4 IMPLEMENTATION

In the following chapter, we detail the steps we took to realize our design, highlighting the problems we encountered.

4.1 FIRMWARE

Our firmware is meant to handle the physical communication between the devices, the extraction of CSI, and the propagation of CSI to user space. We use the *Nexmon Firmware Patching Framework* [19] to inject patches, written in C, into the firmware of Nexus 5 smartphones. More precisely, we use the CSI extractor patch presented in [20] as starting point and modify it to meet our requirements. In particular, we create or make changes in the following files:

IOCTL_5XX.C This file serves as a template for inserting user-specific commands, so-called *ioctls*, into device's firmware. Once such a command is defined, the user can invoke it from a shell using the *nexutil* command line tool.
In this file, we realize the routine for the frame transmission. For this purpose, we first define the frame we want to transmit:

Listing 4.1: Frame structure

```c
struct mpdu_ac qosdata = {
        .frame_control  = 0x0388,
        .duration       = 0x0000,    // set elsewhere to 60ms
        .address1       = { 0xaa, 0xaa, 0x00, 0x0d, 0x0i, 0xii },
        .address2       = { 0x01, 0x00, 0x01, 0x00, 0x01, 0x00 },
        .address3       = { 0x12, 0x34, 0x56, 0x65, 0x43, 0x21 },
        .sequence_control = 0x0010,
        .address4       = { 0x01, 0x00, 0x01, 0x00, 0x01, 0x00 },
        .qos_control    = 0x0007,
        .frame_body     = { 0xa0, 0x39, 0xf7, 'N', 'E', 'X', 'T',
         'O', 'Y', 'O', 'U', 'N', 'E', 'X', 'T', 'O', 'Y', 'O',
         'U', 'N', 'E', 'X', 'T', 'O', 'Y', 'O', 'U', 'N', 'E',
         'X', 'T', 'O' },
        .fcs            = 0x00000000,    // calculated & set
            elsewhere
};
```

We decide to use *Quality of Service (QoS) Data frames* (indicated by `0x88` in the frame control field) because a corresponding *struct mpdu_ac* is already implemented. The *address1 field*, which usually contains the MAC address of the receiver, is particularly important for our application. We use the field's first two bytes `0xaaaa` as a decision pattern,

i.e., a *Collector* should extract CSI from a received frame only if its *address1* starts with these two bytes. While the first two bytes of this field are consequently the same for all our devices, the last three bytes always differ. Each device we use has a unique three-digit inventory number, which is attached to the back of its case. The last two bytes of *address1* contain this inventory number, the corresponding positions in Listing 4.1 are marked with 'i' placeholders. More precisely, we store the two least significant digits of the decimal inventory number in the last byte. The remaining digit is stored in the lower half of the penultimate byte. Correspondingly, if we used a device with inventory number 312, the last two bytes would be 0x030c.
Moreover, we assign consecutive *device numbers* within our experiments to improve clarity. We store the device number at the position marked with the 'd' placeholder in Listing 4.1. The remaining fields of the frame are not important to our application and therefore filled with a dummy content.
In order to be able to send the above frame from an application or shell, we implement *ioctls* for sending over a 20 MHz wide channel and for sending over an 80 MHz wide channel. The sending procedure itself is the same in both of them, except for the encoding of the frame. The corresponding code snippet is shown in Listing 4.2.

Listing 4.2: Send single frame as broadcast

```
// set the retransmission settings
set_intioctl(wlc, WLC_SET_LRL, 1);
set_intioctl(wlc, WLC_SET_SRL, 1);
// deactivate minimum power consumption
set_mpc(wlc, 0);
// pull to have space for d11txhdrs
skb_pull(p, 202);
memcpy(p->data, &qosdata, sizeof(qosdata));
// use with 20 MHz wide channel:
// sendframe(wlc, p, 0, RATES_OVERRIDE_MODE | RATES_ENCODE_HT |
    RATES_BW_20MHZ | RATES_HT_MCS(0));
// use with 80 MHz wide channel:
// sendframe(wlc, p, 0, RATES_OVERRIDE_MODE | RATES_ENCODE_VHT |
    RATES_BW_80MHZ | RATES_VHT_MCS(0) | RATES_VHT_NSS(1));
set_mpc(wlc, 1);
ret = IOCTL_SUCCESS;
```

In our experiments, we do not want to acknowledge receipt of messages. Hence, the *set_intioctl* instructions prevent the device from retransmitting a frame if no corresponding acknowledgment is received. The next instruction *set_mpc* prevents the device to go into a power saving state. After reserving some space for a header which is added later, we finally transmit our QoS Data frame using *sendframe*. In the real application, only one of the two *sendframe* instructions from Listing 4.2 is uncommented. In the case that we currently conduct experiments in the 2.4 GHz band, we use the first of both instructions, i.e., the frame is transmitted using a 20 MHz wide channel and with *Modulation and Coding Scheme (MCS) 0*. This configuration implies that our data is modulated using *Bipolar Phase Shift Keying (BPSK)* and a code rate $R = \frac{1}{2}$, resulting in a maximum data rate of $6.5 \frac{\text{Mbit}}{\text{s}}$ [9]. Accordingly, the second instruction is applied in our 5 GHz experiments. Here, we use a bandwidth of 80 MHz, MCS 0 (i.e., BPSK and $R = \frac{1}{2}$ again) and only one spatial stream, resulting in a maximum data rate of $29.3 \frac{\text{Mbit}}{\text{s}}$ [10].

UCODE-GRINGO-FILTER.ASM The *ucode* is the firmware of a core, called *D11*, that decides whether a received frame should be dropped or further processed. In the case that a received frame is not dropped, the D11 core extracts CSI and sends it to the device's ARM core subsequently. As mentioned in the previous paragraph, we want to extract CSI only from frames that are sent by our application. Thus, the filter needs to be implemented here, recognizing these frames by the first two bytes of the receiver MAC address.
With the *ucode-gringo.asm*, a D11 firmware already exists that is optimized for CSI extraction. Moreover, it includes code samples of how to filter certain MAC addresses, and how to copy a MAC address to the *d11rxhdr* of the frame which saves this address for the further processing steps. We adapt these code samples in such a way that the ucode drops a frame if the first two bytes of the first MAC address are not equal to `0xaaaa`, and copy the first MAC address to the source MAC field of the d11rxhdr. Afterwards, we save the modified file as *ucode-gringo-filter.asm*.

CSI_EXTRACTION.C Once the CSI was received by the ARM core, it is stored in a UDP datagram and sent to the user space. The file *csi_extraction.c* contains the corresponding code. By default, the UDP datagram is created with a set of constant headers (Ethernet, IP, UDP). In *csi_extraction.c*, the function *prepend_ethernet_ipv4_udp_header* from *udptunnel.c* is used for this purpose, which copies information from a predefined *struct ethernet_ip_udp_header* and calculates the IPv4 header checksum. We also want the information currently stored in the source MAC field of the frame from the D11 core to be passed to the user space. All in all, we have to store three bytes of information, i.e., the device number (one byte) and the inventory number (two bytes). We decide to use the source IP field of the IP header for this purpose. Hence, we create a variable *header* of *struct ethernet_ip_udp_header* with the same content as the original variable. Just before a UDP datagram is sent to the user space, we copy the upper four bytes of the D11 frame's source MAC address into *header's* source IP field. However, the function used before to calculate the IPv4 header checksum is not valid anymore, due to the modified header. Listing 4.3 shows the original *calc_checksum* function.

Listing 4.3: Original calc_checksum function

```
/**
 * Calculates the IPv4 header checksum given the total IPv4
     packet length.
 *
 * This checksum is specific to the packet format above. This is
     not a full
 * implementation of the checksum algorithm. Instead, as much as
     possible is
 * precalculated to reduce the amount of computation needed. This
      calculation
 * is accurate for total lengths up to 42457.
 */
static inline uint16_t
calc_checksum(uint16_t total_len)
{
    return ~(23078 + total_len);
}
```

The precalculated value, i.e., 23078, is dependent on the original content of the header. Hence, we must recalculate the checksum. We define a new *calc_checksum* function for this purpose (Listing 4.4). According to [16], the IPv4 checksum is the 16-bit one's complement of the one's complement sum of all 16-bit words in the header. If we assume that the precalculated value of the original *calc_checksum* function is correct, we only have to subtract the share of the former source IP field and add the sum of the 16-bit words of the new source IP subsequently. The source IP of the original header is *10.10.10.10*, which is equal to `0x0a0a0a0a` in hexadecimal. Thus, we obtain $23078 - 2 * 2570 = 17938$ as a precalculated value without the source IP field. Using this value, we define a new *calc_checksum* function, enabling us to calculate the correct IPv4 checksum when device number and inventory number were copied to the header.

Listing 4.4: Modified calc_checksum function

```
uint16_t
calc_checksum(uint16_t total_len)
{
    return ~(17938 + header.ip.src_ip.array[0] * 256 + header.ip.
        src_ip.array[1] + header.ip.src_ip.array[2] * 256 +
        header.ip.src_ip.array[3] + total_len);
}
```

4.2 COLLECTOR APPLICATION

In order to automate the CSI collection process, we implement an Android application that handles the communication with the firmware. In particular, an effortless start of both CSI extraction and frame transmission should be possible. We design separate user interfaces for both tasks, they are shown in Figure 4.1. Moreover, we implement two Android activities *MainActivity* and *SenderActivity* containing the logic necessary for our purpose.

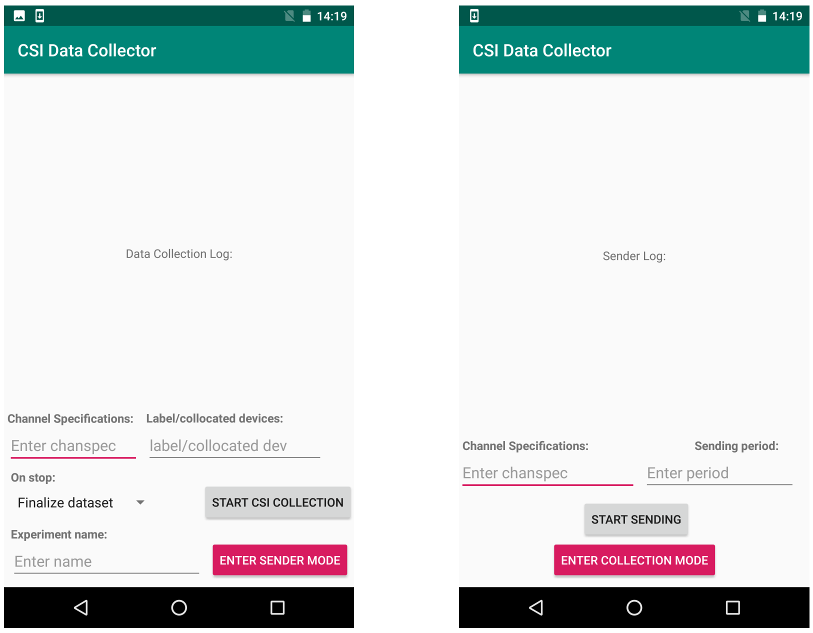

Figure 4.1: Layout of *MainActivity* (left) and *SenderActivity* (right)

4.2.1 *CSI extraction and storage*

By default, the behavior of an Android application after it has been started is defined in its *MainActivity* class. In the context of our collector application, the *MainActivity* contains the implementation of CSI extraction and storage. Once the application has been started, the user is presented with the screen shown in the left part of Figure 4.1. It provides three input fields for the specification of important parameters. The input field with caption *Channel Specifications* is used to determine the Wi-Fi channel and bandwidth in accordance with the format *channel/bandwidth*. For instance, we would have to enter '1/20' here, if we wanted to utilize Wi-Fi channel 1 with a bandwidth of 20 MHz.

Initially, the collector application only facilitates the recording of CSI with an identic label at a certain point of time. The current label ('1' if colocated devices are currently sending, '0' elsewise) can be specified in the input field with the heading *Label/colocated devices*. When the collection is started, all of the CSI subsequently received is labeled according to the input field's value (*single label collection mode*). However, collecting CSI in this way has disadvantages. The experiments we want to conduct and which we introduce in our study design (Section 3.3) include numerous *CSI aggregation rounds*. By our defini-

tion, a single round consists of CSI collection from both colocated and non-colocated devices. With separated recording sessions for both of them, the original number of rounds doubles. The time required for this would make our planned and extensive experiments infeasible. Therefore, we extend the application's functionality. In our enhanced version of the CSI collector, it is still possible to enter only a single label in the input field with the heading *Label/colocated devices*. In this case, the application behaves as described above. Furthermore, the same input field can be used to specify a list of device numbers (as defined in section 4.1) which should be considered as colocated in the collection. As we include the device number in the frame we send for CSI extraction, the application can automatically recognize the originator and assign the proper label. In the case that more than one device is colocated to the collector, their device numbers are separated by colons in the input field (e.g., *4:5:12* for devices 4, 5 and 12 being colocated). One limitation to this approach is that the application cannot be used if a device with number one or zero is the only colocated device of a collector because these numbers are reserved for the single label collection mode. We accept this limitation because it does not matter in our study design, and we consider it more valuable to keep an additional collection mode that allows the collection of CSI even if the device numbers are unknown.
We use the last input field (heading *Experiment name*) to assign a unique name to an experiment. A requirement of our design is to store the CSI of different devices in different files. We conduct several experiments and it is likely for convenience that we do not copy the generated CSI files after each and every single experiment. Unique naming helps to recognize related files in the storage of a phone. However, it is not mandatory to enter a name, just as it is not mandatory to enter anything in the other fields. In the case that we pass a name for the experiment, it becomes part of the name of each file that is created while the collection is active. The final name of a file follows our naming scheme, which is *%e_dev%d_invnr%i_csi_%h-%m.txt* for CSI and *%e_dev%d_invnr%i_labels_%h-%m.txt* for the corresponding labels. We store labels and CSI in separate files for the reason that many machine learning algorithms expect separate arrays for data and labels. This is a design decision, however, other options may work just as well.
The placeholders of our naming scheme are representatives of the following:

- **%e:** The experiment name from the corresponding input field
- **%d:** The device number taken from the received frame
- **%i:** The inventory number taken from the received frame
- **%h:** The hour of the day taken from the current timestamp

- **%m:** The minute of the day taken from the current timestamp

In the case that no name is passed for the experiment, the first placeholder is omitted. The naming scheme is consequently *dev%d_invnr%i_csi_%h-%m.txt* for CSI and *dev%d_invnr%i_labels_%h-%m.txt* for the corresponding labels. We add a timestamp to the end of the filename in order to ensure distinguishability in the absence of a unique experiment name.
The last option that can be changed via the user interface is the application's behavior when we interrupt CSI collection. In particular, we are presented with two options. Choosing the *Finalize dataset* option terminates writing to any file created in the course of the current collection round. Subsequently, these files are considered completed and cannot be modified by the application anymore. When CSI collection is restarted afterwards, the application writes CSI and labels to completely new files. In contrast, the *Extend dataset* option retains references to the current files and hence all of them remain writeable. Restarting the collection extends the existing files.
When the collection procedure is finally started by pressing the *Start CSI Collection* button, the application reads out the input fields containing parameters relevant for the tasks handled by the firmware. In the case that an input field is still empty at collection start, we use predefined default values. These are kept in a central instance of a class called *DataHolder*. It contains default values for channel and bandwidth, saved as static and constant attributes. Moreover, the *DataHolder* instance keeps track of the application's status, i.e., whether the app is currently collecting or sending.
Passing parameters, such as channel and bandwidth, to the firmware is handled via the nexutil command line tool. A Java class *Nexutil* already exists that wraps the most important use cases of this tool in more intuitive Java methods. We use this class as an interface for all communication with the firmware. Before we start listening for incoming messages, we pass the channel specifications to the firmware, activate the CSI extraction functionality with the corresponding *ioctl*, and set the firmware in the monitor mode which is necessary for CSI extraction. In order to keep the application receptive to user input, we start an additional thread for networking. When this thread is started, we listen to network address *255.255.255.255, port 5500* which the firmware uses for passing extracted CSI to the user space. As the firmware is configured to extract CSI only from messages sent by our application (indicated by a receiver MAC address starting with 0xaaaa, see section 4.1), all of the datagrams received here can be considered relevant without any additional filtering.
We know from the firmware modification step that inventory number and device number are contained in the datagram's source IP field. Thus, the source IP field uniquely identifies the datagram's *Transmitter* and we can use its String value as a unique key in a map *(source*

IP address, filename), with the *filename* being the name of the file in the *Collector's* storage that stores all CSI from the respective *source IP address*. The value of *filename* is determined by our naming scheme (as described earlier in this paragraph). In the following, this map has two functions: first, it determines whether a message has been received from a certain *Transmitter* before. In the case that our application has already heard of the *Transmitter*, its source IP is contained in the map's key set. Second, if the *Transmitter* is known, the map provides the name of the file in storage to append CSI of the current datagram to. A corresponding map also exists for the label files.
We implement a method *csiDataToSample* to convert the raw CSI from the datagram to a suitable format for the next processing steps. The datagram contains one complex number per subcarrier, stored in four bytes (two bytes each for the real part $\Re$ and the imaginary part $\Im$). In our application, CSI consists of values for magnitude M and phase shift P. They correspond to the modulus φ and argument r of the received complex number $z = r * e^{i\varphi}$:

$$M = r = \sqrt{\Re^2 + \Im^2}$$

$$P = \varphi = atan2(\frac{\Im}{\Re})$$

We store the CSI of a datagram in a single line of the output file, which means that we obtain a maximum of 510 values per line (for 80 MHz bandwidth). Using this format has the advantage that the structure of our files already resembles that of the arrays we will later use for machine learning. Figure 4.2 shows a simplified illustration of a dataset obtained from the CSI collector application. A single line contains the CSI from one datagram, or, in other words, from one observation. The next line or observation is added when the *Collector* receives the next datagram originating from the same *Transmitter*. As shown in Figure 4.2, we allocate phase and magnitude values to separate blocks within one line. We assume that this segmentation will simplify the evaluation later. Moreover, we agree to round each individual value to two decimal places in order to maintain a consistent structure and to improve clarity.
Before we can process the next datagram, we have to assign a label

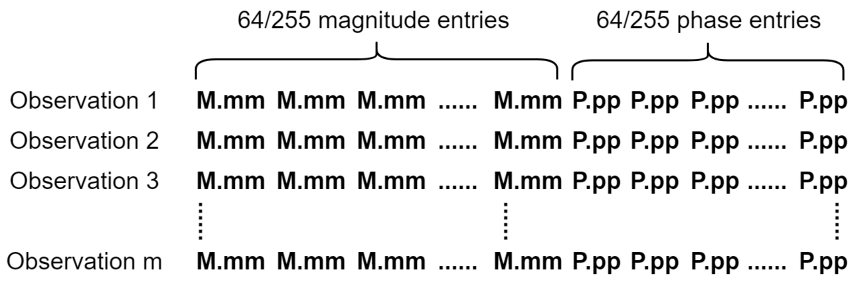

Figure 4.2: Dataset structure

to the CSI just received. For this purpose, if the *Label/colocated devices* input field contains a list of device numbers, we check if the device number from the datagram is among them. If this is the case, we assign the label '1', '0' otherwise. If the input field contains only a single label, this label is assigned. However, in the case that the input field is empty, we assign the default label '1'.
The collection procedure can be stopped with the *Stop CSI Collection* button. When it is pressed, the application evaluates whether the *Finalize dataset* or the *Extend dataset* option was chosen. For *Finalize dataset*, all maps are cleared so that the next collection round will create new files for each source IP. Moreover, we use the *MediaScannerConnection* class to inform the file system about the files we created. Without doing this, the new files appear in the *Collector's* file system only after a restart. In contrast, *Extend dataset* keeps the content of the maps. Hence, a restart of the collection will extend the files of the previous round.

4.2.2 *QoS data frame transmission*

We implement a second Android activity with the name *SenderActivity*. As indicated by the name, this class unites methods that allow an automatic and periodic sending of our QoS data frame. The right side of Figure 4.1 shows the corresponding user interface. It opens when the user presses the *Enter Sender Mode* button in the *MainActivity*. Just like in the user interface of the collection mode, the user may enter the Wi-Fi channel and bandwidth in the input field with caption *Channel Specifications* (format *channel/bandwidth* again). A second input field (*Sending period*) can be used to pass a time in milliseconds which is then used to define a delay between two consecutive transmissions. We will discuss the rationale behind this delay later.
Pressing the *Start Sending* button starts the periodic transmission of the QoS data frame which we examine in Section 4.1. For this purpose, we use the *Nexutil* instance again to pass the channel specifications to our firmware. Afterwards, we start a new thread. It invokes a method *sendPeriodically* which sends *ioctls* to the firmware in dependence of the bandwidth under usage. This distinction is necessary because, at the firmware level, we use a different encoding for 20 and 80 MHz bandwidth. Further details can be found in Section 4.1 and Listing 4.2. In *sendPeriodically*, we schedule the following transmission only after a delay, which is 100 ms by default. The rationale behind this is that without any delay, the transmission rate decreases after some time of sending. We observe a similar behavior when we log a message to appear on the user interface after every transmission. Therefore, we only log the transmission of the first 50 messages to facilitate debugging. Afterwards, log entries are generated in steps of 500 more transmissions. Using this configuration, we observe a rate of about

Dev.Nr.	Inv.Nr.	Dev.Nr.	Inv.Nr.
1	182	2	184
3	244	4	008
5	302	6	175
7	309	8	311
9	186	10	181
11	185	12	180
13	241	-	-

Table 4.1: Inventory number assignment

three transmissions per second, which remains constant for hours of sending. As this rate is monitored using the default delay of 100 ms, we can infer that the code execution itself already causes a measurable delay of about 230 ms per transmission.

4.3 EXPERIMENT DOCUMENTATION

In this section, we document the procedure of our CSI collection. To this end, we examine each scenario individually, highlighting specifics and problems we encounter in the course of our experiments.
Before the experiments can be conducted, we must prepare the devices we want to use as *Transmitters* and *Collectors*. In particular, we install the modified firmware and the collector application. In the course of the firmware installation, we additionally have to insert the individual device number of each device and assign a unique device number to identify the device within the experiments. Table 4.1 shows our assignment of device numbers to the individual devices represented by their inventory number. All in all, thirteen *Nexus 5* smartphones are available. Hence, we assign consecutive device numbers from one to 13. The assignment shown in Table 4.1 remains valid during all of our experiments.
Before we start the actual experiments, we conduct some test runs to evaluate the reliability of our device set. Unfortunately, it turns out that in sender mode, device 3 (inventory number 244) collapses after sending approximately 2500 frames. This means that the phone can transmit reliably for less than 15 minutes. Since the *CSI aggregation rounds* of most of our experiments have a longer duration, and since we need 12 devices at the most in a single scenario, we decide to exclude device 3 and use it as a backup phone only.