RUHR
UNIVERSITÄT
BOCHUM
RUB

RUHR-UNIVERSITÄT BOCHUM

Extending Scapy by a GSM Air Interface

Laurent Weber

Master's Thesis. October 4, 2011.
Research Group Embedded Malware – Prof. Dr. Thorsten Holz

hgi EMMA

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

DATE AUTHOR

Contents

List of Figures v

List of Tables vii

List of Listings ix

1 Introduction 1

1.1 Motivation . . . 1
1.2 Contributions . . . 3
1.3 Related Work . . . 3
1.4 Outline . . . 4

2 Background 5

2.1 GSM . . . 5
2.1.1 General . . . 5
2.1.2 Structure of a GSM network . . . 6
2.1.3 Interfaces in the GSM System . . . 7
2.1.4 GSM protocol stack . . . 7
2.2 Scapy - Interactive packet manipulation tool . . . 9
2.3 USRP - Universal Software Radio Peripheral . . . 10
2.3.1 General . . . 10

3 Scapy-um Design 13

3.1 Overview . . . 13
3.2 Scope of the Code . . . 13
3.3 Presentation of the code . . . 13
3.3.1 From the Specification to the Code . . . 13
3.3.2 Stacking Information Elements to Complete Messages . . . 20
3.3.3 Optional Information Elements . . . 21
3.3.4 Optional Fields . . . 22
3.3.5 Information Element's Optional Headers . . . 22
3.3.6 Sending a Message . . . 22
3.4 Code Quality . . . 25
3.5 Summary . . . 25
3.6 Obtaining the Source Code . . . 25
3.7 Limitations . . . 25

4 Results **27**
4.1 Test Environment Setup . . . 27
4.1.1 Hardware . . . 27
4.1.2 Software . . . 28
4.2 Example: Normal Operations . . . 28
4.2.1 Recreate packets taken of a wireshark capture . . . 28
4.2.2 Call . . . 29
4.3 Case Study: Attacks . . . 33
4.3.1 Classical Attack . . . 33
4.3.2 Novel Attack . . . 38

5 Summary **41**
5.1 Limitation . . . 41
5.2 Future Work . . . 41
5.3 Conclusion . . . 41

6 Annexes **43**
6.0.1 Pcap captures . . . 43
6.1 Universal Software Radio Peripheral Setup . . . 47
6.1.1 General - Howto Ubuntu 11.04 . . . 47

Bibliography **51**

List of Figures

1.1 Presenting the air interface 2

2.1 Structure of a GSM network (key elements) 6
2.2 Parts of the GSM protocol stack 7
2.3 Our USRP1 setup 11

3.1 Graphical representation of the Protocol Discriminator IE and Skip Indicator IE 15
3.2 Graphical representation of a default Protocol Discriminator IE and Skip Indicator IE 16

4.1 Test environment setup 27
4.2 Wireshark analysis of an TMSI Reallocation Command Packet 29
4.3 Graphical representation of a call between two mobile phone users 30
4.4 Call protocol-run initiated by a Base Transceiver Station 31
4.5 Wireshark analysis of a response packet (Call Confirmed) 34
4.6 Wireshark analysis of a response packet (Alerting) 35
4.7 Whole IMSI DETACH INDICATION packet, graphical representation 36
4.8 Call Control protocol/MS side Finite State Machine 39
4.9 Call Clearance 39

List of Tables

3.1 Additional assignment message content, Table 9.2/GSM 04.08 [040b] 14

4.1 IE, presence and length fields of IMSI DETACH INDICATION message 35

List of Listings

3.1 Adding Transaction identifier and Skip identifier to Scapy 15
3.2 Dynamic Message 16
3.3 Returning a configured IE 18
3.4 Adding the dynamic length to the information element 18
3.5 Remove unneeded bytes 18
3.6 The adapt method 19
3.7 Error exception 20
3.8 Stacking Information Elements to Complete Messages 20
3.9 Maximum length additionalAssignment message 21
3.10 Default additionalAssignment message 22
3.11 Sending a Message 23
3.12 Method adding sendum() to Scapy 24

4.1 Recreate TMSI Reallocation Command 29
4.2 Setup of a phone call 32
4.3 OpenBTS logfile while setting up a phone call 32
4.4 Setup a phone call 32
4.5 OpenBTS logfile while setting up a phone call 33
4.6 IMSI DETACH Attack 37
4.7 Authentication reject Attack 37
4.8 Novel Attack Example 1 38
4.9 Novel Attack Example 2 40

6.1 Frame 28 43
6.2 Frame 34 43
6.3 Frame 35 43
6.4 Frame 36 43
6.5 Frame 39 43
6.6 Frame 40 44
6.7 Frame 45 44
6.8 Frame 47 44
6.9 Frame 48 44
6.10 Frame 51 44
6.11 Frame 55 45
6.12 Frame 60 45
6.13 Frame 65 45
6.14 Frame 68 45

6.15 Frame 70 45
6.16 Frame 88 45
6.17 Frame 90 45
6.18 Frame 91 46
6.19 Frame 94 46
6.20 Frame 129 46
6.22 Frame 168 47
6.23 Frame 179 47
6.24 All Ubuntu dependencies needed 47
6.25 Install GNU Radio [gnu] 48
6.26 OpenBTS [opec] & [opeb] 48

Abstract

This thesis describes the enhancement of Scapy, the powerful interactive packet manipulation program, by the layer-3 of the Global System for Mobile Communication (GSM) protocol. Layer-3 of the GSM protocol is part of the UM-interface, which is the air interface connecting the mobile devices to the operators' network. In addition to the demonstration of the addon, we will introduce new attacks on the GSM baseband, targeting the logic of the baseband state-machine.

Thus far attacks on GSM were mainly directed to vulnerable code running directly on the phone. Recently a totally new attack vector was successfully used to exploit Mobile Stations over the air, attacks on the baseband stack. Security researchers working on GSM baseband security lack of open-source tools to analyze the security of the baseband stack. This thesis introduces a Scapy-addon allowing users to create GSM layer 3 packets using simple Python syntax. Furthermore, this thesis will continue the effort of security researchers to test the security of the baseband stack, that has been, until now, neglected. This is done using and enhancing already existing open-source tools. In addition, possible scenarios of novel attacks on the GSM baseband stack are discussed. This thesis demonstrates attacks and tests on the logic of the GSM state machine using our newly created addon. One of our results is that classical attacks, found in the literature, can be easily rebuild using our tool. Furthermore, possibly vulnerable parts of the GSM state machine are explored and discussed.

To the best of our knowledge there is no prior work presenting a tool allowing to build the whole layer 3 of the GSM specification on the command line, as well as there is no work presenting attacks on the state machine of the GSM baseband stack, so far. In a nutshell, while one focus is to introduce the new part of Scapy, another focus is put on classical as well as on novel attacks.

Acknowledgements

First, I would like to express my gratitude to my advisor Prof. Thorsten Holz for this support during this thesis.
Furthermore, my sincere thanks go to *GSMK Gesellschaft für sichere Mobile Kommunikation mbH* for the USRP1 they lend me, and specially to Tobias Engel for testing my software and sharing his immense knowledge in numerous discussions.
I thank my fellow student Martin Steegmanns for the long and stimulating discussions and good coffee as well as StalkR and Philippe Biondi for sharing their experience with Scapy. I also want to thank Tim Kornau for his help as well as Ralf-Philipp Weinmann for his support in debugging a broken USRP and for the discussion and ideas he gave me. Collin Mulliner was a great discussion partner too, so I also want to thank him for his support.
Then I would like to thank the organisation teams of the *hack.lu 2011*, *Hack In The Box Malaysia 2011* and *DeepSec 2011* for letting me present my work on their great conferences.
Last but not the least, I would like to thank my family for supporting me financially and spiritually, among other, during the time of my studies.

1 Introduction

1.1 Motivation

The *Global System for Mobile Communication (GSM)* standard describes a full duplex, digital switched network circuit, optimized for voice communication. This standard has been around for quite a long time now, but the security of that protocol has never really been analyzed by independent security-researchers in the past. This is due to the high amount of money needed to build an own test GSM network allowing testing without interfering with the normal operation of a mobile provider. Recently, the cost needed to operate a test GSM network were reduced. Researchers can now setup their own network for less than 3000 Euro.

Since nowadays nearly everyone owns at least one mobile phone or GSM capable device, it seems very important that the security of that protocol is given. Criminals do not hesitate to exploit vulnerabilities for their own profit, for example in industrial espionage, eavesdropping a GSM conversation of some high responsible of a company as described by Lauri Pesonen [Pes]. Another possible malicious operation could be to trace people and do profiling as shown by the authors of [KYY09]. While a lot of effort has been placed in securing computer-based infrastructures like PCs and networks, GSM security has been ignored for a long time. However a turnover has started, open-source base-stations and open-source baseband stacks have appeared. This brings the cost of building an own test-GSM-network down to an affordable price, which allows a large number of researchers to analyze the protocols. Furthermore, the open-source community invested a lot of work in creating an open GSM baseband stack:*osmocombb* [osm]. The according Base Transceiver Station (BTS) *OpenBTS* [opeb] and Base Station Controller (BSC): *OpenBSC* [opea] were created. All those tools are cooperating with GSM networks owned by telephone companies.

As stated before, a lot of researchers are now focusing on GSM security, for example: Ralf-Philipp Weinmann [Wei] with his attacks on the baseband stacks over the air, Collin Mulliner [MGS] fuzzing Mobile Stations using SMS messages and Karsten Nohl [NP] performing practical attacks on the cipher of GSM. One goal is to spot and fix the problems available in the current GSM protocol. Of course, the research community is not able to force any proprietary stack owner to fix the vulnerabilities detected, but researchers can write and commit patches for the open-source tools [opeb] and [opea] as well as [osm], to achieve a more secure GSM environment.
The analyses that have been done by researchers so far were mainly manual and often static analysis of the code related to the baseband processor or the ciphering, work done, among others, by the authors of [Kas06], [BSW99] and [Cla]. A baseband processor is mostly an ARM processor running the GSM protocol stack. This kind of analysis is one possible approach for a protocol

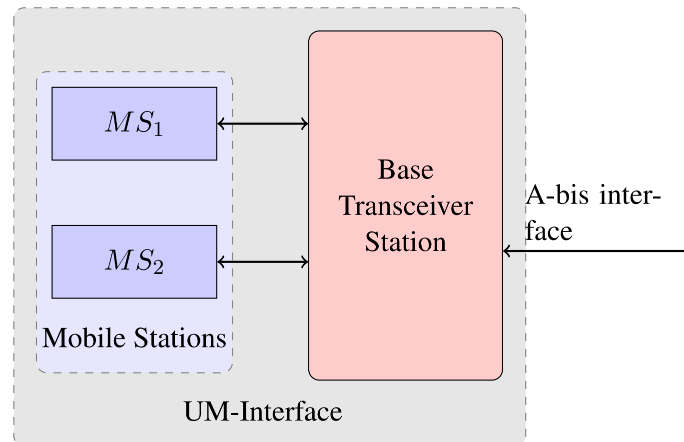

Figure 1.1: Presenting the air interface

analysis, but it will take a huge amount of time to find trivial bugs in the baseband stack. The motivation of this master thesis is to create a tool, allowing researchers to create any possible GSM frames and, for example, send them to mobile devices over an Universal Software Radio Peripheral (USRP) [ett]. The tool that has been build allows researchers to use it in an easy way and is open-source. This gives every user the possibility to enhance the tool or modify it to fit his needs. Such a program allows researcher to cover the different stacks in a faster way than a manual approach.

This thesis will focus on a tool called Scapy [Bioa], a powerful interactive packet manipulation program, widely used by IT-security researchers. Scapy allows users to create, forge as well as decode packets on the command line. Figure 1.1 illustrates the part of GSM network we will focus on for this thesis.

The external frame of the picture presents the UM-interface. The UM-Interface is the air interface between Mobile Station (MS) and the base transceiver station. MS can be every GSM capable device, like for example a mobile phone. This is the part of the GSM network this thesis focuses on. In Figure 1.1 the Mobile Stations are represented by the blue boxes, they can be any kind of GSM capable device. Mobile subscribers are the users of the network. The other entity playing a role in this part of the network is the Base Transceiver Station, represented by a red box in the figure. Usually such antennas are operated by GSM network operators. In this thesis the network will be run by ourselves using an USRP. This allows us to stay very close to a normal operated network since OpenBTS implements a software-based GSM access point without interfering with operators normal operations. OpenBTS implements the GSM protocol stack, and so GSM capable devices are able to connect to it and use it to place phone calls. A large testing network that validates the implementation has been setup in 2010 [niu]. The A-bis interface is used between the BTS and the BSC, and not part of the UM-interface anymore.

The GSM protocol is providing partly encrypted communication to the BTS. Recent attacks

like Ralf-Philipp Weinmanns attack on the baseband stack [Wei], that are not targeting the encryption, showed us that the security feeling created by this encryption is wrong. It is also important to mention that attacks on the A5/1 encryption have been successfully performed for example using a tool called Kraken [Ste]. Anyway, if the baseband stack can be attacked over the air, the encryption is not helping. In order to have a good way to perform protocol analysis we created a tool. This tool gives the possibility to detect flaws in the baseband stack and protocol. These findings allow manufactures and open-source community to fix the problems. This will increase the security of the GSM protocol in general.

1.2 Contributions

This section illustrates which contributions were done through this thesis.

First of all, a new tool was created. The program designed on top of a well known packet manipulation tool called Scapy has around 14000 lines of code and implements the whole layer 3 specifications of GSM [040b]. This open-source piece of code allows to create every imaginable layer 3 message and to send it from a Mobile Station, like for example a mobile phone to a Base Transceiver Station, or vis-versa using hard- and software that allows to act as a network operator.

Secondly, this thesis discusses known, well documented security risks present in the GSM network, by recreating classical attacks. Next to these know risks the work done in this thesis kicks off a completely new attack scenario never documented for the GSM network. Our approach targets the state machine of the mobile devices, nevertheless our tools permits to perform the same attack on the GSM network infrastructure of an operator.

1.3 Related Work

The related work section will be divided in two parts. First, attacks on base station systems and secondly attacks on the mobile phone base bands.
The author of [Gru10] presents attacks on different parts of the GSM network: He attacked the Mobile Station baseband itself while fuzzing the phones over a fake BTS created using an USRP. He also attacked the GSM operator's infrastructure, while sending GSM packets from a malicious phone. He claims to have a possibility to inject prepared packets in layer 3 of the air interface, but never publicly released the code of this. The work presented in this thesis is substantially different from his approach. All the code written to achieve the goal will be released under an opensource licence, in order to allow independent researchers to use it and enhance it.
The author of [Wei] has reached widely publicized and acknowledged results while analysing the baseband stack of current mobile phones. As proof-of-concept he was able to set phones to silently auto-answer calls using vulnerabilities in the stack and executing AT commands [Wei].
One of the aims of this thesis, and the related tool, is to give the ability to researchers to easily build packets from the command line. Such that validation of a security risk can be performed

with ease.
The effort placed in this thesis will arm the security community with a new tool: a packet builder for the layer 3 of the air interface, called *Scapy gsm-um*. The whole specification for layer 3 [040b] has been implemented and is ready to use for fuzzing, crafting packets or rebuilding legit packets. Scapy gsm-um should help the community to understand the protocol in detail and provides a method to build packets in an easy way.
This work differs a lot from most papers released in this field. While the authors of [MGS] say that they are attacking feature phones because they are wider used than smartphones, others for example the authors of [LI] see

smartphones as the interesting target because the impact of a successful attack has an higher importance in terms of data/information that an attacker could gather. This thesis introduces a framework providing the ability to attack any device that has a baseband stack and uses the GSM protocol.

1.4 Outline

The second chapter of this work will focus on the background notions needed to understand what we present in this thesis. GSM will be introduced in a rather compressed and general overview. The software we use as well as the hardware will also be introduced in the second chapter.
A general overview of the code and how it was created, what problems were encountered and how they were solved will be presented in the third chapter.
The forth chapter will focus on the results of this implementation, by starting with a description of the test environment that was used. Then a demonstration of how the tool is used and a validation of the tool by recreating GSM layer 3 packets used in normal GSM operations. The chapter will be concluded by a case study of different attacks that were recreated using the tool.
The last chapter presents the limitation of our work as well as the future work and a general conclusion is presented.

2 Background

This section describes the history of GSM from 1982 till now. A brief insight into the structure of a classical GSM network is also presented as well as some definition of terms used in the context of GSM. The second part of this section introduces Scapy, the powerful interactive packet manipulation tool. The last part of this section focuses on the Universal Software Radio Peripheral.

2.1 GSM

2.1.1 General

The European Telecommunications Standards Institute (ETSI) develops the GSM standard, which specifies how technologies of the second generation (2G) cellular networks have to work. The 2G is a replacement for the first generation of cellar networks. One difference is that the first generation was an analog cellular network while 2G is digital. 2G should also describe an optimized switched network for full duplex voice communication. Later the standard was expanded and it includes support for switched data transport, and even packet data transport via General Packet Radio Service (GPRS). Enhanced Data Rates for GSM Evolution (EDGE) allowed even faster transmission speeds. The *2G* is followed by the third generation *3G*, also known as Universal Mobile Telecommunications System (UMTS) engineered the by 3rd Generation Partnership Project (3GPP). For this thesis we place the focus on 2G.

The GSM standardization was not deployed instantly, in fact a huge amount of stages had to be passed to setup the GSM network around the world. Some of the stages are presented in the following paragraph, starting form 1982 where a working group to establish a standard was founded [Wal01].

1982 A working group for mobile radio is setup in the European Conference of Postal and Telecommunications Administration (CEPT). Its work is to establish a European standard for mobile radio. 26 countries are taking part in the effort.

1985 Germany, Italy and France are subscribing to a development contract for the new standard.

1987 17 GSM Network-provider from 15 countries are signing the GSM *Memorandum of Understanding (MoU)*

1990 The GSM 900 Standard specifications get frozen, so that they can be used to build mobile-phones and network-technology.

1991 The first GSM call is completed by a Finnish operator called *Radiolinja*.

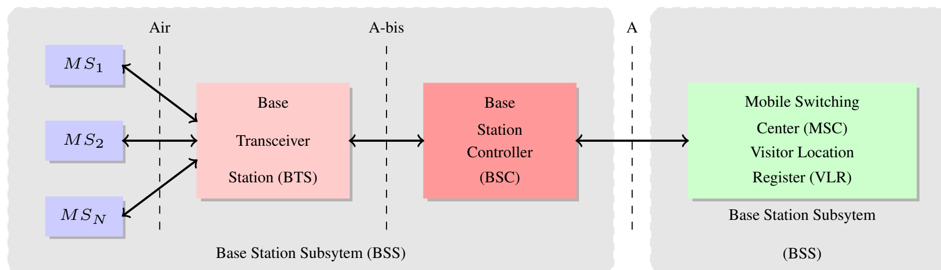

Figure 2.1: Structure of a GSM network (key elements)

1991 Presentation of the first working systems.

1992 First text message (*Short Message Service (SMS)*) is send.

1992 A lot of European GSM 900 Operators start to run their networks.

1993 The first GSM network operating at 1800 Mhz became operational in the United Kingdom.

1996 Pre-Paid SIM cards are launched.

2000 The GSM standardization activities are given to 3GPP.

2003 The number of worldwide GSM subscribers passes 1 billion users.

2.1.2 Structure of a GSM network

GSM networks are structured following a standardized buildup. A coarse representation of a GSM network is presented in Figure 2.1. The representation of the Network Subsystem (NSS) has been simplified a lot in this representation, since it is not very important to understand the rest of the thesis. More information can be found in Bernhard Walkes' book [Wal01]. Figure 2.1 provides a high-level overview of a GSM network. It shows where the part of the network we work on is located within the whole infrastructure. GSM networks are subdivided into four parts, which we present in the following:

Definition. MS *contains all the equipment (hard and software) needed to communicate with a mobile network.*

Definition. Base Station Subsystem (BSS) *has the responsibility to handle traffic as well as signaling between MS and NSS*

Definition. NSS *carries out call switching as well as mobility management functions for roaming on the network of base stations.*

Definition. *GPRS Core Network (optional) allows packet based Internet connections.*

The arrows in Figure 2.1 represent the different interfaces used by the entities to communicate with each other, the relevant interfaces for this thesis are presented in the next section.

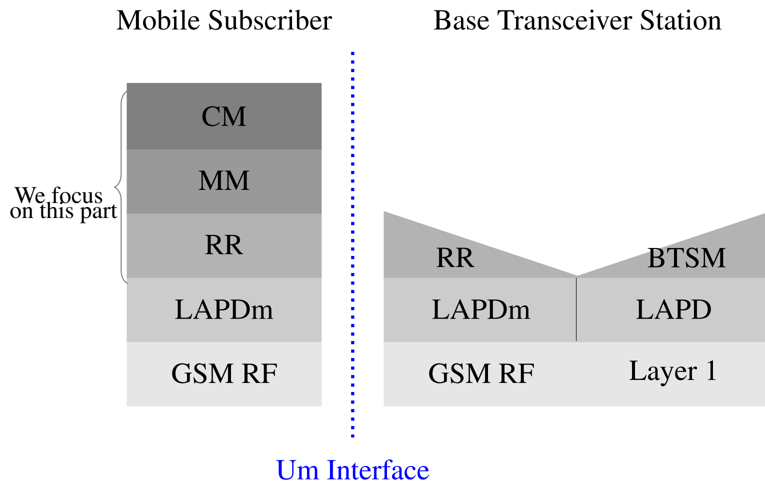

Figure 2.2: Parts of the GSM protocol stack

2.1.3 Interfaces in the GSM System

GSM networks use different interfaces depending on which parts of the network are communication together. The definition of those interfaces follows, for the rest of the thesis we will only focus on the air interface. BTS-BSC *interface:* $\mathcal{A}_{bis}$
Transmissions over the $\mathcal{A}_{bis}$ Interface are based on Pulse-Code Modulation 30 (PCM-30) respectively 64kbit/s interfaces. Since *Public Land Mobile Network (PLMN)*-Operator often are not operating the telecommunication interface a sub-multiplexing technique has been standardized allowing four 16 kbit/s channels on one 64kbit/s channel. This allows the PLNM to save money [Wal01].

BSS-MSC *interface:* $\mathcal{A}$
Voice and data are transmitted digitally over the $\mathcal{A}$ interface. The transmission is done over PCM-30 following the ISDN-Standard (ITU-T-serie G.732). A PCM-30-System consists of 30 full-duplex channels having each 64 kbit/s with a transmission rate of 2,048 Mbit/s in full-duplex. Synchronisation and signalisation are done over 2 64 kbit/s channels, the D_2-channels [Wal01].
MS-BTS *interface:* $\mathcal{U}_m$
The radio interface is located between the MS and the BTS. The speed reached over the air-interface is 270,833 kbit/s [Wal01]. This is the interface we put the focus on for this thesis.

2.1.4 GSM protocol stack

Figure 2.2 represents the GSM protocol stack. The representation of the stack is similar to a representation of an ISO model, known from other network services and is read in the same way. The physical properties of the radio transmission and the GSM specification require an adapted protocol communication to perform the signalisation between the network-elements.

Signaling System No 7 (SS.7) (ITU-T-Serie Q.700-795) is used to perform this signalisation. It was enhanced by Mobile Application Part (MAP) to deal with the special signalisation need of *GSM-PLMN*.
In Figure 2.2 on page 7 the diagonal cuts in the last layers represent the transfer functions used to transform the packets from one GSM node to the next.

2.2 Scapy - Interactive packet manipulation tool

"Scapy can for the moment replace hping, parts of nmap, arpspoof, arp-sk, arping, tcpdump, tshark, p0f, ..."

Scapy[Bioa] is an interactive packet manipulation tool written by *Philippe Biondi* [Biob]. Scapy is written in Python and allows users to use Python programming language within the context of the tool. Direct interaction with the Python interpreter makes Scapy very powerful.
Scapy can be used to perform a wide range of actions, for example:

- generate packets,
- manipulate packets,
- network scanning,
- network discovery,
- packet sniffing.

Furthermore, Scapy can be used to decode or forge packets of a huge list of protocols. It can even capture packets, match request and reply accordingly. The tool supports a large list of protocols, and more are added regularly. The user community behind Scapy is very active and is submitting a lot of new protocols for each release [new].
Scapy has been created having flexibility in mind. In contrast to usual networking tools, where users only have the possibility to build things the author imagined, Scapy for its part allows to perform every possible task, even if the author was not aware that it even could be possible.
Decoding and interpreting are features that are often confused by tools. Machines decode network traffic following strict rules. They are good at this. The user interacting with the tool should be responsible for the interpretation of the decoded traffic. A lot of programs try to overtake this interpretation part too, which takes away a lot of information for the user. For example there are tools that say *"this port is open"* when they receive a *SYN-ACK* packet, instead of providing the user with the entire information. Scapy is providing the whole information to the user. This surely makes Scapy a tool for people knowing what they do, and less attractive for beginners. *"You're free to put any value you want in any field you want, and stack them like you want. You're an adult after all."* this quote from [quo] is clarifying the spirit Scapy is following.
Scapy allows the user to build a new tool within minutes. It has been used to recreate tools that took hundreds of lines in C, but only some little lines in Scapy as the author of [OCo10] points out.
Another aspect that makes Scapy such a great tool is the fact that adding a protocol is really simple. Depending of the protocol complexity the implementation of a Scapy addon might only take several minutes as described in [bui] & [ext].

To summarize, Scapy is a very powerful packet manipulation tool written in Python, that allows fast creation of custom packets for a large number of protocols. The easy way to add new protocols allows us to create an addon for the gsm layer3 protocol without having to create an entire new program.

2.3 USRP - Universal Software Radio Peripheral

In order to be able to impersonate the operators' network we need to use a special piece of hardware called: *Universal Software Radio Peripheral (USRP)* . An USRP allows us to connect Mobile Stations to it like it was a legit GSM network. Even if the USRP can use *Session Initiation Protocol (SIP)* to perform phone calls to other networks, for this thesis no SIP connection is needed.

2.3.1 General

Ettus Research LLC and its parent company, National Instuments developed the USRP which is a hardware device allowing researchers to create and operate their own software radio. The low price of the USRP, compared to other hardware, gives a large number of users the possibility to build their own radio device. Indeed, a price of under 3000 Euros is affordable even for free-time researchers, a lot of people place way more money in their hobbies. The USRP is connected to a host computer through a high-speed USB (USRP1) or Gigabit Ethernet. Through this link the host system is able to control the USRP and compute signals to transmit, or work on received signals.
The USRP relies on an open-source spirit, the board logic as well as drivers are open-source and available online. Furthermore the project is supported by a large online community.
An USRP1 consist of the following hardware:

- Four 64 MS/s 12-bit analog to digital converters,
- Four 128 MS/s 14-bit digital to analog converters,
- Four digital downconverters with programmable decimal rates,
- Two digital upconvertes with programmable interpolation rates,
- High-speed USB 2.0 interface (480Mb/s),
- Capable of processing signals up to 16 Mhz wide,
- Modular architecture supports wide variety of RF daughterboards,
- Auxilliary analog and digital I/O support complex radio controls sucha as RSSI and AGC,
- Fully coherent multi-channel systems (MIMO capable).

This and more information can be on the Ettus web presence [ett]. The hardware of the other versions of the USRP can be found in [pro]. Our exact setup is presented in Figure 2.3.

USRP's are often used, among others, as Radio-frequency identification (RFID), Global Positioning System (GPS) receiver, FM- receiver and transceiver and of course cellular GSM base stations. We use the USRP for the last scenario. In order to operate a GSM network, like we want to do in the scope of this paper we need GNU Radio and OpenBTS.

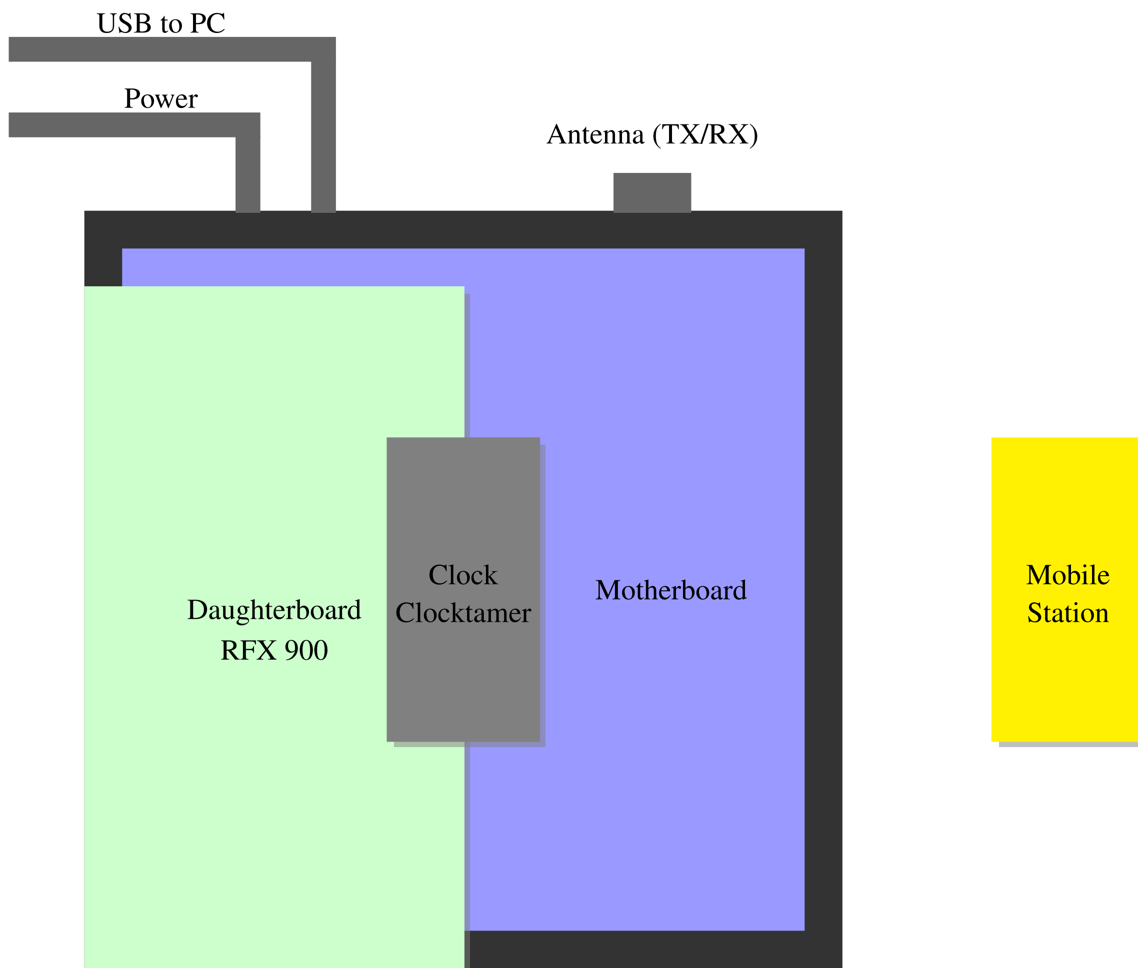

Figure 2.3: Our USRP1 setup

Taken together an USRP allows real time receiving and transmitting of signals. This behaviour is ideal for our use case. We can rely on the USRP to provide a good way to simulate a GSM network and analyze the security of the protocol.

3 Scapy-um Design

After reading and understanding the GSM specification related to layer 3 of the air interface, we started to design and implement the code for Scapy-um.

3.1 Overview

In order to give our tool the best possible coverage, we first implemented a way for Scapy to send packets to three types of sockets: UDP and TCP sockets as well as Unix Domain Sockets. We designed these three sockets to provide the maximum flexibility to the user and existing tools. To this end an USRP running OpenBTS provides a function called *testcall* [tes]. This allows users to use an UDP socket to send layer 3 messages over the USRP. Tools like OsmocomBB are using Unix Domain Sockets for their interaction with the user. Therefore, it seems logical that, if once there is a way to send data over OsmocomBB, it will be an Unix Domain Socket, so gsm-um is ready. Finally, to allow most adaptability to, maybe already existing, tools we rounded up the method with an TCP socket.

3.2 Scope of the Code

The code presented in this chapter does not implement the whole GSM protocol. It places the focus on the core part of the um/Air interface, also known as the layer 3. The specifications that has been implemented is GSM 04.08 [040b]. This allows to use the *Connection Management (CM)*, *Resource Management (RR)* and *Mobility Management (MM)* parts of the GSM standard. Implementation of SMS support is described in [030] and not implemented in the scope of this thesis.

3.3 Presentation of the code

This section introduces different concepts used to create the Scapy addon and illustrate them using code snippets.

3.3.1 From the Specification to the Code

At a first look the more than 700 pages specifications describing the layer 3 of GSM seem quite impressive, nevertheless the author decided to use a manual approach to create the addon and through this learn the standard at the same time. The specifications start with about 300 verbose pages explaining every detail of the layer 3, including state-machines and the like. These pages are interesting for the understanding of how GSM works, but less important for the design of

IEI	Type/Reference	Information Element	Presence	Format	Length
	RR management Protocol Discriminator	Protocol Discriminator 10.2	M	V	1/2
	Skip Indicator	Skip Indicator 10.3.1	M	V	1/2
	Additional Assignment Message Type	Message Type 10.4	M	V	1
	Channel Description	Channel Description 10.5.2.25a	M	V	3
72	Mobile Allocation	Mobile Allocation 10.5.2.21	C	TLV	3-10
7C	Starting Time	Starting Time 10.5.2.38	O	TV	3

Table 3.1: Additional assignment message content, Table 9.2/GSM 04.08 [040b]

Scapy-um. For this part of this thesis we will focus on section 9 of *Message functional definitions and contents* from [040b].
Now, we introduce the first message of the specifications for illustration purposes: The Additional Assignment message. Its content is formed by the Table 3.1.
The table holds all the information needed for us to recreate the message using coding techniques. First of all we explain the different columns and rows of Table 3.1. This paragraph is meant as a help for novice readers to understand the most important information we can take out of the table. Detailed information about the columns are referenced for advanced readers.
The column *IEI* represents the Information Element Identifier (IEI), in hexadecimal notation. IEI's are used to identify information element types. The IEI is only indicated if the format of the Information Element (IE) is: T, TV or TLV (The meaning of these abbreviations will be explained later). Further details can be found in [040b] Section 9.b.1.
The *Type/Reference* column is intended to give an idea of the semantics of the information element. This name is used to reference the IE in the specification. More detail on this can be found in [040b] Section 9.b.2.
The *Information Element* points to the subsection of Section 10 of the specifications where the value of the element can be found.
The *presence* tells us if the Information Element is required, possible values are: *Mandatory (M)*, *Conditional (C)* and *Optional (O)*, these values are defined in GSM 04.07. [040a] Section 11.2.5.
The column called *format* provides the information on which format the IE has to be. There are 5 different formats defined in [040a] Table 11.1 Section 10.2.1.1:

T Type only

V Value only

TV Type, Value

LV Length, Value

TLV Type, Length Value

Please see [040a] for more complete information.
Finally, the *length* column, this column represents the length of the IE in octets. This length might be fixed, like for example Channel Description in Table 3.1, or variable like for the IE Mobile Allocation.

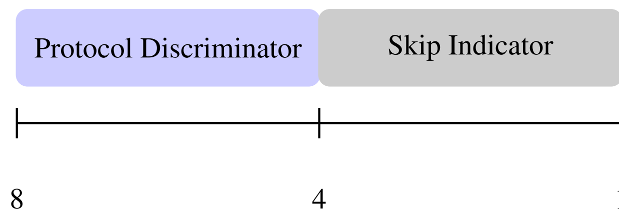

Figure 3.1: Graphical representation of the Protocol Discriminator IE and Skip Indicator IE

Having all these information we can now start to investigate the IEs and then finally implement each of them and stack them one upon the other to reach the message layout described in table 3.1.
How we created the Information Elements is illustrated for two IEs of the Additional assignment message in the following sections. The first shows the creation of a packet having a fixed length, the second introduces a variable length IE.

Fixed Length Information Element

In this part we introduce how we implemented Information Elements having a fixed length. As an example we take the Protocol Discriminator IE and Skip Indicator IE which both have a length of $\frac{1}{2}$ as defined in Table 3.1. The table tells us where we can find the IEs. The IEs are in Section 10.2 and 10.3.1 of the specifications. Since ,these two IEs always come together we implemented them in one layer of one byte. Our byte is organized in the way presented in Figure 3.1.

The Transaction identifier or Skip identifier is taking the bits 8 to 5, and Protocol Discriminator is placed from bit 4 to 1. This is a really easy IE, this IE has been choose to facilitate the understanding for the reader. Extending Scapy by this assembled IE is really easy following [ext] and [bui]. All we have to do is to create a new layer. This is done by creating a class, inheriting from the class *Packet* and define some variables. An example is presented in Listing 3.1.

Listing 3.1: Adding Transaction identifier and Skip identifier to Scapy

```
1 class TpPd(Packet):
2     """Skip Indicator & Protocol Discriminator
3        Section 10.2/10.3 """
4     name = "Skip Indicator and Protocol Discriminator"
5     fields_desc = [
6                  BitField("ti", 0x0, 4),
7                  BitField("pd", 0x7, 4)
8                  ]
```

Line 1 defines the class, inheriting from the Packet class. In line 2 we have an informative string telling us the complete name of the IE and in which section we can find it in the specifications. The *name* variable holds the name of the IE, which will be displayed inside Scapy. The

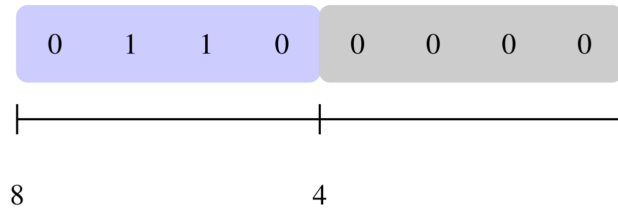

Figure 3.2: Graphical representation of a default Protocol Discriminator IE and Skip Indicator IE

actual structure of the packet is implemented in the array *fields_desc*. In this array, we define two fields of the format *BitField*, one having the name *ti* and a default value set to 0x0 and a size of four bits. The other has the name *pd* a default value set to 0x7 and also a size of four bits. A default *Skip Indicator and Protocol Discriminator* is defined as shown in Figure 3.2.
Scapy follows the idea to provide a fast packet manipulating tool. In order to be really fast it is important to choose decent default values, that do not break the packet. Unfortunately, GSM does not always allow us to choose good default values, but we did our best. In the example above, we choose 0x0 for the *ti* field, we did so because the specification states: *Bits 5 to 8 of the first octet of every Radio Resource management message and Mobility Management message and GPRS MobilityManagement message contains the skip indicator.* ***A message received with skip indicator different from 0000 shall be ignored****. A message received with skip indicator encoded as 0000 shall not be ignored (unless it is ignored for other reasons)* (see [040b] Section 10.3.1.) For the *pd* we choose 0x7 since 0110 is used for Radio Resource management messages, which are used quite often. The *pd* is one of the fields where we cannot take a default value that is works in any case.

Dynamic Length Information Element

While static, fixed length Information Elements are really easy to create we were confronted with a huge number of IEs having dynamic sizes, like for example the Mobile Allocation IE present in Table 3.1 which can have a size from 3 to 10 bytes. This type of Information Elements will be presented here. The concept of creating such a packed does not defer a lot from the packet presented in Section 3.3.1. Listing 3.2 shows an example of the code used to create dynamic messages.

Listing 3.2: Dynamic Message

```
1 class MobileAllocationHdr(Packet):
2     """Mobile Allocation Section 10.5.2.21"""
3     name = "Mobile Allocation"
4     fields_desc = [
5             BitField("eightBitMA", None, 1),
6             XBitField("ieiMA", None, 7),
7             XByteField("lengthMA", None),
```

```
8                ByteField("maC64", 0x12),
9                ByteField("maC56", None),   # optional fields start
10               ByteField("maC48", None),   # here
11               ByteField("maC40", None),
12               ByteField("maC32", None),
13               ByteField("maC24", None),
14               ByteField("maC16", None),
15               ByteField("maC8", None)
16               ]
17
18     def post_build(self, p, pay):
19         aList = []
20         a = []
21         i = 0
22         for i in range(0, len(self.fields_desc)):
23             aList.append(self.fields_desc[i].name)
24         for i in aList:
25             a.append(getattr(self, i))
26         res = adapt(3, 10, a, self.fields_desc)
27         if self.lengthMA is None:
28             p = p[:1] + struct.pack(">B", res[1]) + p[2:]
29         if res[0] is not 0:
30             p = p[:-res[0]]
31         return p
```

This snippet of the Scapy-um code has a higher complexity than the code presented in Section 3.3.1, this is not exclusively due to the bigger size, but more to the dynamic aspect of the code. This information element can have a size of 3 to 10 octets, allowing everything in between.
The beginning of the code is the same as in the previous example and we will skip it to put the focus on the important code. In *field_desc* we introduce new fields, that might not be self-explaining. the 8th bit of the IEI is reserved for possible future extension.(see [040b] page 369). In order to provide most flexibility we decided to model it as a single, addressable bit. The name of this bit is: *eightBitMA*, its default value is set to "None" and it has 1 bit length. Usually this value gets overwritten when a message is created, if not the value of "None" is set to "0x0" once the packet get build.
The next new field is *XByteField*, this has a predefined length (one byte) therefore it only needs two parameters, a name and a default value. The difference between an *XByteField* and a *ByteField* is the output representation. The output of a *ByteField* is decimal while *XByteField* returns hexadecimal values.
The code extract presents a new method: *def post_build()* This method is executed each time the packet is build (for example with the Scapy functions *myPacket.show2()* or *hexdump(myPacket)*) The values of the optional fields are all set to None, this configuration is used in the method, if the user changes the default value to something else we detect it when the packet is build and

we can adapt the *lengthMA* field to the length of the packet. The length is set using the code presented in Listing 3.3:

Listing 3.3: Returning a configured IE

```
1     def post_build(self, p, pay):
2         aList = []
3         a = []
4         i = 0
5         for i in range(0, len(self.fields_desc)):
6             aList.append(self.fields_desc[i].name)
7         for i in aList:
8             a.append(getattr(self, i))
9         res = adapt(3, 10, a, self.fields_desc)
10        if self.lengthMA is None:
11            p = p[:1] + struct.pack(">B", res[1]) + p[2:]
12        if res[0] is not 0:
13            p = p[:-res[0]]
14        return p
```

The post_build class method performs some action we need to make the message correct. First of all we define some variables: *aList*, and *a* which both are arrays and an integer *i*. *aList* contains the names of the fields we have in the *fields_desc array*. After this we add the string self to any element in *aList* and store it in the list *a*. Then we call a function *adapt()* using the minimal length, the maximum length, the list *a* and the array *self.fields_desc*. The function returns a list of two values, the first contains the value of the length we place in the *lengthMA* field. We place it at the correct place using the code of Listing 3.4.

Listing 3.4: Adding the dynamic length to the information element

```
1         if self.lengthMA is None:
2             p = p[:1] + struct.pack(">B", res[1]) + p[2:]
```

If the value of *lengthMA* is not set to *None* this means that the user has modified the value. If this is the case we do not overwrite the length of the packet with the computed one. This makes sense since providing a wrong information element length could be something users of Scapy-um want to test. If *lengthMA* is None we place the correct length value using struct.pack to the correct place in the packet. In this case we place it after the first byte. Which is the correct place if you have a look at the packet structure. How this is computed is described in detail hereafter. The second value of this list is the final length of the packet. The packet gets cut after the given size of bytes. This allows us to remove superfluous bytes that are set to "0". How this is done can be seen in Listing 3.5.

Listing 3.5: Remove unneeded bytes

```
1         if res[0] is not 0:
2             p = p[:-res[0]]
3         return p
```

Finally we return the entire packet.
The computation of the packet length as well as the value we set for the *length*-field of the packet is done in the *adapt* function. The code of this method is presented in Listing 3.6.

Listing 3.6: The adapt method

```
def adapt(min_length, max_length, fields, fields2, location=2):
    # find out how much bytes there are between min_length
    # and the location of the length field
    location = min_length - location
    i = len(fields) - 1
    rm = mysum = 0
    while i >= 0:
        if fields[i] is None:
            rm += 1
            try:
                mysum += fields2[i].size
            except AttributeError: # ByteFields don't have .size
                mysum += 8
        else:
            break
        i -= 1
    if mysum % 8 is 0:
        length = mysum / 8  # Number of bytes we have to delete
        dyn_length = (max_length - min_length - length)
        if dyn_length < 0:
            dyn_length = 0
        if length is max_length:  # Fix for packets that have
            length -= min_length  # all values set to None
        return [length, dyn_length + location]
    else:
        raise ErrorLength()
```

The function counts how many fields are set to *None*. The fields set to *None* are transformed to "*0*" when we build a packet. This behaviour is not what we need. Values of *None* should simply be removed of the information element. We walk through the array containing the values of the fields from the last to the first element. We remove only entire bytes this is why we sum up the size of the fields using the *.size* function. Since *ByteFields* do not implement this function we have to catch *AttributeError*. Once an *AttributeError* is thrown we know we are processing a ByteField: we need to add 8 to our sum. Once we hit the first field that is not set to *None* we break the loop. Since all the fields that are still set to *None* and are left have to be present in the message, they get set to *0*. Using modulo calculation we find out how many bytes we have to remove. The length of the complete information element packet contains the sum of the bytes of the entire packet. The length we have to put in the length field has to be computed. This is done

using:

$$dyn_length = (max_length - min_length - length)$$

The length-field contains only the size of the additional fields, not the overall length of the packet. Therefore we need to know the minimal size of the packet (everything that is smaller or equal to this length gets a length set to 0). We start to count with the first element that is optional and increment the count on every additional optional byte. Finally we return an array containing the overall length of the packet as well as the value that will be inserted in the length field of the information element we process.
If the user specified only a part of an byte and there are fields of that byte that are set to *None* on the right of this byte, an error is raised when the packet is build. This exception simply tells the user to set the last fields belonging to the octet the code can be found in Listing 3.7

Listing 3.7: Error exception

```
1 class ErrorLength(Exception):
2     def __str__(self):
3         error = "ERROR: Please make sure you build entire, 8 bit
4                 fields."
5         return repr(error)
```

3.3.2 Stacking Information Elements to Complete Messages

All the concepts introduced before are worth nothing if we are unable to stack the Scapy layers to get entire messages. How the code creates the whole messages out of IEs is presented in this section.
The 9th section of [040b] presents all the possible GSM messages related to layer 3. All those messages have been build in Scapy-um. This was done using methods for every message. In Listing 3.8 we create an additional assignment message.

Listing 3.8: Stacking Information Elements to Complete Messages

```
1 def additionalAssignment(MobileAllocation_presence=1,
2                          StartingTime_presence=1):
3     """ADDITIONAL ASSIGNMENT Section 9.1.1"""
4     # Mandatory
5     a = TpPd(pd=0x6)
6     b = MessageType(mesType=0x3B)  # 00111011
7     c = ChannelDescription()
8     packet = a / b / c
9     # Not Mandatory
10    if MobileAllocation_presence is 1:
11        d = MobileAllocationHdr(ieiMA=0x72, eightBitMA=0x0)
12        packet = packet / d
13    if StartingTime_presence is 1:
14        e = StartingTimeHdr(ieiST=0x7C, eightBitST=0x0)
```

```
15        packet = packet / e
16    return packet
```

The code creates an instance of every needed class, in our case we instantiate the classes described in Table 3.1. When we create the object we set some values that are non-default, for example we set the *pd* field of the *TpPd* to 0x6, since we know that the additional assignment message is a Radio Resource management message. Then we set the value of *mesType* of the object *MessageType* to 0x3B which is 00111011, we know this is the message type of this message, it is specified on page 368 of [040b]. Furthermore, we set the *iei* and *eightBit* values of the objects *MobileAllocation* and *StartingTime*. We do this since the specification tells us it wants to have IEIs for those two objects, this can be read out of Table 3.1 as well as the value of the IEI. The presence of those information elements is given through the method parameters: *MobileAllocation_presence=1* and *StartingTime_presence=1*, more on this in Section 3.3.3
Using the / operator we stack the objects created. Finally, we return a complete message that follows the specifications. This package is ready to send.

3.3.3 Optional Information Elements

As presented in Section 3.3.1 and implemented in Listing 3.9, some information elements are mandatory and other are not. To be precise, some information elements are mandatory, others are conditional and some are optional. This is clearly defined in the specifications [040b]. In order to reflect his behaviour in Scapy-um, we had to implement some tests. The implementations knows two different states for an information element, either it is present or it is not.

It is important to know that Scapy-um creates the shortest possible packet as default. This means that only information elements that are mandatory are present in messages that are created without any additional parameters. This behaviour has been chosen to speedup the creation of packets.
Creating the whole packet, including all the optional information elements is possible. This can be done by providing method parameters when calling the method. Unfortunately, this also brings some limitations, described in Section 3.7.
For illustration purposes, we now show how to create a message, similar to Listing 3.9 with the difference that it will be the longest possible message, holding all the possible information elements.

Listing 3.9: Maximum length additionalAssignment message

```
1 >>> a=additionalAssignment(MobileAllocation_presence=1,
2                            StartingTime_presence=1)
3 >>> a
4 <TpPd  pd=6 |<MessageType  mesType=0x3b |<ChannelDescription
5 |<MobileAllocationHdr  eightBitMA=0 ieiMA=0x72 |<StartingTimeHdr
6 eightBitST=0 ieiST=0x7c |>>>>>
7 >>> len(a)
8 8
```

While the default messages length 5 as shown in Listing 3.10.

Listing 3.10: Default additionalAssignment message

```
1 >>> a=additionalAssignment()
2 >>> a
3 <TpPd  pd=6 |<MessageType  mesType=0x3b |<ChannelDescription|>>>
4 >>> len(a)
5 5
```

To summarize, packets that are created without any special method parameters take the minimal message size. Longer messages cann be generated using method parameters.

3.3.4 Optional Fields

An Information Element consists of different fields. In Scapy, the fields described in the specifications have been implemented. Some of these fields are mandatory, and some are not. Information Elements that implement optional fields are most often of the Type, Length, Value (TLV) form. In order to deal with the length information as well as which fields are present or not all the optional fields are set to *None* in Scapy gsm-um. Once the packet gets build the superfluous fields are removed and only assigned fields are present in the final packet. The function that performs the computation has been introduced in Listing 3.6.

3.3.5 Information Element's Optional Headers

Depending on the place the information element takes in a message a header is required or not. In order to deal with this, we had to build two different classes for those information element. A first idea was to use Scapy's *ConditionalFields* (Fields that are bound to conditions that decide if they exist or not). This idea had to be dismissed since it is not possible to use *ConditionalFields* as the first field in a layer. The second idea was to build two classes for those elements, this was the approach we used. The information element that need a header have the name of the class ending with *Hdr*, for example:

$$class\ MobileAllocationHdr(Packet)$$

The same class, without the headerfields *eightBitMA* and *ieiMA* has the following definition:

$$class\ MobileAllocation(Packet)$$

This is an important information for users that which to build their messages from scratch without relying on the pre-defined packets. This difference is transparent to users using the pre-defined functions.

3.3.6 Sending a Message

The following section presents how a packet is prepared on the command line, and introduces a method we wrote to send the packet to different devices like an USRP or OsmocomBB.

Listing 3.11: Sending a Message

```
1  Welcome to Scapy (2.2.0)
2  Scapy GSM-UM (Air) Addon
3  >>> a=additionalAssignment()
4  >>> a
5  <TpPd  pd=6 |<MessageType  mesType=59 |<ChannelDescription
6  |<MobileAllocation  eightBitMA=0 ieiMA=114 |<StartingTime  eightBitST=0
7  ieiST=124 |>>>>>
8  >>> a.show()
9  ###[ Skip Indicator and Protocol Discriminator ]###
10 ti= 0
11 pd= 6
12 ###[ Message Type ]###
13 mesType= 59
14 ###[ Channel Description ]###
15 channelTyp= 0
16 tn= 0
17 tsc= 0
18 h= 1
19 maioHi= 0
20 maioLo= 0
21 hsn= 0
22 ###[ Mobile Allocation ]###
23     eightBitMA= 0
24     ieiMA= 114
25     lengthMA= None
26     maC64= 0
27     maC56= None
28     maC48= None
29     maC40= None
30     maC32= None
31     maC24= None
32     maC16= None
33     maC8= None
34 ###[ Starting Time ]###
35         eightBitST= 0
36         ieiST= 124
37         ra= 0
38         t1= 0
39         t3Hi= 0
40         t3Lo= 0
41         t2= 0
42 >>> hexdump(a)
43 000   06 3B 00 10 00 00 00 12  00 00 00 00 00 00 00 7C  .;.............|
44 010   00 00 00                                           ...
```

In line 25 of Listing 3.11 we see that the value of *lengthMA* is not computed when we use the *a.show()* function. The value is changed to 0 when we ask for a *hexdump()*, this is a correct behaviour since we did not specify any value for the optional *MobileAllocation*. The hexdump

shows us a lot of values that are set to 0, nevertheless the created message should be valid. Scapy is usually used for network traffic on computer networks so it is logical that when you call the *send()* method, Scapy wants to send the package over Ethernet. This is not working for us. What we need is a possibility to send layer 3 messages over an USRP or similar device. During our development, we had only an USRP so we were only able to test our code with an USRP. Nevertheless, we implemented solutions for similar working devices as presented in Listing 3.12.

Listing 3.12: Method adding sendum() to Scapy

```
1  def sendum(x,typeSock=0):
2  try:
3  if type(x) is not str:
4      x=str(x)
5  if typeSock is 0:
6      s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
7      host='127.0.0.1'
8      port=28670        # default for OpenBTS
9      s.connect((host,port))
10 elif typeSock is 1:
11     s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
12     s.connect("/tmp/osmoL")
13 elif typeSock is 2:
14     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
15     host='127.0.0.1'
16     port=43797
17     s.connect((host,port))
18 s.send(x)
19 s.close()
20 except:
21  print '[Error]: There was a problem when trying to transmit
22   data. Please make sure you started the socket server'
```

The patch is checking it the given parameter is already of type *str*, *str* represents a binary string. If not it transforms the input to *str*-format. The basic behaviour of the patch is to provide an UDP Socket on localhost port 28670 since that is the default port the *testcall* [tes] utility of OpenBTS uses. If the user prefers to have an Unix Domain Socket on localhost over the file */tmp/osmoL* the *sendum()* function needs to be called with *1* as second parameter. The second parameter has to be set to *2* if the user wants to have an TCP socket on port 43797 on localhost. We believe this patch should provide most flexibility to the users, independent of which hardware they want to use.

3.4 Code Quality

This section describes what has been done to enhance the code quality of gsm-um. Due to the large amount of code ($\approx$ 14000 lines of code), we decided to follow coding guidelines to ameliorate the readability of his code.

3.5 Summary

Since code is usually read more often than written, it is important to allow a quick understand of the code to users and feel comfortable with the source code. In order to allow this, widely used and well known coding guidelines were followed, the style guide for Python code PEP8 [pep]. To guarantee the compliance with this guidelines a PEP8-validator [val] was used.

To summarize this chapter, we are now able to create layer 3 packets of the GSM standard on the command line or through scripts. The code presented in this section provides the ability for researchers to quickly design messages following the GSM specifications and send them using their preferred hardware.

3.6 Obtaining the Source Code

> `Only wimps use tape backup: _real_ men just upload their important stuff on ftp, and let the rest of the rest of the world mirror it ;)`
>
> – Linus Torvalds [lin]

Following the idea of this quote the source code of Scapy gsm-um has been pushed to the contrib repository of Scapy. Due to this every user that clones the Scapy repository will get a copy of Scapy GSM-um Using mercurial this could be achieved using the following command:

hg clone http://hg.secdev.org/scapy my-scapy

3.7 Limitations

Due to the normal operational field of Scapy we encountered some problems when we tried to make it operate with our protocol. The GSM-Layer 3 packets sometimes have multiple occurrences of same information elements in one packet. This behaviour is non existing if you stack IP packets on top of TCP and so on. The impact of this complication on Scapy is that if a information element is used multiple times in the same message, only the values of the first can be modified. Most information elements that gets repeated in a packet are optional information elements, so the problem is only present if packets are needed that implement those optional information elements. The problem can be avoided if the user creates the information element

manually and then assembles the message directly on the command-line or in a script. An implementation walk-around had been implemented, but some important functionality of Scapy were broken after. Due to the rareness of the problems encountered during the development and testing of the addon we choose to let the code stay as it is.
Due to the high amount of different packets that can be build and the different possible structures due to optional fields and optional information elements, dissecting an incoming packet has limited functionality too. The code is at this time unable to dissect packets that implement optional fields and information elements automatically. This time again a manual dissection leads to success.

4 Results

This chapter presents the results we achieved with our tests. First we introduce our test environment, the hardware and software parts we use. The second part of this chapter deals with the tests we performed, starting with easy tasks like recreating packets out of captures and then we go over to everyday day tasks like setting up a phone call. The next part will present classical attacks we recreated with Scapy gsm-um and we finish with novel attacks.

4.1 Test Environment Setup

The following section will introduce the test equipment we used to validate the implementation and development of attacks for testing purposes. This should allow to recreate our test-cases and verify the results. Figure 4.1 shows what our test environment looks like.

4.1.1 Hardware

This section will present the used hardware for the tests described later.

Phones

At the moment of writing, different phones have been involved in the testing/setup of the system: Nokia 3510i, Motorolla C123 and finally a Nokia E71. The large majority of the test were performed on the Nokia phones.

USRP

We used an USRP1 with an RFX 900 [rfx] as well as a clocktamer [clo] for our tests. This setup is sufficient for our purpose. Since we are not running a whole GSM network with plenty

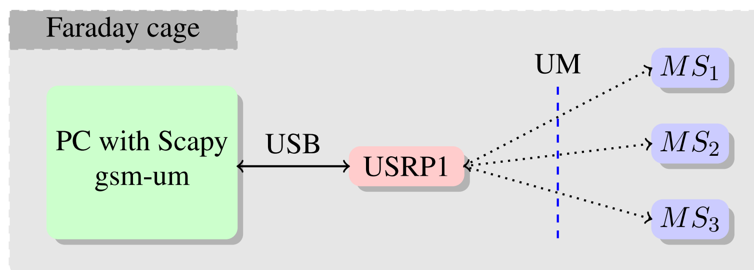

Figure 4.1: Test environment setup

of users over severel meters, we do not necessary need two RFX boards. The advantage of two RFX daughterboards is that we can reduce the crosstalk of the board and hence have better results on longer distances. For our tests the Mobile Stations were placed only several centimeters from the base-station and thus cross-talk was not a problem. USRP1's come with an internal clock of 64Mhz. This is sub-optimal for GSM. Many phones are not able to connect to a base-station having such a clock since the clockdrift is way too high. The clocktamer is a clock that has been especially conceived for interoperability with an USRP. It is a configurable clock generator, that can be used as a GSM reference clock. Furthermore, the clocktamer has the possibility to connect the clock to the *Global Positioning System (GPS)*, this allows higher stability and even better results.

4.1.2 Software

The following section describes the software part of our testing environment.

OpenBTS

Given our hardware setup we had to use a fork of the OpenBTS source code which adds support for single daughter boards. The code can be fetched form the GNU Radio webpage [dlbb].

GNU Radio

The GNU Radio version we used in this paper was check-out of the distributed revision control system git repository of the project, located at [gita]. Since we are using an USRP1 we do not need the UHD [uhd] driver from Ettus, but GNU Radio is managing the communication between our host and the device.

Scapy

The Scapy version we used to develop our Scapy gsm air interface addon is taken from the Scapy mercurial repository located at [gitb]. We did the development on revision: *113ef25f9583*. The code is known to run with Python 2.7.1.

4.2 Example: Normal Operations

Before we start to show that our implementation is an enrichment for security research, we demonstrate basic operations like recreating packets of whireshark dumps and everyday operations like setting up a phone-call using our USRP1 and a Mobile Station.

4.2.1 Recreate packets taken of a wireshark capture

As a first test we build a serie of packets taken out of a publicly available capture of a GSM conversation. The capture we used can be found in [sha]. Listing 4.1 is frame 45 of that capture.

```
▽ GSM A-I/F DTAP - TMSI Reallocation Command
  ▽ Protocol Discriminator: Mobility Management messages
      0000 .... = Skip Indicator: 0
      .... 0101 = Protocol discriminator: Mobility Management messages (5)
    00.. .... = Sequence number: 0
    ..01 1010 = DTAP Mobility Management Message Type: TMSI Reallocation Command (0x1a)
  ▽ Location Area Identification (LAI)
    ▷ Location Area Identification (LAI) - 246/03/4
  ▽ Mobile Identity - TMSI/P-TMSI (0x2e48e5e0)
      Length: 5
      1111 .... = Unused
      .... 0... = Odd/even indication: Even number of identity digits
      .... .100 = Mobile Identity Type: TMSI/P-TMSI (4)
      TMSI/P-TMSI: 0x2e48e5e0

0000  03 62 35 05 1a 42 f6 30  00 04 05 f4 2e 48 e5 e0   .b5..B.0 .....H..
0010  2b 2b 2b 2b 2b 2b 2b                               +++++++
```

Figure 4.2: Wireshark analysis of an TMSI Reallocation Command Packet

Listing 4.1: Recreate TMSI Reallocation Command

```
1 >>> a=tmsiReallocationCommand()
2 >>> a.oddEven=1; a.typeOfId=4; a.idDigit2_1=2; a.idDigit2=0xe;
3 >>> a.idDigit3_1=4; a.idDigit3=8; a.idDigit4_1=0xe; a.idDigit4=5
4 >>> a.idDigit5_1=0xe; a.idDigit5=0; a.mccDigit2=0x4; a.mccDigit1=2;
5 >>> a.mccDigit3=6; a.mncDigit2=0x3; a.mncDigit3=0xf; a.mncDigit1=0;
6 >>> a.idDigit1=0xf; a.oddEven=0; a.lac2=0x4; a.lac1=0x0
7 >>> hexdump(a)
8 0000   05 1A 42 F6 30 00 04 05  F4 2E 48 E5 E0          ..B.0.....H..
```

For validation we compared the hexadecimal value of the packet we generated with the hexdump of the pcap.

The hexdumps are the same. So we successfully recreate a packet of a real intercepted GSM conversation. More packets of this dump can be found in Appendix 6.0.1. Recreating packets of pcap files is one thing, in the next section we will demonstrate that the implementation works using real hardware, too.

4.2.2 Call

Setting up a phone-call can be initiated from two directions. Imagine a situation where a $user_1$ wants to contact $user_2$. Note that both users are mobile phone users.
First, $user_1$'s Mobile Station contacts the base-station, so the GSM device is the initiator of the protocol run. On the other side, $user_2$ is contacted by the base-station. Figure 4.3 illustrates the described scenario. In our case the network is replaced through our USRP and OpenBTS.

What we present in this section is a protocol run initiated by the base station to the Mobile Station. In the example above, this would be how the $user_2$ is contacted. In order to establish a connection to the mobile subscriber a specified series of messages have to be send. A detailed description of each message can be found in [040b]. The author of [Wal01] is also providing information on the run of such a protocol on page 202. Picture 4.4 illustrates how the protocol

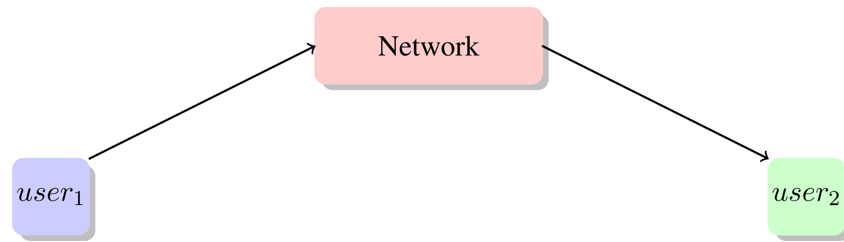

Figure 4.3: Graphical representation of a call between two mobile phone users

has to behave in our case.

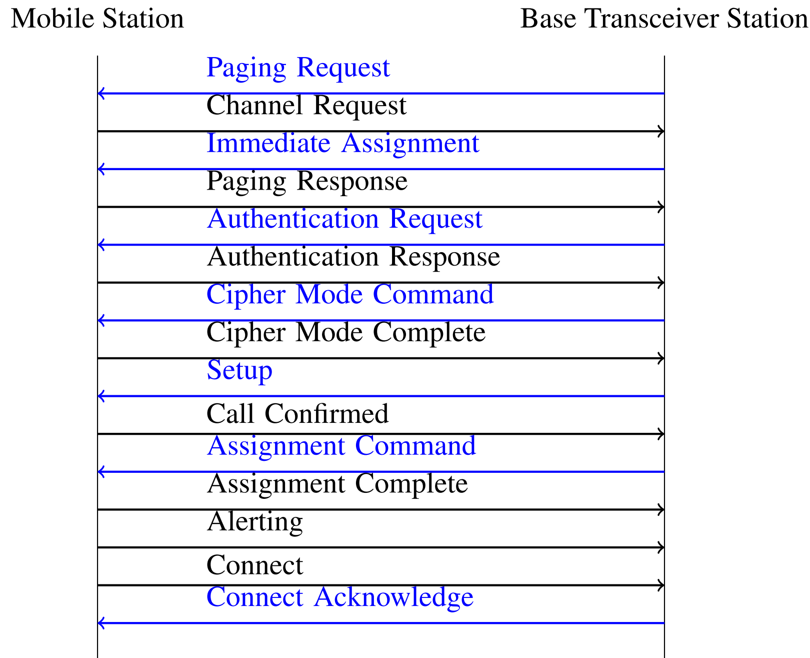

Figure 4.4: Call protocol-run initiated by a Base Transceiver Station

Once the protocol is successfully finished, a connection is established. Using Scapy gsm-um we can reproduce all the messages sent from the BTS to the mobile subscriber (identifiable by the blue color in Figure 4.4), and this way setup a connection. The needed packets are the following:

- pagingRequestType1() (Radio Resource)
- immediateAssignment() (Radio Resource)
- authenticationRequest() (Mobility Management)
- cipheringModeCommand() (Radio Resource)
- setupMobileOriginated() (Call Control)
- assignmentCommand() (Radio Resource)
- connectAcknowledge() (Call Control)

Note that firing up Scapy gsm-um and building and sending these packets will probably not work, since some packet values have to be adapted to the phone you want to communicate with. In our case we used the *testcall* function of OpenBTS. This allows us to test if we can setup a connection to an associated mobile subscriber. Since the *testcall* handles the Radio Resource (RR) and Mobility Management (MM) messages most of the communication protocol above is

handled. We only need to send a *setupMobileOriginated()* message to let a phone ring. This can be performed using Scapy gsm-um as described in Listing 4.2.

Listing 4.2: Setup of a phone call

```
>>> e=setupMobileOriginated()
>>> e
<TpPd  pd=3 |<MessageType  mesType=0x5 |>>
>>> e.show()
###[ Skip Indicator And Transaction Identifier and Protocol
Discriminator ]###
  ti= 0
  pd= 3
###[ Message Type ]###
     mesType= 0x5
>>> hexdump(e)
0000   03 05                                              ..
>>> sendum(e)
```

Listing 4.3 shows what is seen in the OpenBTS logs once we send the message. It contains the send message (0x0305) as well as the answer of the phone (0x830804046002008115 0101).

Listing 4.3: OpenBTS logfile while setting up a phone call

```
1312114461.0385 INFO 3044019056 CallControl.cpp:1174:TestCall:
entering test loop
1312114466.6880 INFO 3044019056 CallControl.cpp:1179:TestCall:
 got 2 bytes on UDP
1312114466.6882 INFO 3044019056 CallControl.cpp:1182:TestCall:
sending  primitive=DATA raw=(0305)
1312114466.8356 INFO 3044019056 CallControl.cpp:1195:TestCall:
received primitive=DATA raw=(8308040460020081150101)
```

The second message needed to setup the call is build in Listing 4.4.

Listing 4.4: Setup a phone call

```
>>> f=connectAcknowledge()
>>> f.show()
###[ Skip Indicator And Transaction Identifier and Protocol
Discriminator ]###
  ti= 0
  pd= 3
###[ Message Type ]###
     mesType= 0xf
>>> hexdump(f)
0000   03 0F                                              ..
```

The logs show that the data sent over the USRP are exactly the one we created in Scapy. The hexdump in Scapy matches the one in the logs.

The answer we get from the Mobile Station is *0x83 08 04 04 60 02 00 81 15 01 01* as shown in Listing 4.3. We can send that packet directly to wireshark over *GSMTAP*.

An analyse of this packet can be found on Figure 4.5. Listing 4.5 presents the logs of OpenBTS after we send the second message. The response of the Mobile Station can be found in Figure 4.6.

Listing 4.5: OpenBTS logfile while setting up a phone call

```
1312233895.8440 INFO 3041938288 CallControl.cpp:1179:TestCall:
got 2 bytes on UDP
1312233895.8441 INFO 3041938288 CallControl.cpp:1182:TestCall:
sending primitive=DATA raw=(030f)
1312233895.8443 INFO 3041938288 CallControl.cpp:1195:TestCall:
received primitive=DATA raw=(8341)
```

4.3 Case Study: Attacks

The following part of the paper describes some attacks that are demonstrated using Sacpy gsm-um. Some attacks are well-known, classical attacks on the network, others are novel and there are no similar attacks known to the author.

4.3.1 Classical Attack

In this paper we refer to the term classical attack for attacks that are well-known, understood and documented by the security community. Implementing some of those attacks is the main idea of this section.

IMSI Detach Indication attack

A first, classical attack will be introduced in this section. This attack is possible due to a conception error in the protocol. In order to perform the attack we need a single message of the type: *IMSI DETACH INDICATION*. The specification describes this message in the following way:
“This message is sent by the Mobile Station to the network to set a deactivation indication in the network.” section 9.2.12 of [040b].
The idea of this message is that the Mobile Station sends it to the Base Transceiver Station when it is turned off. The problem of this message is that it does not require any authentication. The IMSI DETACH message is a service request message. The mobile station is requesting a service, in this case it wants to be detached from the network. The network has to take the decision whether it grants the service or not [040b] page 42.
The really interesting part for us is the following sentence in the specifications, page 42: *The service request message must contain the identity of the Mobile Station and may include further information which can be send* ***without encryption***. This means we can send this message from

```
1961 4.197670 127.0.0.1 127.0.0.1 LAPDm I P, N(R)=2, N(S)=1(DTAP) (CC) Ca
▼ GSM A-I/F DTAP - Call Confirmed
  ▼ Protocol Discriminator: Call Control; call related SS messages
      1... .... = TI flag: allocated by receiver
      .000 .... = TIO: 0
      .... 0011 = Protocol discriminator: Call Control; call related SS message
    00.. .... = Sequence number: 0
    ..00 1000 = DTAP Call Control Message Type: Call Confirmed (0x08)
  ▼ Bearer Capability 1 - (MS supports at least full rate speech version 1 and
      Element ID: 4
      Length: 4
    ▼ Octet 3
        0... .... = Extension: Extended
        .11. .... = Radio channel requirement: MS supports at least full rate s
        ...0 .... = Coding standard: GSM standardized coding
        .... 0... = Transfer mode: circuit
        .... .000 = Information transfer capability: Speech (0x00)
    ▼ Octets 3a - Speech Versions
        0... .... = Extension: Extended
        .0.. .... = Coding: octet used for extension of information transfer ca
        ..00 .... = Spare bit(s): 0
        .... 0010 = Speech version indication: GSM full rate speech version 2(G
        0... .... = Extension: Extended
        .0.. .... = Coding: octet used for extension of information transfer ca
        ..00 .... = Spare bit(s): 0
        .... 0000 = Speech version indication: GSM full rate speech version 1(G
        1... .... = Extension: No Extension
        .0.. .... = Coding: octet used for extension of information transfer ca
        ..00 .... = Spare bit(s): 0
        .... 0001 = Speech version indication: GSM half rate speech version 1(G
  ▼ Call Control Capabilities
      Element ID: 21
      Length: 1
      0000 .... = Maximum number of supported bearers: 1
      .... 0... = MCAT: The mobile station does not support Multimedia CAT
      .... .0.. = ENICM: The mobile station does not support the Enhanced Netw
      .... ..0. = PCP: the mobile station does not support the Prolonged Clear
      .... ...1 = DTMF: the mobile station supports DTMF as specified in subcl

0030  00 00 00 00 00 00 09 00  00 00 01 52 2d 83 08 04   ........ ...R-...
0040  04 60 02 00 81 15 01 01  2b 2b 2b 2b 2b 2b 2b 2b   .`...... ++++++++
0050  2b                                                 +
```

Figure 4.5: Wireshark analysis of a response packet (Call Confirmed)

```
▼ GSM A-I/F DTAP - Alerting
  ▼ Protocol Discriminator: Call Control; call related SS messages
      1... .... = TI flag: allocated by receiver
      .000 .... = TIO: 0
      .... 0011 = Protocol discriminator: Call Control; call related SS messages (3)
    01.. .... = Sequence number: 1
    ..00 0001 = DTAP Call Control Message Type: Alerting (0x01)
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 43 00 00 40 00 40 11  3c a8 7f 00 00 01 7f 00   .C..@.@. <.......
0020  00 01 c6 b1 12 79 00 2f  fe 42 02 04 01 02 40 33   .....y./ .B....@3
0030  00 00 00 00 00 00 09 00  00 00 01 44 09 83 41 2b   ........ ...D..A+
0040  2b 2b 2b 2b 2b 2b 2b 2b  2b 2b 2b 2b 2b 2b 2b 2b   ++++++++ ++++++++
0050  2b                                                 +
```

Figure 4.6: Wireshark analysis of a response packet (Alerting)

Information element	Presence	Length
Mobility management protocol discriminator	M	1/2
Skip indicator	M	1/2
IMSI detach indication message type	M	1
Mobile station classmark	M	1
Mobile identity	M	2-9

Table 4.1: IE, presence and length fields of IMSI DETACH INDICATION message

any possible Mobile Station, the only thing we need to find out is the identity of the Mobile Station. Digging in the packet (see next paragraph) we find out that we need, either the IMSI, IMEI, IMEISV or TMSI/P-TMSI [040b] table 10.5.4.

IMSI DETACH INDICATION, is formed of the packets of Table 4.1. Using the gsm-um Scapy addon it is really easy to program a packet like the one presented in Figure 4.7.

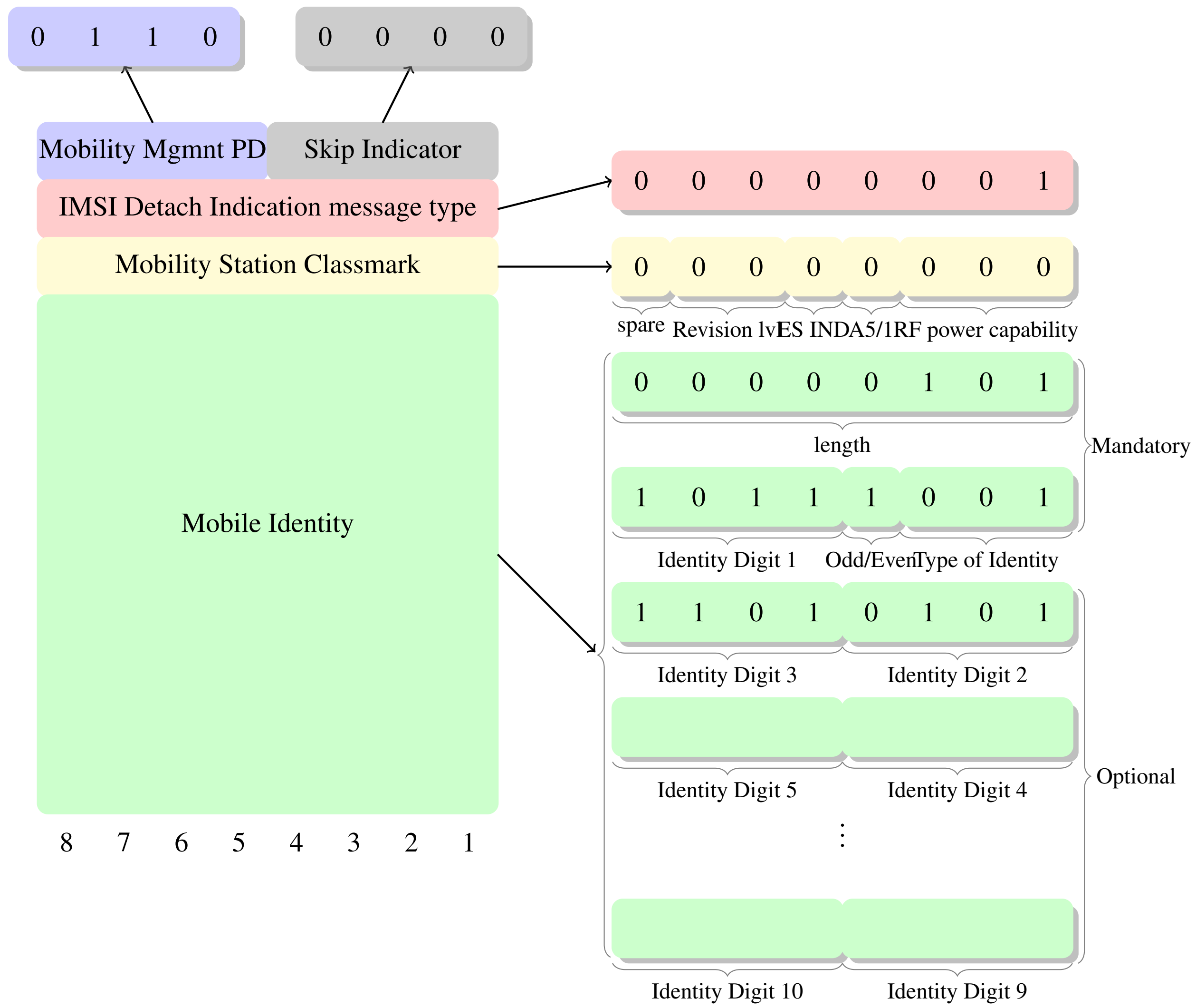

Figure 4.7: Whole IMSI DETACH INDICATION packet, graphical representation

Listing 4.6 is an example script for a fictitious imsi: 270772000125755.

Listing 4.6: IMSI DETACH Attack

```
Welcome to Scapy (2.2.0)
Scapy GSM-UM (Air) Addon
>>> a=imsiDetachIndication()
... a.typeOfId=1; a.odd=1; a.idDigit1=0xF;
... a.idDigit2_1=2; a.idDigit2=7;a.idDigit3_1=0;
... a.idDigit3=7; a.idDigit4_1=7; a.idDigit4=2;
... a.idDigit5_1=0;a.idDigit5=0; a.idDigit6_1=0;
... a.idDigit6=1;a.idDigit7_1=2;a.idDigit7=7;
... a.idDigit8_1=7; a.idDigit8=5; a.idDigit9_1=1;a.idDigit9=4;

hexdump(a)
0000   05 01 00 08 F0 27 07 72  00 01 27 75 14    .....'.r..'u.
```

All we have to do is create an instance of the imsiDetachIndication packet, and set some values that are non default:

- set Mobile Identity type to 0x1 we want to use the International Mobile Subscriber Identity (IMSI) [040b] table 10.5.4,
- set idDigit1 to 0xF (1111), as specified in [040b] page 378. This has to be set if we want to use IMSI/P-TMSI,
- set odd number of identity digits to 1,
- set the IMSI in Binary coded decimal format.

Authentication reject attack

The attack we present here disconnects a mobile user from the GSM network. A message of the type: *Authentication reject* is used to disconnect a user from the network. The user is unable to reconnect to this network, but he is also unable to connect to any other network, until he restarts his mobile station. The specification [040b] describes the message in section 9.2.1 the following way: *This message is send by the network to the mobile station to indicate that authentication has failed (and that the receiving Mobile Station abort all activities)*. Using Scapy gsm-um the message is constructed as described in Listing 4.7.

Listing 4.7: Authentication reject Attack

```
>>> a=authenticationReject()
>>> a.show()
###[ Skip Indicator And Transaction Identifier and Protocol
Discriminator ]###
  ti= 0
  pd= 5
```

```
 7 ###[ Message Type ]###
 8       mesType= 0x11
 9 >>> hexdump(a)
10 0000    05 11                                                 ..
```

After sending this message to the Mobile Stations we the following text appears on the screen of the phones: *SIM card registration failed*. As described before the user is locked out of every network until he restarts his GSM capable device.

4.3.2 Novel Attack

In this thesis we refer to the term novel attack for attacks that haven't been presented on the GSM layer 3 protocol so far. We chose this way to underline the importance of this tool for the security community.
As an attack we decided to have a look at the state-machine of the mobile subscriber side. The state machine and the states are described in Section 5 of the specifications [040b].
For these part we concentrated on a very small subset of the state-machine of the Call Control. This does not mean that other state-machines of the GSM Layer 3 protocol couldn't be analyzed. The attacks we performed were all starting from our BTS, and were send out to the Mobile Stations. Nevertheless this is not a limitation. The state-machines of the Base Transceiver Stations could of course be analyzed too, given a method to send layer 3 messages to this BTS.

The idea

The idea of this attack is to use legit messages that we create using Scapy gsm-um and send them in an unforeseen order. We hope, through this to get the state-machine in an undefined state, and through this to perform operations that we shouldn't be able to perform. Figure 4.8 illustrates a subset of the state machine of the Mobile Subscriber side which should show how a call is setup and which different possibilities there are.

Example

As an example we take the Call Clearance protocol run. This is a very small message exchange and ideal for our tests. Picture 4.9 represents the order in which the messages have to be send. An idea that came to our mind was to try to make the Mobile Station believe that the network really hang-up, but in reality the BTS was still connected. This would allow eavesdropping, which could be a very high security and confidentiality breach if exploited by criminals.
Two different attacks are presented in Listing 4.8 and 4.9, but given the complexity of the protocol and the huge amount of possible messages these two examples are in no case a limitation.

Listing 4.8: Novel Attack Example 1

```
1 >>> a = setupMobileOriginated()
2 >>> b = connectAcknowledge()
3 >>> c = disconnectNetToMs()
4 >>> a = setupMobileOriginated()
```

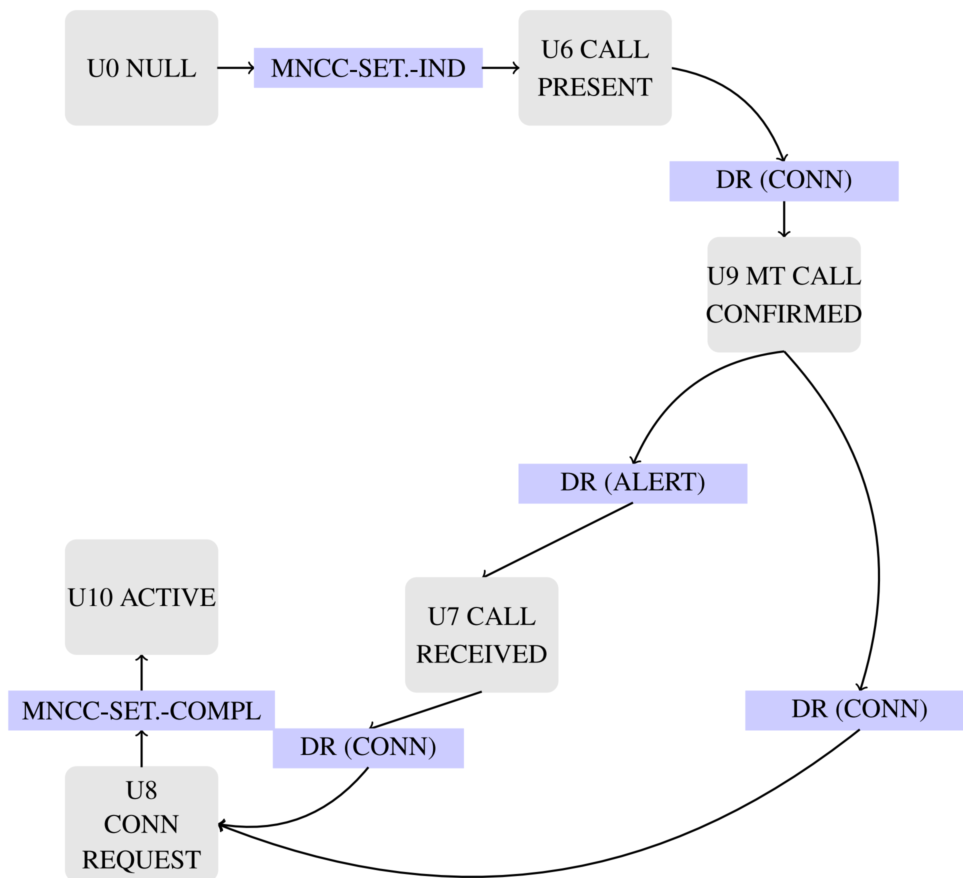

Figure 4.8: Call Control protocol/MS side Finite State Machine

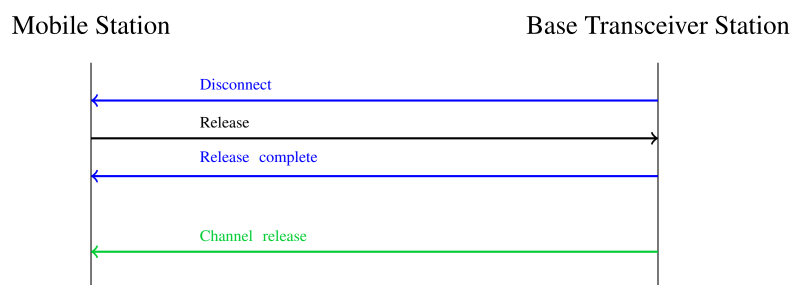

Figure 4.9: Call Clearance

```
>>> sendum(a); sendum(b); sendum(c); sendum(a)
```

In Lisiting 4.8, we first setup a phone call, as described earlier in 4.2.2 and then we send a disconnect message, followed by a *setupMobileOriginated()* message instead of the *Release complete* message that the Mobile Station is expecting.

Listing 4.9: Novel Attack Example 2

```
>>> a = setupMobileOriginated()
>>> b = connectAcknowledge()
>>> c = disconnectNetToMs()
>>> b = connectAcknowledge()
>>> sendum(a); sendum(b); sendum(c); sendum(b)
```

In Listing 4.9 we setup a phone call and then begin the call clearance message exchange, but again we do not send the *Release complete* but a *connectAcknowledge()* message instead.
These examples were used to show that our implementation of Scapy gsm-um allows this kind of analyze. Unfortunately these examples weren't fruitful with our phones. This does not mean that this examples or similar ones are not working on other phones.

To conclude we could say that our implementation of Scapy gsm-um is working as expected and it allows to replay different scenarios of everyday operations using the GSM protocol and that our tool provides a great way to recreate classical attacks in a very fast way and it even allows to attack parts of the GSM state machines that have never been tested by independent researchers. Our freely available tool will hopefully be accepted by the researchers community and used to provide a more secure GSM protocol. We are sure that researchers could even find attacks that we didn't think of, this is due to the high flexibility of the program presented here.

5 Summary

This chapter sums up the thesis, by firstly explaining the current limitations of the work and then presenting the future enhancement we want to perform. Finally we draw some conclusion about this thesis and what we hope we will achieve by releasing our tool.

5.1 Limitation

The limitation of the work presented in this thesis are quite few. The biggest limitation is the missing dissector functionality of Scapy gsm-um. Gsm-um is, in the current time, unable to dissect incoming messages. This feature would be nice to have, but not critical for our work. All the tests we could imagine were successfully performed without dissector. The dissector was not implemented due to time constraints. Nevertheless support for the dissector is planed for a future release and certain beta-testers are currently working on the dissect function, so we are confident that this feature will be supported soon.
Another limitation is the scope of the implementation. Scapy gsm-um implements the specification 04.08 [040b], which is basically the whole layer 3. It would be nice to have support for SMS services too. These messages are specified in 03.04 [030]. The author started to implement this specification but it will only be released in a future version.

5.2 Future Work

The limitations presented in the last section are part of the future work. The author plans to continue working on Scapy gsm-um and to add the missing dissector feature. Another important feature, that is not directly linked to Scapy gsm-um, would be to add support for custom layer 3 messages in projects like osmocomBB. This would allow researchers to play with the security of the Base Transceiver Station, using similar attacks as those presented in this thesis.
Furthermore, more advanced attacks on the state-machine of the mobile stations, using more diversified Mobile Stations are planed. Since no independent researchers tested the security of the state machines it would be a step in the right direction to get a more secure GSM network.

5.3 Conclusion

This thesis described a new addon created for Scapy which implements the specifications of the layer 3 of the GSM protocol. This is the protocol used between the mobile subscriber and the Base Transceiver Station. The addon allows users to build and send correct or forged layer 3 messages created on a command line. Since the whole layer 3 protocol was implemented messages can be send in two directions, from the Base Transceiver Station to the Mobile Station

or vis-versa. Specially the second method seems to be a feature that is really wished by the security community as showed numerous discussions with experts in that field.
Given the fact that the creation of messages is now a very easy task due to our program and not a time consuming effort anymore, we hope his tool gets accepted by the security community and used to make the GSM network more secure. In this thesis examples were given to re-create classical attacks as well as novel attacks on the Mobile Stations. The same attacks could be performed on the Base Transceiver Stations and it is now up to the security community to take over the lead and spot problems. The author believes that his open-source tool give every researcher the possibility to play with the security of the layer 3 of GSM even if the Scapy gsm-um user does not know every detail of the protocol.
Scapy gsm-um has been pushed to the contrib repository of Scapy and we hope it makes it to the mainline of Scapy soon. This means that at the current time every user that pulls Scapy from the main repository of the project has a copy of Scapy gsm-um. Given this wide distribution we hope we can bring more people that were not necessary interested in GSM onto the topic of gsm- security through the easy starting using our addon.

6 Annexes

6.0.1 Pcap captures

The packets presented in this section can be found in: [sha]

Listing 6.1: Frame 28

```
>>> a=immediateAssignment()
>>> a.channelTyp=12; a.tn=2; tsc=6; a.h=1; a.hsn=41; a.tsc=6; a.t1=4
>>> a.t2=0xf; a.t3Lo=1; a.ra=0x49; a.timingVal=1; a.maC64=0x;
>>> a.l2pLength=12
>>> hexdump(a)
0000   31 06 3F 00 62 D0 29 49  20 2F 01 01 03        1.?.b.)I /...
```

Listing 6.2: Frame 34

```
>>> a=cmServiceRequest()
>>> a.keySeq=3; a.serviceType=1
>>> a.revisionLvl=1; a.esInd=1; a.rfPowerCap=3; a.ssScreenInd=1;
>>> a.smCaPabi=1; a.fc=1; a.cm3=1; a.a52=1;
>>> a.idDigit2_1=2; a.idDigit2=0xe; a.idDigit3_1=4; a.idDigit3=8;
>>> a.idDigit4_1=4; a.idDigit4=1; a.idDigit5_1=1; a.idDigit5=5
>>> a.idDigit1=0xf
>>> hexdump(a)
0000   05 24 31 03 33 19 81 05  F4 2E 48 41 15        .$..3.....HA.
```

Listing 6.3: Frame 35

```
>>> a=classmarkChange(MobileStationClassmark3_presence=1)
>>> a.revisionLvl=1; a.esInd=1; a.rfPowerCap=3; a.ssScreenInd=1
>>> a.smCaPabi=1; a.fc=1; a.cm3=1; a.a52=1; a.byte2=0x60; a.byte3=0x14
>>> hexdump(a)
0000   06 16 03 33 19 81 20 60  14 00 00 00 00 00 00 00  ...3..`.......
0010   00 00 00 00                                      ....
```

Listing 6.4: Frame 36

```
>>> a=cipheringModeCommand()
>>> a.sc=1; a.cr=1
>>> hexdump(a)
0000   06 35 00 11                                      .5..
```

Listing 6.5: Frame 39

```
>>> a=releaseCompleteMsToNet(Cause_presence=1)
>>> a.codingStd=3; a.ext2=1; a.causeValue=16
>>> a.mesType=0x6a # due to the sequence number we need to change this
>>> hexdump(a)
0000   03 6A 08 02 E0 90                                    .j....
```

Listing 6.6: Frame 40

```
>>> a=cipheringModeComplete(MobileId_presence=1
>>> a.idDigit1=3; a.typeOfId=3
>>> a.idDigit2_1=0; a.idDigit2=5; a.idDigit3_1=3; a.idDigit3=1
>>> a.idDigit4_1=2; a.idDigit4=7; a.idDigit5_1=6; a.idDigit5=0
>>> a.idDigit6_1=4;a.idDigit6=5;
>>> a.idDigit7_1=5;a.idDigit7=6; a.idDigit8_1=1;  a.idDigit8=4
>>> a.idDigit9_1=0xf; a.idDigit9=0
>>> hexdump(a)
0000   06 32 17 09 33 05 31 27  60 45 56 14 F0              .2..3.1'`EV..
```

Listing 6.7: Frame 45

```
a=tmsiReallocationCommand()
>>> a.oddEven=1
>>> a.typeOfId=4
>>> a.idDigit2_1=2; a.idDigit2=e; a.idDigit3_1=4; a.idDigit3=8 ;
>>> a.idDigit4_1=e ; a.idDigit4=5 ; a.idDigit5_1=e ; a.idDigit5=0
>>> a.mccDigit2=0x4; a.mccDigit1=2; a.mccDigit3=6; a.mncDigit1=0
>>> a.mncDigit3=0xf; a.mncDigit2=0x3; a.lac1=0x0; a.lac2=0x4
>>> a.idDigit1=0xf; a.oddEven=0
>>> hexdump(a)
0000   05 1A 42 F6 30 00 04 05  F4 2E 48 E5 E0              ..B.0.....H..
```

Listing 6.8: Frame 47

```
>>> a=callProceeding()
>>> a.ti=8
>>> hexdump(a)
0000   83 02                                                ..
```

Listing 6.9: Frame 48

```
>>> a=tmsiReallocationComplete()
>>> hexdump(a)
0000   05 1B                                                ..
```

Listing 6.10: Frame 51

```
>>> a=immediateAssignment()
>>> a.tsc=6; a.tn=5; a.channelTyp=1; a.codingStd=3;
>>> a.location=0xa; a.progressDesc=8; a.arfcnLow=0xff
>>> a.arfcnHigh=3; a.powerLvl=5
```

```
>>> hexdump(a)
0000   06 2E 0D C3 FF 05                                  ......
```

Listing 6.11: Frame 55

```
>>> a=assignmentComplete()
>>> hexdump(a)
0000   06 29 00                                           .).
```

Listing 6.12: Frame 60

```
>>> a=progress()
>>> a.ti=8
>>> a.codingStd=3; a.location=0xa; a.progressDesc=8
>>> hexdump(a)
0000   83 03 02 EA 88                                     .....
```

Listing 6.13: Frame 65

```
>>> a=alertingNetToMs(ProgressIndicator_presence=1)
>>> a.ti=8; a.codingStd=3; a.location=0xa; a.progressDesc=8
>>> hexdump(a)
0000   83 01 1E 02 EA 88                                  ......
```

Listing 6.14: Frame 68

```
>>> a=connectNetToMs()
>>> a.ti=8
>>> hexdump(a)
0000   83 07                                              ..
```

Listing 6.15: Frame 70

```
>>> a=connectAcknowledge()
>>> a.mesType=0x4f
>>> hexdump(a)
0000   03 4F                                              .O
```

Listing 6.16: Frame 88

```
>>> a=disconnectMsToNet()
>>> a.codingStd=3; a.ext2=1; a.causeValue=16
>>> hexdump(a)
0000   03 25 02 E0 90                                     .%...
```

Listing 6.17: Frame 90

```
>>> a=systemInformationType6()
>>> a.ciValue1=0x40; a.ciValue2=0x5c
>>> a.mccDigit2=4; a.mccDigit1=2; a.mncDigit3=0xf; a.mccDigit3=6;
```

```
>>> a.mncDigit2=3; a.mncDigit1=0; a.lac1=0; a.lac2=4
>>> a.l2pLength=11
>>> a.nccPerm=0xc
>>> a.dtx=1
>>> a.rLinkTout=4
>>> hexdump(a)
0000   2D 06 1e 40 5C 42 F6 30  00 04 14 0C           -.6@\B.0....
```

Listing 6.18: Frame 91

```
>>> a=releaseNetToMs()
>>> a.ti=8; a.codingStd=3; a.ext2=1; a.causeValue=16
>>> hexdump(a)
0000   83 2D 08 02 E0 90                              .-....
```

Listing 6.19: Frame 94

```
>>> a=measurementReport()
>>> a.baUsed=1; a.dtxUsed=1; a.rxLevFull=39;
>>> a.noNcellHi=1; a.rxlevC1=38;
>>> a.bcchC1=4; a.bsicC1Hi=2; a.rxlevC2=18;
>>> a.bsicC1Hi=1; a.bsicC3Lo=1; a.bsicC3Hi=3;
>>> a.bcchC5Hi=10; a.bsicC6=29; a.bsicC5=18; a.bcchC6Hi=2;
>>> a.rxlevC6Lo=18; a.bcchC6Lo=2; a.bcchC6Hi=2;
>>> a.rxlevC5Lo=3; a.rxlevC5Hi=1; a.bsicC4=25;
>>> a.bcchC4=0xa; a.bcchC2=3;
>>> a.bsicC2Lo=0; a.bcchC2=3; a.bsicC1Hi=1;
>>> a.bsicC3Lo=25; a.bsicC1Hi=1; a.bscicC2Hi=6; a.rxLevSub=39;
>>> a.noNcellLo=2; a.rxlevC4Lo=3;
>>> a.rxlevC3Lo=3; a.bcchC3=12; a.bcchC5Hi=3;
>>> a.bsicC1Hi=2; a.bsicC2Hi=1
>>> hexdump(a)
0000   06 15 E7 27 01 A6 22 12  0D 06 D8 CB 6A 65 33 24  ...'.."...je3$
0010   92 5D                                          .]
```

Listing 6.20: Frame 129

```
>>> a=systemInformationType3()
>>> a.mccDigit2=4; a.mccDigit1=2; a.mncDigit3=0xf; a.mccDigit3=6;
>>> a.mncDigit2=3; a.mncDigit1=0; a.lac1=0; a.lac2=04
>>> a.ciValue2=0x5c
>>> a.t3212=0xc8
>>> a.bsPaMfrms=7
>>> a.dtx=1; a.rLinkTout=4
>>> a.cellReselect=4; a.msTxPwrMax=5; a.neci=1;
>>> a.maxRetrans=1; a.txInteger=9; a.re=1
>>> a.byte1=0x80; a.byte5=0x1b
>>> hexdump(a)
0000   49 06 1B 40 5C 42 F6 30  00 04 48 07 C8 14 85 40  I..@\B.0..H..@
0010   65 00 00 80 00 00 00 1B                        e.......
```

Listing 6.21: Frame 158

```
>>> a=pagingRequestType1()
>>> a.idDigit1=0xf
>>> a.l2pLength=5
>>> hexdump(a)
0000   15 06 21 00 01 F0                          ..!...
```

Listing 6.22: Frame 168

```
>>> a=pagingRequestType1()
>>> a.idDigit1=0xf
>>> a.typeOfId=4
>>> a.idDigit2_1=0x2
>>> a.idDigit2=0xe
>>> a.idDigit3_1=4
>>> a.idDigit3=8
>>> a.idDigit4=0xd
>>> a.idDigit5=0xf
>>> a.l2pLength=9
>>> hexdump(a)
0000   25 06 21 00 05 F4 2E 48  0D 0F             %.!....H..
```

Listing 6.23: Frame 179

```
>>> a=systemInformationType4()
>>> a.maxRetrans=1; a.txInteger=0x9; a.cellBarrAccess=0x0; a.re=1
>>> a.cellReselect=0x4; a.msTxPwrMax=0x5; a.neci=1; a.rxlenAccMin=0
>>> a.lac1=0x0; a.lac2=0x4; a.mccDigit2=0x4; a.mccDigit1=0x2
>>> a.mncDigit3=0xf; a.mccDigit3=0x3; a.l2pLength=12
>>> hexdump(a)
0000   31 06 1C 42 F6 30 00 04  85 40 65 00 00    1..B.0...@e..
```

6.1 Universal Software Radio Peripheral Setup

6.1.1 General - Howto Ubuntu 11.04

Note: Lines starting with "%" are executed as unpriviledged user. Lines starting with "#" are comments.

Listing 6.24: All Ubuntu dependencies needed

```
% sudo apt-get -y install libxi-dev libfontconfig1-dev \
libxrender-dev libfftw3-dev sdcc-libraries \
libpulse-dev swig g++ automake autoconf libtool Python-dev \
libcppunit-dev libboost-all-dev libusb-dev fort77 sdcc \
libsdl1.2-dev Python-wxgtk2.8 git-core guile-1.8-dev \
libqt4-dev Python-numpy ccache Python-opengl libgsl0-dev \
Python-cheetah Python-lxml doxygen qt4-dev-tools \
```

```
libqwt5-qt4-dev libqwtplot3d-qt4-dev pyqt4-dev-tools \
Python-qwt5-qt4 python-cheetah libortp8 libortp-dev asterisk\
git cmake boost-build libusb-1.0-0-dev libusb-1.0-0 \
doxygen  libexosip2-4 libexosip2-dev libosip2-4 libosip2-dev
```

Listing 6.25: Install GNU Radio [gnu]

```
% git clone http://gnuradio.org/git/gnuradio.git
% cd gnuradio
% ./bootstrap
% ./configure
# important: gr-udh is a must: The following GNU Radio
# components have been successfully configured:
# gr-uhd
% make
% make check
% sudo make install > install.gnuradio.log.txt
% sudo addgroup usrp
% sudo usermod -G usrp -a <YOUR_USERNAME>
% echo 'ACTION=="add", BUS=="usb", SYSFS{idVendor}=="fffe",\
SYSFS{idProduct}=="0002", GROUP:="usrp", MODE:="0660"' > tmpfile
% sudo chown root.root tmpfile
% sudo mv tmpfile /etc/udev/rules.d/10-usrp.rules
% sudo udevadm control --reload-rules
```

Listing 6.26: OpenBTS [opec] & [opeb]

```
# Note: Our setup had only one Daughterboard. Therefore we
# need an OpenBTS fork.
% git clone git://github.com/ttsou/openbts-uhd.git
% cd openbts-udh/public-trunk
% ./bootstrap
% ./configure --with-usrp1
% make
% sudo make install
```

Acronyms

GSM Global System for Mobile Communication

BTS Base Transceiver Station

BSC Base Station Controller

USRP Universal Software Radio Peripheral

MS Mobile Station

ETSI European Telecommunications Standards Institute

GPRS General Packet Radio Service

EDGE Enhanced Data Rates for GSM Evolution

UMTS Universal Mobile Telecommunications System

3GPP 3rd Generation Partnership Project

CEPT European Conference of Postal and Telecommunications Administration

SMS Short Message Service

NSS Network Subsystem

BSS Base Station Subsystem

PLMN Public Land Mobile Network

PCM-30 Pulse-Code Modulation 30

SS.7 Signaling System No 7

MAP Mobile Application Part

SIP Session Initiation Protocol

GPS Global Positioning System

RFID Radio-frequency identification

RR Resource Management

MM Mobility Management

CM Connection Management

IEI Information Element Identifier

IE Information Element

TLV Type, Length, Value

BSC Base Station Controller

Bibliography

[030] *GSM 03.04 Specifications*.

[040a] *GSM 04.07 Specifications*.

[040b] *GSM 04.08 Specifications*.

[Bioa] Philippe Biondi. Scapy. http://www.secdev.org/projects/scapy/. [Online; accessed 09-June-2011].

[Biob] Philippe Biondi. Scapy. http://www.secdev.org/. [Online; accessed 23-May-2011].

[BSW99] Alex Biryukov, Adi Shamir, and David Wagner. Real Time Cryptanalysis of A5/1 on a PC. 1999. [Online; accessed 25-July-2011].

[bui] Scapy Build and Dissect. http://www.secdev.org/projects/scapy/doc/build_dissect.html. [Online; accessed 23-May-2011].

[Cla] Christophe Clavier. An Improved SCARE Cryptanalysis Against a Secret A3/A8 GSM Algorithm. http://www.springerlink.com/content/e73522084t478460/.

[clo] Clocktamer. http://code.google.com/p/clock-tamer. [Online; accessed 26-June-2011].

[dlba] OpenBTS download. http://sourceforge.net/projects/openbts/. [Online; accessed 26-June-2011].

[dlbb] OpenBTS fork download. http://gnuradio.org/redmine/projects/gnuradio/wiki/OpenBTSExpandedDboard. [Online; accessed 26-June-2011].

[ett] USRP. http://www.ettus.com/products/. [Online; accessed 09-June-2011].

[ext] Scapy Extend. http://www.secdev.org/projects/scapy/doc/extending.html. [Online; accessed 23-May-2011].

[gita] GNUradio Git Repository. http://gnuradio.org/git/gnuradio.git. [Online; accessed 26-June-2011].

[gitb] Scapy Mercurial Repository. http://hg.secdev.org/scapy. [Online; accessed 26-June-2011].

[gnu] GNU Radio - Ubuntu Install Howto. http://gnuradio.org/redmine/wiki/gnuradio/UbuntuInstall. [Online; accessed 24-May-2011].

[Gru10] The Grugq. Attacking GSM Base Station Systems and Mobile Phone Base Bands. 2010.

[ima] UDH images. http://www.ettus.com/downloads/uhd_releases/003_001_000/images-only/. [Online; accessed 24-May-2011].

[Kas06] Emilia Kasper. Complexity analysis of hardware-assisted attacks on A5/1. Master's thesis, University of Tartu, 2006.

[KYY09] Ryuichi Kitamura, Toshio Yoshii, and Toshiyuki Yamamoto. *The expanding sphere of travel behaviour research: selected papers from the* 11th *International Conference on Travel Behaviour Research*. Emerald Group Publishing Limited, 2009.

[LI] Bo Li and Eul-Gyu Im. Smartphone, promising battlefield for hackers.

[lin] Linus Torvalds Quote. http://groups.google.com/group/linux.dev.kernel/msg/76ae734d543e396d?pli=1. [Online; accessed 24-September-2011].

[MGS] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. SMS-of-Death: from analyzing to attacking mobile phones on a large scale. To Appear In the Proceedings of the 20th USENIX Security Symposium San Francisco, CA, USA 10-12 August 2011.

[mld] Baseband Devel Mailing List Archive. http://lists.osmocom.org/pipermail/baseband-devel/2011-June/002073.html. [Online; accessed 26-June-2011].

[new] New protocols. http://comments.gmane.org/gmane.comp.security.Scapy.general/4239. [Online; accessed 23-May-2011].

[niu] OpenBTS: Niue Pilot System, Spring 2010. http://openbts.sourceforge.net/NiuePilot/. [Online; accessed 11-July-2011].

[NP] Karsten Nohl and Chris Paget. GSM: SRSLY? http://events.ccc.de/congress/2009/Fahrplan/attachments/1519_26C3.Karsten.Nohl.GSM.pdf.

[OCo10] T.J. OConnor. Grow Your Own Forensic Tools: A Taxonomy of Python Libraries Helpful for Forensic Analysis . 2010.

[opea] OpenBSC. http://openbsc.osmocom.org/trac/. [Online; accessed 21-June-2011].

[opeb] Openbts. `http://sourceforge.net/projects/openbts/`. [Online; accessed 24-May-2011].

[opec] OpenBTS Building And Running. `http://gnuradio.org/redmine/wiki/1/OpenBTSBuildingAndRunning`. [Online; accessed 24-May-2011].

[osm] OsmocomBB. `http://bb.osmocom.org/trac/`. [Online; accessed 21-June-2011].

[pep] Style Guide for Python Code. `http://www.python.org/dev/peps/pep-0008/`. [Online; accessed 6-July-2011].

[Pes] Lauri Pesonen. GSM interception. `http://flur.net/archive/research/GSMinterception.pdf`. [Online; accessed 11-July-2011].

[pro] Ettus Products. `http://www.ettus.com/products`. [Online; accessed 24-May-2011].

[quo] Scapy Quote. `http://www.secdev.org/projects/Scapy/doc/introduction.html`. [Online; accessed 23-May-2011].

[rfx] RFX 900 specifications. `http://code.ettus.com/redmine/ettus/attachments/10/rfx900.pdf`. [Online; accessed 26-June-2011].

[sha] Wireshark GSM Capture. `http://wiki.wireshark.org/SampleCaptures?action=AttachFile&do=view&target=gsm_call_1525.xml`. [Online; accessed 30-August-2011].

[Spa] Dieter Spaar. Playing with the GSM RF Interface. `http://events.ccc.de/congress/2009/Fahrplan/attachments/1507_Playing_with_the_GSM_RF_Interface.pdf`. [Online; accessed 25-July-2011].

[Ste] Frank A. Stevenson. [A51] The call of Kraken. `http://lists.lists.reflextor.com/pipermail/a51/2010-July/000683.html`. [Online; accessed 09-June-2011].

[SWWJ] Hermant Sengar, Duminda Wijesekera, Haining Wang, and Sushil Jajodia. VoIP Intrusion Detection Through Interacting Protocol State Machines.

[tes] OpenBTS Mailing List Archive. `http://sourceforge.net/mailarchive/forum.php?thread_name=519145817-1307484753-cardhu_decombobulator_blackberry.rim.net-1267004548-%40b11.c5.bise6.blackberry&forum_name=openbts-discuss`. [Online; accessed 6-July-2011].

[udh] UDH-driver. `http://ettus-apps.sourcerepo.com/redmine/ettus/projects/uhd/wiki`. [Online; accessed 24-May-2011].

[uhd] Ettus UHD driver. http://code.ettus.com/redmine/ettus/projects/uhd/wiki. [Online; accessed 26-June-2011].

[use] USRP Uses. http://en.wikipedia.org/wiki/Universal_Software_Radio_Peripheral#Uses. [Online; accessed 24-May-2011].

[val] Pytest Plugin to Check Source Code Against PEP8 Requirements. http://pypi.python.org/pypi/pytest-pep8. [Online; accessed 6-July-2011].

[vdB10] Fabian van den Broek. Catching and Understanding GSM-Signals. Master's thesis, Radboud University Nijmegen, 2010.

[Wal01] Bernhard Walke. *Mobilfunknetze und ihre Protokolle 1 Grundlagen, GSM, UMTS und andere zellulare Mobilfunknetze*. Teubner, 2001.

[Wei] Ralf-Philipp Weinmann. All Your Baseband. Are Belong To Us. https://cryptolux.org/media/deepsec-aybbabtu.pdf. [Online; accessed 09-June-2011].

[Wel] Harald Welte. Fuzzing your GSM phone using OpenBSC and Scapy.