the HLOS loads metadata of the actual firmware and its authenticity is checked by the MBA. This metadata contains information about the memory sections of the firmware, like their length and destination address in the system's memory. In addition, it contains hash values of the data of all sections.

The different memory sections of the firmware are then loaded one by one into memory and after each of those "blobs" the MBA is instructed to check the authenticity. Since it already has an authenticated hash value for the contents of each segment, it just needs to verify if these match.

When all these steps did not result in an error, finally control is passed to the main firmware code. In case one of the steps fails, the responsible bootloader will return an error and wait for repetition of this step.

Note that, before the authentication step, the MBA locks all memory regions containing the data with an access protection unit (see

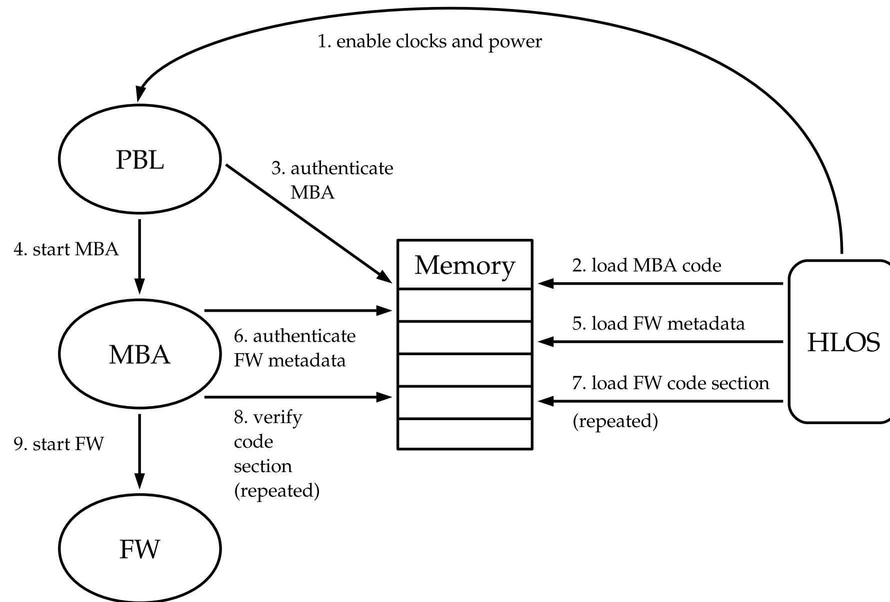


Figure 9: Modem firmware loading process

Section 6.2.1), exclusively allowing the modem processor to access it, withdrawing access rights from the HLOS. Therefore, it is not possible to simply modify code after authentication. Similarly, the PBL moves the loaded MBA code to a location solely accessible by the modem processor. This is the modem's internal TCM memory in which the complete MBA code and data fits. We assume that *Qualcomm* has chosen this solution to simplify the one-time-programmable PBL as it does not need to configure XPUs in this approach, minimizing the possibility for security issues in code that cannot be updated after shipping the chip.

Memory sections are protected from accesses of other subsystems before they are authenticated.

The method described is not valid for all *Qualcomm* modems. Older phones use the devices *ARM TrustZone* implementation to authenticate the code, instead of directly performing the computations on the modem's processor. The *TrustZone* kernel is invoked by the HLOS and informs the modem directly in case of success.

7.2 FIRMWARE AUTHENTICATION

In the last section, we saw a classical chain-of-trust with the PBL and its included certificate as root of trust. How exactly authentication is handled between the chains elements is topic of this section. First, we present authentication of the main firmware code and then show differences for authenticating the MBA.

The firmware image contains a chain of certificates (usually three). The first of these is signed with a key from the last bootloader stage. All memory sections of the firmware are hashed using the *SHA-256* algorithm (*SHA-1* is also defined but not used in any of the firmware

images analyzed). These hashes are then placed in a table. Together with a header structure (*table_header*), they build the data which is actually signed. This header includes information where to find the signature, the image size and the destination address of the firmware in memory. This data is part of the firmware metadata (steps 5 and 6 in Figure 9). It is then hashed again.

Metadata of the firmware is hashed and checked against a signature with certificates contained in the firmware image.

A scheme similar to the construction of a Keyed-Hash Message Authentication Code (HMAC) (refer to [20] for the general scheme and used names) includes two identification numbers, found in the certificate, into the calculation: a firmware identification number (*SW_ID*) is used as key in the first step, XORed with the *ipad* constant and a SoC identification number (*MSM_ID*) XORed with the *opad* constant as key in the second step.

For verification, the signature found in the image is checked by applying the public RSA key in the firmwares certificate (*pk_fw*) and checking the result against the calculated hash value. The overall scheme can be summarized as:

$$\begin{aligned} H &: \text{SHA-256 hash function} \\ \| &: \text{concatenation} \\ \oplus &: \text{XOR} \\ RSA_verify &: \text{RSA signature verification (encryption)} \end{aligned}$$

$$\begin{aligned} hash_table &= H(segment_1) \,\|\, H(segment_2) \,\|\, \ldots \,\|\, H(segment_n) \\ m &= H(table_header \,\|\, hash_table) \\ hmac &= H((MSM_ID \oplus opad) \,\|\, H((SW_ID \oplus ipad) \,\|\, m)) \\ \text{with} \quad ipad &= 0x3636363636363636 \\ opad &= 0x5C5C5C5C5C5C5C5C \end{aligned}$$

$$\text{verify} \quad hmac == RSA_verify_{pk_fw}(stored_signature)$$

To check the actual firmware memory sections, the authenticated hash values from the table are compared against the computed hash value for each section. Note that the metadata already contains one firmware section in addition to the authenticated part as described above, the Executable and Linkable Format (ELF) headers. These are also verified by checking the sections hash value.

In order to check the firmware's sections, authenticated hashes in the metadata are compared with the computed values.

Authentication of the MBA code from the PBL is handled in a similar way. Its binary also contains a certificate chain which is verified by the previous stage. However, since the MBA only contains one section, the hash table is replaced by directly using the code data. This is hashed together with the same header structure as before and used as input to the HMAC process. After this, the signature verification is done in the same manner as before.

7.2.1 *Issues*

The used authentication method seems to be well designed, as it uses a common structure and builds on well-researched cryptographic algorithms. Problems on deployed devices emerge from another reason: negligent smartphone vendors. For developer devices, the authentication can be disabled or configured to accept special development certificates by settings in the one time programmable *QFuses*. For devices on the end-user market, manufacturers should blow these fuses (irreversibly program) so that a device only accepts correctly signed modem firmwares. Therefore, this in itself is not a problem.

Smartphone vendors are largely responsible for security, not all of them activate the authentication.

However, not all vendors activate the authentication mechanism on their devices, be it with settings in the *QFuses* or with modifications of the PBL or MBA. In [11] the *Icon 225 USB modem* was found to skip the check of the MBA signature. During this work, thanks to an anonymous hint in an Internet forum, we found devices from *Asus* to be configured insecure concerning this aspect. Either they use publicly available development keys or no signature checks at all. It might be the same for other vendors or at least some devices of other vendors. This allows modifications of the firmware as we discuss them in Part iii of this thesis. It leaves an open door to attackers, especially as the modem firmware can simply be overwritten from the running Android system (requiring root access rights) and will be loaded on the next boot. On the other side, it opens the phones for research purposes also discussed in this document.

7.3 OPERATING SYSTEM

Historically, *Qualcomm* used their completely self-developed operating system called *Rex* and later shifted to use *Rex* components on top of an *OKL4* microkernel from *Open Kernel Labs* [41]. However, when *Qualcomm's* basebands started to be *Hexagon* based, the company decided to change to a newly implemented operating system. It was first introduced as *Blast* but then got renamed to Qualcomm Real Time operating system (QuRT), with only minor changes. The *Hexagon* SDK also includes this OS for user applications running on the audio DSP. Even in *Hexagon* modem firmwares, still a lot of strings refer to the old *Rex* OS, clearly showing the origin of the software stack running on top.

The baseband runs Qualcomm's self-developed QuRT real-time OS.

QuRT is a real-time OS tailored for the needs of the baseband. It offers common OS functionality, abstracting from the DSP hardware, including:

- Thread/Task management
- Communication and synchronization between Threads
- Cache/Translation Lookaside Buffer (TLB) management

- User memory management
- Interrupt and exception handling
- Timers and system clock

In the following sections, we review selected parts of the OS relevant for this thesis in detail.

7.3.1 *Tasks*

In order to support multiple tasks, QuRT includes (software) threads. These abstract from *Hexagon's* hardware threads (presented in Section 6.1.4). QuRT takes care of scheduling many software threads on the limited available hardware threads, similar to scheduling in a multi-core system. A priority value ensures that important tasks, which need to be finished fast, for example the physical layer processing in LTE, get the computing resources first. Additionally, interrupts always suspend the hardware thread processing the task with the lowest priority in the set of currently scheduled tasks, leading to the minimal possible disruption.

QuRT implements threads with priorities to support multiple tasks, which are initialized by an init system.

After system boot, QuRT initializes and starts tasks with its init system *rcinit*. It is also capable of starting legacy *Rex* tasks which were not yet ported to the QuRT implementation.

Synchronization between threads is provided by usual means, including mutexes, semaphores, barriers and also atomic instructions for more flexibility. Tasks can sleep and wait for signals of other components to trigger continuation of their execution. A readable name is also assigned to each task. Table 11 lists all tasks started on our test smartphone's baseband (Asus PadFone Infinity 2, baseband version *M3.12.15*), obtained by modifying the task creation function of *rcinit* and extracting the task information of interest. Shown are the tasks name and its priority, which ranges from 0 (lowest) to 255 (highest). Obviously, the active tasks depend on the features supported by the modem and are therefore different for each model. For further details, Appendix A.1 contains a listing of the *rcinit* startup sequence, including task starts and thread creations. It allows to observe the loading order of the various tasks and we can also guess which threads are associated with which task. We see that one main thread is created for each task with the same name as the task, some tasks spawn more threads from this main thread and also create and destroy child threads dynamically during runtime, similar to usual multi-threaded programs.

7.3.1.1 *Inter Process Communication*

Communication between different tasks can be achieved by memory regions shared between the tasks and utilizing the synchronization

mechanisms to access them. However, QuRT also includes some more advanced techniques, summarized in Table 12.

Various IPC mechanisms allow for communication between tasks.

With the *signal* mechanism presented before, a thread can sleep and wait for an event of another thread without consuming computation time, leaving the resources to other tasks. As a consequence, this is useful for notification about events. If a larger amount of data, arriving in a stream pattern, should be transferred from one sender to one receiver, QuRT's *pipes* are the right choice.

For more complex communication schemes, *Qualcomm's* firmware implements its own *Inter-Process Communication (IPC) router*, based on the mechanisms provided by the OS but not part of it. Clients can register to this router for message types they want to receive. The sender of a messages tags it with an identification number structured such that similar message types get similar IDs and, as a result, clients can register also for ranges of IDs to get all messages from a module or for a technology such as LTE. The sender of a message does not need to know the receiving clients, leading to a loose coupling between the modules and point to multipoint schemes being covered.

7.3.2 *Memory management*

QuRT takes care of cache and TLB administration as well as memory address translation from virtual to physical addresses. To its user code, it offers memory management in terms of a *heap* as almost every operating system nowadays.

A DSM pool, for allocations with changing size, is provided in addition to classical memory management.

An implementation named *Data Services Memory (DSM)* pool expands this. It is tailored for flexible size memory allocations which might grow during their lifetime. To achieve this, it creates a chain of memory buffers for each allocation, implemented as a linked list. Whenever the buffers of the current chain are full, a new chain element can be created to increase the capacity of the DSM chain. A disadvantage is that the access is slightly slower compared to a continuously allocated memory region since accesses need to check in which buffer which data bytes reside and potentially write or read over multiple buffer items, leading to the need to perform some bound checks on each item. Note that still, even for large buffer allocations, this structure can be advantageous as only smaller memory sections are needed for the buffers compared to the requirement of a single large region. This allows the heap to make better use of the available memory and, in case the remaining memory gets limited, it is more likely that a set of small DSM buffers can be allocated rather than one large continuous block.

The API includes basic functions for allocation and deallocation of elements, to read, write and extract data as well as more advanced features like locked access to allow sharing of a DSM chain between multiple threads.

Table 11: List of active tasks

Name	Priority	Name	Priority	Name	Priority
dog	164	gsm_mac	119	hitapp	76
rfa_fwsw	164	gsm_rlc_dl	118	cm	75
wfw_eulstrt	164	gsm_rlc_ul	117	qmi_mmode	75
wfwsw_evt	164	locotdoactrl	112	ftm	74
tds_fwsw_evt	164	tds_rrc	110	audioinit	73
timer	163	tplt	110	wms	72
tmr_slave2	162	rrc	109	nf	71
tmr_slave1	162	gsm_rr	108	trm	70
tmr_slave3	162	dswcsd_ul	107	cd	70
npascheduler	162	ps_rm	107	tlm	69
cc	162	a2_ul_per	106	xtm	68
slpc	161	dswcsd_dl	106	ui	67
tds_l1	161	ds_gcsd	105	uim	65
gsm_l1	161	gsm_llc	104	time_ipc	64
hdrrx	161	comp	103	fs	62
rf_fwrsp	160	mm	102	nv	60
rf	160	reg	101	modem_cfg	60
rf_ic	160	tc	100	locotdoamp	60
tx	156	sm	99	qmi_pbm	56
fc	155	mn_cnm	98	pbm	56
wcdma_l1	153	mc	97	gsdi	55
sys_m_qmi	152	hdrmc	96	gstk	54
sys_m_smsm	152	sd	96	ds	53
sys_m	152	auth	95	qmi_modem	52
hdrdec	151	rf_apps	94	dcc	51
hdrtx	150	mmoc	94	ds_sig	50
rx	149	mgpmc	93	dh	49
srch	148	lm	92	ps	47
pgi	147	sm_tm	91	ims	44
hdrsrch	146	sm_gm	91	qvp_rtp	40
rxtx	143	gnss_msgr	91	secips	37
smdtask	142	gnss_sdp	90	secrnd	34
a2	125	pdcommtcp	88	secssl	30
pp	124	pdcommwms	87	sec	29
tds_l2_ul	123	tcxomgr	86	cb	26
wcdma_l2_ul	123	tdso	85	seccryptarm	24
tds_mac_hs_dl	122	cxm	80	gpsfft	21
wcdma_mac_hs_dl	122	limitsmgr	80	gps_fs	20
tds_l2_dl	121	loc_middleware	79	fs_async_put	8
wcdma_l2_dl	121	thermal	78	a2_log	8
gsm_l2	120	diag	76	sleep	1

Table 12: Inter Process Communication mechanisms

Mechanism	**Suited for**
Signals	Event notification
Pipes	Point-to-point serial data streams
IPC router	Point-to-multipoint messages, possibly with unknown destinations
Shared memory with synchronization	Custom implementation, data shared between two tasks

7.4 SECURITY FEATURES

After the severe security issues of basebands as shown in the related work (Section 2.1), for example in [40], *Qualcomm* added attack mitigation techniques to their firmwares in the *Hexagon* era.

The XPUs of Section 6.2.1 enforce memory access rights, with rules depending on the requesting component. Their initialization during boot by the MBA and PBL implementations withdraws all rights of the main system for the basebands memory regions.

Well-known exploit mitigation techniques harden the baseband's security.

The build chain used to generate the firmware image analyzes all functions and automatically adds stack canaries to functions that might contain stack buffer overflows: a random canary value is chosen during the system start. It is then written as the last element of the function's stack frame and verified at the end of the function. An exploit of the memory corruption would need to overwrite the canary value in order to write into other stack frames and, for example, modify the function's return address. As a result, if the attacker does not know the correct canary value, the stack modification will be detected and an exception will be raised. We can clearly see this mechanism during reverse engineering *Qualcomm's* firmware.

According to R.P. Weinmann in [41], these additional security measures are present:

- Safe unlinking for the heap: prevent exploitation of heap overflows
- Data execution prevention by a non-executable stack/heap (optional, OEM can decide)
- Kernel/user mode separation (optional, OEM can decide)

7.5 COMMUNICATION WITH HLOS

Historically, modems were connected over a serial interface to the main system. To control the modem, human readable *AT* commands (from *attention*) were used, encoded in the American Standard Code for Information Interchange (ASCII). The command *AT+CPIN="1234"*, for example, would try to unlock the SIM with the PIN code *1234*. Data communication was also handled over such a serial link using the Point-to-Point Protocol (PPP).

The HLOS communicates with the modem over a serial link. In a shared memory architecture, this behavior is emulated.

Especially the AT protocol is a rather inefficient encoding, for instance a simple byte value uses up to three bytes instead of just one and to encode the value it has to be converted into an ASCII character string.

Therefore, more efficient and flexible protocols were developed, we review *Qualcomm's* QMI protocol in Section 7.5.2. Since these protocols are proprietary, we will see that still an AT interface is implemented in the modem for legacy and compatibility reasons.

Section 5.1 shows that the communication between the baseband and the main system is realized by shared memory, replacing a physical serial link. The lower layers in the used protocol stack emulate behavior similar to the serial interface.

7.5.1 *Protocol stack*

Table 13 gives an overview of the used protocols and their layering. The Shared Memory (SMEM) driver provides access to the physical available memory. It implements allocation and safe access to the shared memory areas, taking care that no access conflicts between the two entities, communicating over the shared memory, happen. It also translates addresses from a virtual address space to the physical locations where the shared memory area was placed.

Abstraction from the raw memory is done in multiple layers, on the top providing protocols independent of the actual connection.

On top of this layer operates the Shared Memory Driver (SMD). It offers independent data channels, identified by names. Its API for these channels includes functions to read and write a serial stream of data and also supports data transfers in packets.

Finally, various protocols for the different purposes use these channels. For example, the implementations include a serial interface (TTY) emulation. By using this, the modem looks like being connected over the same kind of serial interface as in an architecture with modem and main processor separated. Another protocol example is the *NMEA 0183* implementation to connect the GPS system, which then seems to be connected like a typical GPS receiver and standard drivers can be used to process its data.

The most interesting protocol for this thesis is the Qualcomm MSM Interface (QMI), used to send commands to the modem. In the practical part, we also use it to exchange messages between the Android kernel and our injected code in the modem firmware (Section 10.1).

7.5.2 *Qualcomm MSM Interface (QMI)*

The QMI protocol contains three types of messages: requests, responses and indications. Requests can be issued to services which then later answer with a response message. When multiple requests

Table 13: Protocol layering for communication between Modem and HLOS

<table>
<tr><td>QMI</td><td>TTY</td><td>...</td><td>purpose specific protocol</td></tr>
<tr><td colspan="3">SMD</td><td>independent serial channels</td></tr>
<tr><td colspan="3">SMEM</td><td>access and allocation of memory</td></tr>
<tr><td colspan="3">Shared Memory</td><td>physical memory</td></tr>
</table>

are issued in a short time, the responses might arrive in another order but include the request ID and can be matched by this. Indication messages are send by services without a previous request in case a certain event happens, removing the need to poll for state changes. They can be broadcasts to all clients or send to only a single recipient. Before a client can access a service, it has to allocate an ID at a special control service (CTL). After this, it can send messages to the services implementing the different functions, for example to the Device Management Service (DMS) or the Network Access Service (NAS). [27]

QMI is a flexible protocol supporting different patterns of communication.

Be aware that, depending on the used kernel driver, this registration might be implemented inside the driver and occur transparently to the code using the QMI interface. This is notably the case for *Qualcomm's* QMI driver in the here used Android kernels.

A 32 bit service ID together with a 32 bit value, distinguishing between different instances implementing the same service, for example in two different subsystems like the modem and the audio processing system, identify services. These should be unique, however, in Section 10.1.2.1, we give a practical example in which this is not the case. The parameters in QMI messages can be mandatory or optional, which can also depend on other parameter values. They are encoded using Type-Length-Value (TLV) fields.

8 REVERSE ENGINEERING THE MODEM

As the internals of *Qualcomm's* modems are the company's secret, only few information is available in publicly available documentation, or can be derived from this. Therefore, the remaining information needs to be obtained through reverse engineering. Results from previous projects, presented in the related work chapter, can replace this reverse engineering or give at least a starting point. Even work on older modem models is included here, since obviously *Qualcomm's* engineers do not redesign the whole modem hardware and software from scratch in each revision but reuse large parts of the design. However, we have to verify that a design aspect is still valid and if not analyze how it changed.

We used multiple techniques to reverse engineer the target modem system. Inspecting the related Android kernel code, which is open source, allows to study some aspects. Examples for this are the communication mechanism between the modem and the HLOS (Section 7.5) as well as the firmware loading scheme (Section 7.1) as they require corresponding implementations in the Android kernel. The firmware loader, together with investigation of the raw firmware image in a hex editor, respectively the files contained in the image, allowed to derive the firmware format used (Section 9.1), a prerequisite for all further analysis of the modem's software and the key for modifications.

We combined multiple sources to reverse engineer the modem. Together, they allow to deduce the presented results.

Knowing the firmware format, we can convert it into a common format (ELF) which allows using well established reverse engineering tools such as *IDA pro*. In Section 6.1, we already introduced the available plugins for the used *Hexagon* processor for IDA. Well established techniques then allow to analyze the firmware. Especially, strings inside the binary were found useful as we show in the next section.

Additionally, *Qualcomm* released an SDK for the *Hexagon* DSP, targeting user applications on the audio accelerator. As it includes the same OS, information about the QuRT Real Time Operating System (RTOS) can be collected from the code of the SDK. It also contains documentation of the build tools which were useful in the development of the patching framework.

These are completed by techniques used in special cases, for specific purposes. An example are dynamic observations of variable values in the patch code implementation phase. Allowing simple dynamic analysis scenarios is also the aim of the modem memory access patch (Section 10.3).

The mainly used static techniques can be summarized by:

- Android kernel code analysis (firmware loader, QMI, ...)
- Raw binary inspection
- SDK analysis (QuRT, build tools)
- IDA pro reverse engineering
  - String analysis
- Other techniques for special purposes

8.1 STRING ANALYSIS

When loading the firmware of our target device, the *Asus PadFone Infinity 2 (A86)*, the IDA tool found a total of 79,198 strings in it. Some of these are false detects and do not contain a useful string. Another portion of the strings is never used, or at least IDA could not find a reference. Still, a large number of helpful strings remains.

The most helpful tool to analyze the modem were strings in the firmware binary as they reveal many details of the code.

These strings include, among other things, file and function names, conditions of assertions as text representation as well as status and error messages. Listing 1 gives a few examples. The first string is obviously an error message. By comparing the condition under which it is printed, we can identify the "payload_ptr" referred to in the message in the assembly code referencing the string.

Even more useful is the second example in the listing. It first tells that the function it is used in is a "CSF::CALLBACK", which we can identify as acronym for *channel state feedback callback*, allowing to label the function with a meaningful name. It also tells that we are now "entering csf post processing", so what the following code of the function is doing. With this knowledge, we might be able to also label several called functions with names and thus successively get a better understanding of the firmware's internals. The variable value outputs in the format string allow to label a list of variables (frame, subframe, carrier_index, sys_bandwidth, csf_config_flag) when we

Listing 1: Example strings found in modem firmware

```
1 Fatal Error: 'payload_ptr == NULL' %d%d%d
2
3 CSF::CALLBACK frame=%d, subframe=%d, carrier_index=%d, sys_bandwidth=%d,
     ↪ csf_config_flag=%d, entering csf post processing
4
5 D:/builds/build/2000.1_test0726/modem_proc/core/securemsm/
     ↪ x509/shared/src/secx509.c
```

compare them with the parameters passed to the print function together with the string. These may be stored in registers or stack locations local to the function and the identification of them greatly helps in understanding the purpose and the way of working of the target assembly code. If the variables are even global variables, stored in system wide memory locations, they also provide information for analyzing further functions.

The last example of Listing 1 simply contains a filename. Interestingly, it contains the whole path of the file on the build system. Strings like this one allow to know what file the surrounding code comes from and with that estimate its purpose. They help to shrink the amount of code to search for the correct target function, for a given feature, in the large modem code.

Part III

MODIFICATION OF FIRMWARES

In this part, we present the developed patching framework enabling firmware modifications. For this, we analyze the structure of the firmware image and demonstrate the needed steps to successfully modify the image. In the end, we show implemented example use cases and explain them.

9 PATCHING FRAMEWORK

As mentioned in Section 1.2, the aim of this work is to modify the modem's firmware in order to extend its functionality, get access to internal data or to mount attacks. We have to modify the binary image to achieve the desired behavior as the modem's source code is not publicly available. In early stages of this work, firmware patches were first done by hand, either in assembler or before by even directly computing the binary encoding of the instructions and placing them in the binary. Obviously, this is time consuming and only feasible for small patches like changing a string constant, re-directing a jump to another function or removing a condition check (for example to remove a SIM lock).

Simply writing in a higher level language and using a compiler is not possible as the code has to be linked with the existing firmware binary and integrated into it. Also, the firmware contains checksum values which need to be corrected, as seen in Section 7.2. Therefore, in addition to the code compile step, we need to perform many more operations, which we detail in Section 9.2.

The developed framework allows to modify firmware images with patches written in C.

It should now be obvious that for complex firmware modifications some more sophisticated solution than applying patches to the binary by hand are desired. As a result, inspired by the *nexmon* project [36], we implemented a patching framework using Python scripts to realize custom functionality. *Makefiles* control the build process and call the different tools (*Hexagon* compiler, linker, ...) and Python scripts.

This framework includes the following main features for patch code:

- Overwrite functions by annotating a patch code function with an attribute
- Place functions in pointer tables (e.g. handler dispatch tables) using an attribute
- Call functions of the firmware
- Call an original (overwritten) function of the firmware by using an automatically generated function named in the form "FUNC_NAME_fw_org" (explained in Section 9.2.3)

To enable these functions, the framework needs to know the positions of functions and other symbols (like strings) inside the target firmware binary. These can be specified in additional firmware wrapper header files, in Section 9.2.2 we give more details.

Unfortunately, it is not possible to generate a valid digital signature for the patched binary, needed to pass the authenticity checks described in Section 7.2. Therefore, the framework can only be used for smartphones with this check disabled (Section 7.2.1).

9.1 FIRMWARE IMAGE FORMAT

A special partition stores the modem firmware in multiple files, basically consisting of ELF files and metadata for the MBA and the actual firmware.

A special partition, formatted with a File Allocation Table 16 (FAT16) file system, stores the modem firmware. System updates contain an image file of this partition and can, therefore, be used as source to obtain the firmware. Alternatively, the partition can be read and dumped from a running phone. Our target devices use a size of 64 MiB. Other phones use slightly different capacities, mainly depending on what firmware files for other subsystems are also placed on this partition. Table 14 lists the set of files it contains for the modem, together with their purposes.

The files can be grouped into two categories: metadata files (*.mdt*) used by the firmware loader in the Android kernel and the files containing the actual content being loaded (*.bXX*) into the basebands memory section. For the MBA, only one content file is used. The actual firmware, on the other hand, uses between 20 and 30 files, depending on the modem model and enabled features. Each of these files contains one of the sections specified in the ELF header. The header can also define sections with a file size of zero or a memory size bigger than the actual size of a section file. The loader then initializes these memory regions with zero bytes.

Since the first section contains an ELF header, all content files concatenated result in a valid ELF file of the firmware (padding bytes between the files might be required).

Note that the firmware metadata file is equal to the concatenation of the first two content files and therefore these files contain redundant information. As these are only required for firmware loading, the first two content sections are not actually loaded and the files are

Table 14: Files in a modem firmware image

File(s)	Purpose	Content
mba.mdt	MBA loader information	MBA metadata (ELF header)
mba.b00	MBA content	MBA code, signature, certificates
modem.mdt	FW loader information	FW metadata (ELF header), hash table, signature, certificates
modem.b00	FW content	FW metadata (ELF header)
modem.b01	FW content	hash table, signature, certificates
modem.b02 - modem.bXX	FW content	FW code sections

not required. They are only residuals of *Qualcomm's* internal firmware build process.

Please also refer to Section 7.1 to see how the files are loaded and how the metadata is used.

9.2 PATCHING PROCESS

In this chapter, we present each step of the patching process performed by the framework in detail. The general steps are:

1. Extract a base ELF binary from the image file (Section 9.2.1)
2. Generate the firmware binary wrapper files (Section 9.2.2)
3. Analyze the patch code and create needed *fw_org* functions (Section 9.2.3)
4. Compile the patch files (Section 9.2.4)
5. Link the patch code files together and with the firmware wrapper (Section 9.2.5)
6. Combine the (patch) base ELF file with the generated patch binary (Section 9.2.6)
7. Generate an image file which can be loaded (Section 9.2.7)

Firmware patching is done in many steps, starting from a base image and resulting in the patched image ready to be loaded.

If one of these steps was already performed before and the input files did not change, it will not be executed again. A good example for this is the extraction of the base ELF file which needs to be done only for the first run.

In addition to the (annotated) patch code, the framework only requires the firmware image file and a wrapper defining existing functions and their locations as inputs. In order to analyze the source code, the framework uses a Python implementation of a C language parser called *pycparser* [6] with the extension *pycparserext* [19]. For the generation of *fw_org* functions, the *hexagondisasm* disassembler [26] is required, we detail this in Section 9.2.3.

Due to the simple annotation interface of the framework, interactions between the steps and generated intermediate files passed between the steps are rather complex. Figure 10 illustrates the process and the used files. Numbers on arrows in the figure refer to the step numbers in the list above. Table 15 details the contents and purposes of all occurring files. In both of these, symbolic names (indicated by all uppercase writing) are used for input files or files with names depending on the actual patch project.

The framework overwrites functions by replacing the first instruction of the original function with a jump instruction to the new function's first instruction. Therefore, it destroys the original function. In

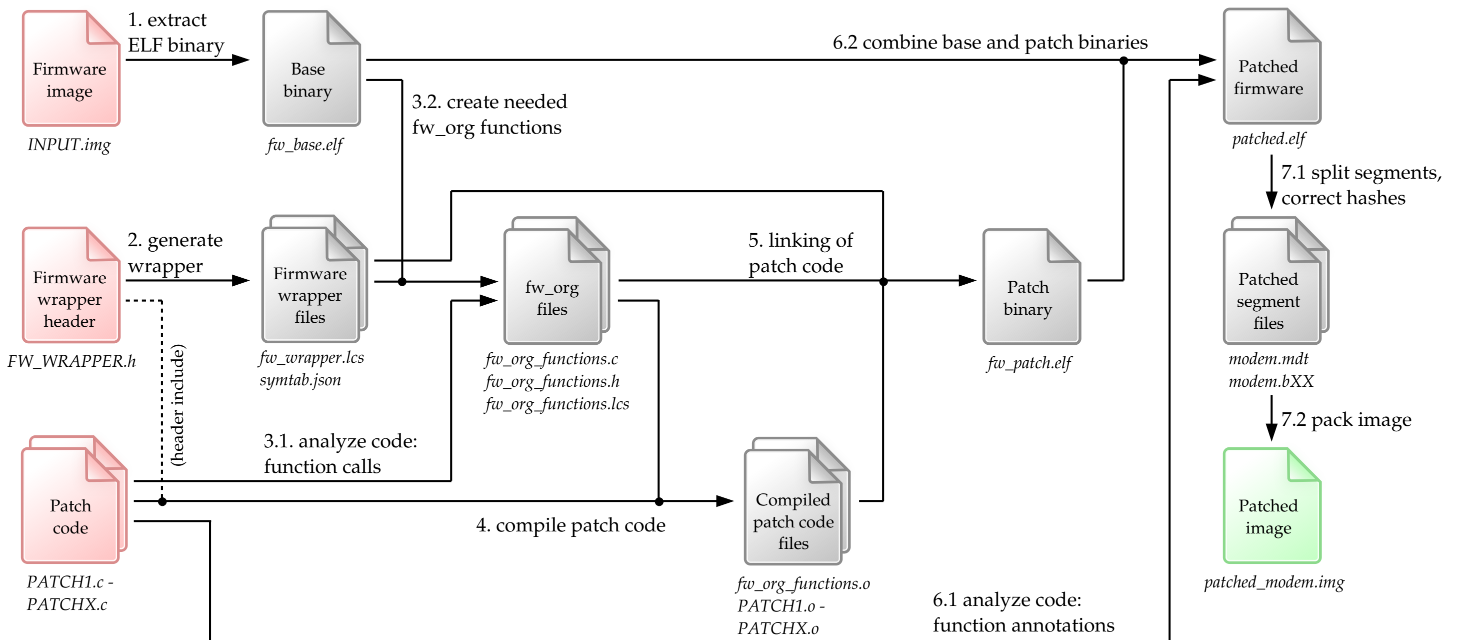


Figure 10: Overview of the patching process of the framework (input files in red, output file in green and intermediate files in gray)

The framework overwrites functions by replacing their first instructions with a jump to the patch code.

case we still want to use the old function, we have to re-generate a function with the same behavior. The *fw_org* functions, which we introduce in Section 9.2.3, target this problem.

An alternative approach to overwrite functions would be to patch all calls and jumps to this function with the new address. Obviously, this leaves the old function untouched and it could still be called from patch code if needed. However, for this we need to find all calls to the function. For call instructions with directly encoded destination addresses this is feasible, for calls to an address stored in a register, on the other hand, it is not trivial anymore to correctly patch them. These occur, for example, when using function pointers in the source code, a feature used extensively in the modem code. The chosen approach guarantees to re-direct all occurring calls without special treatment of this case.

Due to *Hexagon's* interleaved execution of hardware threads, as seen in Section 6.1.4, the processor completes all previous instructions before it loads the inserted jump instruction and finishes the jump instruction before it will load the next instruction of this thread. Therefore, in contrast to other processor designs, the computation of the next program counter address is finished and the processor can load the correct instruction, without the need to stall the pipeline or having to remove incorrectly loaded instructions. As a consequence, there are no extra pipeline penalties for jump instructions and the total penalty of the inserted jump is only one execution cycle.

9.2.1 *Extracting the base firmware*

The firmware obtained from a system update file or directly from a phone is a partition image as seen in Section 9.1. We also discovered that all the *modem.bXX* files together form an ELF file. Since this is a standard file format supported by many tools and therefore easy to process, the first step the framework performs is to generate such a file from the base firmware.

As first step, the framework extracts an ELF file from the base image.

For extracting the section files from the image, the Unix program *mcopy* (part of *mtools*) is used. A Python script (*blob-merge.py*) then merges these. This script reads in the ELF information from the first section file and then combines all sections into a single file by writing the individual section file contents to the file offset indicated in the read header data.

Table 15: Files used by the patching framework. Input files are listed first, followed by intermediate files and the output image as last file.

File(s)	Purpose	Content
INPUT.img	firmware input	partition (FAT16) with firmware files
FW_WRAPPER.h	input of information about functions in firmware	declarations of functions in firmware (annotated with addresses)
PATCH1.c - PATCHX.c	patch code input	patch source code
fw_base.elf	firmware in convenient format for further processing steps	base firmware (ELF format)
fw_wrapper.lcs	information for linker about base firmware binary	base firmware symbol definitions
symtab.json	aggregated information for fw_org functions generation	base firmware symbol and function declaration information (in JSON)
fw_org_functions.c	code for fw_org functions	fw_org functions definitions
fw_org_functions.h	knowledge of fw_org functions in compilation of patch code	fw_org functions declarations
fw_org_functions.lcs	correction of relative addresses in fw_org functions by linker	fw_org symbol definitions
PATCH1.o - PATCHX.o, fw_org_functions.o	binary patches ready for linking	compiled patch code (with relocations/symbols information)
fw_patch.elf	complete binary patch ready for merge with base firmware	linked patch code
patched.elf	patched firmware, input to image generation	patched firmware (ELF format)
modem.mdt, modem.boo - modem.bXX	intermediate files for hash correction and image generation	patched firmware sections and metadata (see also Table 14)
patched_modem.img	patched firmware output	partition with patched firmware files

Note that we based the idea of the script and the ELF header processing on the *pil-splitter.py* script available at [37], which performs the inverse operation, extracting segment files from an ELF file. We also use this script later for generating the output image file in Section 9.2.7.

9.2.2 *Firmware wrapper generation*

The binary file obtained in the previous step contains all the (machine) code of the firmware but no information about the functions and their parameters inside these set of instructions. With the knowledge of a function's address and its parameters, it is possible to call this address directly and place the parameters at the right positions. However, this is not convenient for the patch code programmer and since the function addresses are different between modem models and even between different firmware versions for the same modem, this patch would not be portable and only work with a single firmware binary.

9.2.2.1 *Purpose of the wrapper header*

A wrapper header file contains signatures and locations of functions present in the base firmware.

Therefore, information about the existing functions in the binary is stored in a header file (wrapper). By including this header, functions from the firmware binary can be called like a normal function in the *C* code: they are well defined for the compiler. The framework also uses this file to define symbols (name to value/address mappings) for the functions and feed them to the linker, so that it can resolve the destination addresses for calls to these functions. We give more details in sections 9.2.4 and 9.2.5 which are dealing with these steps. A header file needs to be written by hand once for each firmware version, however, in Section 9.3 we show scripts able to port the file from one target to another automatically.

The framework also uses this wrapper to determine which instructions (addresses) need to be changed to overwrite a certain function (Section 9.2.6) and to construct fw_org functions (Section 9.2.3).

As a result, the firmware image file itself and the wrapper header are the only modem and version dependent files. Patch projects can be processed for all targets with these available, as long as all used firmware functions are declared in the header. Please keep in mind that the internal structure of the firmwares could still be different, meaning that discrepancies between available functions, their signature and their internal behavior might exist. Consequently, patch code depending on such a function is not portable anymore or can behave differently on particular targets. Luckily, the observed *Qualcomm* firmwares were found to be very similar, especially the actual mobile communication stack implementations seem to originate from the same code base.

9.2.2.2 *Wrapper header content*

Like a usual header file in the C language, the firmware wrapper contains declarations of variables and functions with their signatures (name, parameters and return value). Our framework extends the *__attribute__* mechanism for code annotations with additional information of *GNU C* with an *address* attribute in order to define the memory address of a function. Listing 2 shows an example to illustrate the syntax of this. To simplify the annotation even further, we defined a macro *ADDRESS* here.

The wrapper stores function information as usual C declarations, with additional annotations to define memory addresses.

We can see that this approach keeps the effort to add the needed data about a firmware binary small. It encodes all required information for a function together. On the contrary, function signatures need to be declared again for every new firmware, which could be avoided by splitting the location data and signatures into separate files. Then, however, we still need to write the function names in the address file to identify the symbols resulting in only a small advantage. Since available functions and their signatures might also differ between firmwares as seen before (Section 9.2.2.1), we preferred the other approach.

Note that the wrapper header file might also include other header files, for example to group function declarations by purposes. Since this is just a normal C include treated by the preprocessor, for simplicity, we can consider the wrapper header as a single file.

In addition to functions and variables, the base address of patch code and data segments can be defined in the wrapper by declaring pseudo (void) pointers called *__patch_addr_text_base__*, respectively *__patch_addr_data_base__*, with the same annotation scheme as before to define the address.

9.2.2.3 *Wrapper processing*

Two files are generated from the wrapper header file as input for later processing steps. The *fw_wrapper.lcs* file is a linker script containing symbol mappings from all functions and variables declared in the header to their address specified in the attached attribute. The main

The header is transformed into a linker script and into JSON format for further processing.

Listing 2: Firmware function declaration example

```
1 #define ADDRESS(x) __attribute__ ((address (x)))
2
3 ADDRESS(0x09F88260) void* malloc(unsigned int size);
4 ADDRESS(0x09F87F40) void free(void* ptr);
5 ADDRESS(0x092B9AB0) int snprintf(char* str, unsigned int size, const
    ↪ char* format, ...);
```

control script handed to the linker can directly include this file as the symbol definitions are valid linker control instructions.

As a second file, *symtab.json* is generated. It contains all information of the header file (names, function signatures, addresses) in a JavaScript Object Notation (JSON) format. Its solely purpose is to simplify reading in of the information in scripts of consequent steps and avoiding the need to slowly parse the code again. It is used in the fw_org functions generation (Section 9.2.3).

The *create_wrapper_lcs.py* script does this processing of the header file. It parses the input wrapper file, using a Python *C* parser (*pycparser* [6] with *pycparserext* [19]), and then writes out the data in the formats of the two files.

9.2.3 *Preparation: fw_org functions generation*

In the general discussion of the patching process (Section 9.2), we already saw the way to overwrite functions used by the framework: the first instruction of the function is replaced by a jump to the patch function. Obviously, this will destroy the original function as its first instructions are missing and the remaining instructions will not be reached by the processor anymore, at least as long as the code originates from a standard *C* compiler and contains no jumps into the middle of the function. This seems to be no problem since we anyway wanted to replace the function with a new one and, as a consequence, do not need to original version anymore.

In many cases, it is desirable to call an original function from the function overwriting it. fw_org functions enable this.

However, there are cases in which it is desirable to call the original version of the function. Instead of changing a function completely, often one just wants to add some functionality to it. An example for this would be to add a call counter to a function. The counter has to be incremented after the function call, the rest of the function should perform as the original function before. Another similar case could be retrieving cryptography keys from a generator function. Here, we first want to do the same calculations as the original function and then do additional processing with the result. If the original function is not available anymore, we have to re-implement the complete behavior of it, instead of just calling the old code at the desired position in the patch function.

As a solution, the framework offers the *fw_org* functions. These reproduce the behavior of the original (overwritten) function of the firmware and, therefore, can be utilized in the examples given before. They are used by simply calling a function with the name of the original function appended with the postfix *_fw_org*. The framework will then automatically detect this call and create the needed function with appropriate content.

9.2.3.1 *Detection of required functions*

The generation of these *fw_org* functions is quite complex and slow. Therefore, as a first step, we need to detect which ones are needed and should be generated, instead of creating all possible ones. For this, with the Python *C* parser (*pycparser* [6] with *pycparserext* [19]) previously used for the wrapper header, the framework parses the code and checks function calls for names ending with the identifier prefix, leading to the list of required *fw_org* functions.

9.2.3.2 *Function generation*

To re-generate the behavior of the original function, we generate a new function containing the first instructions of the original function, followed by a jump to the rest of the original code sequence, as only the first few instructions were destroyed by the redirecting jump.

fw_org functions reimplement the destroyed instructions of an overwritten function and then jump to the rest of the original code.

A jump instruction on *Hexagon* has a limited range for the destination address relative to the current execution address. Since the patch code is located in another memory segment than the original function, the redirection jump is usually far, with a value bigger than this range. As a result, we might need a constant extender (Section 6.1.3) and, in total, two instructions of the original function have to be modified.

When re-generating the first instructions of the function, splitting an instruction packet (Section 6.1.2) would lead to undefined behavior. As a consequence, always complete packets need to be copied. Mostly, only the first packet needs to be re-implemented, solely in case when the first packet contains a single instruction and a constant extender is necessary for the jump, two packets need to be copied.

Since the instructions might contain relative addresses for calls or jumps and they will be relocated to another address in the *fw_org* function as before, simply copying the binary encoding values would change their meaning. As a solution, the *generate_fw_org_functions.py* script passes them to a disassembler implementation (available at [26]), resulting in assembler code. The code, together with a jump to the first instruction of the first unchanged packet of the original function, is then used as inline assembler implementation in the surrounding *fw_org* *C* function.

Relative references in the instructions are replaced by symbols, as a result, the linker solves all problems.

Our scripts replace all relative address values in this by symbols. Thereby, the linker will take care of correcting the values depending on the instructions final position. It will also handle problems when a distance is now too far to be encoded in the instructions immediate field by creating a jump to another location where it has enough space to place a jump instruction together with a constant extender, a so called "trampoline".

Listing 3 shows an example of a generated *fw_org* function implementation. In line 3, we see that a relative address was replaced by a symbol and in line 6 is the jump to the remaining original function

code. Note that the *C* compiler will generate a function return after this jump which will never be reached. Instead, the function ends with the return instructions of the original code, which directly jump back to the calling instruction of the *C* function.

These functions are written to the *fw_org_functions.c* file and signature declarations for them are available in the corresponding header file (*fw_org_functions.h*). Definitions of the symbols occurring in the inline assembler code are generated in *fw_org_functions.lcs* which has the same format as the *fw_wrapper.lcs* file and therefore can also be used directly as input to the linker.

9.2.4 *Compiling*

Compilation is straightforward and uses the compiler of the SDK.

Next, the framework compiles all source code files individually. Every called function is well declared for this step now, functions of the firmware are given in the wrapper header and the header file generated in the previous step contains the *fw_org* functions. In addition to the patch code files, the generated code for the original firmware functions in the *fw_org_functions.c* file is compiled. For this, we invoke *Qualcomm's* compiler for *Hexagon* from the available SDK [28], giving a set of object files as output.

Note that we ignore custom function annotations to overwrite functions in this step (we show how these work later). These only influence the combination of the patch code with the base firmware file but not the compilation process.

9.2.5 *Linking*

The individual object files need to be combined into a single binary. This is done by passing them to the linker, again, we utilize *Qualcomm's* own implementation from the SDK [28]. The framework hands the linker files generated in the previous steps to the tool in addition to the object files, defining the symbols needed for *fw_org* functions and the base firmware binary.

Listing 3: Example of a generated *fw_org* function

```
1 void* memset_fw_org(void* ptr, int value, unsigned int num) {
2         asm(
3                 "{ if (r2 == #0) jump:nt sym_0x9130750\n\t"
4                 "r7 = vsplatb (r1)\n\t"
5                 "r6 = r0 }\n\t"
6                 "{ jump sym_0x913065C }"
7         );
8 }
```

The main work done by the linker can be summarized as:

The framework hands all compiled files and control scripts to the linker, which resolves all symbols in the patch code but does not yet combine it with the base firmware.

- Usual linking process: link object files together, resolve undefined symbols with symbols defined in other files (e.g. calls of functions in other file, data references)
- Resolve calls to firmware functions and references to data of the base firmware by using the symbol definitions in *fw_wrapper.lcs*
- Resolve symbols used in the inline assembler code of *fw_org* functions with the symbol definitions in *fw_org_functions.lcs*

As a result, the linker generates a single file containing all the patch code and data sections in ELF format (*fw_patch.elf*). This file contains no more undefined symbol references, the functions in it are in the final version and will not be changed by later steps anymore.

9.2.6 *Patching*

After the patch code is in its final form, it needs to be combined with the base firmware. Therefore, we have to merge the sections of the two ELF files. In addition, changes to the base firmware code to overwrite functions need to be done. This step then results in a patched firmware ELF (*patched.elf*) file, containing all the final modem instructions and data.

9.2.6.1 *Source code annotations*

The patching step merges patch code and base firmware. It is directed by annotations in the patch code, e.g. to overwrite functions.

For overwriting functions, the framework's patching process needs to know which patch code should replace which function in the base firmware. To make this as simple as possible, the framework uses a mechanism similar to the address specification of base firmware functions in the firmware wrapper header. To specify a function replacement, patch functions can be annotated with an attribute to overwrite another function. This annotation includes the destination function as a parameter.

Listing 4 shows an example of how to define the function *memset_hook* so that it replaces the function *memset* (lines 1 to 5) of the base firmware. Lines 7 to 19 use another feature included in the framework, the *pointer_table* attribute. It allows to directly place a pointer to a function in a table of function pointers. This scheme is often used for handler functions which dispatch further message processing to a next function depending on the message type, using this value as index to a table of pointers which contains the processing functions. It needs the table base and an offset value as parameters. Note that patches also use it to replace a simple function pointer (not inside a table) by using an offset of zero.

9.2.6.2 *Generation of a list of required patches*

As a first substep to prepare the actual patching, the *patcher.py* script generates a list of required changes. Afterwards, it processes the patches contained in this list together.

To first step of the patcher is to identify which modifications are required.

The first patch added to the list is the task to include all memory sections of the patch ELF file into the base binary. Then, character string replacement tasks are added for the firmware version text as well as the build time and date, if this is requested by defining a new firmware name in the project's *Makefile*.

The framework parses the patch code source files the *C* code parser (*pycparser* [6] with *pycparserext* [19]) to find the annotations on function definitions as discussed in the previous section (9.2.6.1). For each found *overwrite* directive, it adds a task to write a jump instruction to the start address of the function which should be replaced and the address of the new function as destination address to the list. Similarly, for all *pointer_table* directives, it appends a binary replacement task for the content of the pointer table entry with the patch function's address to the list.

It then hands the resulting list to the *binary_patcher.py* script to perform the actual patching of the ELF file.

9.2.6.3 *ELF section combination*

To combine a patch ELF file with the base firmware ELF, the framework needs to insert all segments of the patch file into the base file.

Listing 4: Examples of function annotations

```
__attribute__ ((overwrite("memset")))
void* memset_hook(void* ptr, int value, unsigned int num) {
    memset_counter++;
    return memset_fw_org(ptr, value, num);
}

__attribute__ ((pointer_table("qmi_ping_svc_req_handle_table", 0x21)))
unsigned int services_response_handler (
    void*             clnt_info,
    void*             req_handle,
    unsigned int      msg_id,
    void*             req_c_struct,
    unsigned int      req_c_struct_len,
    void*             service_cookie)
{
    /* custom code here */

    return 0;
}
```

Two main cases need to be considered in order to insert a segment: it can either contain data for memory already defined by a base segment or lie in a different address range than all existing segments.

The sections of patch and base ELF files need to be combined carefully to produce the patched firmware.

In the latter case, we can simply append it to the binary, maybe with padding bytes before it to deal with alignments inside the ELF file. We have to add its metadata to the header structure.

However, in most cases, the new segment's memory address range will overlap with an existing segment. Then, we split this existing segment in up to three new segments, one with the same address range as the patch segment and potentially one before and after this. We correct the ELF header structure accordingly and replace the fitting segment with the new data.

Note that in case a new segment ends in another existing base file segment than it starts, an error will be thrown as this is very likely due to a size overflow error of the patch code and not intended, another target segment as the one chosen for the patch code (or data) would be overwritten.

9.2.6.4 *Other binary patches*

The framework handles patches to the base firmware data depending on their type. Simple data word patches (for example for the *pointer_table* attribute) replace the content at their target address with a new data word. String patches used to adapt the firmware version strings change a sequence of bytes beginning at a start address until the terminating null character.

We need to perform additional modifications to realize directives from the annotations.

More complex are *jump* patches used to overwrite a function. These generate the required instruction encoding for a jump from the instruction's address to a destination address. If the jump distance is too large for the immediate value in the jump instruction encoding, a constant extender (Section 6.1.3) is used. For this, we have to write one or two data words containing these instructions to the target address of the patch. The original firmware function now immediately jumps to the patch code version and, as a result, is overwritten with the new version.

9.2.7 *Image generation*

As a last step in the firmware patching process, we need to generate an image file which can be written to the device. Therefore, the framework needs to convert the patched ELF file into an image file with the same structure as the input image, which we describe in Section 9.1.

At the beginning of this step, our code extracts the files needed for a firmware image from the ELF file. For that, we write every segment contained in the file to a separate file named in the form *modem.bXX*. Additionally, we copy the firmware metadata stored in the first two segments (ELF header and integrity/authenticity data)

of the ELF binary to the *modem.mdt* file. This is handled by the *pil-splitter.py* script which is again largely based on the code available at [37]. We applied only minor changes to the script, mainly refactoring of the code.

As a last step, the framework generates an output firmware image from the ELF file. This also includes correction of section hashes.

For the MBA to accept the firmware data, the hash values in the firmwares header table in the metadata need to be valid for the loaded segment files (Section 7.1). Since the hashes are for now simply copies of the values for the base firmware segments and the patch code changed these, all hashes need to be computed again and replace the old values at the matching offsets in the hash table. The *pil-patcher.py* script (again based on code available at [37]) takes care of this correction.

Note that we would also need to recompute the signature in the metadata, used for firmware authentication (Section 7.2), in the general case. However, since we do not have the right private keys to generate valid signatures which would be accepted by a device and we therefore focus this work on devices which do not use a signature or skip the check, no signature needs to be calculated.

In order to get the final image file, we need to write the individual files to a single image file of a FAT16 partition. For that, we overwrite the modem's firmware files in a copy of the base image with the output files of the previous step by using the *mcopy* tool. This image now contains the patched modem firmware, together with all other files stored on the partition, like the MBA code or firmwares of other subsystems. It is ready to be copied to a target device.

9.3 PORTING TO NEW TARGETS

Adapting to a new target should be as simple as possible and should require the smallest amount of work possible. A target in our context means a certain firmware image for a device in a specific version and, therefore, porting to another target includes both, adapting to another firmware version of the same device used before and targeting a completely new device.

To simplify this, the patching framework itself is independent of the target as long as it has a similar architecture and is based on the *Hexagon* DSP, which is true for all recent modem models of *Qualcomm*. In Section 9.2, we saw that the patching framework uses the following inputs: the patch code, the firmware image and a firmware wrapper header. By design, the patch code with its annotations is independent of the target and the firmware image is the target itself. Consequently, we only need to port the wrapper header file in order to support a new target.

We developed a tool which assists in porting to other targets. It transfers a wrapper header file from one firmware to another.

Since it is tedious to adapt the wrapper header by hand, which basically means to find back the correct locations of all known functions in the new target firmware, we developed scripts to automate this

process. They are able to automatically generate a wrapper header for a new target, given a base firmware image with its wrapper. Obviously, it can only find functions declared in the base wrapper for the new target.

As additional feature, the scripts are able to locate functions contained in an object file in a firmware image. This is useful for the initial reverse engineering, so to identify new functions which were not previously labeled in another firmware wrapper header, for example if one gets access to pre-compiled library files used for *Qualcomm's* firmware builds.

9.3.1 *Implementation*

The large similarity between the different target firmwares, originating from the same source and many parts coming even from the same pre-compiled library, simplifies the implementation of this function search mechanism. As a consequence, it is sufficient to compare functions at a binary level, with fixed and encoded instructions, rather than at a higher level by analyzing the behavior of the function as it is, for example, done by the *BinDiff* tool (available at [45]). Since the location of the function is known in the base firmware image, we can extract its byte sequence and try to locate the same sequence in the new target firmware.

The biggest problem with this are relocations: the linker adapts some of the immediate values used in instructions. For example, offsets in *call* instructions depend on the destination's function address as well as the call's instruction location. Both of these might be different in distinct targets and, therefore, the offset is likely also different. Similar situations occur for other instructions with code offsets or data references. As it is hard to know which immediate values contain relocations and which contain fixed values, we chose to mask out all immediate values for the search as a simple strategy to deal with this problem. This is done by determining the bits containing immediate values of each read word (32 bits, one instruction) and setting them to ones. By doing this for the base function and for the data of the destination firmware, all immediates are identical and not part of the comparison anymore. The remaining structures, that means the sequence of instructions and used registers, still contain enough information to identify most functions uniquely.

Our script uses simple binary comparison to identify functions, with masking out of all immediate values in instructions.

A second problem is that we do only know the starting address of functions but not their length. Due to jumps in the assembly code and multiple return instructions used for different code paths of the function, it is not trivial to determine the correct end of a function. However, we need this information to generate a search sequence for a function containing exactly all its instructions. As a result, we can only create a search sequence not perfectly matching the length of

the function. If it is too short, multiple other functions will match, if it is too long, it will exceed the end of the function, leading to no results as in the new firmware the order of functions might differ. To solve this issue, the script repeats the function search with different search string length, starting with a small value which will match the function for sure. The next iteration is determined with the following logic, with the maximum search length initialized to the remaining file size after the function:

- Unique match: found the function, return match position
- Multiple matches: double search sequence length, limit with maximum search length
- No match: set maximum search length to current length minus one, new length is arithmetic mean between last matching length and maximum search length

In addition to the stop condition of a unique match, we stop the search when multiple matches were found for the current maximal length or no matches were found for the minimal length. In other words, this is done whenever multiple matches are found for one length but no matches for one more instruction in the search sequence. In this case, it is not possible to conclude the correct match with the algorithm, either the function is not present (in the same form) in the new target or two functions only differ in the immediate values used in their instructions.

For a complete wrapper header transfer, the script repeats this for every function declared in the header. It then generates the new wrapper from the base wrapper and the obtained mapping between the functions in the base firmware and the new target firmware.

9.3.2 *Limitations*

The developed tool can only port function locations automatically, other symbols have to be identified by hand for now.

Obviously, the described method to transfer the wrapper header from one firmware to another only works for functions. Other symbols like data (global variables) still needs to be located by hand. However, as the data declared in the wrapper will be the data encountered while reverse engineering the functions in the wrapper and these locations are known, thanks to the automatic transfer scripts, it is not complex to locate the data addresses by hand afterwards. Target locations for the patch code and data also need to adapted by hand for a new target.

In addition, we saw that the implementation can return without a unique match for a function. The algorithm cannot assign an address to such functions. If it could restrict the possible addresses to a small set with a size smaller than a threshold, for example up to four possible locations, it will indicate these in the generated wrapper to simplify the necessary search by hand.

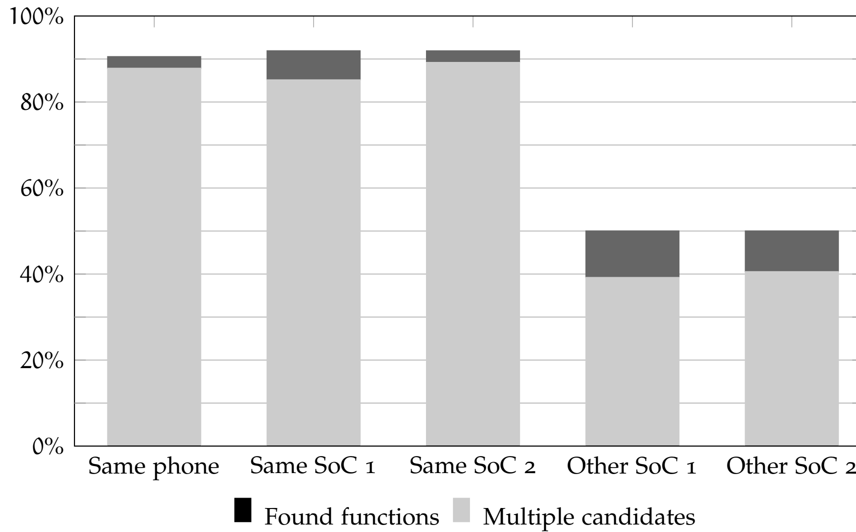


Figure 11: Percentage of automatically detected functions. Table 16 defines the scenarios and gives the exact numbers.

9.3.3 *Matching performance*

In this section, we evaluate the matching performance, meaning how well the automatic detection works in different cases. For this, we ran the wrapper transfer scripts on different new targets, with the main target firmware of this thesis as base firmware: version *M3.12.15* for the *Asus PadFone Infinity 2 (A86)*. At the time of this evaluation, 89 symbols were declared in the wrapper header. 74 of these were functions, 2 define the memory locations to place patch code and data, 5 indicate the position of version strings and the remaining 8 are variables of the firmware.

Table 16 lists the performance results of the function matching. Figure 11 additionally visualizes the percentages of found functions and functions with candidates given for the different targets, with the targets referring to the ones of the table.

Table 16: Automated porting results to different targets, from the base firmware *M3.12.15* for the *Asus PadFone Infinity 2 (A86)*, 74 declared functions

	Same phone	Same SoC 1	Same SoC 2	Other SoC 1	Other SoC 2
Phone	Asus PadFone Infinity 2 (A86)	Google Nexus 5	LG G2 (D802)	OnePlus One	Asus ZenFone 2 Laser (ZE601KL)
Version	M3.15.4	2.0.50.2.28	1.0.190036	DI.3.0.c6-00241	1.16.40.1524
SoC	MSM8974	MSM8974	MSM8974	MSM8974AC	MSM8939
Found Functions	65	63	66	29	30
Multiple candidates	2	5	2	8	7

We can see that for the same phone a large portion of the labeled functions is detected in another firmware version (roughly 88 %, respectively almost 91 % if we also count functions with candidates suggested). Running the automated porting for another phone with the same SoC results in approximately the same amount of matched functions. This is because *Qualcomm* ships large parts of the firmware precompiled in libraries to the manufactures for their SoCs and therefore our algorithm, working by direct instruction compare, works well.

The tool works well if the base and the target firmware are for the same SoC. More advanced techniques are required for porting to new SoCs.

On the other hand, if we do the wrapper transfer for a target with a different chip, many libraries will be compiled again to adapt to the internals of the SoC and, therefore, loose the instruction level similarity as the compiler has some freedom in translating the source code to machine level code. This explains the significantly lower matching rates for these targets of ~40 %, respectively 50 % if we include functions with suggested candidates. Since the matches include important functions like *memset* and *memcpy*, the scripts can at least provide a starting point for the reverse engineering.

An example of found function suggestions is a pair of a function to cipher and the corresponding decipher function. Their implementations contain indeed the same instructions and only differ by the called functions (which are again similar). As a consequence, our script finds both functions in the new target firmware as candidates for both base functions and the ambiguity has to be resolved manually.

10 IMPLEMENTED PROJECTS

We developed a couple of firmware modifications to test the framework, exemplary demonstrate its functions and make first use of it. As it can be seen in Table 17, they range from simple test patches (*func_counter_snprintf*) over tools to simplify debugging and reverse engineering (for example *mem_access*) up to real final applications (*lte_mac*, ...).

It is possible to reuse already developed patches by including them in another patch project. This is utilized by the *all_app* project to bundle all the different projects into a single patched firmware which supports all features included in an Android application that we implemented. This app realizes a user interface showing data sent by the patched modem and allows the user to send inputs to the custom code inside the modem.

10.1 INFORMATION EXCHANGE

Before looking at the implemented projects, we now have a detailed look on how the modem communicates with the user space application since the same mechanism is used for all of the projects and can also be used as basis for information exchange for new patch projects.

Communication with patch code is based on the QMI protocol.

Figure 12 shows the involved entities and protocols. The modem communicates with an Android kernel module over *Qualcomm's* QMI protocol. This kernel driver then interfaces to user space applications, allowing them to send messages to the modem or receive messages from it. We explain all details of this figure in the following sections.

Table 17: Implemented projects

Name	Description
func_counter_snprintf	Counting of calls of standard C functions, forwarding of *snprintf* outputs
mem_access	Direct access (read & write) to the modem memory
lte_mac	LTE MAC Layer messages sniffing
lte_sec	LTE security: key extraction, cryptography functions
channel_estimation	Access to LTE channel estimation matrices
all_app	Metaproject to combine all patches used by the app

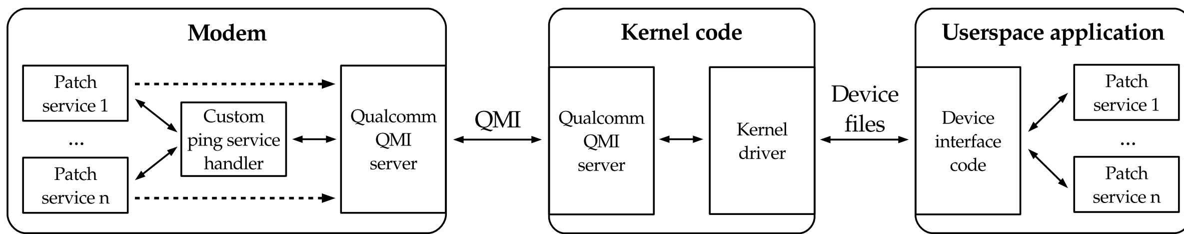


Figure 12: Information exchange with patch code in the modem

10.1.1 *Using the QMI protocol*

It would be possible to realize communication between the main system and the modem by directly accessing the memory regions shared between them. However, we would then need mechanisms to signal new messages, indicate which memory areas can be used again as the message we wrote there is completely processed and so on. Also, we would need to be careful not to use memory ranges currently needed by the modem's original operation.

As the modem's firmware already implements all these functions with a stack of protocols as seen in Section 7.5, it is much simpler to reuse these. Therefore, we base our communication on the QMI implementation.

On the other hand, registering a new service with the QMI implementation on the modem is complex since a large set of structures describing the service and all possible messages is required, with many data fields each. To generate the correct structures for an own QMI service, lots of reverse engineering work to identify the meaning and encoding of all of these data structures is necessary.

Adding an own message type to QMI is complex, thus we leverage an existing ping service and tunnel our messages through it.

Luckily, the modem contains a ping test service solely used for debugging purposes but active in all observed firmware versions. It supports a message type with an array of 8,192 bytes as a data field. Normally, the modem replies to this message with an echo message containing the same data. This allows us to overwrite the handler for this message and encode our own messages inside this array. Unfortunately, then we cannot use the message type of QMI anymore to differentiate between message destinations, as all messages need to be sent to the ping test service. Also, the decoding into different data fields with appropriate types of *Qualcomm's* code cannot be used.

As a result, we overwrite the receive function for this data echo message of the QMI test service and tunnel our own protocol through it. As Table 18 indicates, this protocol only adds a destination service identifier to the raw message content. The different custom services are responsible for encoding and decoding their data in the remaining data bytes. If a message is larger than the available length (8,192 bytes), it has to be split into multiple messages. However, for most purposes the available size should be sufficient.

The modem code implementing this protocol only needs a table which associates handler functions with service identification numbers. If this is configured, incoming messages will be automatically sent to the appropriate handler function which can then further decode the service specific message content. This handler function also gets a pointer to a buffer where it can write a response message. It will be sent back directly after the handler function returns. In addition, patch code can always send QMI indications (described in Section 7.5.2) to the main system. In Figure 12, these are indicated by dashed arrows in contrast to the solid drawn arrows of the request/response message path.

10.1.2 *Kernel module*

Since the QMI server on the Android system is only accessible from kernel space, we included a kernel module responsible to forward message between user space applications and the QMI server. Therefore, it connects to *Qualcomm's* QMI kernel implementation and registers to receive all messages from the ping test service. On the other side, the user space application can communicate with it over usual device files.

A kernel module routes messages between QMI and user space applications, accessible over a device file interface.

It was desired to keep the kernel driver code as simple as possible to reduce the possibility of critical kernel errors and to keep the module as flexible as possible. Therefore, the driver only implements the message routing between the device file interface for the user space applications and the QMI interface to the modem. It forwards messages as they are received and not modified, all protocols have to be implemented in user space. As a result, the kernel driver is independent of the used tunneling protocol. However, only communication with the QMI ping test service is possible and no other requests can be issued.

The driver creates two device files. Reading */dev/seemoo_qmi_status* returns all available status and error messages from the driver as human readable ASCII text. The */dev/seemoo_qmi_data* file is used for the actual message passing. Every read returns the complete content of the next available message or nothing if no more messages are available. A write to this file is interpreted as a new message and all data bytes written in a single write operation will be sent as one QMI message.

Table 18: Message structure to tunnel messages through QMI ping service

Byte	0	1	2	3	4	...	MSG_SIZE - 1
Content	SVC_ID				Service specific message		

10.1.2.1 *QMI service identification problem*

As seen in Section 7.5.2, QMI uses two numbers to identify a service instance: a service ID distinguishes between different types of services (*0xF* for the ping service) and an instance id (*1* for the modem) differs between instances of the same service.

The identification information of the ping service is not unique. We fixed Qualcomm's kernel code to communicate with the modem.

The ping service exists on multiple subsystems of the smartphone and not only the modem. Unfortunately, *Qualcomm* assigned the same instance ID to many of them, making the identification scheme not unique anymore. The Android kernel implementation simply returns the first fitting service it finds when requesting connection to a service which was not the desired instance running in the modem. Note that the QMI message router uses information stored in the handle structure returned by the connection function, the IDs are only used to identify the service in the connection phase. Therefore, once a connection with the right service is established, all messages will be routed correctly.

As a quick fix for this bug, we changed the connection function to return the last found service instance instead, in case multiple matching service instances are found. This works since the instances are always found in the same order and the modem is the last of these. However, a cleaner solution would be to try all the services by sending a test message and concluding from the response message whether or not it is the modem's instance, either by checking if we received an echo or by adding a special test response for this purpose in the modem code.

10.1.2.2 *Indications and request/response message bug*

Qualcomm's QMI server implementation, contained in the Android kernel, crashes when responses to request messages and indications arrive interleaved at the same registered client. To overcome this problem, our kernel module registers two clients with the server. The first of these is used for the request/response scheme while the second client handle is only used to receive indications and to send registration messages for these.

As a consequence, the kernel driver needs to know whether a message is a normal request or a registration for an indication to choose the appropriate client handle. For this, on the user space interface, we use the highest bit of the service ID. This breaks a bit what we said in the introduction. Now, the first bit of byte four (the data is encoded in little endian) of the message has a special meaning and the kernel driver is no longer independent of the messages content. However, only this bit is interpreted, the rest of the encoding remains in user space. The driver also sets this bit for incoming indication messages, allowing the user space application to differ between response mes-

sages and indications. Consequently, the service ID should contain only 31 bits and not use the highest bit of the data word.

10.1.3 *User space implementation*

We implemented the user space code as a usual Android application in *Java*. A class called *SeemoQMI* implements the complete device file interface to the kernel module. It abstracts from the interface itself and the used protocol. Other classes can add a listener for the status and error messages from the driver. Service implementation can register a listener for a specific service ID for incoming response messages and another listener for indications. Sending a message can be requested easily by a method call with the destination service and payload as parameters. Remember that it is the task of the services to encode and decode the actual data it wants to transmit in the message in an appropriate way.

Our Android application implements the last part of the communication scheme, the user space code.

Thereby, we now have communication between a patch service implementation on the modem and a corresponding service implementation in the Android application.

10.2 FRAMEWORK USAGE EXAMPLE

The purpose of the *func_counter_snprintf* project is to give a simple example of how to use the framework. During the development, we also used it to check if the framework is working as desired. Therefore, it includes examples of all features supported by the framework.

The example func_counter_snprintf demonstrates the capabilities of the patching framework.

It consist of two parts. First, it hooks standard C functions and adds call counters to them. These counter values are sent back as response to a QMI request message. Second, it modifies the *snprintf* function used by the modem firmware to format various status strings such that, on every call of this function, a QMI indication with the resulting formatted string is sent, in case a client previously registered for this. The counters and the *snprintf* messages can be viewed in the developed Android application.

Listing 5 shows some example messages obtained by the *snprintf* function hook, only a small set of different message from the output is printed here. We can see that, even in the firmware version deployed on customer devices, a lot of parts of the firmware use the *snprintf* function to print debug and status messages. In line one to five, we can see location based information. Lines seven to nine show internal messages from the LTE and Wideband Code Division Multiple Access (WCDMA) modules. The remaining messages contain network cell discover and selection information.

10.3 MODEM MEMORY ACCESS

In Section 6.2.1, we saw that access rights are given to different subsystems and protection units prohibit access from one subsystem to a memory region of another subsystem (except for shared regions between these two entities). As a result, the Android system cannot access internal memory data of the modem during runtime. However, for reverse engineering it is crucial to be able to look at memory data of the system during runtime, for example to observe values assigned to variables or to find destinations of jumps to a variable function pointer.

With the mem_access project, we can read and write memory from the context of the modem . This includes regions only accessible by this subsystem.

Therefore, we implemented a service allowing to access memory from the modem's processor. It enables to issue requests for a memory address together with the amount of bytes to read. The service implementation then reads these bytes and sends back a response message with the data. In addition, it supports write requests. For this, a message with the start address and the bytes to write is sent. The modem patch code then writes this data to its memory. Note that the service does not check the memory addresses passed to it. The user is responsible to only use memory regions for which the modem has the appropriate access rights for the requested operation.

Listing 5: Captured example snprintf output messages

```
1  TPC: Lat:47.000000, Lon:4.000000, Alt:0.000000, Punc:700000.000000,
       ↪ AltUnc:10000.000000
2  TPC: Sub:1, TimeValid:0, Wk:0, msec:0
3  TPC: SendPosReport: TimeTickMsec=77242, GpsTimeValid=0, GPSWk=0, GPSTow=0
4  TPC: SendPosReport: PosValid=1, Lat=47.000000, Lon=4.000000,
       ↪ Punc=700000.0
5  TPC: Sending position update to user:ALE
6
7  [ LTE: pci 279 fcn 1501 ]
8  [ WCDMA: psc 144 fcn 10564 ]
9  [ LTE: pci 488 fcn 2825 ]
10
11 Cell Change ClosestCellID:20 20 ClosestPosEst:20 21500000.000000
12 ME updating LTE CellDB: Valid=1, MCC=208,MNC=10, cellID=ac06, tac=c15c
13 ME updating LTE CellDB: Valid=1, MCC=208,MNC=10, GlbCellID=44038,
       ↪ PhyCellID=488, tac=49500
14 MCC:208, MNC:10, CellId:44038, Physical CellId:488, TAC:49500, DL
       ↪ Freq:6300
15 MCC:208, MNC:10, LAC:49542, CellId:31702, Band:3, BsIc:50, Arfcn:612
16 GSM PhyId, Bsic:50
```

Due to the used communication scheme with the main system (Section 10.1.1), the maximal amount of memory read or written at once is limited to:

8192 bytes	(maximum message size)
− 4 bytes	(SVC_ID)
− 4 bytes	(start address)
− 4 bytes	(length)
= 8180 bytes	

Multiple requests have to be sent if a larger amount of data needs to be read or written. Since all messages contain the complete memory address, we can issue these fast after each other and even if they are received by the modem out of order due to the underlying QMI protocol, it will still process the requests correctly. The developed Android application also supports this service, either by directly showing the read bytes in the Graphical User Interface (GUI) and typing the bytes to write or by reading from and writing the data to files (Intel HEX and raw binary data files are supported).

10.3.1 *Security aspects*

Obviously, this service breaks most of the modem's security mechanisms. An Android user space application gets the same memory access permissions as the basebands code itself. Consequently, a malicious application accessing this service could perform many unwanted things, like monitoring all network traffic or accessing encryption keys. However, the patch is only intended for debugging and reverse engineering and not for a large set of phones in real everyday use. An attacker (if he would really exist) could only get access to test equipment used during reverse engineering without sensitive user data being processed.

This patch introduces large security issues. It is no problem as the example is only intended for testing devices.

The same discussion holds for other patch projects implemented in this thesis. Especially, the *lte_sec* project (Section 10.5) also opens doors for attackers by extracting cryptography keys.

10.3.2 *Outlook: Debugger*

Often it is hard to identify what a piece of assembly code does and to infer its purpose by solely statically looking at the instructions. With the presented *mem_access* service, it is possible to read memory locations during runtime, leading to first dynamical analysis possibilities. Unfortunately, we cannot see the memory content at a specific time step, for example when the processor is executing a certain function of the code. Also, local variables of a function on the stack are almost impossible to observe and we cannot read parameters passed to functions in processor registers at all.

Therefore, a complete remote debugger implementation offering single step execution, execution breakpoints, access to the current processor state (registers) and similar common debugger functionality would be nice to have.

Regrettably, implementing it is quite complex due to *Hexagon's* hardware and software thread execution. We need further details on how these are managed and how the used *QuRT* operating system interacts with them to implement a proper debugger. Also, the support of the DSPs architecture for debuggers needs to be reviewed in detail as Qualcomm keeps the documentation of this in the reference manual minimal (compare [33]).

At last, the modem system sends RF signals at high power in licensed bands. Breaking inside signal processing code might be dangerous and leave the device in a sending state with radiation of unintended signals. As a consequence, it could disturb the operation of the mobile network and the service quality for other users might be influenced.

Note that similar cases could occur for all kinds of modifications on the modem's firmware as presented in this work. Modified devices should only be operated in a laboratory environment and turned off whenever no experiments are conducted.

10.4 LTE MAC LAYER FRAME SNIFFING

The *lte_mac* project's objective is to get access to messages from the LTE MAC layer in order to be able to analyze these. It allows us to observe messages send in a real deployed network and to see how the device reacts. Modifying the modem's firmware with this patch enables the analysis of MAC layer communication on cheap devices, removing the need for special development equipment, one of the main purposes guiding this thesis.

LTE MAC layer sniffing, provided by the lte_mac project, allows real-time analysis of the communication between a device and a base station in Wireshark.

Possible applications include:

- Understand (better) how LTE works by looking at actual data, also for teaching purposes
- Demonstrate what a malicious attacker could do with access to modem firmwares (all user data is passed here)
- Study differences between deployed networks (for example per operator)
- Analyze network problems
- See how a network reacts to attacks (e.g. send on resources assigned to other users, try to authenticate as other user, more sophisticated attack scenarios, ...)
- Observe what a device really sends to a network (at a low level)

Some of these applications require additional modifications of the firmware. We see that this sniffing project can be useful in many different scenarios, the list contains only some ideas from the author, many more applications might exist.

Figure 13 shows an example capture of LTE MAC frames in Wireshark obtained with this project. The capture includes the initial connection and configuration negotiation of the phone with the network as well as receiving an SMS message. The figure focuses on a few interesting messages, other frames are omitted for better readability. Frames in the uplink are highlighted in purple in contrast to the white downlink frames. The detailed packet window displays the first RAR message to get an idea of how the data can be analyzed.

Before the connection establishment of the device, we can see System Information Block (SIB) messages with configuration parameters of the network. In frame three to 31, the device then performs a RACH procedure and after that negotiates connection parameters with the network, including the used security features (frames 30 and 31). This is followed by a phase of inactivity, we can see paging requests for other devices in frames 53 and 54. In frame 56 then the device received a paging request for itself, triggering it to establish an active connection to the network with a RACH procedure. In this connection, the network then delivers the SMS message to it. Note that we shortened the message delivery here and consists of many more messages than shown. The capture also includes unsuccessful message transmissions. For example, in the second RACH procedure, the preamble (*Msg1*) is sent three times. Similarly, the UE repeats the RRC connection request (*Msg3*) (frames 61 and 62).

10.4.1 *Limitations*

Only MAC frames intended for the device are forwarded as the PHY layer solely listens on the resource blocks assigned to the device or for broadcast messages (remember the resource grid from Chapter 3). By reverse engineering the details of the physical layer, it should be possible to modify it to receive all messages from the connected base station on the same channel (downlink). Unfortunately, since LTE uses a different modulation scheme in the uplink as in the downlink and for FDD LTE the uplink uses distinct frequency resources, it is impossible to receive uplink messages send by other phones in the network. The only way to overcome this is to largely rewriting the baseband's firmware with demodulation capabilities for the uplink modulation scheme. As a result, this work focuses on giving access to the MAC layer communication between a single device and the network, instead of monitoring the whole communication in the network.

We can only observe messages intended for the device, no other network traffic.

No.	Time	Protocol	Length	Info
1	0.000	LTE RRC DL_SCH	77	SystemInformationBlockType1
2	0.001	LTE RRC DL_SCH	108	SystemInformation [SIB2 SIB3]
3	0.002	MAC-LTE	59	RACH Preamble chosen for UE 7315 (RAPID=10[GroupA], attempt=1)
4	0.002	MAC-LTE	68	RAR (RA-RNTI=1, SFN=35 , SF=6) (RAPID=10[GroupA]: TA=20, UL-Grant=54808, Temp C-RNTI=29841)
5	0.003	LTE RRC UL_CCCH	69	RRCConnectionRequest
6	0.008	LTE RRC DL_SCH	77	SystemInformation [SIB5]
7	0.008	LTE RRC UL_CCCH	69	RRCConnectionRequest
8	0.009	LTE RRC DL_CCCH	102	RRCConnectionSetup
9	0.009	MAC-LTE	66	DL-SCH: (SFN=37 , SF=8) UEId=7315 (Timing Advance) (Padding:remainder)
10	0.010	LTE RRC UL_DCC…	167	RRCConnectionSetupComplete, Attach request, PDN connectivity request
11	0.010	RLC-LTE	66	UEId=7315 [DL] [AM] SRB:1 [CONTROL] ACK_SN=1
12	0.112	LTE RRC DL_DCC…	120	DLInformationTransfer
13	0.112	RLC-LTE	111	UEId=7315 [UL] [AM] SRB:1 [CONTROL] ACK_SN=1
14	0.113	LTE RRC UL_DCC…	147	ULInformationTransfer
15	0.114	RLC-LTE	70	UEId=7315 [DL] [AM] SRB:1 [CONTROL] ACK_SN=2
16	0.126	LTE RRC DL_DCC…	100	DLInformationTransfer, Security mode command
17	0.228	LTE RRC UL_DCC…	223	UEId=7315 [UL] [AM] SRB:1 [CONTROL] ACK_SN=2 || ULInformationTransfer
18	0.228	RLC-LTE	70	UEId=7315 [DL] [AM] SRB:1 [CONTROL] ACK_SN=3
19	0.229	LTE RRC DL_DCC…	100	DLInformationTransfer
20	0.230	RLC-LTE	195	UEId=7315 [UL] [AM] SRB:1 [CONTROL] ACK_SN=3 || UEId=7315 [UL] [AM] SRB:1 [DATA] sn=3 [122-bytes..
21	0.230	LTE RRC UL_DCC…	109	ULInformationTransfer
22	0.231	RLC-LTE	70	UEId=7315 [DL] [AM] SRB:1 [CONTROL] ACK_SN=5
23	0.373	LTE RRC DL_SCH	74	SystemInformation [SIB6]
24	2.819	MAC-LTE	70	DL-SCH: (SFN=316 , SF=4) UEId=7315 (Timing Advance) (Padding:remainder)
25	4.649	MAC-LTE	70	DL-SCH: (SFN=502 , SF=2) UEId=7315 (Timing Advance) (Padding:remainder)
26	4.973	LTE RRC DL_DCCH	85	UECapabilityEnquiry
27	4.974	RLC-LTE	159	UEId=7315 [UL] [AM] SRB:1 [CONTROL] ACK_SN=4 || UEId=7315 [UL] [AM] SRB:1 [DATA] sn=5 [86-bytes..
28	4.975	LTE RRC UL_DCCH	183	UECapabilityInformation
29	4.975	RLC-LTE	70	UEId=7315 [DL] [AM] SRB:1 [CONTROL] ACK_SN=7
30	4.976	LTE RRC DL_DCCH	85	SecurityModeCommand
31	5.090	LTE RRC UL_DCCH	159	UEId=7315 [UL] [AM] SRB:1 [CONTROL] ACK_SN=5 || SecurityModeComplete
51	13.295	RLC-LTE	167	UEId=7315 [UL] [AM] SRB:1 [CONTROL] ACK_SN=11
52	13.297	PDCP-LTE	85	UEId=7315 [DL] [AM] SRB:1 [DATA] (P) sn=10
53	95.763	LTE RRC PCCH	66	Paging (1 PagingRecords)
54	105.647	LTE RRC PCCH	66	Paging (1 PagingRecords)
55	128.159	LTE RRC DL_SCH	87	SystemInformationBlockType1
56	197.783	LTE RRC PCCH	66	Paging (1 PagingRecords)
57	197.785	MAC-LTE	59	RACH Preamble chosen for UE 7315 (RAPID=14[GroupA], attempt=1)
58	197.891	MAC-LTE	59	RACH Preamble chosen for UE 7315 (RAPID=24[GroupA], attempt=1)
59	197.893	MAC-LTE	59	RACH Preamble chosen for UE 7315 (RAPID=24[GroupA], attempt=2)
60	197.895	MAC-LTE	68	RAR (RA-RNTI=1, SFN=364 , SF=6) (RAPID=24[GroupA]: TA=16, UL-Grant=54808, Temp C-RNTI=29021)
61	197.897	LTE RRC UL_CCCH	69	RRCConnectionRequest
62	197.909	LTE RRC UL_CCCH	69	RRCConnectionRequest
63	197.911	LTE RRC DL_CCCH	102	RRCConnectionSetup
64	197.913	MAC-LTE	66	DL-SCH: (SFN=366 , SF=9) UEId=7315 (Timing Advance) (Padding:remainder)
75	197.970	RLC-LTE	137	UEId=7315 [UL] [AM] SRB:1 [CONTROL] ACK_SN=4 || UEId=7315 [UL] [AM] SRB:1 [DATA] sn=3 [64-bytes..
76	197.971	PDCP-LTE	167	UEId=7315 [UL] [AM] SRB:1 [DATA] (P) sn=4 ..37-bytes] |

```
> RAR Headers:  (1 RARs)
v RAR Body: (RAPID=10[GroupA]: TA=20, UL-Grant=54808, Temp C-RNTI=29841)
    0... .... = Reserved: 0x0
  v .000 0001 0100 .... = Timing Advance: 20
      > [Expert Info (Note/Sequence): RAR Timing advance not zero (20)]
  v .... 0000 1101 0110 0001 1000 = UL Grant: 54808
      .... 0... = Hopping Flag: 0
      .... .000 1101 011. = Fixed sized resource block assignment: 107
      .... ...0 000. .... = Truncated Modulation and coding scheme: 0
      ...1 10.. = TPC command for scheduled PUSCH: 6 dB (6)
      .... ..0. = UL Delay: 0
      .... ...0 = CQI Request: 0
    Temporary C-RNTI: 29841
  Padding data: b300
  [Padding length: 2]
```

Figure 13: LTE MAC sniffing capture in Wireshark

10.4.2 *Choice of the target layer*

The main aim of the *lte_mac* patch is to enable research work directly on the MAC layer but more applications are possible. In Chapter 3, we introduced the protocol layering of LTE. The MAC layer resides directly on top of the PHY layer. As a result, MAC frames include the data of all higher layers and can therefore also be analyzed by decoding the MAC messages. For example, this patch also intercepts layer 3 messages as the *Azenqos* tool presented in the related work section can do. However, this requires to implement the complete decoding of the messages and all intermediate layers. Even though this can be done easily by using an existing implementation, we use a *Wireshark* dissector for this. In case a project specifically targets a higher layer, it should also be considered to let the modem do the decoding and to intercept the messages directly on this layer. This could, for example, be beneficial in a case where the whole project needs to run directly on the smartphone over a long time and is power constraint (on battery) or when high throughput data needs to be analyzed. On the other hand, in most cases it is sufficient to use the MAC layer sniffing also for projects targeting higher layers.

With this argumentation, the reader might ask why not to start even lower in the stack then, in the physical layer. Here, we would get the same information plus the data from the PHY layer. First, we would have to implement decoding of the data ourselves as all decoders known to the author start from the MAC layer upwards. Then, the outputs and inputs to the physical layer are sample values of an analog signal with a bandwidth of 20 MHz or even more with carrier aggregation in recent LTE releases (up to 100 MHz aggregated bandwidth from LTE Release 10 onwards). Forwarding all these samples in real-time leads to many QMI messages and processing them would consume a large amount of resources on the main processor which is not optimized for signal processing. Next, the PHY layer processing has to meet strict timing constraints which are hard to respect with our still limited knowledge of the modem's firmware internals and, as a result, might get violated if we inject code in these functions.

The lowest feasible layer was chosen for the sniffing, allowing to intercept the largest amount of communication data in exchanged messages.

10.4.3 *Sniffing scheme*

The sniffing project uses the mechanism developed for all projects in Section 10.1 to communicate with the Android user space application. The application's code can then process the MAC frame data. Analyzing the MAC traffic on the screen of the phone is rather inconvenient. Instead of processing the data directly on the phone, we included a feature into the app to forward the frames to another device in the network over the User Datagram Protocol (UDP). The *Wireshark* network analyzer tool includes a dissectors to analyze LTE MAC frames

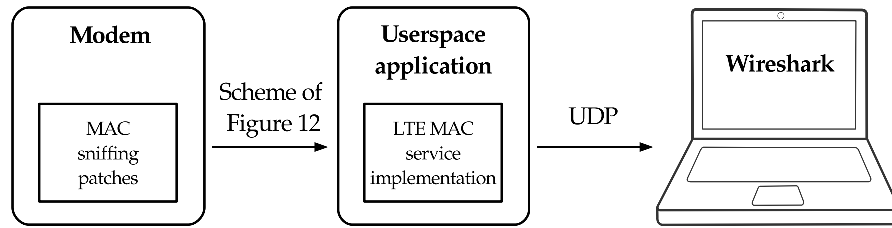


Figure 14: LTE MAC sniffing setup

Our Android app forwards messages over UDP. This allows to analyze messages on another device running Wireshark.

[42] and the packets are sent in the format this dissector expects. Consequently, to pass the data to the sniffer tool, it just needs to capture the incoming packets on the network interface and all its analysis features can be used. Figure 14 illustrates this scheme.

In Figure 13, we see that some PDCP frames are not analyzed higher than this layer since they are encrypted, for example frame 52 as security was enabled in frames 30 and 31 before. Luckily, the Wireshark PDCP dissector is also able to decrypt these, if the correct keys are configured in its settings. These can be obtained by running the patch for cryptography key extraction, which we present in Section 10.5, together with the MAC layer sniffing and entering the required keys in Wireshark. Unfortunately, neither the MAC nor the PDCP dissector directly allow to pass the keys in metadata of the frames they process, meaning they cannot be passed directly to Wireshark but have to be entered by hand. If this is a problem, it can be overcome by extending one of the dissectors, favorably the MAC dissector, with this capability. The PDCP dissector already includes the needed functions to set the keys from other dissectors.

10.4.4 *Implementation*

In this section, we discuss how this patch was realized. Remember that the aim of the project is to capture MAC frames. Therefore, we need to collect the data at the low end of the MAC layer, at the interface with the PHY layer. As a consequence, we first need to figure out how this interfacing is done in *Qualcomm's* modem firmware. We can divide this into two subtasks: downlink frames which are received by the MAC layer from the physical layer and uplink frames passed in the opposite direction.

10.4.4.1 *Intercepting downlink frames*

Downlink frames are easy to intercept, all required values are passed to a single function.

Frames in the downlink are passed to the MAC layer as a *DSM* item (as introduced in Section 7.3.2). This item contains all bytes of the received MAC frame. A pointer to this element is passed as a part of a data structure containing additional meta-information on the received frame, like the time it was received as system frame number and subframe number.

The patch overwrites the main frame processing function in the MAC layer. It gets a pointer to the previously described data structure as parameter. With this, we have all relevant data and can forward the metadata and the bytes extracted from the *DSM* item to the Android application. This then converts it to a packet compatible to Wireshark's LTE MAC protocol. Especially, it needs to convert the metadata from the modem's internal representation, consisting of two structures packed in two 32 bit words, into the protocols encoding. For this, it first extracts the different fields from the data words. Additionally, in case of the RNTI type value, a mapping between the values used by the modem and the values used by the Wireshark protocol is required.

10.4.4.2 *Intercepting uplink frames*

Intercepting messages in the other direction, sent from the MAC to the PHY layer, is not as simple as for the downlink frames, the patch code consists of a total of 16 overwritten functions in order to achieve the desired functionality. This is because the frame is not constructed as a single array of bytes by the MAC layer but instead passed in parts. The physical layer is then responsible to build the final bytes to transmit during the sending phase.

This is implemented by a queue of "tasks", each of these specifying data to be written to the frame. The queue is ordered and tasks can only be appended to the end, meaning that tasks are executed in the sequence they are created and the bytes of the frame are written in exactly this order. To still allow the MAC layer to first write data in a different pattern, functions to reserve tasks without adding actual data and corresponding functions to fill them with information at a later point in time exist. This is, for example, used to write the frames Service Data Unit (SDU) before building the MAC header.

The mechanism to pass uplink frames to the PHY layer is complex. Only the lower layer builds the final frame.

There exist different types of tasks for various purposes as listed in Table 19. Thereby, for example, padding bits can simply be specified by the number of bits instead of copying a large number of zero bytes.

Table 19: MAC uplink frame generation tasks

Task	Description
Gather	copies an array of data bytes
Fill	same as gather task but optimized for small amounts of data (maximal 16 bytes)
DSM	copies data from a *DSM* item
Padding	adds a number of padding bits (all zero)
Cipher	ciphers the bytes written by the next task
Done	pseudo task, sets a done flag to indicate the frame is complete

The cipher type is a special task which does not write bytes itself. Instead, it indicates that the data of the next task should be ciphered with certain parameters. This is needed as the higher PDCP layer in *Qualcomm's* stack only marks data to be ciphered but does not perform the actual cryptographic operation. The cipher operation is postponed to the building of the final frame in the PHY layer. For this, it also implements a key store with a corresponding function to add new cipher keys which are then referenced by a key index in the cipher tasks.

Even more task types exist in the firmware. However, they are either not used by the observed firmware versions or internally map to one of these tasks.

To patch the firmware and sniff the uplink frames, we modified all these task creation functions, including their reserve and information adding derivates. The patch code implements a copy of the task queue. The function hooks add or modify the tasks in this queue, in addition to calling the original function. On creation of a "done" task, our code processes the content of the queue and builds the MAC frame's byte data by evaluating the tasks. It then forwards the resulting data over the QMI based communication scheme to the user space application.

10.4.4.3 *Metadata for uplink frames*

We have to recover metadata for uplink frames, especially frame numbers, from other sources.

Unfortunately, the tasks contain no metadata information for the described frame. At least the send time in terms of a LTE system frame number and subframe number is crucial for a meaningful analysis of the data, allowing to put them in reference to received downlink frames. We cannot recover this information from simply observing the arrival time in the user space app due to varying latencies of the QMI protocol used for communication. Also, an offset between the times of processing received frames and the construction of uplink frames exist since downlink frames are processed after they were sent over the air interface whereas uplink frames obviously have to be built before the LTE system time at which they are supposed to be transmitted.

An initialization function called before each MAC uplink frame generation exists. It gets a value at which the frame is supposed to be sent in a unit of some timer ticks. Note that this value has a fixed increase per time (19200 ticks/ms) but no relation to the current LTE system time.

We can recover this relation by utilizing a detail of the LTE specification: in the LTE RACH procedure, a RRC connection request message (*Msg3*) in the uplink has to follow a RAR message (*Msg2*) in the downlink exactly six subframes later. Additionally, these are the first messages of each data exchange between a UE and an eNodeB. With this information, we could generate a mapping function to conclude

the LTE system times from the timing values of the task initialization function.

The timer value can be shifted slightly during operation by the timing advance feature of LTE adjusting send times to take care of signal propagation delays to the base station. As this is limited to a maximum of approximately $667,7\,\text{us} < 1\,\text{ms}$ (one LTE subframe), the mapping function can deal with this by just placing decision intervals for the LTE times carefully. Therefore, it does not need to keep track of the current timing advance value. This calculation is implemented in the Android application, the modem firmware patch directly forwards the timer value. With this, after a successful RACH procedure, the app is also able to forward uplink messages including LTE system frame and subframe numbers to a device running Wireshark.

10.4.5 *RACH preamble*

The RACH preamble (*Msg1*) is not a real MAC layer message but rather a special mechanism of the physical layer to offer random access of devices to the network to allow them to request a new connection. On the other hand, the sending process of it is initiated by the MAC layer and, therefore, it makes sense to also see these messages in the analysis. The Wireshark LTE MAC dissector also supports RACH preambles by setting a special metadata flag and leaving the message content empty. Therefore, the patch code overwrites two more functions, one used to send a RACH preamble and one used to retry in case the first attempt failed.

10.5 LTE CRYPTOGRAPHY KEY EXTRACTION

The *lte_sec* projects allows to extract cryptography keys used by LTE for encryption and integrity protection. Obtaining these allows, for example, to decrypt payload data of the LTE MAC frames from the *lte_mac* project, allowing further applications when these two patches are combined as we are then able to analyze all of the data contained in the MAC frame. It can also be used as base for generating own MAC frames with payload data, useful to test new MAC or higher layer approaches. Another use case is to implement attack scenarios with an attacker having access to the modem, allowing him to extract the keys and impersonate the target's identity, showing only a single kind of many severe attacks possible with the ability to modify the modem's firmware code.

The lte_sec project gives access to session keys. These can, for example, be used to decrypt data in intercepted MAC messages.

To perform the key extraction, the patch project overwrites the firmwares key generation function and forwards the generated key together with parameters of the function determining the data it will be used for (NAS/RRC/user data), the purpose (integrity or ciphering) and the target ciphering or integrity algorithm of the key.

The patch can also report calls of related security functions, including all values used for the cryptographic operation.

In addition to this, we modified the firmware functions for ciphering, deciphering and integrity MAC calculation and our patch code can inform the user space application on calls including all metadata needed as input for the calculation operations: bearer ID, count value, message length and the direction of the data (uplink/downlink).

The user space code can choose from a variety of options when registering for the LTE security patch service of the modem. First, it can define about which function calls it wants to be informed. For the actual cryptographic operations, it can then further choose if the used key should be included and whether or not the input and output data should be part of the messages (all independently), reducing the communication overhead to the required minimum.

Listing 6 shows example output messages with all options enabled for cryptographic operations but only the resulting key for the generation function. At the beginning, interestingly, the modem firmware generates keys for all possible algorithms of the LTE specification. Logically, the cryptography functions only use the keys for the algorithms chosen for the communication. We can also see that the output of decipher calls, without the last four bytes containing the transmitted authentication code, prepended with the *count* value, is passed as input to the integrity protection function in order to check the MAC of the message.

The listing contains only a few messages for brevity, in a real LTE communication many more MACs are generated and many messages are ciphered. Also, new keys are generated under certain conditions. Depending on the cause that triggered this key generation, the old keys might still be used for a while before the new keys replace them.

In Section 10.4.4.2, we describe that the PDCP ciphering for the uplink is delegated to the final frame building step inside the PHY layer. There, an independent set of cryptographic functions is used and, as a result, cipher operations in the uplink are not seen by this patch.

As already mentioned in Section 10.4, one use case of this patch is the combination with MAC layer sniffing allowing to decrypt protected content in the intercepted messages.

10.6 LTE CHANNEL ESTIMATION

In the LTE background introduction (Section 3.2.3), we saw that the LTE UE has to estimate the current channel to the base station continuously. This results in information on how the transmission channel changes the phase and amplitude of different frequency components of the signal, which maps to how the symbols of the subcarriers are influenced in case of the used OFDM. This characterization is done for each pair of antennas of the UE and the base station. When, for example, 2x2 MIMO is used, four distinct channels exist and need to

be estimated. This data is then used to compensate the channel's influences on the transmitted symbol in the decoding phase and allows to reconstruct the symbol that was originally sent by the transmitter. Moreover, the UE bases the channel information values sent back to the base station to optimize resource allocations in the cell on the channel estimates.

LTE continuously estimates the communication channel. The channel_estimation patch extracts this CSI data.

In addition to these primary use cases, it is possible to leverage the channel estimates to derive other information which is not directly obvious. As the signals spread from the transmitter to the UE through the area around these and, for example, get reflected on objects in between, the channel characteristics contain information about the environment.

Listing 6: LTE key extraction output messages

```
1  new algorithm key for RRC_INT, used algorithm EIA1:
       ↪ 315A6F461457AA158CA7F25C4C506D06
2  new algorithm key for RRC_INT, used algorithm EIA2:
       ↪ 4A3DC770975265CF0A62642A796D97EA
3  new algorithm key for RRC_INT, used algorithm EIA3:
       ↪ 9898D1C2AA988B406670B9656AEB23BF
4  new algorithm key for RRC_ENC, used algorithm EEA1:
       ↪ D90C0A1DEC414BA89A7C77E6EB466F5E
5  new algorithm key for RRC_ENC, used algorithm EEA2:
       ↪ 824B9118660B0CCC211FB7AF076A89A9
6  new algorithm key for RRC_ENC, used algorithm EEA3:
       ↪ D4E43C747D47B8F67E033B6331417B08
7  new algorithm key for UP_ENC, used algorithm EEA1:
       ↪ D5728315F0ABD010B8D80DF79912DACE
8  new algorithm key for UP_ENC, used algorithm EEA2:
       ↪ F64E8F372D22A0B2410C901E5E3958C0
9  new algorithm key for UP_ENC, used algorithm EEA3:
       ↪ A20198CD3E3B3565980EEBF61ED7A71C
10
11 MAC-i call uplink: algorithm: EIA2, bearer: 0, count: 7, message bytes: 3
12 used key: 4A3DC770975265CF0A62642A796D97EA
13 input message: 071400
14 output data: EAAE8E39
15
16 decipher call: algorithm: EEA2, bearer: 0, count: 7, message bytes: 28
17 used key: 824B9118660B0CCC211FB7AF076A89A9
18 input message: 87020DF253015134EC107FC05BA95A4FBC3D4EF3E0B81ED1630BAA8A
19 output data: 26101540023002EEC300300C800A1184648031E0284B45809E0260D8
20
21 MAC-i call downlink: algorithm: EIA2, bearer: 0, count: 7, message
       ↪ bytes: 25
22 used key: 4A3DC770975265CF0A62642A796D97EA
23 input message: 0726101540023002EEC300300C800A1184648031E0284B4580
24 output data: 9E0260D8
```

We can use CSI information for many other purposes. A similar extraction tool was released for WiFi before.

Many research projects focused on making use of CSI data from WiFi channels after Daniel Halperin et al. released a tool to extract these from commonly used *Intel* WiFi cards [16]. Example areas are indoor localization and activity respectively event detection, a list of papers is available at the page of the extraction tool at [15]. Some of these ideas might be also applicable to LTE CSI data. However, the conditions for LTE channels are different to the ones found for WiFi:

- The distance between sender and receiver is longer.
- A larger area is covered.
- Usually no line of sight exists between the stations.

As a consequence, an LTE channel is usually exposed to a larger variation than the WiFi channels. Its channel estimation contains information about a larger area but with that also less details and it gets harder to conclude the precise cause of an observed change. Therefore, some applications developed for WiFi might not work with LTE CSI data or at least large adaptations to the algorithms could be required. On the other hand, especially due to the larger area covered, new use cases can be created. For example, it might be possible to monitor the traffic on a street between an UE and a base station.

With this patch project we intend to provide a tool for LTE CSIs similar to the presented WiFi CSI tool and enable research in this direction based on LTE.

10.6.1 *Implementation*

The moment of reading the CSI data has to be chosen with care to avoid memory conflicts. We use a function which generates reports for the base station.

Due to the continuous updating of the channel matrix estimation, the implementation needs to be careful at which moment it fetches the data. Reading during the same time as a new matrix write will lead to unusable data as a part of it contains already new values while the rest might still be the previous data. There is no way to detect this situation or to find out which values were already updated. Depending on the concrete implementation and configuration, the firmware might even crash if a read and write to the same item of the channel matrix occur.

As a result, we need to find a point in the firmware where it is safe to read the channel estimation. Logically, this is the case where the channel state feedback report messages for the base station are generated, as this also needs to access the channel information. The baseband code does this in a function referred to as "CSF::CALLBACK" (for channel state feedback callback) in strings of the firmware. This function is called once per LTE subframe, so once per millisecond. However, it does not generate a report in each call but only when necessary to fulfill the configuration of the network.

It also calls another function with the purpose of logging, according to string messages found in the code directly before it is executed. We assume that the target of this is the *DIAG* interface used by the tools presented in Section 2.2, even though we could not get the data with any of these. Note that the code calls this function only when a channel state feedback report is generated and then only every n-th time, with n being 100 by default. Thus, although the channel estimation data should be available at the diagnostic interface, the report interval is far beyond the actual measurement rate and too low for many applications. For now, it is unclear if it is possible to change the report interval to a lower value with a command on the exposed interface.

This is where the patch code presented in this work is advantageous. With our full access to the firmware, we can simply change the calling frequency of the logging function, fetch the CSI data here and forward the matrices together with metadata over QMI, at the rate they are generated. Reading the data at each execution of the callback function did not lead to further improvement: we read the same values multiple times, the channel seems to be estimated only if a report is generated. Thanks to the modified interval value, the logging function is also called in this case and we can get the data at the actual measurement rate here.

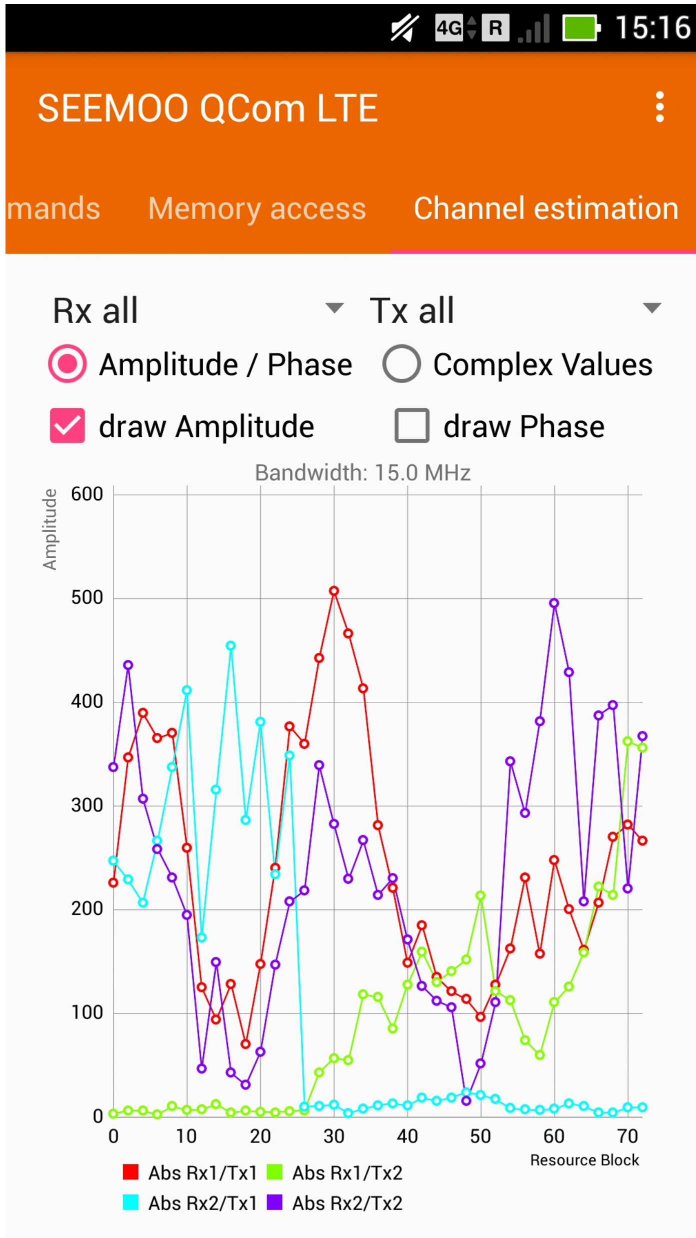
10.6.2 *Output data*

The output of this patch is the described channel matrix, respectively a set of channel vectors. The developed Android app includes functionality to visualize this data in different ways as it can be seen in Figure 15. It allows to get a first idea of how a LTE channel fades in time and frequency domains and how paths between different antenna pairs change. We can estimate general characteristics and decide for what projects the data might be suitable, for example, by observing variations under static conditions or when moving the phone.

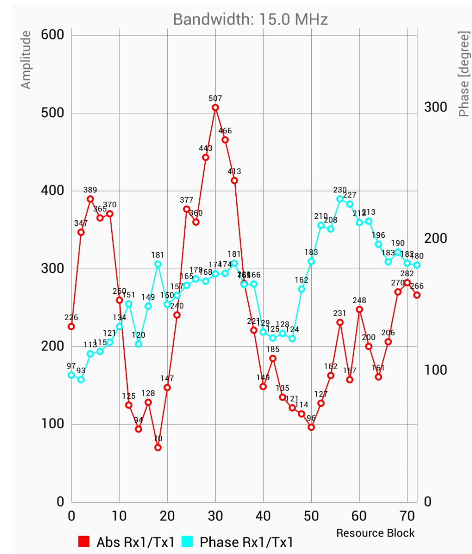
Our code provides the CSI to users over a simple interface. The developed app visualizes the data to get a first impression.

Later, the service can provide the data for a real use case. As all presented patch service implementations, the Android code implements a simple interface and allows to register listeners for data updates. Figure 15 (a) shows the amplitude of all four channels between two transmit antennas (Tx) of a base station and two receiving antennas (Rx) of the phone for a network with 15 MHz bandwidth. We can see that the channels Rx1/Tx1 and Rx2/Tx2 show a similar characteristic. The signal received by Rx2 from Tx1 is strong for the lower 5 MHz of the used frequency range and then negligible. The channel from Tx2 to Rx1 behaves inversely, its signal is weak for low frequencies and gets stronger in the higher 10 MHz.

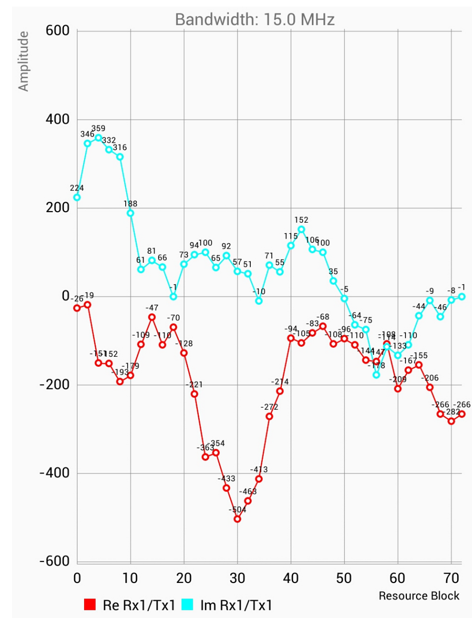
Figure 15 (b) shows a detailed view of the Rx1/Tx1 channel, including its phase. This channel shows why feedback of the channel



(a) Amplitude of all channels between base station and UE



(b) Amplitude and phase of Rx1/Tx1



(c) Complex value representation of Rx1/Tx1

Figure 15: Channel estimation plots

conditions to the base station is important: resource blocks around zero to ten and 22 to 36 of this channel currently have much better properties for signal transmission than the remaining resources and therefore, if possible in conjunction with allocations to other users, transmissions to this UE should occur on the subcarriers of these blocks. Similarly, Figure 15 (c) shows the complex representation of the channel coefficients for the same channel.

11 PROBLEM: FIRMWARE AUTHENTICATION

In Section 7.2, we showed that the modem's firmware image is digitally signed. As only the OEM of a phone has access to the needed private keys, the previously presented patching framework is not able to generate such a valid signature and can only be used on devices with this authentication deactivated. In order to broaden the range of target devices, we analyzed opportunities which could allow to bypass this problem and also get execution of modified firmwares on devices with the signature check enabled. Unfortunately, until now, we were not able to achieve this goal. We still shortly present our methods and findings here, as they reveal additional knowledge about the system and some have the potential to be the basis for a successful exploit when developed further in the future.

Modem images are digitally signed for most devices. Until now, we could not find a way to bypass this.

In contrast to previous work, *Qualcomm's* firmware code of the target device seems to be hardened well to us, for example all parameters passed to externally available interfaces, such as QMI, are validated, especially the length of the contained values. As a consequence, we expect low chances in finding an exploitable runtime vulnerability and decided to investigate other attack vectors first.

11.1 CONFIGURING PROTECTION UNITS

The modem's firmware in the shared memory cannot be modified after the authentication process by the HLOS because it is protected from accesses through a XPU. Thus, the straightforward idea is to reconfigure those and allow the accesses. Obviously, it is not possible to configure XPUs from the Linux kernel as this would be equal to no protection, malicious code with root privileges could also simply change the XPU configuration.

Therefore, configuration of the protection units is only possible in the *TrustZone* from the main processor system. In this trusted execution environment, the ARM processors can run code for security critical tasks, isolated from the non-secure code execution of, for example, the Android operating system. Analogous to the authentication of modem code, the *TrustZone* code needs to be signed such that no custom code can be executed. This means that it is also not possible to modify this code.

XPUs can only be configured from the TrustZone. With exploits it is possible to gain the needed rights.

However, exploits exist allowing to gain full access to *TrustZone* implementations. Especially, for the *MSM8974* SoC, a vulnerability and detailed exploit code with an extensive explanation can be found in [7]. A bug in one of the handlers for commands from the non-

secure kernel allows to get a zero write primitive. The exploit then uses this to modify other command handlers to gain an arbitrary write primitive. Since handler functions for commands are called by pointers stored in a table, we can now modify these with our crafted primitive and directly call internal functions of the *TrustZone* code. For the details, please refer to the well written original source. The author also presents ways to execute own instruction in the secure environment, however, the step to write these to secure memory did not work stable in our tests and caused the system to reboot in many cases.

After identifying the interface for configuration of XPUs in the *TrustZone* code, this exploit allows us to execute these functions and with that to modify rules enforced by the XPUs. By this means, we were, for example, able to successfully configure locks for memory regions in the main memory MPU which are then no longer accessible from the non-secure world.

We are able to configure own XPU rules, however, not to modify existing rules of the modem.

Regrettably, this is limited by ownerships of the rules (Section 6.2.1). Thus, even the *TrustZone* does not have the required rights to remove the XPU locks from the modem's memory regions. We discovered that for multiple rules owned by the modem only the last rule is enforced. It is not possible to place a new rule with access permissions for the main processor after the modem rules as the last rules in the main memory MPU are owned by this subsystem. As a consequence, the straightforward idea of unlocking the XPUs is not possible.

Please note that the knowledge about XPUs and the used rules, including rule ownerships, was not available before and we only gained it by this process, although presented earlier in this document in Section 6.2.1.

11.2 FUSE BITS

The SoC includes a set of one-time-programmable fuse bits. Similar to a regular fuse, they can be either intact or blown, with the first state representing a zero bit and the blown state a one. This means that zeros can be programmed to ones later by blowing a fuse but it is physically impossible to change a one back to a zero. Bits are logically grouped and accessed in registers of 32 bits.

One-time-programmable fuse bits control if the authentication of modem images is required. Once enabled, it is physically impossible to disable it again.

The first and second stage modem bootloaders, PBL and MBA, read one of these fuse bits in order to determine if secure boot is enabled, meaning that the loaded image needs to be authenticated. If the bit is zero, the signature checking code is skipped. As a blown fuse bit enables the feature, it is not possible to deactivate it again later. For the main target *MSM8974*, this fuse bit is accessible by the sixth lowest bit of the physical address 0xFC4BE040 for authentication of the MBA image, respectively 0xFC4BE044 for the actual firmware image.

As expected, in the *Google Nexus 5* we found these fuses to be blown, in contrast to the *Asus PadFone Infinity 2* with deactivated signature checks where they contain zeros.

11.2.1 *Locking access to fuse bits*

Our next idea was to combine the findings of the last two sections. When a region is locked by a XPU, read accesses to these location return only zeros. As the value of an unblown fuse bit is zero, we can in consequence make blown fuse bits look to subsystems again as if they were unblown. We can thus say that we are inversing the purpose of XPUs, from protection units to using them to mount attacks against the systems security.

We were indeed able to use this idea to trick functions checking fuse bits of the *TrustZone* to return manipulated values. For example, the secure world code allows to write fuses but blocks writes to some regions depending on fuse bit configurations. A XPU lock allowed us to let the function responsible for checking the address return that a write is allowed, even for previously blocked regions.

However, an attack against the MBA or PBL of the modem was not successful. In theory, locking access to the fuse bits for the modem would also make it read only zero bits. Therefore, the previously described check of the bit responsible to indicate the secure boot status would return that the image does not need to be authenticated. Regrettably, the bootloaders hung up completely when we tried this. A command to the MBA to authenticate the image metadata, for example, never returns a result, not even an error code.

Using XPUs to prevent accesses to fuse bits makes blown fuses read as unblown again. Unfortunately, the modem bootloaders crash during such an attack.

It is also only possible to lock all fuse bits together since the fuse peripheral is protected by an APU and not an MPU. As a consequence, the lock might cause problems for other fuses which now also read zero. We thus analyzed all accesses of the MBA to fuse bits but could not find any access that would explain the observed behavior.

Nevertheless, we tried to place the lock only at the time when the intended target fuse bit is read. As we do not get any feedback from the modem after sending the authentication command, we determined this timing indirectly. After a known delay, we modified the passed metadata such that it is incorrect and the authentication should fail, in case we did the modification before the authentication started. During the actual authentication, we cannot modify the data as the MBA removes write rights from other subsystems for the memory region containing the data. After, it would not change the result anymore. With this method, we know if the write occurred before the authentication process started or not. By slowly increasing the delay time and observing the result of the authentication, it gets possible to determine the actual delay between the command and the start of the authentication function. As the check of the target fuse bit is done

directly before this in the code, we now know when we have to lock the XPU. Unfortunately, the timed lock also did not prevent the MBA from crashing.

As a consequence, we assume that the modem DSP sees the access error on the system bus and, in contrast to the main application processor, raises an exception. The entries for bus errors in the exception vector tables of the MBA and the modem's firmware point to code which leads to a "halt" instruction, after some other code. Thus, this would explain the observed behavior well.

Although we could not identify any possibility to fully verify this assumption yet, another observation supports it. We tried to lock the access for the modem to a part of its memory region, containing only text strings, by an additional rule overlapping the modem's rules. If we configure the owner of this not as the modem, according to Table 10, the target region gets locked for all accesses. Indeed, the modem's behavior is influenced by the new rule. The modem did not just read empty strings but again stopped operating completely.

Please note that, even though it seems not to be possible to use XPU locks on fuse bits to tamper with the modem's operation, it might be possible to use it in the other direction: to tamper with *TrustZone* code from the modem subsystem, as it should be possible to place the same locks we used in our experiments from the modem's code.

11.2.2 *Changing the root-of-trust*

A hash of a root certificate is used as the root-of-trust for the modem's firmware loading. This hash is compared with the computed hash of a certificate embedded in the loaded image and the certificate is only accepted if the two hashes match.

For the MBA, its image file usually includes this root-of-trust. However, in the function responsible for reading the hash, we found that OEMs have the option to place the hash in fuse values instead, activated again by another fuse bit.

The source of the root-of-trust can be changed to a region of fuse bits. Regrettably, it was not possible to write own data to this location.

Since the fuse to active reading the hash from fuses is set to zero, we can blow it and change the source of the root-of-trust. This is only interesting if it is possible to write the fuse bits containing the hash, thus giving us the ability to provide our own root certificate and sign images with the corresponding key. Other fuses configure the related region as write protected which is enforced in software in the *TrustZone* code. Even after bypassing these checks, either with the previously described XPU locks or by directly calling the internal functions after the checks, we were not able to change the needed fuse bits to place our own hash value.

The reason for this could be that the hardware also enforces the write protection additionally. Another possibility is that a rule of the modem in the XPU for the fuse bit hardware prevents the access.

Part IV

DISCUSSION AND CONCLUSIONS

In the final part of this thesis, we summarize and discuss the results. An outlook on possible further work follows. At last, we draw conclusions from the results.

12 DISCUSSION

After presenting all details of our work, we now discuss the implications of it. Especially, we have a detailed look at the cases in which our contributions are useful, in three senses:

- Considered scenario with full access to the device (Section 12.1)
- Target devices (Section 12.2)
- Actual applications (Section 12.3)

Even though these were outlined in the motivation already, we can now review how well we satisfy these and extend to a broader range of use cases.

Before the discussion, let us shortly summarize the main achievements presented throughout this thesis:

ANALYSIS OF THE MODEM INTERNALS We analyzed the hardware of the modem system by identifying its components and their system level architecture inside *Qualcomm's* SoC, including a detailed description of the utilized *Hexagon* DSP. We then used this knowledge in the study of the software which is executed by the processor.

DEVELOPMENT OF A PATCHING FRAMEWORK Building on the results of the analysis, we developed a framework allowing to apply code patches written in a high-level programming language, augmented with simple annotations, to firmware images. Unfortunately, images are authenticated on many devices and, without the private keys of the Original Equipment Manufacturer (OEM), the framework cannot include correct signatures in a patched image, meaning we cannot target these devices yet.

IMPLEMENTATION OF EXAMPLE APPLICATIONS We created a set of example applications. These range from simple test and debugging patches to complex implementations, which directly can be used for research projects (e.g. LTE channel matrix extraction). For this, we reverse engineered the corresponding parts of the firmware in detail.

12.1 CONSIDERED SCENARIO

Our approach for firmware modifications and all the example applications consider a scenario where we have full access to the phone.

As we want to modify our own phone, we consider a scenario with full access to the phone.

We are able to gain superuser (root) rights in the Android OS. In addition, we can modify the system's kernel, including changes to existing modules and extending it with own kernel modules. This is, for example, necessary to add our module for direct communication from a user space application with patch code in the modem over the QMI protocol. Furthermore, we can replace the modem firmware, however, this is actually a consequence of the previous abilities as we can overwrite the firmware partition with root rights or simply change the modem firmware loader in the kernel to load it from another location.

Event though these assumptions might sound unrealistic at first, they are all given if we recall our aim again: we want to modify the modem of a phone owned by ourselves, to which we have full access and not to attack a device under control of another party which would grant us only limited rights. On the other hand, modifying the modem's firmware by end-users is not an intended use case of the device. As we saw, even with highest rights an end-user can get on a device and the ability to change all data stored in the device's memory, this is still prevented.

12.1.1 *Extending the scenario*

The results of this work can also be applied to a broader range of scenarios, in particular attack cases. Although we can already consider our main scenario as an attack since a modification of the modem's firmware is somehow not a legitimate use even if we own the device, attacks here especially mean intrusion of devices not in our possession, thus having limited rights. We discuss two such scenarios in the following.

The results also apply to other scenarios, in particular attack cases.

In the first case, an attacker first targets the Android system. By exploiting vulnerabilities of the HLOS, the intruder gains full access to the phone. Being then in a similar position as considered by our main scenario, the attacker can apply the same methodology to modify the modem's firmware. Since the intruder first needs to break the Androids security scheme, its usefulness is questionable. Nevertheless, there are still advantages for an attacker in escalating further to the modem system. By deleting all malicious code on the Android system after changing the modem, the intrusion can be hidden well and all known countermeasures like virus detectors are unable to detect it. In addition, the attacker gains a set of new possibilities which is the topic of Section 12.3.1.

The attacker in the second case is less strong. He targets the modem directly and finds a vulnerability, or a collection of multiple vulnerabilities, directly in the modem code, for example in a message parser, be it for an interface to the Android system or for messages in a mobile communication technology. With this, the intruder manages

to craft a write gadget allowing arbitrary writes to all location of the modem's memory, including code and data sections. Subject to the strength of this write primitive and the available payload, the attacker can then either also completely replace the modem firmware or act more targeted and just apply the modifications needed for his patch. Note that the output of the patching framework could also be used in this case. The intruder can then mount the same variety of attacks as in the first scenario, depending on the used attack vector, possibly injecting the malicious code over-the-air.

12.2 TARGET DEVICES

We conducted the work in this thesis mainly on a single SoC, even on a single device with this chip (MSM8974, Asus PadFone Infinity 2). However, as already mentioned, the hardware and software of the modem part in *Qualcomm* SoCs obviously is not redesigned from scratch for every new chip but only slightly modified. The only bigger change in the baseband design was the shift to the fully *Hexagon* based architecture, running all logic on this DSP instead of a separate ARM core as used before by *Qualcomm*. Thus, most results of the analysis part are also valid for many other SoCs, as long as they are based on the new *Hexagon* design. Note that large parts of the software analysis are even valid for ARM based designs as, although the implementation was moved to the DSP processor, it still originates from the same code base.

In general, the modification framework itself is independent of the target. After all, due to the used tools in the processing steps, it is limited to *Hexagon* modems. Apart from this, only the firmware image and its wrapper header used as input files are target dependent, which means that we only have to generate a new wrapper header in order to support a new target. The evaluation of the presented porting tools (Section 9.3.3) shows that moving to a new device with the same SoC can be done almost automatically, only few details have to be added manually. On the other hand, for porting to another SoC, our tools do not provide sufficient performance yet, meaning that the reverse engineering steps to discover functions and their locations have to be done by hand again, or better tools for the automation of this step need to be developed. Luckily, this is only a small part of the reverse engineering process. The more complicated analysis on how the code works, how functionality is implemented, for example the purposes of functions, still applies. We described this, for firmware parts related to the implemented patches, in the corresponding section of Chapter 10.

Even though this thesis focuses on a single device, most results can also be applied to other phones. For devices with the same SoC, we can already do this automatically.

Keep also in mind that locating functions was not the main contribution in Part iii but instead the framework allowing easy modifications of the firmware binary, after the function information is

collected, as well as the implementation of example patches which are independent of the actual location of functions in the target binary. Anyway, the reverse engineering process has to be performed for new patches targeting not yet discovered parts of the firmware code. This can then be done directly for the desired target.

12.2.1 *Bypassing the firmware authentication*

We explained in Section 7.2 that the firmware image on most devices is digitally signed and in Chapter 11 we showed our efforts to break this security feature in order to be able to run modified modem firmwares also on these devices. However, our attempts were not yet successful. Therefore, we discuss further possibilities here.

As explained before, the modem shares the system's memory with the application processor. Accesses to regions of the other entity are prevented by XPUs. Thus, a low-level *rowhammer* attack against the memory chip itself might be able to compromise the modem's security. This attack uses a hardware bug present in many memory chips which allows to modify bits in the memory by repeatably reading other bytes that are physically located next to the target bit in the memory chip. The first bytes owned by the modem are physically adjacent to bytes owned by the application processor. As a consequence, they might be vulnerable to bit flips by only accessing memory not belonging to the modem. XPUs would not prevent such an attack as the target location does not need to be actually accessed.

Most devices authenticate the modem's firmware which we could not break yet. Low-level attacks like rowhammer might have the potential to achieve this.

In "Drammer: Deterministic rowhammer attacks on mobile platforms" [38], the authors indeed could show that *rowhammer* attacks are possible on mobile devices, including the *Nexus 5*. The problem with this approach is that only a few bits are vulnerable on each DRAM chip and that these are different for each instance. As a result, for each individual device, we have to check which vulnerable bits lead to exploitable situations. The chances for such a case are low if we look at the numbers presented in the paper: 12 out of 15 tested *Nexus 5* devices showed bit flips, in the best case with an average distance of 1 kB between two bits that could be flipped. Due to the row size of the used DRAM chip of 64 kB, we can thus expect approximately 64 bits in the modem firmware region that can be flipped by the application processor, for such a best case device. We now need that one of these flips is, for example, in the distance value of a jump that we can modify to let the DSP start executing code in an non-secure location, which is not very likely.

Using such a simple change might cause other problems, for example the modified location will not be mapped for the modem, further decreasing the chances of a successful exploit. It might, however, be possible to use this method to tamper with the authentication of metadata as here the attacker can freely choose the location of the data and

by this select a position such that a vulnerable bit is present at the desired location. On the other hand, this way is limited to apply single bit changes to the information in the firmware metadata which might not be sufficient for an attack. As our test device did not show any bit flips with the *drammer* example code, we did not perform further experiments in this direction.

In contrast to these attack vectors, we should evaluate classical runtime attacks. Although we already explained why we estimate the chances for these to be rather low, there might still be exploitable vulnerabilities in *Qualcomm's* code. These could be discovered by statical code analysis as well as dynamical methods such as *fuzzing*.

12.3 APPLICATIONS

After discussing the scenarios in which the results of this work can be used and evaluating the range of target devices, we now have a look at the actual applications, meaning for what exactly we can use the modem modification possibilities on the target devices. The implemented example applications give a first idea of what can be done. Roughly speaking, everything related to the mobile communication technologies implemented in the modem is possible, not limited to the focus on the LTE technology in this thesis. In the following, we detail these further, categorize and give examples.

All these applications can in general be used in all scenarios, nevertheless, many only make sense in a subset of the scenarios. An attacker, for example, might not be interested in improving video streaming applications on the target device (we explain this example later) but instead like to intercept SMS messages. On the other hand, if we modify our own phone we are (usually) not interested in spying ourselves but instead might want to improve the service quality in the aforementioned use case. However, there are also applications which can be used in both scenarios. Sniffing LTE MAC layer messages is such an example as it can be utilized by an attacker to intercept user data and by researchers to analyze the mobile network.

12.3.1 *Malicious applications*

When extending our scenario, we outlined why attackers would be interested in changing code of the modem but we still miss a discussion of the possibilities they gain by this, thus which malicious applications modem firmware modifications provide. As this is not the main focus of this work, we keep the section short and only provide first ideas. Attackers with malicious intentions will easily find many more application ideas.

Access to the modem's firmware gives an attacker a large set of additional capabilities, from spying users to jamming attacks.

Obviously, the ability to modify the baseband's code allows to intercept all user traffic passing over the mobile network interface, includ-

ing IP traffic as well as phone calls and other services like SMS. With the GPS being integrated with the modem subsystem, an exact user location is also available to the adversary. In addition to these passive attacks, attackers can actively suppress messages or calls, send messages in the name of the phone user or listen to the devices environment even when the phone is not in use by triggering a call to themselves. Intruders can go even further and try to hide their attacks also from the network side by leaking the information over another communication channel than the mobile network. With the modem hardware, they can send arbitrary signals and are not bound to the mobile standards. Therefore, attackers can implement their own communication directly with a remote station under their control, in addition to the usual mobile network operation such that no disruption of the service is noticeable. Thanks to the high transmit power and with that transmission range of the modem, the attacker can communicate with the target device from a large distance, at least equal to the range of the supported mobile network technologies.

All these attacks have in common that they attack the device itself, respectively its user. They are contrasted by attacks against the network which can also influence other devices in the proximity using the same network cell or directly try to harm the base station. Sending manipulated messages to the base station is such a case. Here, an attacker could try to exploit vulnerabilities of the base station's implementation with messages containing data to trigger behavior not desired by the operator. Instead of such a rather focused attack, only modified values could be sent to the base station to influence mechanisms of the mobile technology, for example to manipulate the downlink resource scheduling in LTE.

Attackers controlling multiple devices could also turn all phones under their control into jammers, all sending with the maximal available transmit power, possibly leading to severe service disruptions for all communication in network cells in the proximity.

12.3.2 *Modifications wanted by the user*

Users may want to modify their own modem to use it as a research tool as well as to add new features.

In contrast to these malicious applications, the motivation of this work was to enable modifications to the modem which are actually wanted by the user of the device. These again contain two subcategories: tools for research and end-user applications, or experience enhancements for existing ones. Note that the categories are by no means including all possible applications but just represent use cases we can currently think of. Similarly to the categorization into wanted and malicious applications, the subcategories are strongly related to each other, for example the development of most new user applications will need to be preceded by research.

Let us consider the following scenario: we want to develop an application able to detect burglars in an apartment by monitoring variations in the LTE channel estimations, similar to solutions that have been shown for WiFi, for example in [2]. For this, we first need to extract the CSI values as done in the example application in Section 10.6. We then have to develop algorithms which are able to detect the desired event in this data. Until now, we used the modification capability only as a tool for our research, we could have done this research also on another platform, for example with an SDR. In the last step, we use the results of the research to implement the actual end-user application, including the necessary modifications to the modem firmware.

12.3.2.1 *Research tools*

One of the main motivations for our work was to open the modem in off-the-shelf smartphones for research projects, leading to a cheap platform and removing the need for special equipment in many cases. With the presented modification framework, this was realized. All implemented example applications fall into this category.

Smartphone modems can serve as cheap and portable platforms for research on mobile communication networks.

Research applications include:

- Logging low-level messages and modem internal data, e.g. for analysis of network cells
- Testing new algorithms, e.g. improved implementations of parts of the LTE stack
- Evaluating new features directly on the final target devices
- Modifying data sent to a base station, e.g. for active analysis of network behavior

The last case is close to an attack against the network. However, it differs by our intention to analyze how current implementations in base stations react to certain cases instead of harming the system or other users. Such message modifications could, for example, contain manipulated Quality of Service (QoS) tags for data packages or report wrong CSI values in order to try to trick the base station to assign more of the available resources to us, thus stealing them from other users. Note that too high CSI values might actually degrade our throughput as LTE then chooses higher modulation schemes, leading to high error rates if the real conditions are not good enough to support these. Nevertheless, an attacker to the network could still be interested in doing so as it might also degrade service quality for other users and, therefore, research on the base station's behavior is also useful in these cases.

12.3.2.2 *End-user applications*

We can also use firmware modifications to implement additional features which are only available in other modem models but not included in the firmware of the desired device, for example Voice over LTE (VoLTE) support.

Even more interesting are features which are not implemented by today's modems of smartphones at all. This could be because the manufacturer sees no advantages in implementing it or because the new feature is a recent research result which did not yet reach consumer devices, maybe even our own new improvement idea in a research project.

A modifiable modem allows to include new features, to improve existing applications or to realize completely new ones.

Such features can obviously be completely new applications, like the burglar detector example drawn in the previous section, but also include improvements of the user experience of an already existing smartphone application. For instance, one could use information currently only available inside the baseband to allow user applications a better estimation of the current network communication state and with that enhance their performance. Such an example is adaptive video streaming, enriched with information from the LTE physical layer as used in the piStream project [43]. To gather the needed information in their test implementation, the authors used an SDR in addition to the target smartphone. With the ability to modify the phone's modem firmware, the need for this additional device gets removed and the whole solution could be implemented on the phone, leading to a practically usable and directly deployable solution.

We see that the modification capability removes the need to wait for manufacturers to include new features. Producing a working implementation example of usually rather abstract research results allows evaluation of the new feature in user studies. It is also easier to show the usefulness of the results to companies and end-users, increasing the chances for it to be adapted in commercial products later. Even if not adapted, it can at least be used on the target devices supported by the example implementation, generating a practical outcome from the research project directly to end-users in any case.

12.4 OUTLOOK

This thesis covered a large range of topics, from analyzing the modem's hardware and software, over the development of a patching framework, to the implementation of patch projects. As a result, there are many opportunities to improve, supplement or to extend our work. We already mentioned some of these in the corresponding chapters. The main open topics are:

- Finding a way to bypass the firmware authentication. For now, the patching framework is limited to devices on which this security feature is disabled, thus to a small set of targets.
- Improvement of the automated porting tools by using more advanced matching techniques, in order to allow faster porting to other SoCs and smartphones. This also extends the list of possible target devices.
- Analyzing further parts of the communication stack in detail and identifying more functions, including their purpose, parameters and locations. We did such a deep study only for relevant parts in the example projects. However, new applications will likely focus on other sections of the firmware. As the modem firmware contains a lot of code, it is, unfortunately, not feasible to reverse engineer the whole code such that the most interesting parts have to be selected.
- Implementing more tools for the most common research purposes, removing the need to do so before an actual project. Thereby, we could also enable users unfamiliar with the still rather low-level coding of patches and the required reverse engineering steps to use the modems in off-the-shelf phones as a research platform. In addition, for many projects, the time to implement the required patches will not be available, thus still commercial equipment supporting the needed features will be bought, which can only be overcome by already supplying patch code for such features.

A few parts of this work leave room for small improvements. In addition, we lack the demonstration of an end-user application.

- Demonstration of attack scenarios as outlined in the discussion of the considered scenario. This includes showing an attack gaining access to the modem firmware as well as indicating malicious actions the intruder could perform.
- Realization of an end-user application with the modification framework. Since all the presented example projects are research tools, we lack the demonstration of an application interesting for end-users.
- Extending the firmware modifications to new use cases. For example, it should be possible to transform the modem into a complete SDR.

In summary, we see that although further work can be done on the presented work itself, the biggest open topics are concerned with extending the range of target devices and using the resulting implementations in real world application cases.

13 CONCLUSIONS

In this work, we were able to shed light on a black box in today's smartphones, the modem. Reverse engineering revealed several internal details. Furthermore, we could demonstrate that modifying the modem firmware is possible and practical. It allows to collect internal data, as well as modifying functionality and adding completely new features.

We successfully looked into the modem's black box and could modify its firmware. The developed tools allow to build complex patches.

We learned that reverse engineering code for *Qualcomm's Hexagon* DSP is feasible and supported by some tools, although not yet as advanced as for ARM processors for example. Plugins for the *IDA* tool exist but contain problems including incomplete detection of functions and displaying wrong immediate values in assembly code views. Also, as far as we know, no decompiler exists. This means that we have to perform all analysis directly on the assembly code.

From our analysis of the firmware code, we can say that security measures strongly improved compared to the results of previous research on mobile communication modems that we saw in the related work in Section 2.1. This includes tests for a variety of error conditions in all functions. Also, checks for the length of passed arguments, especially in parsers for messages originating from external sources (Android system or over-the-air interface), were added to the code. Since a programmer can always oversee a security issue, these are complemented by automated mechanisms using well known exploit mitigation techniques. For instance, stack canaries are inserted against stack overflow exploitation and access permissions of memory segments prevent execution of data and the modification of instructions.

Qualcomm increased security significantly compared to previous observations.

The runtime security features supplement the authentication of the loaded firmware image, protecting from loading modified code in the first place. Afterwards, hardware units lock the access to the corresponding memory regions, making it impossible to manipulate internals of the baseband from other subsystems integrated in the SoC, for example the main processor. As a result, we can say that *Qualcomm* got aware of the importance of a secure modem subsystem nowadays and raised the bar for successful attacks significantly. Our discussion of malicious applications underlines this need for strong attack countermeasures.

Unfortunately, security strongly depends on the OEMs which can decide which security features they activate. In the case of our primary target device, we found the firmware authentication to be disabled completely, leaving an attacker who gained root rights in An-

droid in a sufficiently strong position to replace the modem firmware permanently.

Fortunately, even though weak security is bad for protection of end-users, it also is the key to allow us to modify the baseband. In this way, we can use the modem in commercial devices for our intentions as research platform and to implement new features for end-users.

All in all, this work should represent a good basis for further research on *Qualcomm* modems. Furthermore, we would be glad if the presented modification framework, or even the applications we implemented with it, proof themselves to be useful in future projects.

Part V

APPENDIX

A APPENDIX

A.1 TASK STARTUP SEQUENCE

In addition to the list of running tasks presented in Section 7.3.1, we observed the startup sequence of the modem with the hooked *rcinit* task start functions and additions to the thread creation function of QuRT. Listing 7 shows the results. Please note that QuRT threads use a priority value inverse to the one of the init system, with zero being the highest priority and 255 the lowest.

Listing 7: Startup sequence of tasks and corresponding thread creations

```
qurt_thread_create: DPC Task, 87
qurt_thread_create: Main Task, 144
qurt_thread_create: rcworker, 144
qurt_thread_create: NPA_ASYNC_E, 123
qurt_thread_create: NPA_ASYNC_R, 93
qurt_thread_create: IST38, 81
qurt_thread_create: IST42, 77
qurt_thread_create: IST50, 77
qurt_thread_create: DAL_WL_0, 252
qurt_thread_create: IST239, 77
qurt_thread_create: DAL_WL_1, 123
qurt_thread_create: DAL_WL_2, 123
qurt_thread_create: IST37, 78
qurt_thread_create: IST41, 77
qurt_thread_create: SMDL_PROFIL, 112
qurt_thread_create: IST36, 78
qurt_thread_create: IST40, 77
qurt_thread_create: IST48, 77
qurt_thread_create: IST32, 81
rcinit starting task: smdtask, 142
qurt_thread_create: smdtask, 113
qurt_thread_create: SMD_HIGH, 95
qurt_thread_create: SMD_LOW, 178
qurt_thread_create: IST33, 78
rcinit starting task: tmr_slave2, 162
qurt_thread_create: tmr_slave2, 93
rcinit starting task: tmr_slave1, 162
qurt_thread_create: tmr_slave1, 93
rcinit starting task: tmr_slave3, 162
qurt_thread_create: tmr_slave3, 93
rcinit starting task: timer, 163
qurt_thread_create: timer, 92
rcinit starting task: time_ipc, 64
qurt_thread_create: time_ipc, 191
rcinit starting task: sys_m_qmi, 152
qurt_thread_create: sys_m_qmi, 103
rcinit starting task: nv, 60
qurt_thread_create: nv, 195
rcinit starting task: fs, 62
qurt_thread_create: fs, 193
rcinit starting task: sys_m_smsm, 152
qurt_thread_create: sys_m_smsm, 103
rcinit starting task: sys_m, 152
qurt_thread_create: sys_m, 103
qurt_thread_create: QURT_fatalNotif, 99
qurt_thread_create: ssm, 195
rcinit starting task: dog, 164
qurt_thread_create: dog, 91
qurt_thread_create: IST68, 77
qurt_thread_create: IST74, 77
qurt_thread_create: FS Benchmar, 252
qurt_thread_create: QMI_PING_SV, 245
qurt_thread_create: modem_sec_xpu_I, 144
rcinit starting task: secrnd, 34
qurt_thread_create: secrnd, 221
rcinit starting task: npascheduler, 162
qurt_thread_create: npaschedule, 93
rcinit starting task: thermal, 78
qurt_thread_create: thermal, 177
rcinit starting task: seccryptarm, 24
qurt_thread_create: seccryptarm, 231
rcinit starting task: sleep, 1
qurt_thread_create: sleep, 254
rcinit starting task: fs_async_put, 8
qurt_thread_create: fs_async_pu, 247
rcinit starting task: sec, 29
qurt_thread_create: sec, 226
rcinit starting task: diag, 76
qurt_thread_create: diag, 179
qurt_thread_create: IST242, 78
qurt_thread_create: time_srv, 241
qurt_thread_create: IST75, 77
qurt_thread_create: DAL_WL_3, 123
rcinit starting task: modem_cfg, 60
qurt_thread_create: modem_cfg, 195
```

```
rcinit starting task: audioinit, 73
qurt_thread_create: audioinit, 182
qurt_thread_create: VOCSVC, 104
qurt_thread_create: MVS_IST, 91
qurt_thread_create: TOYSERVER, 132
qurt_thread_create: startup_first_t, 253
qurt_thread_create: FW_CRM_THREAD, 5
qurt_thread_create: INT01B, 15
qurt_thread_create: INT01C, 15
qurt_thread_create: INT01D, 15
qurt_thread_create: INT0D0, 15
qurt_thread_create: FW_LM_THREAD, 235
qurt_thread_create: FW_FWS, 63
qurt_thread_create: FWS, 230
qurt_thread_create: IST96, 77
qurt_thread_create: IST97, 77
rcinit starting task: a2_log, 8
qurt_thread_create: a2_log, 247
rcinit starting task: slpc, 161
qurt_thread_create: slpc, 94
rcinit starting task: a2_ul_per, 106
qurt_thread_create: a2_ul_per, 149
rcinit starting task: rf_apps, 94
qurt_thread_create: rf_apps, 161
rcinit starting task: rf_fwrsp, 160
qurt_thread_create: rf_fwrsp, 95
rcinit starting task: rf, 160
qurt_thread_create: rf, 95
rcinit starting task: a2, 125
qurt_thread_create: a2, 130
rcinit starting task: rf_ic, 160
qurt_thread_create: rf_ic, 95
rcinit starting task: ftm, 74
qurt_thread_create: ftm, 181
rcinit starting task: rfa_fwsw, 164
qurt_thread_create: rfa_fwsw, 91
qurt_thread_create: IST125, 77
qurt_thread_create: IST153, 75
qurt_thread_create: IST154, 75
qurt_thread_create: IST174, 78
qurt_thread_create: IST175, 78
qurt_thread_create: IST178, 78
qurt_thread_create: IST199, 80
qurt_thread_create: IST198, 77
qurt_thread_create: IST200, 80
qurt_thread_create: IST201, 80
qurt_thread_create: IST138, 75
qurt_thread_create: RFCMD, 63
qurt_thread_create: startup_second_, 253
qurt_thread_create: ML1 GM, 91
qurt_thread_create: ML1 MGR, 93
qurt_thread_create: ML1 OFFLOAD, 93
qurt_thread_create: WCN Driver, 91
qurt_thread_create: LTE MAC DL, 101
qurt_thread_create: LTE MAC UL, 97
qurt_thread_create: LTE MAC CTRL, 100
qurt_thread_create: LTE RLCDL, 103
qurt_thread_create: LTE RLC UL, 102
qurt_thread_create: LTE PDCP DL, 105
qurt_thread_create: LTE PDCP UL, 104
qurt_thread_create: LTE PDCPOFFLOAD, 99
qurt_thread_create: LTE RRC, 141
qurt_thread_create: LTE TLB CTRL, 107
qurt_thread_create: CFM, 97
qurt_thread_create: DSMSGR RECV, 108
rcinit starting task: rxtx, 143
qurt_thread_create: rxtx, 112
rcinit starting task: tds_mac_hs_dl, 122
qurt_thread_create: tds_mac_hs_, 133
rcinit starting task: gsm_rr, 108
qurt_thread_create: gsm_rr, 147
rcinit starting task: tds_rrc, 110
qurt_thread_create: tds_rrc, 145
rcinit starting task: gsdi, 55
qurt_thread_create: gsdi, 200
rcinit starting task: hdrdec, 151
qurt_thread_create: hdrdec, 104
rcinit starting task: tds_l2_ul, 123
qurt_thread_create: tds_l2_ul, 132
rcinit starting task: rrc, 109
qurt_thread_create: rrc, 146
rcinit starting task: cxm, 80
qurt_thread_create: cxm, 175
rcinit starting task: wfw_eulstrt, 164
qurt_thread_create: wfw_eulstrt, 91
rcinit starting task: gsm_mac, 119
qurt_thread_create: gsm_mac, 136
rcinit starting task: tx, 156
qurt_thread_create: tx, 99
rcinit starting task: rx, 149
qurt_thread_create: rx, 106
rcinit starting task: hitapp, 76
qurt_thread_create: hitapp, 179
rcinit starting task: tds_l2_dl, 121
qurt_thread_create: tds_l2_dl, 134
rcinit starting task: hdrtx, 150
qurt_thread_create: hdrtx, 105
rcinit starting task: wcdma_l2_ul, 123
qurt_thread_create: wcdma_l2_ul, 132
rcinit starting task: gstk, 54
qurt_thread_create: gstk, 201
rcinit starting task: tdso, 85
qurt_thread_create: tdso, 170
rcinit starting task: limitsmgr, 80
qurt_thread_create: limitsmgr, 175
rcinit starting task: hdrsrch, 146
qurt_thread_create: hdrsrch, 109
rcinit starting task: wfwsw_evt, 164
qurt_thread_create: wfwsw_evt, 91
rcinit starting task: dh, 49
qurt_thread_create: dh, 206
rcinit starting task: tplt, 110
qurt_thread_create: tplt, 145
rcinit starting task: uim, 65
```

```
qurt_thread_create: uim, 190
rcinit starting task: srch, 148
qurt_thread_create: srch, 107
rcinit starting task: wcdma_mac_hs_dl, 122
qurt_thread_create: wcdma_mac_h, 133
rcinit starting task: auth, 95
qurt_thread_create: auth, 160
rcinit starting task: fc, 155
qurt_thread_create: fc, 100
rcinit starting task: trm, 70
qurt_thread_create: trm, 185
rcinit starting task: wcdma_l1, 153
qurt_thread_create: wcdma_l1, 102
rcinit starting task: tds_l1, 161
qurt_thread_create: tds_l1, 94
rcinit starting task: wcdma_l2_dl, 121
qurt_thread_create: wcdma_l2_dl, 134
rcinit starting task: gsm_rlc_dl, 118
qurt_thread_create: gsm_rlc_dl, 137
rcinit starting task: gsm_l1, 161
qurt_thread_create: gsm_l1, 94
rcinit starting task: hdrrx, 161
qurt_thread_create: hdrrx, 94
rcinit starting task: gsm_llc, 104
qurt_thread_create: gsm_llc, 151
rcinit starting task: gsm_rlc_ul, 117
qurt_thread_create: gsm_rlc_ul, 138
rcinit starting task: tcxomgr, 86
qurt_thread_create: tcxomgr, 169
rcinit starting task: tds_fwsw_evt, 164
qurt_thread_create: tds_fwsw_ev, 91
rcinit starting task: gsm_l2, 120
qurt_thread_create: gsm_l2, 135
qurt_thread_create: RFRPE, 241
qurt_thread_create: IST87, 77
qurt_thread_create: IST89, 77
rcinit starting task: gps_fs, 20
qurt_thread_create: gps_fs, 235
rcinit starting task: ds_sig, 50
qurt_thread_create: ds_sig, 205
rcinit starting task: dswcsd_ul, 107
qurt_thread_create: dswcsd_ul, 148
rcinit starting task: cm, 75
qurt_thread_create: cm, 180
rcinit starting task: cc, 162
qurt_thread_create: cc, 93
rcinit starting task: cb, 26
qurt_thread_create: cb, 229
rcinit starting task: gpsfft, 21
qurt_thread_create: gpsfft, 234
rcinit starting task: pgi, 147
qurt_thread_create: pgi, 108
rcinit starting task: pdcommtcp, 88
qurt_thread_create: pdcommtcp, 167
rcinit starting task: locotdoamp, 60
qurt_thread_create: locotdoamp, 195
rcinit starting task: pdcommwms, 87
qurt_thread_create: pdcommwms, 168
rcinit starting task: loc_middleware, 79
qurt_thread_create: loc_middlew, 176
rcinit starting task: sm_tm, 91
qurt_thread_create: sm_tm, 164
rcinit starting task: ps, 47
qurt_thread_create: ps, 208
rcinit starting task: pp, 124
qurt_thread_create: pp, 131
rcinit starting task: mn_cnm, 98
qurt_thread_create: mn_cnm, 157
rcinit starting task: tc, 100
qurt_thread_create: tc, 155
rcinit starting task: wms, 72
qurt_thread_create: wms, 183
rcinit starting task: nf, 71
qurt_thread_create: nf, 184
rcinit starting task: gnss_sdp, 90
qurt_thread_create: gnss_sdp, 165
rcinit starting task: ims, 44
qurt_thread_create: ims, 211
rcinit starting task: dswcsd_dl, 106
qurt_thread_create: dswcsd_dl, 149
rcinit starting task: xtm, 68
qurt_thread_create: xtm, 187
rcinit starting task: hdrmc, 96
qurt_thread_create: hdrmc, 159
rcinit starting task: reg, 101
qurt_thread_create: reg, 154
rcinit starting task: qmi_pbm, 56
qurt_thread_create: qmi_pbm, 199
rcinit starting task: comp, 103
qurt_thread_create: comp, 152
rcinit starting task: dcc, 51
qurt_thread_create: dcc, 204
rcinit starting task: qvp_rtp, 40
qurt_thread_create: qvp_rtp, 215
rcinit starting task: mmoc, 94
qurt_thread_create: mmoc, 161
rcinit starting task: ps_rm, 107
qurt_thread_create: ps_rm, 148
rcinit starting task: cd, 70
qurt_thread_create: cd, 185
rcinit starting task: ds, 53
qurt_thread_create: ds, 202
rcinit starting task: lm, 92
qurt_thread_create: lm, 163
rcinit starting task: sm_gm, 91
qurt_thread_create: sm_gm, 164
rcinit starting task: mc, 97
qurt_thread_create: mc, 158
rcinit starting task: qmi_mmode, 75
qurt_thread_create: qmi_mmode, 180
rcinit starting task: mm, 102
qurt_thread_create: mm, 153
rcinit starting task: gnss_msgr, 91
qurt_thread_create: gnss_msgr, 164
```

```
rcinit starting task: mgpmc, 93
qurt_thread_create: mgpmc, 162
rcinit starting task: ui, 67
qurt_thread_create: ui, 188
rcinit starting task: tlm, 69
qurt_thread_create: tlm, 186
rcinit starting task: sm, 99
qurt_thread_create: sm, 156
rcinit starting task: pbm, 56
qurt_thread_create: pbm, 199
rcinit starting task: ds_gcsd, 105
qurt_thread_create: ds_gcsd, 150
rcinit starting task: sd, 96
qurt_thread_create: sd, 159
rcinit starting task: qmi_modem, 52
qurt_thread_create: qmi_modem, 203
rcinit starting task: locotdoactrl, 112
qurt_thread_create: locotdoactr, 143
rcinit starting task: secips, 37
qurt_thread_create: secips, 218
rcinit starting task: secssl, 30
qurt_thread_create: secssl, 225
qurt_thread_create: RFCMD, 63
qurt_thread_create: IST0, 77
qurt_thread_create: IST99, 77
qurt_thread_create: IST98, 77
qurt_thread_create: IST100, 77
qurt_thread_create: WFW_CRASH_DUMP, 16
qurt_thread_create: WFW_SW_CMD_PROC, 44
qurt_thread_create: WFW_RX_AGC, 34
qurt_thread_create: WFW_SRCH_SCHED, 46
qurt_thread_create: INT0B6, 25
qurt_thread_create: WFW_DFRONT_CFG, 37
qurt_thread_create: WFW_TX_BETANORM, 43
qurt_thread_create: WFW_CHAN_PROC, 42
qurt_thread_create: WFW_DBACK_NOHS, 45
qurt_thread_create: WFW_DBACK_HS_0, 40
qurt_thread_create: WFW_DBACK_HS_1, 41
qurt_thread_create: INT0B5, 26
qurt_thread_create: INT0B4, 27
qurt_thread_create: RFCMD, 63
qurt_thread_create: IST119, 77
qurt_thread_create: GFWSCHEDULER, 34
qurt_thread_create: GFWASYNC, 36
qurt_thread_create: INT016, 26
qurt_thread_create: GFW GENERIC 0, 40
qurt_thread_create: GFW GENERIC 1, 40
qurt_thread_create: GFW GENERIC 2, 40
...
```

A.2 OCCURRED PROBLEMS

During the practical modification part of our work, we encountered a set of problems. In Section 10.1.2.1, we already discussed the not unique assignment of identification information for the QMI test service and gave a solution. In the following, we explain further occurred problems.

A.2.1 *Hexagon issues*

We start with issues related to the *Hexagon* DSP found in *Qualcomm's* modems.

A.2.1.1 *Unaligned accesses*

The *Hexagon* processor only supports aligned accesses to data, meaning that, for example, the address of a data word (32 bits) needs to be divisible by four. Since patch codes will unlikely require more than 16 bits to distinguish between patch services, we used only a half word for service identification in our first protocol for tunneling QMI messages. As a consequence, service handlers got a pointer which was not properly aligned for four byte accesses from our service dispatcher code and the patch programmer would need to take care of this. It is not possible to simply shift the whole buffer by two bytes as it is part of a structure passed to a QMI function from *Qual-*

comm which also needs to be word aligned. Thus, even though not required and increasing the overhead by two bytes, we now reserved a full word for service identification, making the programming of handlers more intuitive and less error prone.

A.2.1.2 *Hexagon assembler*

Disassembled *Hexagon* code contains immediate extenders (*immext*). Unfortunately, the DSP's assembler does not support these in its input but rather expects immediate values which should be extended preceded by two sharps. As a result, code produced by a disassembler, be it *Qualcomm's* own implementation or the *hexagondisasm* disassembler from [26], cannot be fed into the assembler again directly. We have to remove the *immext* instructions before. This is required in the generation of *fw_org* functions where we disassemble the first instructions of the original function and implement them again in inline assembler in the generated function.

A.2.1.3 *Hexagon linker*

In Section 9.2.3.2, we presented how *fw_org* functions are generated. We showed that some immediate values are relative to the instructions position and, therefore, need to to be adapted when the instruction is moved to another location. We explained that we can simply define symbols for these which will be resolved automatically by the linker.

The obvious way is to define a symbol with a value of the instructions original address, or the start address of the packet to be precise, and add the original immediate value to it in the generated assembly code. Thereby, the linker will handle problems with too large immediate values. If the instruction is, for example, a jump further than it can be encoded in the immediate field, the linker will generate "trampoline" code which performs the far jump and redirect the limited range jump to this code.

Unfortunately, the linker has a bug which can be triggered by this. Whenever two jumps in the same instruction packet are so far that they require a trampoline, meaning we would use the same symbol for them and just add different immediate values, the linker generates only a single trampoline, to the destination of the first jump. As a result, the second jump ends in a wrong destination.

We solved this problem by not using a sum of a symbol and an immediate value but rather generating a symbol directly for the absolute value (destination) and using this in the generated assembly code.

A.2.2 *Fastboot with the Asus PadFone Infinity 2*

Android devices can be booted into a special *fastboot* mode. The boot process then stops in the bootloader and it is, for example, possible to boot a custom kernel, if the bootloader is unlocked to allow this. Usually, pressing a certain key combination when turning on the device (e.g. *volume up* + *power*) or a reboot option from a running Android system allows to reach this mode.

According to several communities in the Internet, this should also be possible with the *Asus PadFone Infinity 2*. Regrettably, our device simply ignores the pressed key or the reboot target and boots the Android system instead. However, we needed to load our own kernel image.

This issue could be overcome with a simple trick. Android smartphones contain another mode called *recovery* which starts without problems on the device. We intentionally break this recovery, so that it cannot be executed anymore. Then, we tell the device to boot into this mode, which results in an error and the PadFone staying in the bootloader, giving us the desired *fastboot* mode. Listing 8 contains a short script for this.

Note that the *recovery* mode is automatically repaired. Therefore, at the next start of the device, the *recovery* mode is working again and our trick did not break any functionality of the phone.

Listing 8: Script to start *fastboot* on the Asus PadFone Infinity 2

```
1 #!/bin/sh
2 # write broken recovery
3 adb shell "su -c dd if=/dev/zero
     ↪ of=/dev/block/platform/msm_sdcc.1/by-name/recovery bs=2048
     ↪ count=6995; sync;"
4 # reboot to recovery
5 adb reboot recovery
```

[1] ARM Limited. "ARM1156T2-S Technical Reference Manual." Revision: r0p4. July 2007.

[2] M. A. A. Al-qaness et al. "Device-Free Home Intruder Detection and Alarm System Using Wi-Fi Channel State Information." In: *International Journal of Future Computer and Communication (IJFCC)* 5.4 (2016), p. 180. IACSIT Press.

[3] Asus Global. "The new PadFone Infinity (A86)." (last access: March 15, 2017). URL: `https://www.asus.com/Phone/The_new_PadFone_Infinity_A86/specifications/`.

[4] Azenqos. "AZQ Android - DriveTest Solution for LTE, WCDMA & GSM." (last access: March 3, 2017). URL: `http://www.azenqos.com/`.

[5] A. Basir. "Random Access Procedure in LTE." (last access: March 2, 2017). URL: `http://www.simpletechpost.com/2013/04/random-access-procedure-rach-in-lte.html`.

[6] E. Bendersky. "pycparser: Complete C99 parser in pure Python." (last access: February 4, 2017). URL: `https://github.com/eliben/pycparser`.

[7] G. Beniamini (laginimaineb). "Bits, Please!: Full TrustZone exploit for MSM8974." (last access: April 22, 2017). URL: `http://bits-please.blogspot.de/2015/08/full-trustzone-exploit-for-msm8974.html`.

[8] N. Bui and J. Widmer. "OWL: a reliable online watcher for LTE control channel measurements." In: *ArXiv e-prints* (2016). arXiv: `1606.00202 [cs.NI]`.

[9] J. Cichonski, J. M. Franklin, and M. Bartock. "LTE Architecture Overview and Security Analysis." In: *NIST Internal or Interagency Reports (NISTIR)* Draft 8071 (2016). NIST.

[10] L. Codrescu et al. "Hexagon DSP: An architecture optimized for mobile multimedia and communications." In: *IEEE Micro* 34.2 (2014), pp. 34–43. IEEE Computer Society.

[11] G. Delugré. "Reverse engineering a Qualcomm baseband." In: *28th Chaos Communication Congress (28C3)*. Berlin, Germany, 2011. CCC.

[12] S. Dhagle. "LTE Resources by Sandesh Dhagle." (last access: February 27, 2017). URL: `http://dhagle.in/LTE`.

[13] N. Golde and D. Komaromy (Comsecuris). “Breaking Band reverse engineering and exploiting the shannon baseband.” In: *Reverse engineering conference (REcon) 2016*. Montreal, Canada, 2016.

[14] L. Gwennap. “Qualcomm Extends Hexagon DSP.” In: *Microprocessor Report* 27 (2013), pp. 13–15. The Linley Group.

[15] D. Halperin et al. “Linux 802.11n CSI Tool.” (last access: February 23, 2017). URL: `http://dhalperi.github.io/linux-80211n-csitool/`.

[16] D. Halperin et al. “Tool Release: Gathering 802.11n Traces with Channel State Information.” In: *Computer Communication Review (CCR)* 41.1 (2011), p. 53. ACM SIGCOMM.

[17] W. Hengeveld. “IDA processor module for the hexagon (QDSP6v55) processor.” (last access: Januar 14, 2017). URL: `https://github.com/gsmk/hexagon`.

[18] International Data Corporation (IDC). “Smartphone OS Market Share.” (last access: Januar 10, 2017). URL: `http://www.idc.com/promo/smartphone-market-share`.

[19] A. Kloeckner. “pycparserext: Extensions for Eli Bendersky’s pycparser.” (last access: February 4, 2017). URL: `https://github.com/inducer/pycparserext`.

[20] H. Krawczyk, M. Bellare, and R. Canetti. “HMAC: Keyed-hashing for message authentication.” RFC 2104. 1997. IETF.

[21] S Kundojjala (Strategy Analytics). “Baseband Market Share Tracker Q2 2016: MediaTek and Spreadtrum Together Take One-third LTE Volume Share.” Tech. rep. Sept. 2016. Strategy Analytics.

[22] LG Electronics. “D820 Sprint Black.” (last access: March 15, 2017). URL: `http://www.lg.com/us/cell-phones/lg-D820-Sprint-Black-nexus-5`.

[23] Y. Li et al. “Mobileinsight: Extracting and analyzing cellular network information on smartphones.” In: *The 22nd Annual International Conference on Mobile Computing and Networking (MobiCom 2016)*. New York, USA, 2016, pp. 202–215. ACM SIGMOBILE.

[24] L. Miras. “Baseband playground.” In: *Ekoparty 2011*. Buenos Aires, Argentina, 2011.

[25] L. Molas et al. “Hexagoon IDA plugin.” (last access: Januar 14, 2017). URL: `https://github.com/programa-stic/hexag00n`.

[26] L. Molas et al. “Qualcomm Hexagon Disassembler.” (last access: February 4, 2017). URL: `https://github.com/programa-stic/hexag00n/tree/master/hexagondisasm`.

[27] A. Morgado (Lanedo GmbH). “Qualcomm Gobi devices in Linux based systems.” 2013.

[28] Qualcomm Technologies Inc. "Hexagon DSP SDK - Qualcomm Developer Network." (last access: Januar 14, 2017). URL: https://developer.qualcomm.com/software/hexagon-dsp-sdk.

[29] Qualcomm Technologies Inc. "Qualcomm 205 Mobile Platform." (last access: April 12, 2017). URL: https://www.qualcomm.com/products/qualcomm-205-mobile-platform.

[30] Qualcomm Technologies Inc. "QxDM Professional QUALCOMM eXtensible Diagnostic Monitor." (last access: March 3, 2017). URL: https://www.qualcomm.com/documents/qxdm-professional-qualcomm-extensible-diagnostic-monitor.

[31] Qualcomm Technologies Inc. "Snapdragon 800 Processor." (last access: March 15, 2017). URL: https://www.qualcomm.com/products/snapdragon/processors/800.

[32] Qualcomm Technologies Inc. "Hexagon V2 Programmers's Reference Manual." 80-NB419-1 Rev.A. Aug. 2011.

[33] Qualcomm Technologies Inc. "Hexagon V5x Programmers's Reference Manual." 80-N2040-8 Rev.H. Mar. 2016.

[34] J. Ryu. "Sharetechnote." (last access: February 24, 2017). URL: http://www.sharetechnote.com/.

[35] A. Satpathy. "Channels in LTE." (last access: February 26, 2017). URL: http://lteinwireless.blogspot.de/2011/05/channels-in-lte.html.

[36] M. Schulz et al. "Nexmon: The C-based Firmware Patching Framework for Broadcom/Cypress WiFi Chips that enables Monitor Mode, Frame Injection and much more." (last access: February 2, 2017). URL: https://github.com/seemoo-lab/nexmon.

[37] Unknown (published from GitHub user "remittor"). "Qualcomm mbn tools repository." (last access: February 7, 2017). URL: https://github.com/remittor/qcom-mbn-tools.

[38] V. van der Veen et al. "Drammer: Deterministic rowhammer attacks on mobile platforms." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Vienna, Austria, 2016, pp. 1675–1689. ACM.

[39] R.-P. Weinmann. "All your baseband are belong to us." In: *27th Chaos Communication Congress (27C3)*. Berlin, Germany, 2010. CCC.

[40] R.-P. Weinmann. "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks." In: *6th USENIX Workshop on Offensive Technologies (WOOT '12)*. Bellevue, USA, 2012, pp. 12–21. USENIX.

[41] R.-P. Weinmann. "Baseband exploitation in 2013: Hexagon challenges." In: *PACific SECurity (PacSec) 2013*. Tokyo, Japan, 2013. dragostech.com inc.

[42] Wireshark-Community. "MAC-LTE - The Wireshark Wiki." (last access: February 20, 2017). URL: `https://wiki.wireshark.org/MAC-LTE`.

[43] X. Xie et al. "piStream: Physical Layer Informed Adaptive Video Streaming over LTE." In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom 2015)*. Paris, France, 2015, pp. 413–425. ACM.

[44] tutorialspoint. "LTE Protocol Stack Layers." (last access: February 24, 2017). URL: `https://www.tutorialspoint.com/lte/lte_protocol_stack_layers.htm`.

[45] zynamics / Google Inc. "zynamics.com - BinDiff." (last access: March 17, 2017). URL: `https://www.zynamics.com/bindiff.html`.

THESIS STATEMENT

pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I have written the submitted Master Thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, May 1, 2017

Carsten Gerald Bruns

ERKLÄRUNG ZUR ABSCHLUSSARBEIT

gemäß § 22 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich die vorliegende Master Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, 1. Mai 2017

Carsten Gerald Bruns