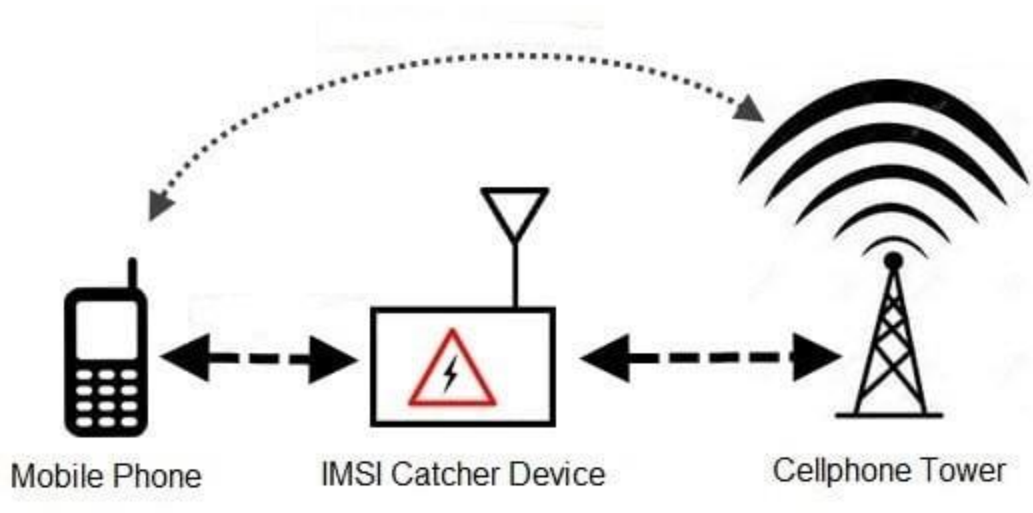


Capture Cellular Signals

ADVANCED TOPICS IN NETWORKS

Kotsiaridis Konstantinos 2547
Pavlidis Panagiotis 2608
Fall 2021-2022
Semester Project

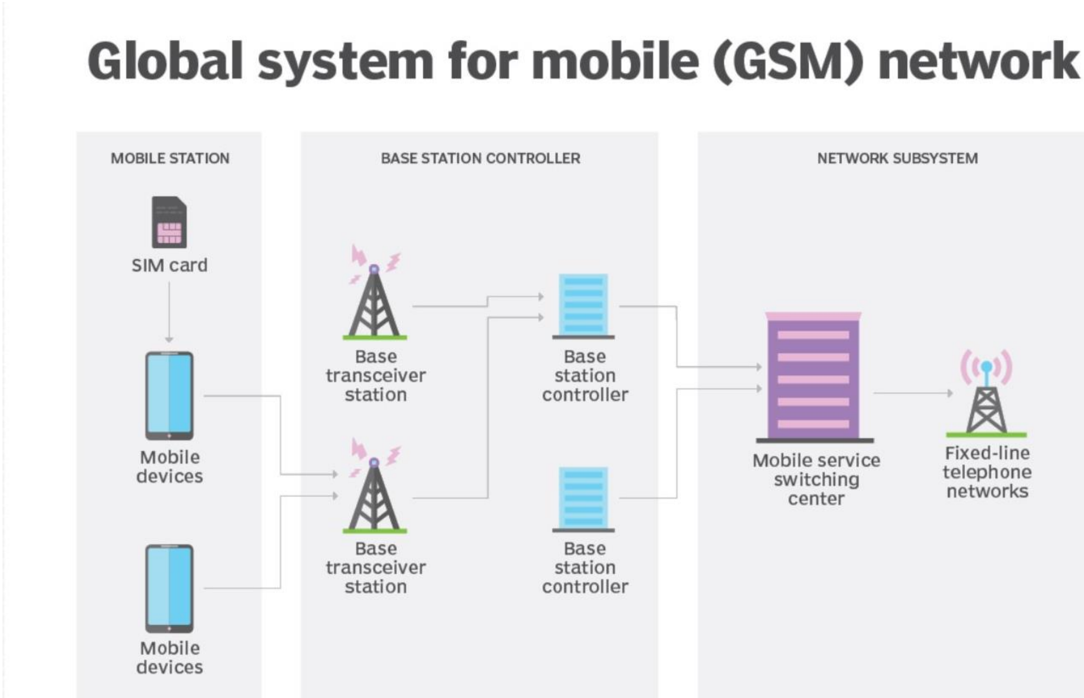
Introduction

This project is composed of two parts. In the first part, our goal was to capture Cellular Base Station Signals (GSM signals only), decode them and extract information about the Base Stations near the scanned region. This information was inserted in a database and after analyzation, we rendered the Base Stations in a map.

In the second part, we created a check in mechanism using the information from the captured Base Station Signals that keeps track of the time duration that each person stays in the scanned area. In order to capture the Base Station Signals we used the ADALM – PLUTO SDR.

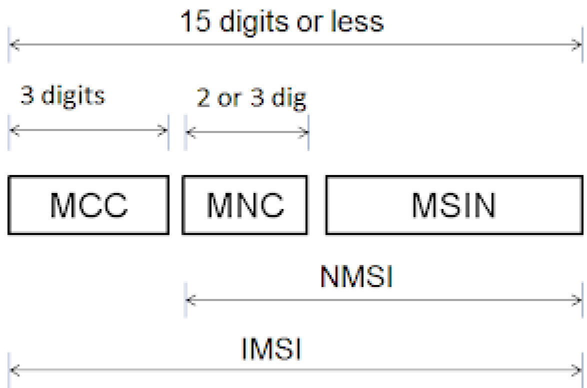
GSM ARCHITECTURE

Our project is based on Global System for Mobile Communication(GSM).

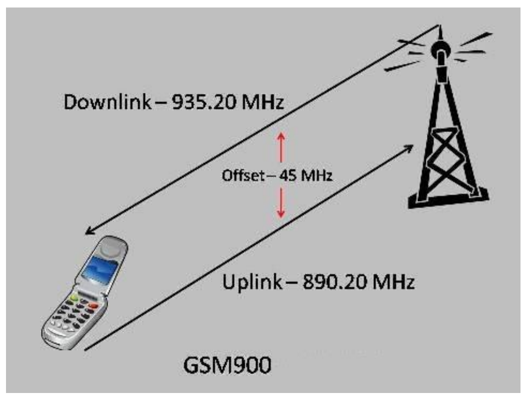


As shown above a mobile phone connects to a base station using its SIM card and exchanges packets. Each SIM card has a unique identifier which is called International Mobile Subscriber Identity (**IMSI**) . IMSI is a number that uniquely identifies every user of a cellular network. It is sent by the mobile device to the network. To prevent eavesdroppers from identifying and tracking the subscriber on the radio interface, the IMSI is sent as rarely as possible and a randomly-generated TMSI is sent instead.

An IMSI is usually presented as a 15-digit number but can be shorter. The first 3 digits represent the mobile country code (MCC), which is followed by the mobile network code (MNC), either 2-digit (European standard) or 3-digit (North American standard). The length of the MNC depends on the value of the MCC. The remaining digits are the mobile subscription identification number (MSIN) within the network's customer base, usually 9 to 10 digits long, depending on the length of the MNC.



For the communication between mobile station and base station two channels are used, the uplink and the downlink, and they differ in frequency.



In Greece, there are two main frequency bands for GSM (**GSM-900, GSM/DCS- 1800**). Each provider has a license to transmit in a specific band as shown in the below tables.

900 MHZ	COSMOTE	925-930	880-885	χωρίς περιορισμούς	30/9/2012	29/9/2027
	COSMOTE	930-935	885-890	χωρίς περιορισμούς	9/9/2002 9/9/2017	08/09/2017 29/9/2027
	WIND	935-945	890-900	χωρίς περιορισμούς	30/9/2012	29/9/2027
	VODAFONE	950-960	905-915	χωρίς περιορισμούς	30/9/2012	29/9/2027
	VODAFONE	945-950	900-905	χωρίς περιορισμούς	6/8/2001 6/8/2016	05/08/2016 29/9/2027

1800 MHz	WIND	1805-1810	1710-1715	χωρίς περιορισμούς	6/8/2001	5/8/2016
	WIND	1810-1820	1715-1725	χωρίς περιορισμούς	6/8/2001	5/8/2016
	VODAFONE	1820-1830	1725-1735	χωρίς περιορισμούς	1/7/2012	31-06-2027
	VODAFONE	1830-1845	1735-1750	χωρίς περιορισμούς	6/8/2001	5/8/2016
	COSMOTE	1855-1880	1760-1785	GSM/DCS (στο τμήμα 1855-1865 MHz και 1760-1770 MHz χωρίς περιορισμούς)	5/12/1995	4/12/2020
	COSMOTE	1845-1855	1750-1760	χωρίς περιορισμούς	1/7/2012	31/06/2027

SW AND HW TOOLS

As part of the software tools that have been used in our project are:

- GNURADIO-COMPANION – PlutoSDR extension
- Libiio
- Pluto SDR drivers
- gr-gsm module
- IMSI-CATCHER
- Node.js module bscoords
- OpenCellID API
- Folium Map library
- SQLite3
- Docker

The only hardware equipment that has been used is the ADALM-PLUTO SDR.

PROJECT STAGES FOR PART 1

1st Stage

- Docker image creation.
- Installation of libiio for interaction with PlutoSDR.
- Installation of GNURADIO-COMPANION and PlutoSDR extension which contains the PlutoSDR source block for capturing signals.

- Installation of gr-gsm module that uses GNURADIO blocks for capturing and demodulation of GSM packets. (**grgsm_livemon.py**)
- Installation of IMSI-CATCHER module that contains **simple_imsi_catcher.py** which decodes the information of GSM packets.

2nd Stage

Adding functionality in **grgsm_livemon.py** for channel hoping in band frequencies between 925MHz – 960MHz and 1805MHz – 1880 MHz with 0.2MHz step. In this way, we cover the whole spectrum of downlink channels for GSM communications and we are able to capture all the GSM packet transmissions. Also, on each frequency band we scanning for 2 seconds.

```
conn = create_connection("/root/cell_info.db")
cursor = conn.cursor()
try:
    fc_slider = 925400000
    print(int(fc_slider))
    for i in range(int(fc_slider),960400000,200000):
        start_time = time.time()
        tb.set_fc_slider(i)
        print(tb.get_fc_slider())
        end_time = time.time()
        if conn:
            conn.execute(
                u"INSERT INTO observations (freq) " + "VALUES (?);",
                [str(tb.get_fc_slider())]
            )
        conn.commit()
        while(end_time - start_time <= 2):
            end_time = time.time()

    fc_slider = 1805000000
    print(int(fc_slider))
    for i in range(int(fc_slider),1880400000,200000):
        start_time = time.time()
        tb.set_fc_slider(i)
        print(tb.get_fc_slider())
        end_time = time.time()
        if conn:
            conn.execute(
                u"INSERT INTO observations (freq) " + "VALUES (?);",
                [str(tb.get_fc_slider())]
            )
        conn.commit()
        while(end_time - start_time <= 2):
            end_time = time.time()
```

Figure 1: grgsm_livemon.py channel hopping.

```python
    for i in range(len(freq_val)):
        #print(i)
        sql_delete_query = '''DELETE from observations where cell is NULL; '''
        cursor.execute(sql_delete_query)
        conn.commit()
    print("Records deleted successfully ")
    cursor.close()
    conn.close()

    f = open("/root/.close.txt", "w+")
    f.write("1")
    f.close()
```

Figure 2: grgsm_livemon.py deletion empty database entries.

The **simple_imsi_catcher.py** decodes the captured GSM packets from **grgsm_livemon.py** and exports the desired information in an sqlite3 database called **cell_info.db**.

```
Nb IMSI ; TMSI-1     ; TMSI-2     ; IMSI                ; country                  ; brand      ; operator                              ; MCC  ; MNC  ; LAC   ; CellId ; Timestamp
1       ;            ;            ; 202 01 0909376968   ; Greece                   ; Cosmote    ; COSMOTE - Mobile Telecommunications S.A. ; 202  ; 01   ; 2120  ; 41427  ; 2022-02-23T16:34:00.550356
2       ;            ;            ; 202 01 0924876455   ; Greece                   ; Cosmote    ; COSMOTE - Mobile Telecommunications S.A. ; 202  ; 01   ; 2120  ; 40308  ; 2022-02-23T16:34:15.043042
3       ; 0x01ccae59 ;            ; 202 01 0909560541   ; Greece                   ; Cosmote    ; COSMOTE - Mobile Telecommunications S.A. ; 202  ; 01   ; 2120  ; 40268  ; 2022-02-23T16:34:55.451461
4       ;            ;            ; 202 01 0928094880   ; Greece                   ; Cosmote    ; COSMOTE - Mobile Telecommunications S.A. ; 202  ; 01   ; 2120  ; 40268  ; 2022-02-23T16:34:55.567783
6       ;            ;            ; 310 410 175736042 ; United States of America ; AT&T       ; AT&T Mobility         ; 202  ; 01   ; 2120  ; 41448  ; 2022-02-23T16:35:17.388446
6       ;            ;            ; 310 410 175736042 ; United States of America ; AT&T       ; AT&T Mobility         ; 202  ; 01   ; 2120  ; 41448  ; 2022-02-23T16:35:17.390776
7       ;            ;            ; 202 01 0927403122   ; Greece                   ; Cosmote    ; COSMOTE - Mobile Telecommunications S.A. ; 202  ; 01   ; 2120  ; 41448  ; 2022-02-23T16:35:17.697214
8       ;            ;            ; 202 01 0923894958   ; Greece                   ; Cosmote    ; COSMOTE - Mobile Telecommunications S.A. ; 202  ; 01   ; 2120  ; 41448  ; 2022-02-23T16:35:24.656284
9       ;            ;            ; 310 260 297996167 ; United States of America ; T-Mobile   ; T-Mobile USA          ; 202  ; 05   ; 55    ; 19353  ; 2022-02-23T16:37:47.671174
10      ;            ;            ; 202 05 2930592757   ; Greece                   ; Vodafone   ; Vodafone Greece        ; 202  ; 05   ; 55    ; 59563  ; 2022-02-23T16:37:53.274205
11      ; 0xe350c93e ;            ; 202 05 2936636843   ; Greece                   ; Vodafone   ; Vodafone Greece        ; 202  ; 05   ; 55    ; 59563  ; 2022-02-23T16:37:53.456269
12      ;            ;            ; 202 05 2967004435   ; Greece                   ; Vodafone   ; Vodafone Greece        ; 202  ; 05   ; 55    ; 59563  ; 2022-02-23T16:37:56.752894
13      ;            ;            ; 214 01 8906612123   ; Spain                    ; Vodafone   ; Vodafone Spain         ; 202  ; 05   ; 55    ; 19351  ; 2022-02-23T16:37:56.984435
14      ;            ;            ; 202 05 2988616899   ; Greece                   ; Vodafone   ; Vodafone Greece        ; 202  ; 05   ; 55    ; 19351  ; 2022-02-23T16:39:18.689920
```

Figure 3: IMSI-CATCHER stdout.

The initial schema of the cell_info.db contained the columns:

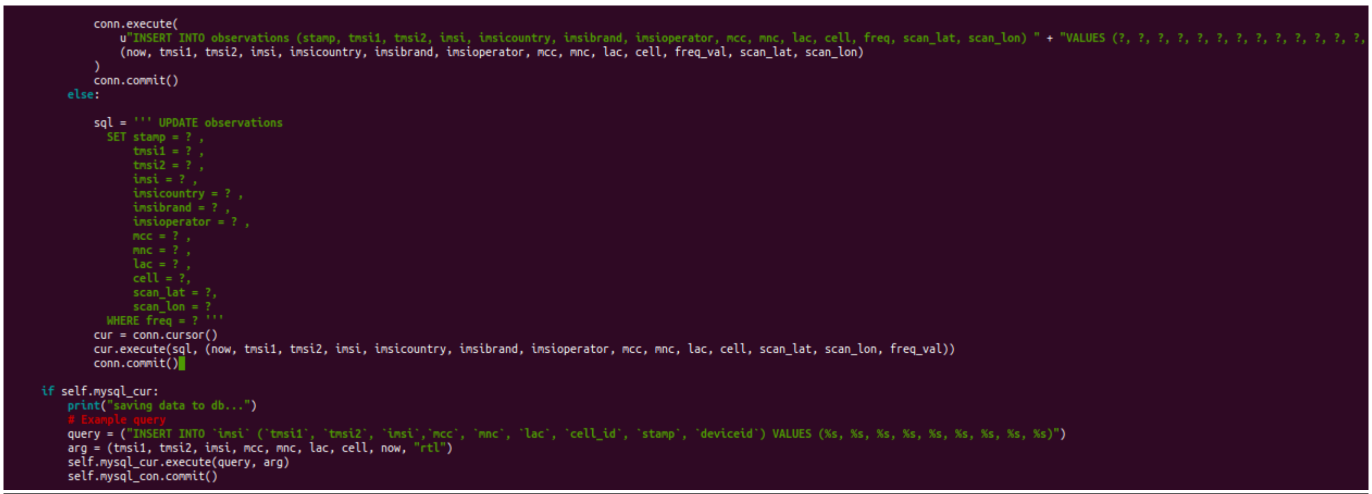
Timestamp	Tmsi1	Tmsi2	Imsi	Imsicountry	Imsibrand	Imsioperator	Mcc	Mnc	lac	cell

The lac and the cell columns are the part that comprise the msin part of the IMSI. LAC is the location area of the BTS and cell is the identification number of the area that an antenna in BTS covers.

The schema of **cell_info.db** is altered and **freq, scan_lat, scan_lon** columns are added. Scan_lat and scan_lon values are used to classify the entries based on the scanning location. They are taken using google maps' coordinates by the user and are added as command-line arguments in the simple_imsi_catcher.py.

The freq column corresponds to the channel frequency that PlutoSource captured a specific packet. Because, this value doesn't exists in GSM packets, simple_imsi_catcher.py can't access it, so we add a cell_info.db entry for every channel hop(all columns except freq are empty). In the end of scanning, we delete the empty entries and only preserve the entries that imsi-catcher added.

We execute simple_imsi_catcher.py and grgsm_livemon.py in parallel and every time the cell_info.db is deleted. We use a file named .close.txt for signaling the simple_imsi_catcher.py to close when grgsm_livemon.py finish the execution.



```
            conn.execute(
                u"INSERT INTO observations (stamp, tmsi1, tmsi2, imsi, imsicountry, imsibrand, imsioperator, mcc, mnc, lac, cell, freq, scan_lat, scan_lon) " + "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
                (now, tmsi1, tmsi2, imsi, imsicountry, imsibrand, imsioperator, mcc, mnc, lac, cell, freq_val, scan_lat, scan_lon)
            )
            conn.commit()
        else:

            sql = ''' UPDATE observations
              SET stamp = ? ,
                  tmsi1 = ? ,
                  tmsi2 = ? ,
                  imsi = ? ,
                  imsicountry = ? ,
                  imsibrand = ? ,
                  imsioperator = ? ,
                  mcc = ? ,
                  mnc = ? ,
                  lac = ? ,
                  cell = ?,
                  scan_lat = ?,
                  scan_lon = ?
              WHERE freq = ? '''
            cur = conn.cursor()
            cur.execute(sql, (now, tmsi1, tmsi2, imsi, imsicountry, imsibrand, imsioperator, mcc, mnc, lac, cell, scan_lat, scan_lon, freq_val))
            conn.commit()

    if self.mysql_cur:
        print("saving data to db...")
        # Example query
        query = ("INSERT INTO `imsi` (`tmsi1`, `tmsi2`, `imsi`,`mcc`, `mnc`, `lac`, `cell_id`, `stamp`, `deviceid`) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)")
        arg = (tmsi1, tmsi2, imsi, mcc, mnc, lac, cell, now, "rtl")
        self.mysql_cur.execute(query, arg)
        self.mysql_con.commit()
```

Figure 4: cell_info.db creation and update in simple_imsi_catcher.py

<u>3rd Stage</u>

We create a .js file, which uses the bscoords module, in order to make a request to the Opencellid API that responds back with coordinates of the BTS.

To begin with, we access the cell_info.db and we select the mcc, mnc, lac, cell , imsioperator , freq, scan_lat and scan_lon values. In the request to the opencellid API we include the mcc, mnc, lac and cell values and we retrieve the lat and lon values of the BTS.

We create a new persistent database named coords.db with the following schema:

lat	lon	provider	cell	freq	Scan_lat	Scan_lon

We insert the values that we have in the corresponding columns in cords.db.

For the request we need an API key that we got by signing up in the https://opencellid.org/ site.

```
'use strict';

const bs = require('../lib/bscoords');
const sqlite3 = require('sqlite3').verbose();

let db = new sqlite3.Database('/root/cell_info.db', sqlite3.OPEN_READONLY, (err) => {
  if (err) {
    console.error(err.message);
  }
  console.log('Connected to my database.');
});

let db_coords = new sqlite3.Database('/root/coords.db',(err) => {
        if(err) {
                console.error(err.message);
        }
        console.log('Database coords created');
});

//db_coords.run('DROP TABLE IF EXISTS coords');
db_coords.run('CREATE TABLE IF NOT EXISTS coords(lat double, lon double, provider string, cell string, freq string, scan_lat double, scan_lon double)');

const services = ['opencellid'];

bs.init({
    apikey_opencellid: 'pk.6d18b1a25cb511b416c7ebc11b8ddac1',

    'timeout': 3000
});

db.serialize(() => {
  db.each(`SELECT mcc,mnc,lac,cell,imsioperator,freq,scan_lat,scan_lon
           FROM observations`, (err, row) => {
    if (err) {
      console.error(err.message);
    }
```

Figure 5: Creation of coords.db and access of cell_info.db in coords.js.

```
    if ((row.mcc != '') && (row.mcc != null)) {
        console.log(row);
        bs
        .opencellid(row.mcc,row.mnc,row.lac,row.cell)
        .then(coords => {
                console.log(coords);
                        db_coords.run(`INSERT INTO coords(lat, lon, provider, cell, freq, scan_lat, scan_lon) VALUES (?, ?, ?, ?, ?, ?, ?)`,
                                [coords.lat,coords.lon,row.imsioperator,row.cell,row.freq,row.scan_lat,row.scan_lon], function(err) {
                        if (err) {
                                return console.log(err.message);
                        }
                        });
        })
        .catch(err => console.log(err));
    }
  });
});

db.close((err) => {
  if (err) {
    console.error(err.message);
  }
  console.log('Close the database connection.');
});
```

Figure 6: Retrievement of BTSs' coordinates and insertion in coords.db in coords.js.

4th Stage

We created a map_render.py file in which we render the BTS locations and the scanning locations in a map which is hosted in the localhost IP.

Our implementation is based on the folium map library. Using the lat and lon values for every entry in cords.db we create the markers for the BTS locations. We present the provider, cellid and freq values for every BTS in a pop-up label.

The scan_lat and scan_lon values are used to group the BTSs by scanning locations and each group has a specific colour.

```python
conn = create_connection(database)
with conn:
    print("2. Query all tasks")
    rows = select_all_tasks(conn)
    scan_coords = select_scan_coords(conn)
    scan_coords = tuple(set(scan_coords))

    print (scan_coords)
    lat_list = []
    for lat in scan_coords:
        lat_list.append(lat[0])

    print(lat_list)

colors = ['red', 'blue', 'green', 'purple', 'orange', 'darkred,lightred', 'beige', 'darkblue', 'darkgreen', 'cadetblue', 'darkpurple', 'white', 'pink', 'lightblue', 'lightgreen', 'gray', 'black', 'lightgray']

# create map
data = (np.random.normal(size=(100, 3)) *
        np.array([[1, 1, 1]]) +
        np.array([[48, 5, 1]])).tolist()
folium_map = folium.Map(location=[39.36345892631218, 22.948074885711076],
                tiles = 'Stamen Terrain')
#HeatMap(data).add_to(folium_map)
#dct = dict((y, x) for x, y in rows)
#tuple(set(rows)
for row in tuple(set(rows)):

    counter = 0
    for lat in lat_list:
        if(lat == row[5]):
            break
        counter += 1
        if(counter == 19):
            counter = 0

    if (row[2] == ""):
        provider = "Unknown"
    else:
        provider = str(row[2])
```

Figure 7: Map creation and retrievement of coordinates for rendering in map_render.py.

```python
    folium.Marker(
        location=[row[0], row[1]],
        popup=folium.Popup('<b>provider=</b><br>' + provider + '<br>'
                + '<b>cellid=</b>' + str(row[3]) +'<br>'
                + '<b>freq=</b>' + str(row[4]),min_width = '500%', max_width='500%'),
        icon=folium.Icon(colors[counter])
    ).add_to(folium_map)

counter = 0
for coords in scan_coords:
    folium.Marker(
        location=[coords[0], coords[1]],
        popup=folium.Popup("<b>scan_location_%d </b>" % (counter)),
        icon=folium.Icon(colors[counter], icon='home')
    ).add_to(folium_map)
    counter += 1
```

Figure 8: Markers and pop-up placements in map_render.py

5th Stage

This is the last stage of part 1 .We created a wrapper for part 1 named wrapper_part1.sh.

```
cd /root/IMSI-catcher


if [ $# -gt 1 ]
then
        xterm -e "bash -c 'python /root/grgsm_livemon.py'" &
        python3 simple_IMSI-catcher.py -s -w /root/cell_info.db -l $1 $2

else
        xterm -e "bash -c 'python /root/grgsm_livemon.py'" &
        python3 simple_IMSI-catcher.py -s -w /root/cell_info.db -l 39.36044374110071 22.949124812591084

fi

cd /root/node_modules/bscoords/test/

node test.js

cd


python3.8 map_render.py &
sleep 15
killall -e python3.8
```

Figure 9: wrapper_part1.sh script.

6th Stage

For part 2 of our project we used the already developed grgsm_livemon.py and simple_imsi_catcher.py in stage 2.

We created a check_in.py file in which we access the cell_info.db and retrieved the stamp, timsi1, timsi2, imsi and cell values. We also created a persistent database named check_in.db with the following schema:

First_check_in	imsi	cell	status	counter

In the imsi field we assign either the imsi, or timsi1 or timsi2 whichever exists in this order.

In the first_check_in field we assign the date of the first occurrence of this imsi entry in the check_in.db.

Our goal is to determine if an imsi subscriber has either “active” and “offline” status. We achieve this by using the counter value.

If subscriber’s imsi exist in the cell_info.db after the scanning, subscriber is present, counter is set to 3 and status is “Active”.

If subscriber’s imsi doesn’t exist in the cell_info.db after the scanning, subscriber is missing and counter is reduced by 1.

If counter > 0 subscriber is still “Active”, else if counter is 0, subscriber’s status is set to be “Offline”.

```
def sqlite_file(filename):
    try:
        sqlite_con = sqlite3.connect(filename)
    except Error as e:
        print(e)

    sqlite_con.execute("CREATE TABLE IF NOT EXISTS subscribers(first_check_in datetime, imsi text, cell integer, status text, counter integer);")

    return(sqlite_con)

def select_all_tasks(conn,name):

    cur = conn.cursor()
    cur.execute("SELECT * FROM %s" % (name))

    rows = cur.fetchall()
#    for row in rows:
#        print(row)
    return(rows)


if __name__ == "__main__":

    sub_conn = sqlite_file("/root/check_in.db")
    obs_conn = sqlite3.connect("/root/cell_info.db")

    subs = select_all_tasks(sub_conn,'subscribers')
    obs = select_all_tasks(obs_conn,'observations')
```

Figure 10: Connection and retrievement of entries of check_in.db and cell_info.db in check_in.py.

```python
flag = False
imsi = 0
for row in obs:
    if(row[3] == ''):
        if(row[1] == ''):
            if(row[2] == ''):
                continue
            else:
                imsi = row[2]
        else:
            imsi = row[1]
    else:
        imsi = row[3]

    if len(subs) == 0:
        sub_conn.execute(
            u"INSERT INTO subscribers(first_check_in, imsi, cell, status, counter) " + "VALUES (?, ?, ?, ?, ?);",
            (row[0], imsi, row[10], "Active", 3)
        )
        sub_conn.commit()
    else:
        for sub in subs:
            if(imsi == sub[1]):
                flag = True
                break
            else:
                continue
```

Figure 11: IMSI selection and insertion of subscriber's entry in check_in.py.

```python
subs = select_all_tasks(sub_conn,'subscribers')

print("       First Check In       |         IMSI        | Cell-ID |  Status    | Counter ")
print("---------------------------------------------------------------------------------")
for sub in subs:
    if(len(sub[1]) < 17):
        print ("%s |      %s     |  %s  |  %s  |     %s" % (sub[0],sub[1],sub[2],sub[3],sub[4]))
    else:
        print ("%s | %s |  %s  |  %s  |     %s" % (sub[0],sub[1],sub[2],sub[3],sub[4]))

print("\n")
```

Figure 12: Output formation of IMSI subscribers in check_in.db in check_in.py.

7th Stage

This is the last stage of part 2 . We created a wrapper for part 2 named wrapper_part2.sh.

```bash
cd /root/IMSI-catcher


if [ $# -gt 1 ]
then
        xterm -e "bash -c 'python /root/grgsm_livemon.py'" &
        python3 simple_IMSI-catcher.py -s -w /root/cell_info.db -l $1 $2

else
        xterm -e "bash -c 'python /root/grgsm_livemon.py'" &
        python3 simple_IMSI-catcher.py -s -w /root/cell_info.db -l 39.36044374110071 22.949124812591084

fi

cd /root/node_modules/bscoords/test/

node test.js

cd

python3 check_in.py
```

Figure 13: wrappper_part2.sh script.

RESULTS OF THE PROJECT

In this section we present the results of both part1 and part2 as shown in the figures below.

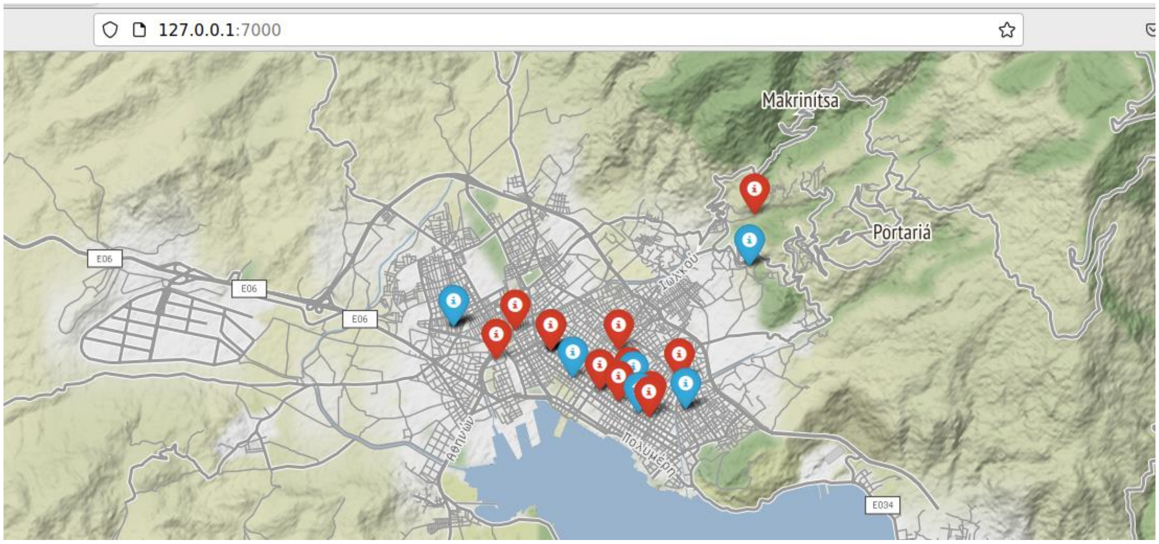


Figure 14: Map with BTS locations as a result of part1's project.

```
root@ubuntu2004:~# python3 check_in.py
       First Check In        |        IMSI        | Cell-ID |  Status   | Counter
---------------------------------------------------------------------------------
2022-02-15 00:12:41.664118 | 202 01 0922638500 |  52008  |  Offline  |    0
2022-02-15 00:12:41.942549 |     0xcb04050f     |  52008  |  Offline  |    0
2022-02-20 18:17:45.192027 | 202 01 0915493630 |  41427  |  Offline  |    0
2022-02-20 18:18:19.138191 | 202 01 0926193754 |  40258  |  Offline  |    0
2022-02-20 18:18:19.251674 | 202 01 0911063644 |  40268  |  Offline  |    0
2022-02-20 18:18:34.166237 | 202 01 0924583711 |  40268  |  Offline  |    0
2022-02-20 18:20:30.991404 | 202 09 1200115407 |  34473  |  Offline  |    0
2022-02-20 18:21:26.449773 | 202 05 2936566550 |  10473  |  Offline  |    0
2022-02-20 18:21:27.497675 | 208 10 3793365024 |  11602  |  Offline  |    0
2022-02-20 18:21:30.386405 | 202 05 2933557923 |  19353  |  Offline  |    0
2022-02-20 18:21:41.429789 | 202 05 2967930138 |  19351  |  Offline  |    0
2022-02-20 18:23:02.169930 | 202 05 2937499762 |  19351  |  Offline  |    0
2022-02-15 00:12:41.942549 | 202 05 2967329171 |  52008  |  Offline  |    0
2022-02-23 21:02:37.161252 | 222 01 3104336236 |  00000  |   Active  |    3
2022-02-23 21:02:46.498127 | 202 01 0922525746 |  40308  |   Active  |    3
2022-02-23 21:02:46.523334 | 202 01 0928011579 |  40308  |   Active  |    3
2022-02-23 21:02:47.340196 | 202 01 0927803952 |  40308  |   Active  |    3
2022-02-23 21:04:56.969903 | 202 10 0242533888 |  34473  |   Active  |    3
2022-02-23 21:06:14.872293 |     0x807bde75     |  11602  |   Active  |    3
2022-02-23 21:06:18.396981 | 202 05 2970338096 |  11602  |   Active  |    3
2022-02-23 21:06:18.560105 | 202 05 2966526303 |  11602  |   Active  |    3
2022-02-23 21:06:28.752449 | 284 01 3160943528 |  19351  |   Active  |    3
2022-02-23 21:14:13.003114 | 206 01 8070166946 |  61575  |   Active  |    3
2022-02-23 21:14:18.956255 |     0xd9308f16     |  61575  |   Active  |    3
```

Figure 15: check_in.db for connection status as a result of part2's project.