Connected Car Hacking

Telematic Control Unit (TCU) hacking

Quarkslab

Overview

Telematic Control Unit is the key element in a car's connectivity, offering:

- Backend connectivity
  - OEM monitoring
  - Mobile application interaction (door lock/unlock, positioning, charging status, HVAC...)
  - Firmware updates
- Emergency/Assistance call (xCall)
- Internet connectivity
  - On-board navigation
  - 3rd application connectivity

Role of IoT in connected vehicle technology

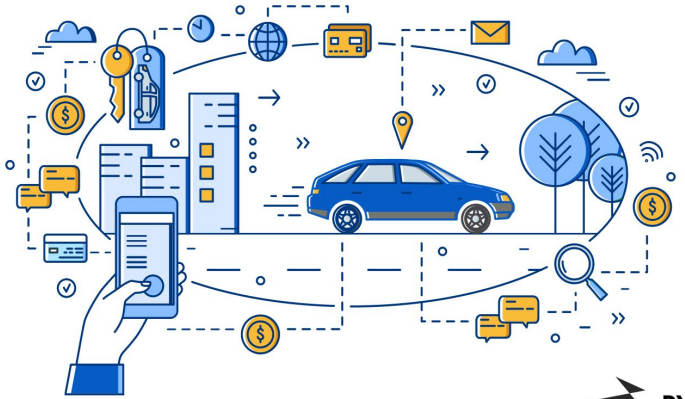


Illustration: [link]

BYTEBEAM

Hardware analysis

Key elements of a TCU are:

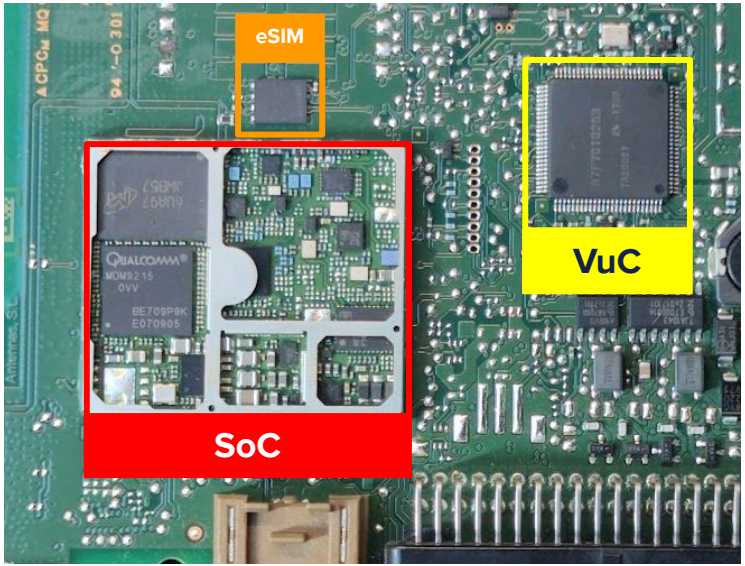
- **Dedicated vehicle microcontroller:** or **VuC** (Vehicle uC), which interacts with the on-board networks like CAN and handles diagnostic requests
- **3GPP embedded modem (SoC):** allows Internet/backend connectivity and usually runs an operating system handling car's connected features (remote unlocking, updates…)
- **eSIM**: embedded SIM card use to connect to the cellular network of programmed carrier

The **VUC** and the **SoC** communicate through an on-board communication, which could be **SPI**, **UART**, **USB3**…, look for **level shifter** as **VuC** and **SoC** often use different voltage levels (3.3V/1.8V)

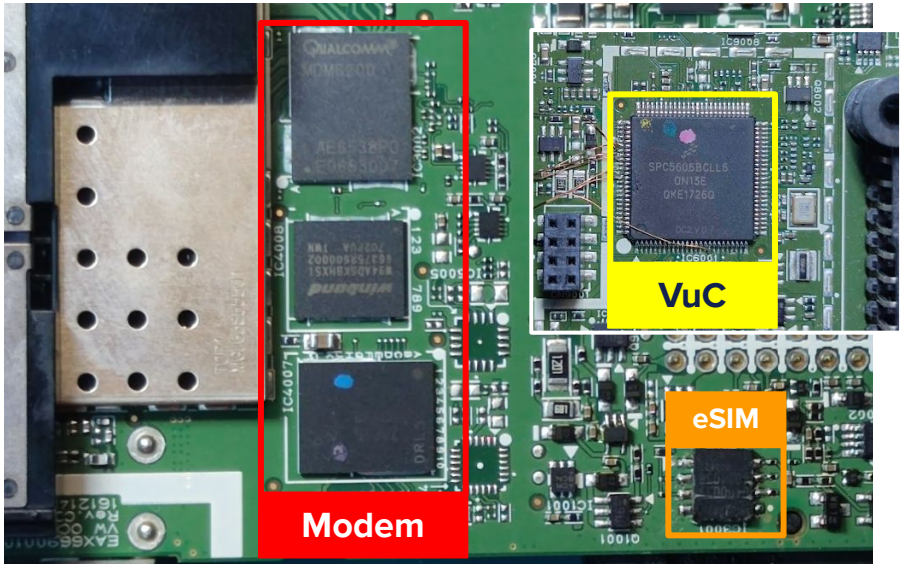
TCU offers **data connectivity** to other ECUs, using **Automotive Ethernet** or acting like an **USB to Ethernet** adapter

Hardware analysis

TCU #A



TCU #B



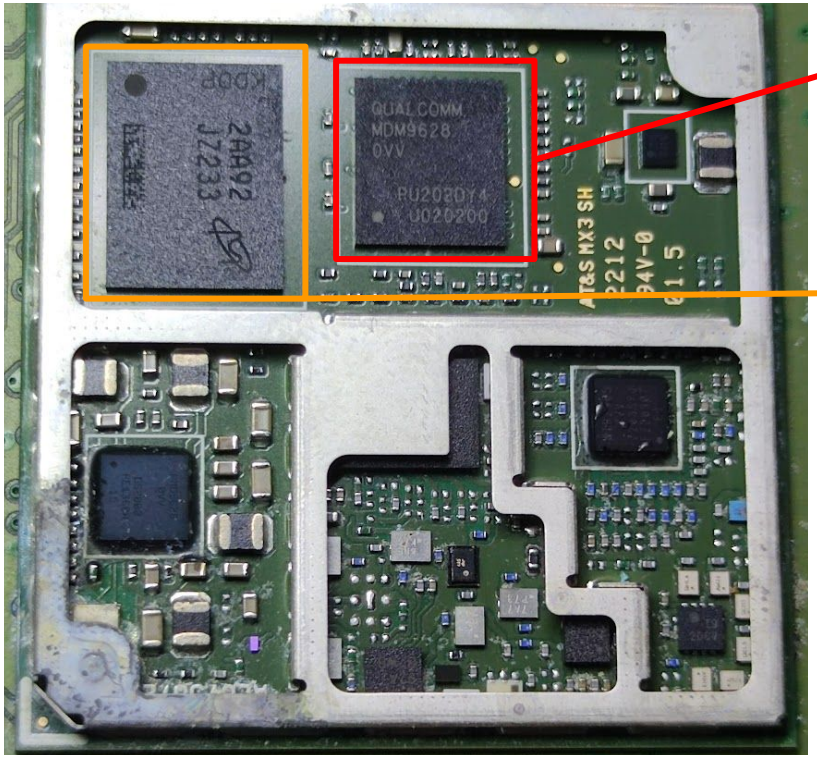
TCU Firmware extraction

Quarkslab

Embedded firmwares

- To fully assess a TCU, at least 2 firmwares need to be analysed:
  - **VuC** with its **embedded firmware** for in-vehicle communications
  - **SoC** running an **operating system** and a **baseband**
- However, **connected features** will be managed by the **SoC**,
- Those features are usually **binaries** or **Java applications** running on a framework provided by the **SoC manufacturer**
- **SoC file system** can be retrieved through **firmware updates** or by dumping its **Flash memory**
- Firmware updates are often **encrypted**
- Dumping SoC Flash memory requires **chip-off techniques**, being stored on **Parallel NAND Flash** or **MCP** (Multi Chip Package) chips (Flash + RAM)

SoC hardware analysis



Qualcomm MDM9628
Automotive grade communication chipset
ARM Cortex A7 1.2GHz
ARM Cortex M3 100MHz

Micron JZ233
MCP memory
4Gbit NAND Flash
2Gbit LDRR2 RAM

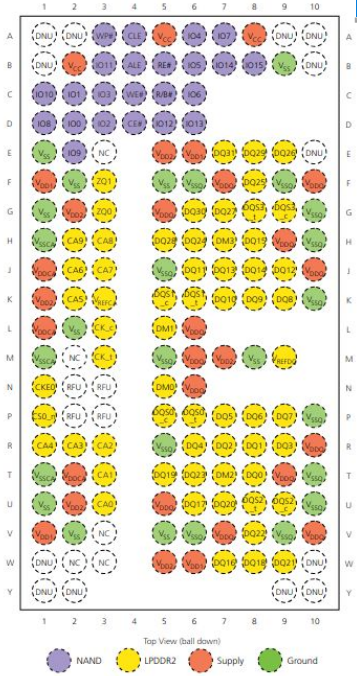
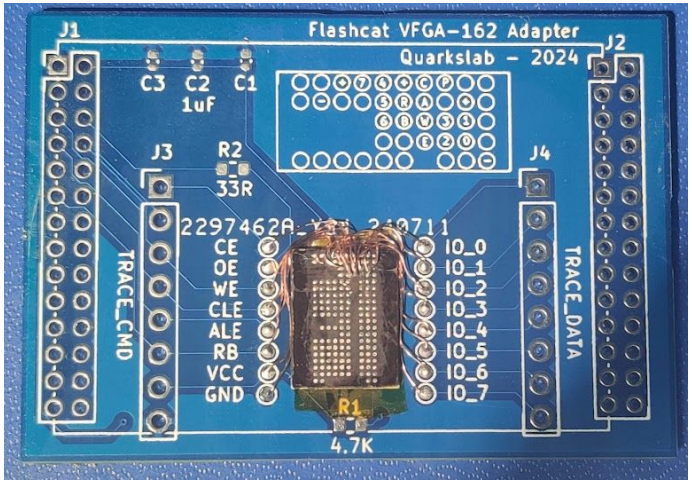


Illustration: [link]

Firmware storage - NAND Flash extraction

High storage capacity **NAND Flash** uses a **Parallel interface** to reach high bandwidth, requiring to remove the chip and using a memory programmer.

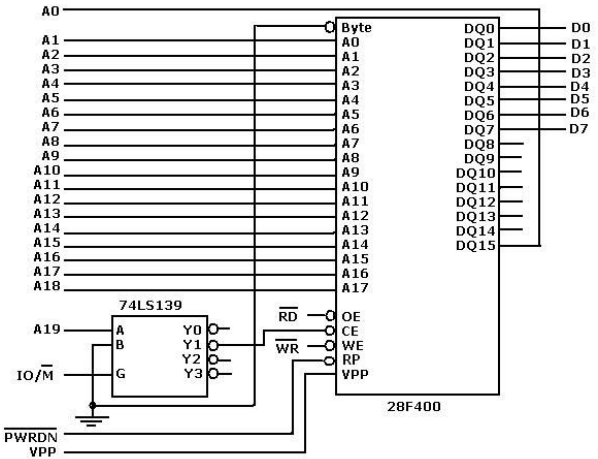
Common format are **TSOP48** and **BGA** like. Some integrated chip combines Flash memory and RAM requiring some microsoldering if no sockets are available off the shelf.



Firmware storage - NAND Flash - Parallel interface

For high-speed data transfer, increasing the clock speed of previous protocols has limits. Using **multiple data lines** allows reaching such rates and are commonly used by NAND memories and or SRAM chips, like the following example:

Command Latch Enable (**CLE**), Address Latch Enable (**ALE**), Write Enable (**WE**) and Read Enable (**RE**) and Chip Enable (**/CE**) control bus allow controlling the behaviour of the memory while data and address are read/written through the 8/16 corresponding bus [**DQ0-15**] and [**A0-A15**].



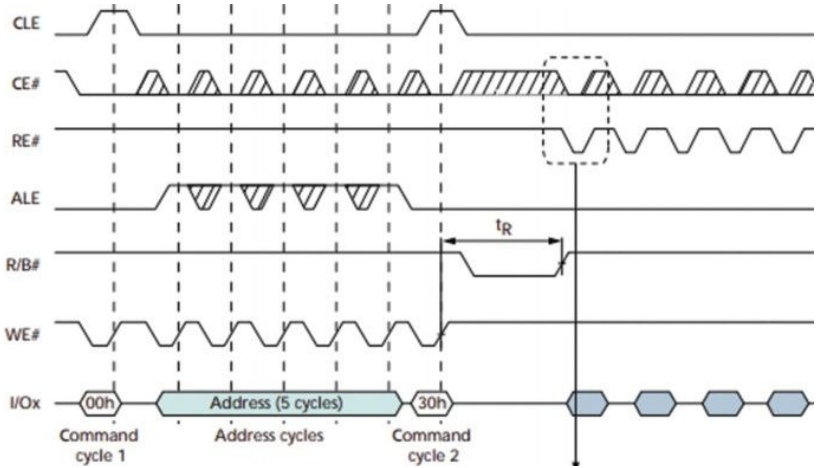


Illustration: [link]

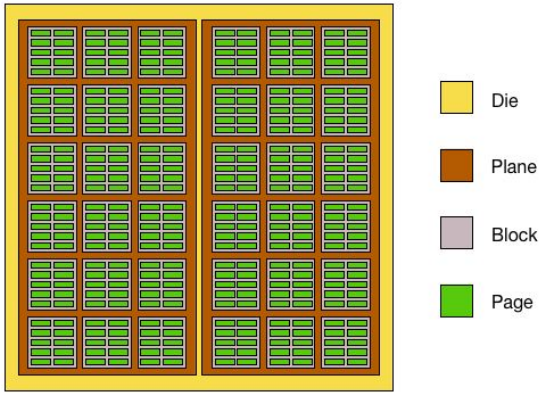
Firmware storage - NAND Flash layout

NAND Flash memory are commonly seen into **SoC**.

Data is written in **page** and erased in **block**. Once a bit in Flash is set to 0, it cannot be set back to 1 unless an erase operation is performed, i.e. writing a whole **block** to 1.

By its physical nature, **NAND Flash** memory is not 100% reliable, error when storing bits may occur.

To mitigate this issue, such memory have **spare area** to store **Error Correction Codes** (ECC) and more available memory space to provide defined storage capacity over the life cycle of the chip.



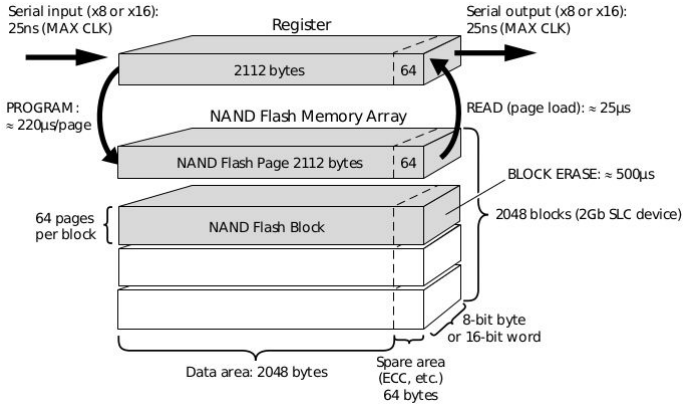


Illustration: [link] & [link]

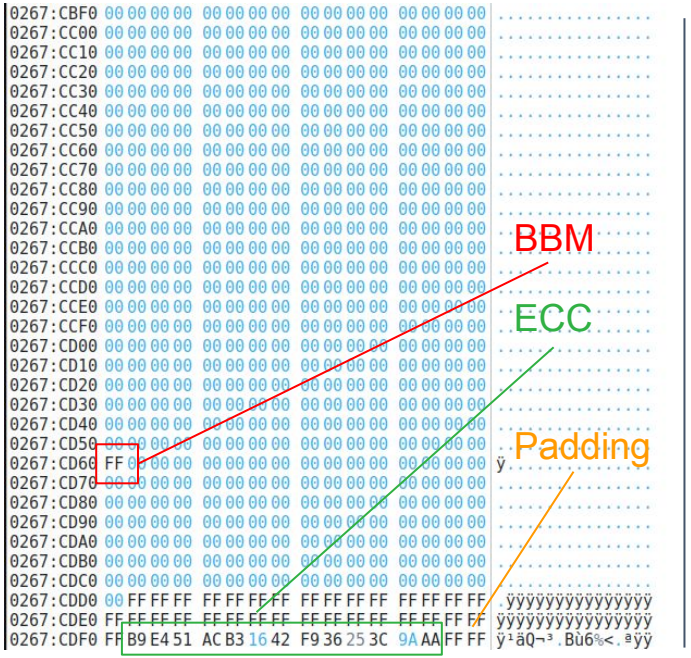
Firmware storage - NAND Flash wear levelling data

When dumping a **NAND Flash**, trying to analyse its content with **binwalk** or **unblob** will return error.

It is necessary to first identify **bad block marker** (**BBM**), **error correction code** (**ECC**) and potential **padding** that could be used.

- **Bad block marker** is a byte present at a specific interval, specifying if the block is valid or corrupted
- **ECC** will allow using algorithm like **BCH** or **Reed Solomon** to determine and correct invalid bits
- **Padding** could be used depending on the **ECC** size

Using the **ECC** to check and correct the content of the block and removing **BBM**, **ECC** and padding will allow getting the proper content of the **NAND** Flash



Firmware storage - BCH

- **Bose-Chaudhuri-Hocquenghem** (BCH) is a cyclic error-correcting code
- **ECC** It detects **error** in a set of symbols and allows **bits correction** depending on the codeword length (a 13-bytes BCH corrects up to 6-bit errors)

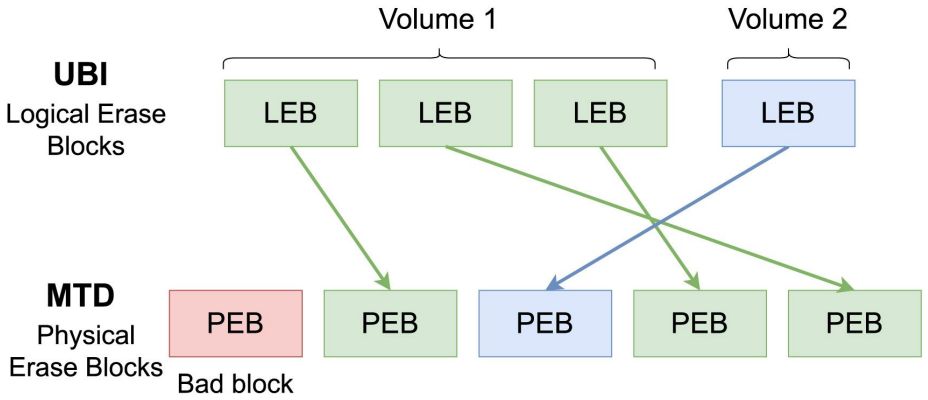
```
from bchlib import BCH

bch = BCH(8, m=13)  # BCH(BCC_bits, ECC_Polynomial)
data = b'...'       # data to analyse
bch_ecc = b'...'    # 13 bytes BCH code word

# Check if errors are present
error_qty = bch.decode(data, bch_ecc)

# Correct errors
data = bytearray(data)
bch.correct(data, ecc)
```

NAND specific file systems: UBIFS

Unsorted Block Images file system, known as **UBIFS** is a file system developed for unmanaged Flash memory to spread erase and write operation across all the blocks/pages.

It uses **Logical Erase Blocks** (**LEB**) that are dynamically mapped to **Physical Erase Blocks** (**PEB**). Data are stored into volumes, that consists of one or several **LEB**.

To do so, it adds at least an **Erase Counter header** at the beginning of the block and a **Volume IDentifier header**, recognizable with the magic bytes `UBI#` and `UBI!`



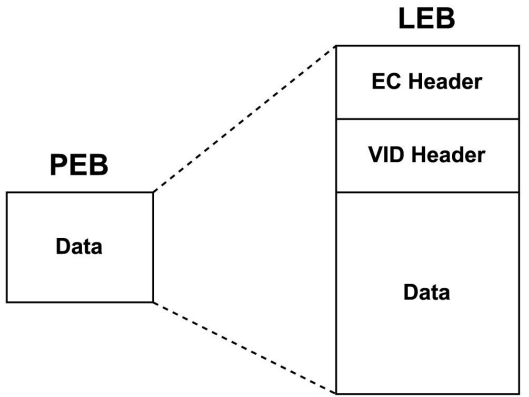


Illustration: [link]

NAND specific file systems: SquashFS

SquashFS is a **compressed** and **read-only** file system. It compresses files, inodes and directories and supports block sizes from 4KB up to 1MB. It is well suited for embedded devices.

Magic bytes *sqsh* (*0x73717368*) is used at the beginning of a **SquashFS** superblock.

Modification done to files (configuration, binaries) stored on a **SquashFS** using shell access will not be saved. To handle **persistent** data, such device will have other storage file system, like **UBIFS** on its Flash memory.

0x73717368	inode count		timestamp		block size
fragment entry count	compr-ession algo.	block size (log)	flags	id count	version
root directory inode			archive size (bytes)		
id table start offset			xattr id table start offset		
inode table start offset			directory table start offset		
fragment table start offset			export table start offset		

Illustration: [link]

Operating system overview

Core logic of a **Telematic Control Unit** is usually performed by the **System-on-Chip**

It will run a Linux based distribution with an **API provided by the SOC** manufacturer to interact with mobile networks

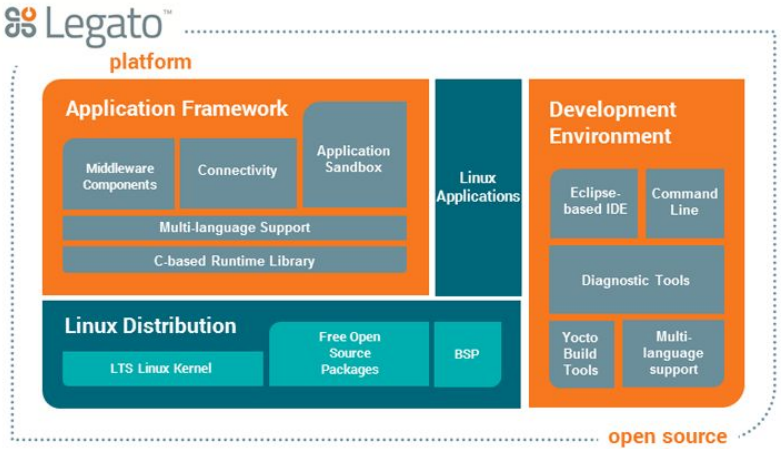


Illustration: [link]

Lab 1 - TCU NAND memory analysis

Goals

- For this lab, you'll practice **NAND Flash** memory analysis
- From the documentation, you'll look for the **NAND Flash** parameters
- And find **wear levelling data** that need to be removed/processed to dump the stored content
- Complete challenges **Telematic - NAND Flash**

Connecting TCU to a cellular test network

Quarkslab

Assessing cellular devices

- Analysing communication of **cellular devices** is less trivial than Wi-Fi or Bluetooth
- It requires **specific equipment** to emulate cellular network
- Device need to be modified to connect to the simulated network as **mutual authentication** is used in 3G+ networks
- As using radio frequency is regulated, it is recommended to use a **Faraday cage** to avoid emitting on restricted frequencies

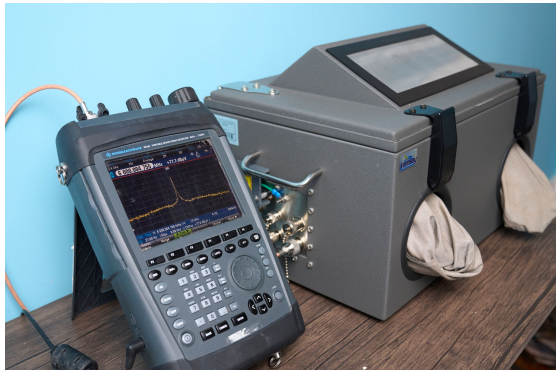
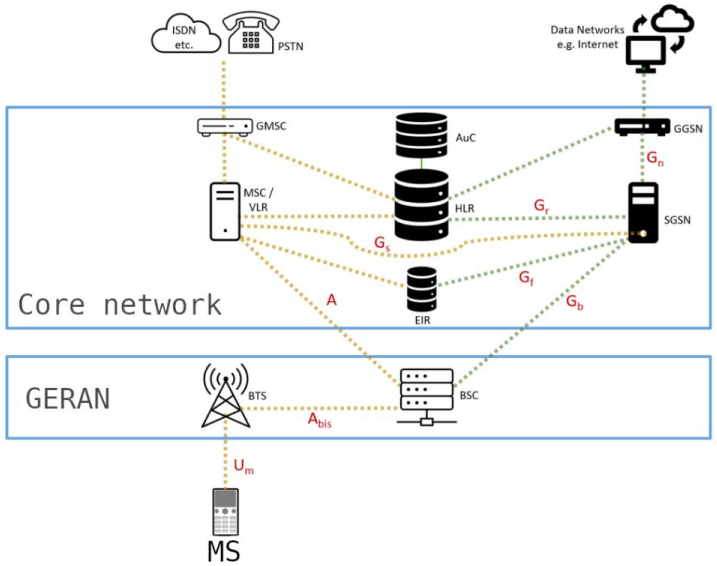


Illustration: [link]

2G network overview

- In 2G the **Mobile Station (MS)** connects to the **BTS** radio antenna
- The **Base Station Controlle**r exchanges with the **Core Network** to authenticate the subscriber, get data service, send/receive text messages and calls
- The **Home Location Register** holds details about subscribers
- The **Authorization Center** is responsible for authentication and ciphering of data



2G network limitation

- In 2G, the **authentication** of a subscriber is performed by the **HLR** only
- The **mobile network** choose the encryption algorithm that will be used, based on a list of supported one sent by the **Mobile Station**
- If the algorithm **A/0** is used, no encryption will be performed and data is sent in clear text
- Exploiting those flaws can be useful if the device under test could not be altered, to swap the SIM card for example

Emulating 2G BTS: YateBTS

- **YateBTS** is an open-source emulating 2G network using compatible SDR devices
- It provides to the user equipment **GPRS** data connectivity
- Voice **calls** could be made
- Using additional script, **SMS** and **binary SMS** could be generated from the host computer to the user equipment

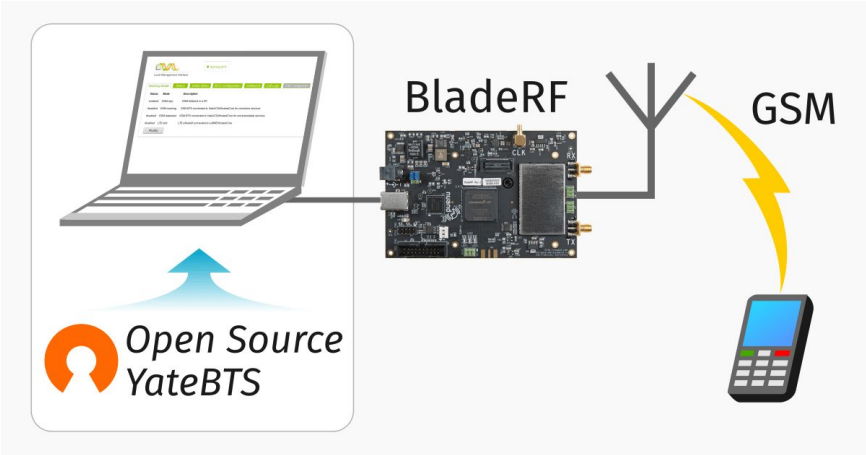
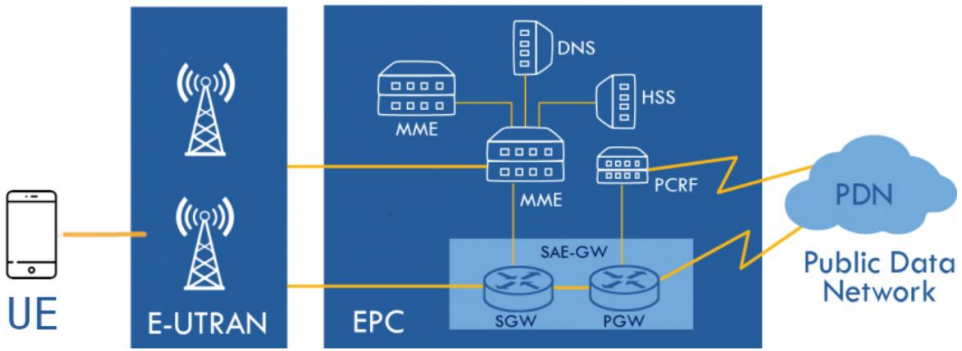


Illustration: [link]

LTE network overview

- **Long Term Evolution** (**LTE**) network improve mobile communication by offering full IP solution, replacing of circuit-switched architecture
- It offers **higher data rate**, reduce delay and improve responsiveness for Internet services
- A **User Equipment** connects to an **eNodeB** antenna that routes data to the **Evolved Packet Core**
- It uses **mutual authentication**



Emulating LTE network: srsRAN

- To emulate LTE and 5G network, the open-source **srsRAN** project is useful
- It could emulate **eNodeB**, but also **user equipment** (UE)
- **srsEPC** binary act as an Evolved Packet Core
- **srsENB** emulates the **eNodeB** using compatible SDR device
- It can also work with **Open5GS** to have a working **5G SA** network

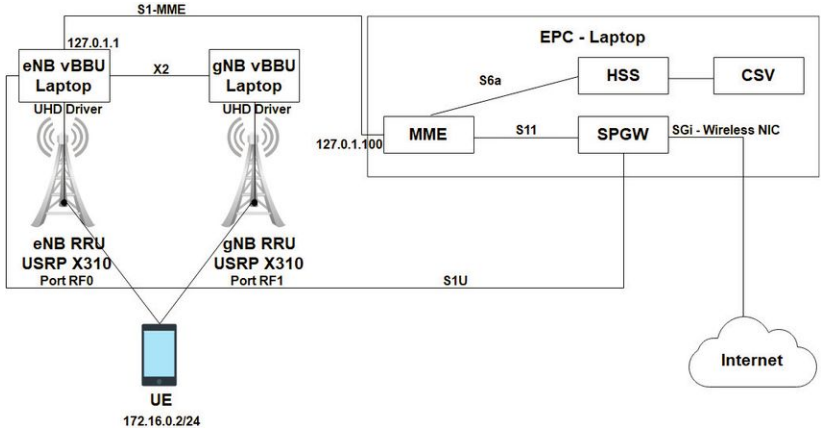


Illustration: [link]

Combining Open5GS EPC and srsRAN

- **srsRAN** does not support **IPv6 PDN**
- A device requiring such IP type could drop the connection
- **srsRAN** could be configured to use **Open5GS** EPC
- The following online resource explains how to configure srsRAN and Open5GS for this purpose

LTE evolved packet core

- **Mobility Management Entity (MME)**: manages user mobility, authentication and bearer session establishment
- **Serving Gateway (SGW)**: routes and forwards user data packets
- **Home Subscriber Server (HSS)**: central database containing user subscription information and authentication credentials
- **Packet Data Network Gateway (PGW)**: provides connectivity between user equipment and external packet data networks, like Internet
- **Policy and Charging Rules Function (PCRF)**: policy rules for quality of services and handles real-time charging control
- **IP Multimedia Subsystem (IMS)**: enables delivery of multimedia services over IP, supporting voice and video communications

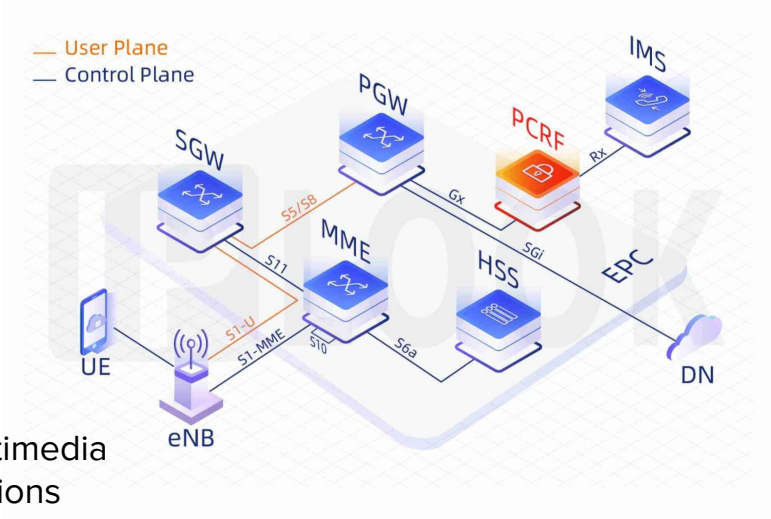


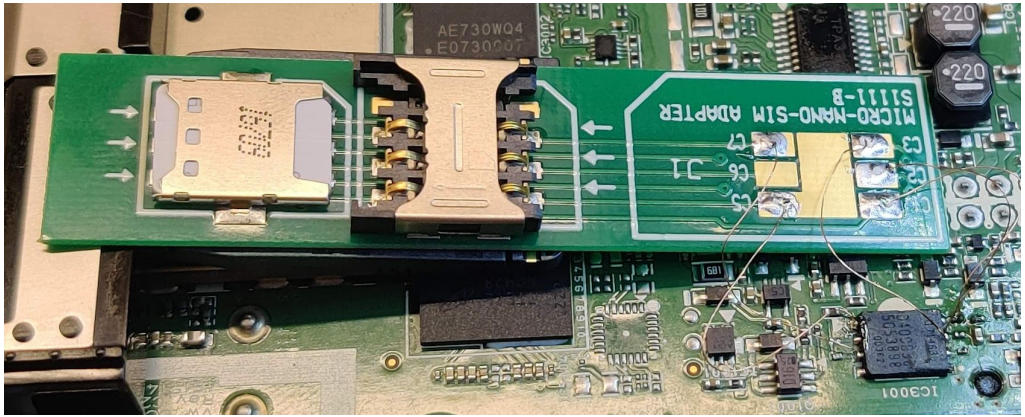
Illustration: [link]

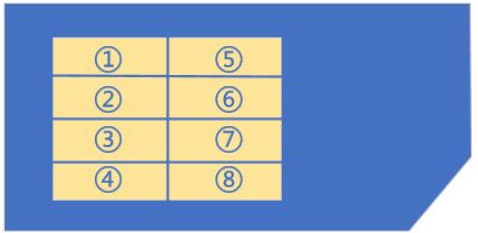
Preparing device under test

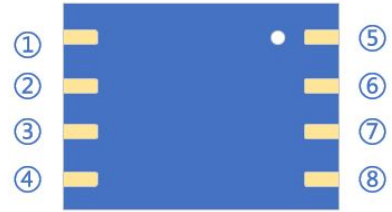
Quarkslab

eSIM

- To connect to the **cellular network**, the device will need a **SIM** card
- Old physical SIM cards are now replaced by **eSIM** integrated circuit
- eSim share pins with standard SIM
- By **desoldering** the eSIM or **cutting traces**, it could be replaced by a test SIM





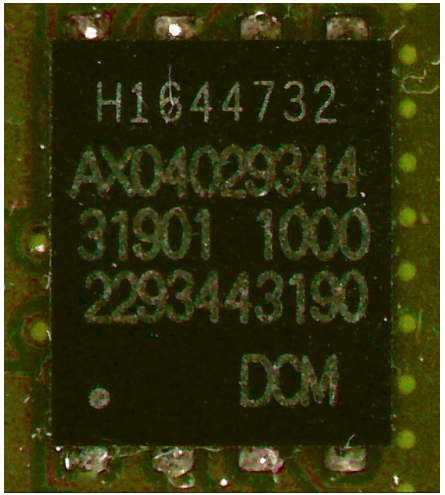


1	VCC	5	GND
2	RST	6	VPP
3	CLK	7	I/O
4	NC	8	NC

Illustration: [link]

eSIM - identification

- **eSIM** chips are easily identifiable
- The **ICCID**, **Integrated Circuit Card Identifier**, is a unique identifier of the eSIM is written on the package
- ICCID is **19-22 digits** number



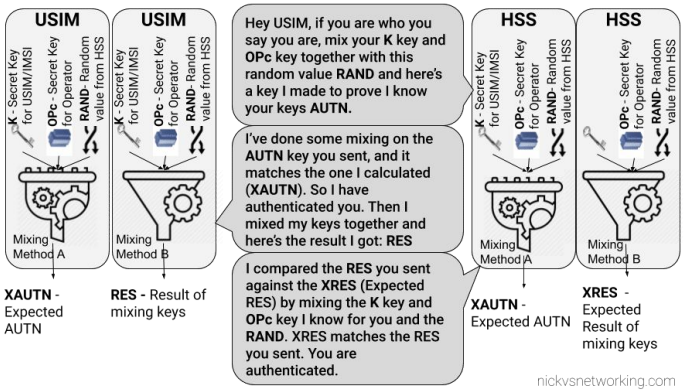
USIM card: MMC, MNC, Ki & OPC

- USIM hosts several data, including an authentication key **Ki** and the **operator code**
- **Operator code** could be derived (**OPC**) or in clear (**OP**)
- Each USIM has a unique **15-digit ID**, the **IMSI**
- The **Mobile Country Code** (**MCC** - 3 digits) and the **Mobile Network Code** (MNC - 2 digits) of the operator issuing the USIM is also registered
- **MNC** are defined for each country, like 208 for France, 310 for the US or 001 for test networks
- Some SIM cards, like **Sysmocom** ones, are easily programmable to be used with **YateBTS** and **srsRAN**

Illustration: [link]

Network authentication

- When connecting to the network of the operator who issued the USIM, authentication will be made with the **Home Subscriber Server** (**HSS**)
- The **HSS** knows the **Ki** and the **OPC** associated with the **IMSI** of the **USIM**
- **Ki** and **OPC** will be used to perform **mutual authentication** between the **USIM** and the **HSS**
- The **Ki** will be derived multiple times to cipher communications



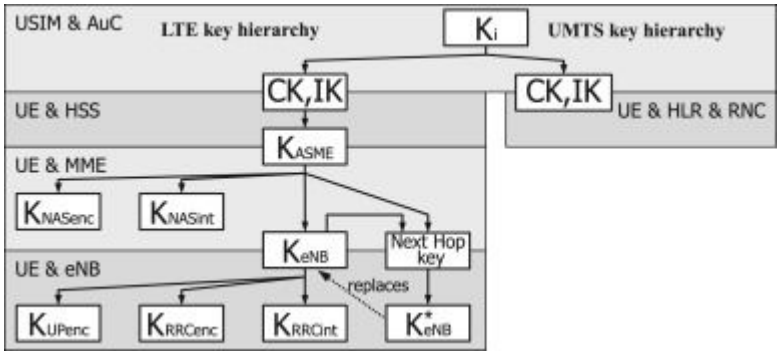


Illustration: [link] & [link]

Programming a Sysmocom SIM card

- Using a **PCSC** compatible reader and pySim, it is possible to read some content of a SIM card
- To activate the pcsc daemon on Linux, and check if reader is recognized:

```
$ pcscd
$ pcsc_scan
Using reader plug'n play mechanism
Scanning present readers...
0: Alcor Link AK9563 00 00
[...]
```

- To read data of the SIM card:

```
$ python3 pySim-read.py -p0
```

- To modify IMSI using the ICCID and the admin code ADM1 of the card:

```
$ python3 pySim-prog.py -p0 -a [ADM1] -s  [ICCID] -i [IMEI]
```

- To read the OPC and Ki, sysmo-isim-tool will be required:

```
$ sysmo-isim-tool.sja5.py --adm1 [ADM1] -ok
```

Configuring the device

- To be able to get a **PDN** context and access network, an **APN** must be set in the device
- For **TCU**, this parameter could often be written using a **UDS Write Data By Identifier** request
- If changing the value is not possible, retrieve the stored value using **UDS Read Data By Identifier** and change it in:
  - srsRAN only: `/etc/srsran/epc.conf`
  - srsRAN and Open5GS: `/etc/open5gs/smf.yaml` and `/etc/open5gs/smf.yaml`

Lab 1 - Using Sysmocom SIM card

Goals

- For this lab, you'll practice the basic command to prepare a programmable SIM card and your test mobile network setup
- Complete challenge **Telematic - SIM card**

Emulating 2G network

Quarkslab

YateBTS - introduction

- Once installed, **YateBTS** can be run using command

```
$ sudo yate -vv
```

- To route data packet, **ipv4 forwarding** and **iptables** rules must be set

```
$ sudo sysctl -w net.ipv4.ip_forward=1
$ sudo iptables -A POSTROUTING -t nat -s 192.168.99.0/24 ! -d 192.168.99.0/24 -j MASQUERADE
$ sudo iptables -A POSTROUTING -t nat -s 192.168.99.0/24 ! -d 192.168.99.0/24 -j MASQUERADE
```

- It exposes a telnet shell using port 5038 where some base commands could be sent

```
$ telnet 0 5038
Trying 0.0.0.0...
Connected to 0.
Escape character is '^]'.
YATE 6.2.1-devel1 r (http://YATE.null.ro) ready on user.
nipc list accepted
IMSI            MSISDN
--------------- ------------
001010000071399 | 8820071399
```

YateBTS - configuration

- Individual subscribers can be registered in `/usr/local/etc/yate/subscribers.conf`
- Using **regex** value, subscribers authentication can be bypassed to accept corresponding **IMSI**

```
$ cat /usr/local/etc/yate/subscribers.conf
[general]
country_code=882
smsc=8822003

# If set, regexp will bypass IMSI verification
# regexp=^001

[001019901001390]
msisdn=88221390
iccid=89882999000010013900
imsi_type=3G
active=on
op=54C927F80684484858F74D8CD6E53A35
opc=on
ki=*
imsi=00101990100139
```

YateBTS - commands

- Basic commands to simulate a call or send SMS can be sent using the Telnet access

```
$ telnet 0 5038

YATE 6.2.1-devel1 r (http://YATE.null.ro) ready on user.
nipc list accepted
IMSI            MSISDN
--------------- ------------
001010000071399 | 8820071399

smsend 8820071399 12345 Hello from 12345 !
message successfully sent.

callgen set called=8820071399
Set 'called' to '8820071399'

callgen set caller=888

Set 'caller' to '888'
callgen single
```

YateBTS - advanced behavior

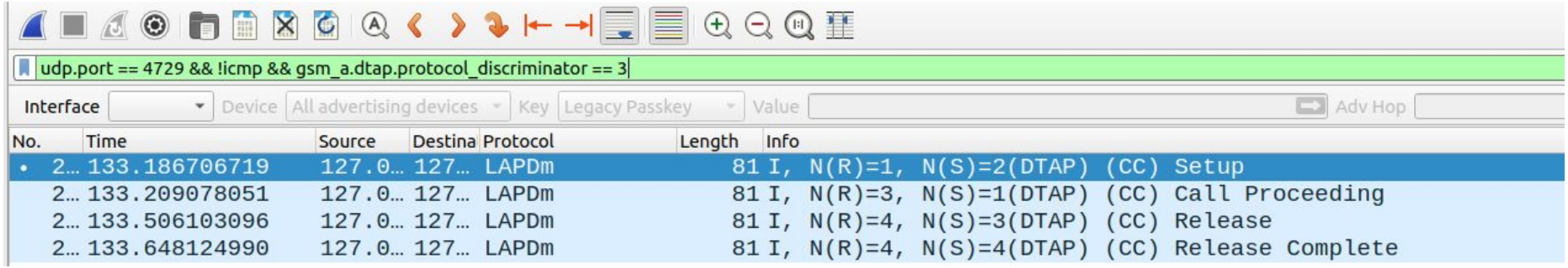
- The **Network-in-a-PC** version of **YateBTS** supports **Javascript** scripting, to perform automated or advanced task

- Editing **nipc.js** or **welcome.js** in */usr/local/share/yate/* allows for instance to automate action on calls or message sent by the device under test

```
// Automate action on called number
if (message.called == "888") {
  Engine.debug(Engine.DebugInfo, "Got call to 888");
  Channel.callTo("wave/record/-",{"maxlen":10000});
}

// or caller number
if (message.caller == "8820071399") {
  Engine.debug(Engine.DebugInfo, "Got call from MSISDN 8820071399");
  Channel.callTo("wave/record/-",{"maxlen":10000});
}
```

YateBTS - monitoring

- In `/usr/local/etc/yate/ybts.conf` tapping could be set to monitor **GSM/GPRS** traffic in **Wireshark** on port 4729
- To find **IMSI/IMEI** that connects to the BTS, use filter: **gsm_a.dtap.msg_mm_type**
- To find SMS sent/received: **gsm_a.dtap.protocol_discriminator == 9**
- For calls: **gsm_a.dtap.protocol_discriminator == 3**



Lab 2 - Analysing YateBTS data

Goals

- Analysing a network capture from a GSM/GRPS tap of YateBTS, can you retrieve key information ?
- Complete challenges **Telematic - YateBTS**

Emulating LTE network

Quarkslab

Training setup

- For this training, we will use **ZMQ** protocol to exchange packets between the **Telematic Control Unit** and your **Virtual Machine**
- Doing so, no **Software Defined Radio** device are required, avoiding issues related to radio traffic
- In the **Telematic Control Unit**, **srsRAN** has been compiled with libpcsclite-dev installed
- This allows to execute **srsUE** to emulate an User Equipment on the **Telematic Control Unit**, with the **PCSC** support to get data (IMSI, Ki, OPC) from the provided SIM card
- If you want to reproduce this setup, look at **srsRAN** documentation: https://docs.srsran.com/projects/4g/en/next/app_notes/source/zeromq/source/
- **Open5GS** is also installed, script **srsenb_zmq_start.sh** will manage the correct initialization between Evolved Packet Core used: **srsRAN** or **Open5GS**
- The **Telematic Control Unit** will emulate an **User Equipement** when a SIM card is detected and IP connection to the VM through IPv4 address 192.168.111.2/24 active

Training setup

srsRAN: configuring SIM data

- To accept the assessed device, **srsRAN** need to know the **IMSI**, **Ki** and **OPC** of our **SIM** card
- By default, **srsRAN** use the `/etc/srsran/user_db.csv` file

```
$ cat /etc/srsran/user_db.csv
[...]
# card_name, auth, imsi, ki, op_type, OP/OPC, AMF, SQL, QCI, IP Allow
sysmo_97,mil,001010000071397,7ce2…..f97ef8ce97,opc,ac5e562dd0c64……3428203,8000,000000001635,7,dynamic
sysmo_98,mil,001010000071398,0b32…..12dc1b78f1,opc,e73c8a8e7a8c…….e234dee,8000,000000001696,7,dynamic
```

- If **OPC** is used in the **USIM**, auth type will be `mil` (milenage)
- For **USIM** with only **OP** set, auth type will be `xor` and the OP/OPC field need to be set at `op`

srsRAN: other parameters

- In `/etc/srsran`, other configuration files are available
- **MCC** and **MNC** could be set in `/etc/srsran/enb.conf`
- If changed, they must also be configured in `/etc/srsran/epc.conf`
- `/etc/srsran/epc.conf` also sets the **APN** (internet) and the default IP address range given in the **PDN** context (172.16.0.1)
- **srsRAN** does not support **IPv6**

srsRAN: starting emulated network

- **eNodeB** and **EPC** are independant in srsRAN and must be started individually
- For testing, we will use the srsRAN virtual radio feature, emulating the **UE** and **eNodeB** using **ZMQ**
  https://docs.srsran.com/projects/4g/en/next/app_notes/source/zeromq/source/

```
sudo srsenb --rf.device_name=zmq
--rf.device_args="fail_on_disconnect=true,tx_port=tcp://*:4000,rx_port=tcp://tcu_ip:4001,id=enb,base_srate=23.04e6"
```

```
$ sudo srsepc
---  Software Radio Systems EPC  ---
[...]
Reading configuration file /etc/srsran/epc.conf...
HSS Initialized.
MME S11 Initialized
MME GTP-C Initialized
MME Initialized. MCC: 0xf001, MNC: 0xff01
SPGW GTP-U Initialized.
SPGW S11 Initialized.
SP-GW Initialized.
S1 Setup Request - eNB Name: srsenb01, eNB id: 0x19b
[...]
Sending S1 Setup Response
```

```
$ sudo srsenb [...]
---  Software Radio Systems EPC  ---
[...]
Supported RF device list: bladeRF zmq file
CHx base_srate=23.04e6
[...]
==== eNodeB started ===
Current sample rate is 11.52 MHz with a base rate of
23.04 MHz (x2 decimation)
Current sample rate is 11.52 MHz with a base rate of
23.04 MHz (x2 decimation)
Setting frequency: DL=2680.0 Mhz, UL=2560.0 MHz
for cc_idx=0 nof_prb=50
```

Lab 3 - LTE network over ZMQ

Goals

- You'll configure **srsRAN** to accept **User Equipment** using the provided SIM Card
- Complete challenges **Telematic - srsRAN**

open5GS: configuring SIM data

- In **Open5GS**, subscribers data are stored in a mongoDB database
- To add a new subscriber, use tool **open5gs-dbctl**

```
$ open5gs-dbctl add IMSI Ki OPC
```

- By default, **Open5GS** will register **APN internet**, if a custom APN is required use:

```
$ open5gs-dbctl add IMSI Ki OPC APN
```

- To remove a subscriber:

```
$ open5gs-dbctl remove IMSI
```

open5GS: usage

- **Open5GS** runs as several system services
- When starting **srsENB**, you can check the proper S1 connection to the Open5GS EPC:

```
$ sudo tail /var/log/open5gs/mme.log
09/16 16:04:32.909: [mme] INFO: eNB-S1 accepted [127.0.1.1]:51780 in s1_path module (../src/mme/s1ap-sctp.c:114)
09/16 16:04:32.909: [mme] INFO: eNB-S1 accepted[127.0.1.1] in master_sm module (../src/mme/mme-sm.c:108)
09/16 16:04:32.909: [mme] INFO: [Added] Number of eNBs is now 1 (../src/mme/mme-context.c:3035)
```

- To monitor User Equipment association and IP assignment, use the following log files:

```
$ tail /var/log/open5gs/upf.log
[upf] INFO: [Added] Number of UPF-Sessions is now 1 (../src/upf/context.c:212)
09/16 16:34:48.536: [upf] INFO: UE F-SEID[UP:0xf21 CP:0x500]
APN[internet] PDN-Type[3] IPv4[10.45.0.3] IPv6[2001:db8:cafe:2::3] (../src/upf/context.c:498)

$ tail /var/log/open5gs/mme.log
09/16 16:24:58.815: [mme] INFO: [Added] Number of MME-Sessions is now 1 (../src/mme/mme-context.c:5166)
09/16 16:24:58.849: [esm] ERROR: Invalid APN[internet2] (../src/mme/esm-handler.c:275)
09/16 16:24:58.850: [mme] INFO: Removed Session: UE IMSI:[001010000071397] APN:[Unknown]
```

open5GS: configuring PDN

- To set custom IP pool and APN, you'll need to modify `/etc/open5gs/smf.yaml` and `/etc/open5gs/upf.yaml`
- Do not forget to add supported networks in `/etc/systemd/network/99-open5gs.network`

```
session:
        - subnet: 192.168.11.0/24
          gateway: 192.168.11.1
          dnn: internet
          dnn: secret
- subnet: 2001:db8:cafe::/48
          gateway: 2001:db8:cafe::1
          dnn: internet
- subnet: 10.45.0.0/16
          gateway: 10.45.0.1
          dnn: secret2
```

Lab 3 - Using Open5GS EPC

Goals

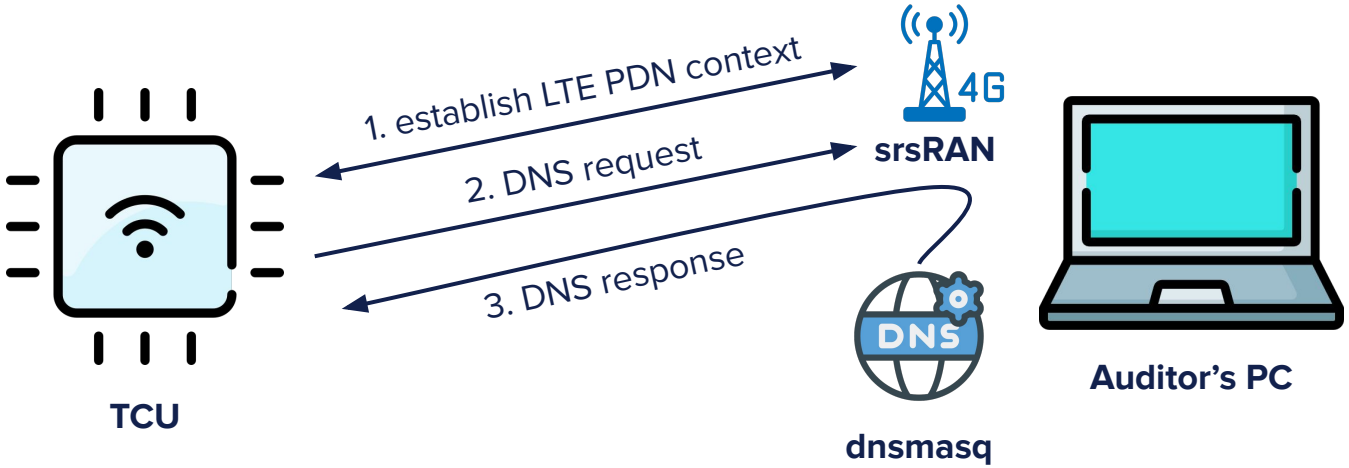
- You'll now use **Open5GS** Evolved Packet Core
- Complete challenge **Telematic - Open5GS**

Intercepting and downgrading communications

Quarkslab

Intercepting and redirecting traffic

- TCU commonly use **private APN**, having access to servers not exposed on Internet
- Setting a local **DNS server** will allow redirecting requests to various domains on our machine
- `/etc/srsran/epc.conf` need to be modified to set the correct rogue DNS IP, `172.16.0.1` by default
- For Open5GS, `/etc/open5gs/smf.yaml` need to be edited



Intercepting and redirecting traffic - training setup

- For this training, the **DNS hijacking** will be performed using **DNSmasq**
- It is already installed and configured on your VM for this purpose
- To add a domain to intercept, edit `/etc/dnsmasq.conf`

```
$  cat /etc/dnsmasq.conf
[...]
# Add domains which you want to force to an IP address here.
address=/hijack.me/172.16.0.1
```

- To apply changes, DNSmasq needs to be rebooted

```
$  sudo systemctl restart dnsmasq
```

Lab 4 - DNS MitM

Goals

- You'll edit **DNSmasq** to redirect **Telematic Control Unit** requests to your machine
- Complete challenges **Telematic - DNSmasq**

Downgrading MQTT TLS traffic

- **MQTT** is often used in TCU, with ciphered communication using **TLS** on port **8883**
- Server and clients each have:
  - a **private key** (*.key)
  - a **signed certificate** from a **Certificate Authority** (*.crt)
  - the certificate of the **Certificate Authority** (ca.crt)
- Those materials may not be alterable in the device, but could be reuse to create a **rogue server** if **stored in clear**

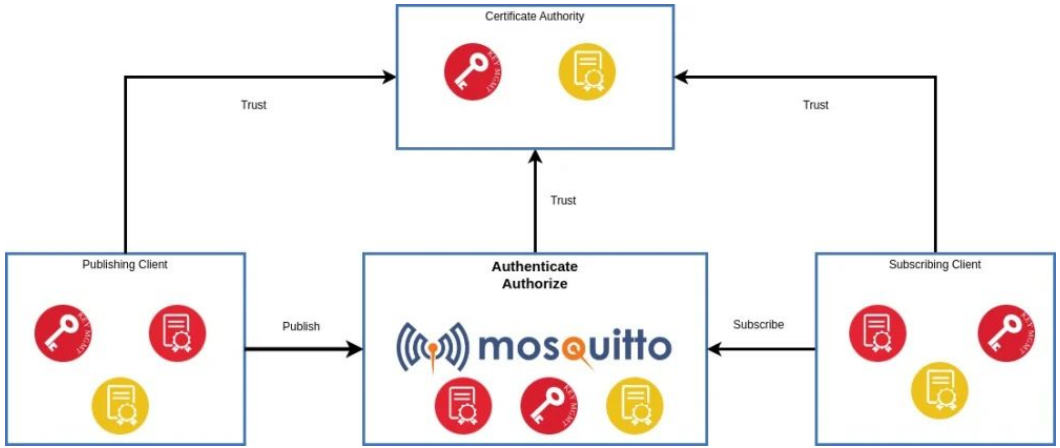


Illustration: [link]

Extracting certificates from binaries

- Some binaries could store **certificates** for **asymmetric encryption**
- Standard storage format is **ASN.1 DER** (Distinguished Encoding Rules)
- If stored encoded in **base64**, search for **“MII”**
- Otherwise, look for pattern `0x30 0x82 xx yy` where `xx yy` will be the length of data to analyse
- Using **Python cryptography** module, check if it returns a valid certificate:

```
from cryptography import x509
data = open(binary.bin','rb').read()
pos = data.find(b'\x30\x82')  # Add iteration to find more cert
cert_len = int.from_bytes(data[pos+2:pos+4],'big')
try:
    decoded_cert = x509.load_der_x509_certificate(data[pos:pos+cert_len+4])
    print(f"\nFound X509 at {pos:#010x}:")
    print(f"Issuer: {decoded_cert.issuer}, subject: {decoded_cert.subject}")
    print(f"OID: {decoded_cert.signature_algorithm_oid}")
except Exception as e:
    pass
```

Certificates - explore using OpenSSL

- Raw certificate in **ASN.1 DER** format could be analysed using OpenSSL

```
$ openssl asn1parse --inform DER -in undefined_cert.bin
    0:d=0  hl=4 l=1213 cons: SEQUENCE
    4:d=1  hl=2 l=   1 prim: INTEGER           :00
    7:d=1  hl=2 l=  13 cons: SEQUENCE
    9:d=2  hl=2 l=   9 prim: OBJECT            :rsaEncryption
   20:d=2  hl=2 l=   0 prim: NULL
   22:d=1  hl=4 l=1191 prim: OCTET STRING      [HEX DUMP]: xxxxxx
```

- Get more details for **X509** certificates:

```
$ openssl x509 -noout -text -in client.crt
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            2d:32:43:2f:1d:d0:d1:c1:97:96:12:f0:22:69:a1:16:52:08:cf:31
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = AU, ST = Some-State, O = TrustedCertAuth
        [...]
```

Lab 5 - Setting a rogue MQTT server

Goals

- The **TCU** tries to connect to a **MQTT** server
- Analyse the **frames** sent using **Wireshark** and try to redirect requests to the hosted **Mosquitto MQTT** server
- Check if you can **extract certificates and key** from the binary which is provided in the `~/Lab` folder
- Configure the **MQTT** server so the TLS communication between the binary and the server could be established and **capture the flag sent**
- The **Mosquitto MQTT** server configuration is available in `/etc/mosquitto` with logs in `/var/log/mosquitto` and can be restarted using command

```
$ sudo systemctl restart mosquitto
```

Thank you

Contact information:

Email: contact@quarkslab.com

Phone: +33 1 58 30 81 51

Website: www.quarkslab.com

@quarkslab

Quarkslab