Information Security

Signal Intrusion Detection for Remote Keyless Entry Systems

Diplomarbeit

zur Erlangung des akademischen Grades

Master of Science (MSc)

eingereicht von

Simon D. Hasler

is161514

im Rahmen des
Studienganges Information Security an der Fachhochschule St. Pölten

Betreuung
Betreuer/in: Dipl.-Ing. Dr. Henri Ruotsalainen

St. Pölten, May 30, 2018 ______________________ ______________________

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

*

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

Ort, Datum *Unterschrift*

Acknowledgments

While it was my task to plan and conduct this research project, I could never have achieved the final result without the support and encouragement of a number of people inside and outside of academia, some of whom I would like to show my appreciation by making mention of their names.
First, I would like to give my thanks to Dipl.-Ing. Dr. Henri Ruotsalainen, my project supervisor, for always helping me out with input that would steer me in the right direction whenever I seemed to get stuck on some technical detail during my experiments. Also for his patience, the well-working communication over the entire phase of this research project, and for lending me a piece of equipment that I could not have done without.
I would also like to thank Dipl.-Ing. Christian Hinterdorfer, who by the time of this writing is my superior and the head of the IT department of the company I am working for, for giving his permission to try out my completed proof-of-concept on one of the company-owned nVidia Jetson TX2 modules, which I felt was adding great value to my diploma thesis.
As the successful completion of this research work marks the end of my IT Security and Information Security studies, this seems like the perfect opportunity to express my gratitude to my parents, Dipl.-Päd. Elisabeth and Dipl.-Ing. Wolfgang Hasler, my family, and everybody else who contributed in getting me to this point. It has been a tough road at times but the countless moments of joy, surprise, and success easily made up for all the difficulties experienced, and all the fond memories of the great people I had the pleasure of meeting during my university years in St. Pölten will always remain.

Abstract

Recent years have brought to light a surprising number of hacking techniques that circumvent the security measures implemented in automobiles, allowing car thieves to remotely open vehicles without the use of the legit keyfob. White-hat hackers and security researchers have revealed how these kinds of attacks are possible and what kind of hardware and software is used. In this research work, I review the RollJam attack, which aims at replaying captured signals after preventing the car from receiving them by jamming the receiver frequency during legit transmissions. I show that this attack scheme can be reproduced to remotely unlock a 2008 model VW Group vehicle with a selection of low-cost transmitter devices and open-source software. After visualizing captured signals from different transmitters and analyzing their unique characteristics, I proceed by demonstrating that a number of features can be extracted that allow to distinguish between signals based on their origin. Based on my findings, I present a technique that applies two different machine learning algorithms for the classification of data points on a pre-built dataset, and subsequently use it to create a proof-of-concept for a Signal Intrusion Detection System capable of classifying unknown signals based on known signal data. I show how both machine learning algorithms perform in various use cases on the provided signal data in terms of resource utilization and accuracy, and reveal where their individual strength and weaknesses lie. Lastly, I introduce the nVidia Jetson TX2 module that I chose as the hardware platform for the tested proof-of-concept and explain why it is especially well-suited for AI computing tasks in embedded environments such as automobiles.

Contents

1. Introduction

The more cars become computerized and rely on electronical rather than mechanical security measures to prevent theft, the more it becomes obvious how incapable or neglecting car manufacturers seem to have been for many years of implementing strong and robust cryptography into their products. Research publications of recent years and the work of hackers and hobbyists alike have revealed a number of astonishingly simple hacks that circumvent whatever cryptography is implemented in vehicles altogether.
In 2016, at the 25th USENIX Security Symposium, Garcia et al. presented a *wireless attack* [1] on the remote keyless entry (RKE) system of Volkswagen Group cars that affects approximately a *100 million vehicles* built since 1995 worldwide. While this attack is based on *reverse engineering* and the recovery of the used cryptographic algorithms, the paper subsequently describes another attack that focuses on the *HITAG 2 rolling code scheme*, showing that only a few minutes of computation on a laptop are sufficient to create a *fully functional clone* of the remote control of the car. This second attack affects another couple of million cars by manufacturers such as Alfa Romeo, Citroën, Fiat, Ford, Mitsubishi, Nissan, Opel, Peugeot and Renault. NXP, the semiconductor company that sells the chips that use this vulnerable, out-of-date crypto system states that it has been recommending customers upgrade to newer schemes for years [2].
In [3], which was published in 2012, Verdult et al. present three practical attacks on the HITAG 2 transponder used in *electronic vehicle immobilizer systems*, which are supposed to prevent the engine from starting without the remote control containing the transponder chip being present and in close proximity to the reader unit that is usually placed somewhere in the dashboard of the car. The security researchers describe how they were able to *start the engine* of more then 20 vehicles of different makes and models by *recovering the used secret key* using only wireless communication.
In 2011, Francillon et al. have been able to use relay attacks on passive keyless entry and start systems (PKES) [4] to successfully *transmit data between a car and the belonging keyfob* over physical distances as far as 50 meters, non line-of sight. This allowed them to *open and start the attacked vehicle* simply by placing one malicious device in close proximity to the car, and another close to the keyfob. According to their findings, 10 recent car models of 8 manufacturers are vulnerable to certain variations of replay attacks due to the car only verifying whether it can communicate with the keyfob, not if it actually is in

its physical proximity.

Not only are the computational efforts and the execution time of these attacks considerably lower than what might be expected, there is also another, even far more simple way of opening a car without being in possession of the belonging keyfob, requiring no expensive equipment, no knowledge about cryptography at all, and only a minimal technical understanding of digital signals. At DEF CON 23, which took place in Las Vegas in 2015, hacker and hobbyist Samy Kamkar presented a new attack scheme that he named *RollJam* [5, 6]. It basically allowed him to *capture the unlock signal* sent by a keyfob while at the same time *jamming the frequency* so that the car would not receive and, after unlocking, invalidate the signal, thus making it possible to later *resend* it and successfully open the car [7]. On CBC News, Kamkar demonstrated this attack on a Cadillac luxury SUV [8] using nothing more than a small self-made device that was powered only by a low-cost Teensy 3.1 micro-controller.

In this research work I demonstrate that it is both feasible and practical to reproduce the RollJam attack by utilizing easily purchasable low-cost hardware. In Section 4.1 I further analyze the characteristics exhibited by captured signals of various origins and explain how carefully selected features can be extracted to numerically represent the individual signals based on previously obtained reference data. In Section 4.2 I present a method for building a dataset by accumulating a sufficiently large number of captured signals, and continue by showing how the *k*-nearest Neighbors and the Support Vector Classifier machine learning algorithms can be applied to that dataset. A complete and tested proof-of-concept for a Signal Intrusion Detection System capable of determining whether a received signal originated from the legit keyfob is presented in Section 4.3. Section 5.1 and Section 5.2 explain the findings made during my machine learning tests and also describe various limitations to this method. Section 6.1 summarizes the methodology applied in this research work and the results obtained, and Section 6.2 concludes this research work by proposing an entirely different yet equally possible machine learning approach to the same research problem, as well as the option to utilize the CUDA capability of the hardware platform I selected for significant performance gains to my approach in future work.

2. Research Questions

Considering the wide array of radio based attacks against automobiles, I want to demonstrate that it is in fact possible to implement an intrusion detection system capable of recognizing malicious signal sources by detecting differences in signals received on the frequency of the legit keyfob. To this end the following questions need to be answered:

a) Is it at all possible to find differences in captured and replayed signals depending on which device is used as a transmitter, and if so, what characteristics do these differences exhibit?

b) What could a proof-of-concept for a Signal Intrusion Detection System (Signal IDS) look like in theory, which hardware and software could be used, and how could it actually be implemented?

c) To what degree is it possible to automate such a Signal IDS so that it is not overly error-prone due to heavy user interaction?

3. Background

This chapter introduces the concept of rolling codes in remote keyless entry systems and presents a capture and replay attack aimed at circumventing said concept. Furthermore, a selection of receiving and transmitting devices is introduced, all of which have been evaluated for the purpose of this research work. For each device a brief technical overview is given, as well as a description on how it may be used in malicious activities regarding signal capture and replay.

3.1. Rolling Codes and the RollJam Attack in Theory

3.1.1. KeeLoq Rolling Codes

Usually, modern vehicles have a *rolling code scheme* implemented in their keyless entry systems, which in most cases is based on the *KeeLoq*[1] algorithm from `MICROCHIP` and is supposed to prevent *replay attacks*. Such attacks would occur if an attacker could simply *capture* the unlock signal sent by the legit keyfob and *replay* it at any time of their choosing.
Both the transmitting device, *i.e.*, the keyfob, and the receiver inside the car use a *pseudo random number generator* (PRNG) that calculates a 32- or 48-bit *seed value*, which is used as a synchronised starting code [10]. The synchronization process between the keyfob and the car that involves this initial code is also referred to as "pairing". After that, every time the car owner presses the unlock button on the keyfob, the next 16-bit value in the sequence is calculated, taking the result of the previous button press as input. In theory, if the code received from the transmitter, including the calculated value, matches the one pre-calculated by the receiver unit, the car is unlocked and the value is automatically invalidated.
In practice, most rolling code implementations store a list of pre-calculated valid next codes, usually 256 in number, and compare received codes against all of them until a match is found. The reason being that otherwise the keyfob and the receiver unit would be out of sync as soon as even a single transmitted code

[1] It should be noted that unlike older versions of KeeLoq, which are still widely used in cars manufactored in the past 20 years, recent implementations include a timer driven message counter that continuously increments after a short amount of time, thus providing a security measure specifically against replay attacks [9]. This causes rolling codes to become invalid long before an attacker could attempt to use them.

is missed for any reason. If a match is found at the n^{th} position in the list, the sequence is updated to range from $n + 1$ to $256 + n$, thus always giving the car owner 256 chances to successfully transmit a signal to their car and unlock it.

3.1.2. The RollJam Attack

The wireless attacking scheme that hacker and hobbyist Samy Kamkar named "RollJam" when he presented it at DEF CON 23 in Las Vegas in 2015 [5] is able to circumvent the security measures implemented in rolling code schemes such as older implementations of KeeLoq by preventing the car from receiving unlock signals while at the same time saving them for later use.

What is important to know is that the receiver unit inside a car does not just listen for signals on an exact frequency but rather inside a *receive window*, which, in essence, is a small frequency range with the programmed frequency—for my Škoda Fabia II 2008 model that is 434420000 Hz—at its center, as can be seen on the spectrogram in Figure 3.1. This allows to *jam the receiver window of the car*, thus preventing it from receiving and invalidating unlock signals without overlapping the keyfob signal data with jamming data. If at the same time the attacker is able to *capture and save* a keyfob signal, they can then easily *replay* it to the car later and the receiver unit won't invalidate the signal as it has not seen it before.

This process becomes a bit more sophisticated if a victim really wants to unlock their car and, after one failed attempt due to the jamming, presses the unlock button on the keyfob again, thus producing another rolling code that invalidates the previous one. To still be able to perform a successful attack, RollJam therefore captures the keys of both unlock attempts that occur during the jamming, then stops jamming and replays the first captured key. Since the frequency window now is not jammed anymore, the receiver unit is able to "see" the signal, invalidates it, and unlocks the car, leaving the victim none the wiser. However, as the attacker is in possession of another key that includes a later calculated value in the rolling code sequence, they are still able to unlock the car at a later time.

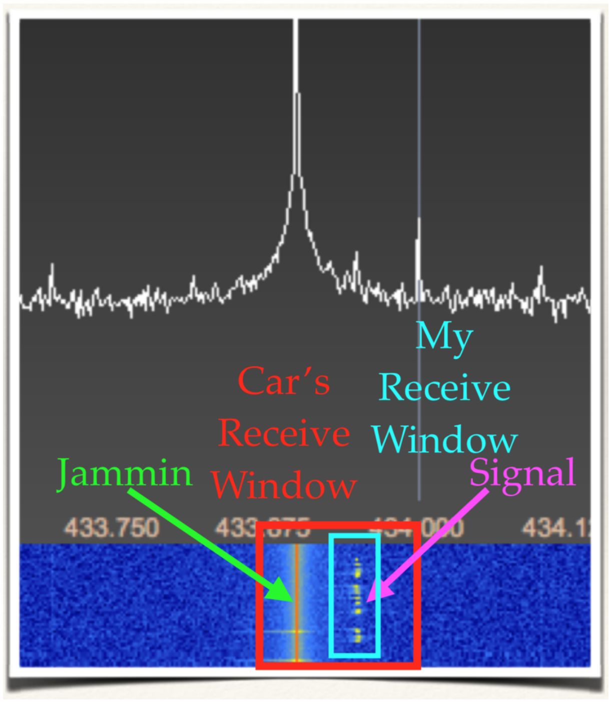


Figure 3.1.: RollJam attack inside the receive window of the car, as observed in the spectrogram [5].

Considering that there are still millions of cars on the road that have old, insecure versions of KeeLoq rolling code schemes implemented and are thus susceptible to RollJam, this attack technique serves as motivation and as technical basis for this work.

3.2. Signal Capture and Replay in Practice

Unlike micro-processor manufacturers and cryptographers who could swap out old microchips in their hardware against newer, better ones and implement stronger, more secure cryptographic algorithms, respectively, I show that even with low-cost hardware such as RTL-SDRs and transceivers based on the Texas Instruments CC1111 chipset it is possible to implement a system capable of detecting attacks like RollJam by recognizing different signal origins. On the software-side I use freeware like the Python-based GNU Radio, the rich Python scientific ecosystem, including libraries such as NumPy and Pandas and the SciPy and Scikit-learn frameworks for machine learning, as well as the powerful MATLAB and Simulink platform together with the Communications System Toolbox and the RTL-SDR support package, which I was lucky enough to purchase with a generous student discount.
However, before testing of different approaches on a Signal Intrusion Detection System could be started, first, reproducing the RollJam attack had to be attempted as being able to transmit captured signals myself would be an integral part of this research work. The devices I had available for that purpose are presented

in the following sub-sections.

3.2.1. RTL2832U based SDR Dongle

So-called Software Defined Radios (SDRs) utilize Digital Signal Processing algorithms to implement most of the functionality usually associated with the Physical Network Layer in software. All the hardware needed for a standard SDR receiver is nothing more than a high-speed GHz sampler chip capable of digitizing a broad spectrum of radio frequencies, and an antenna [11]. Most consumer-level SDRs, additionally, come with a small housing that holds the micro-chip and a USB connector so that they can easily be used with a wide variety of computers and notebooks.
The "RTL" in RTL-SDR stems from the `REALTEK` RTL2832u demodulator [12] that is used in many low-cost SDRs. For this work, the NooElec NESDR Nano 2+ RTL-SDR [13] was chosen, which also features a `Rafael Micro` R820T2 second generation tuner with improved sensitivity and selectivity compared to the older R820T model, and boasts an approximate tuning range of 25 MHz to 1.7 GHz.

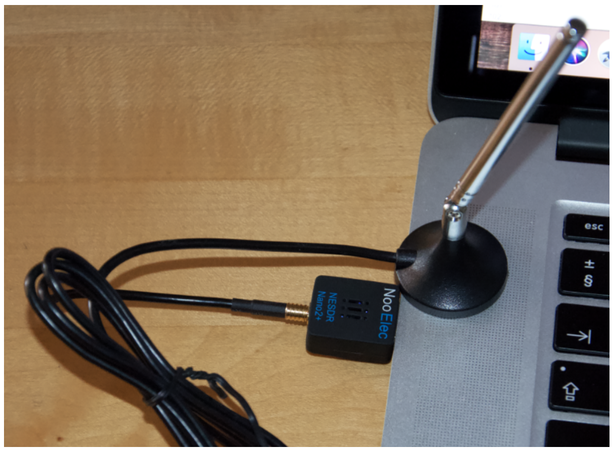
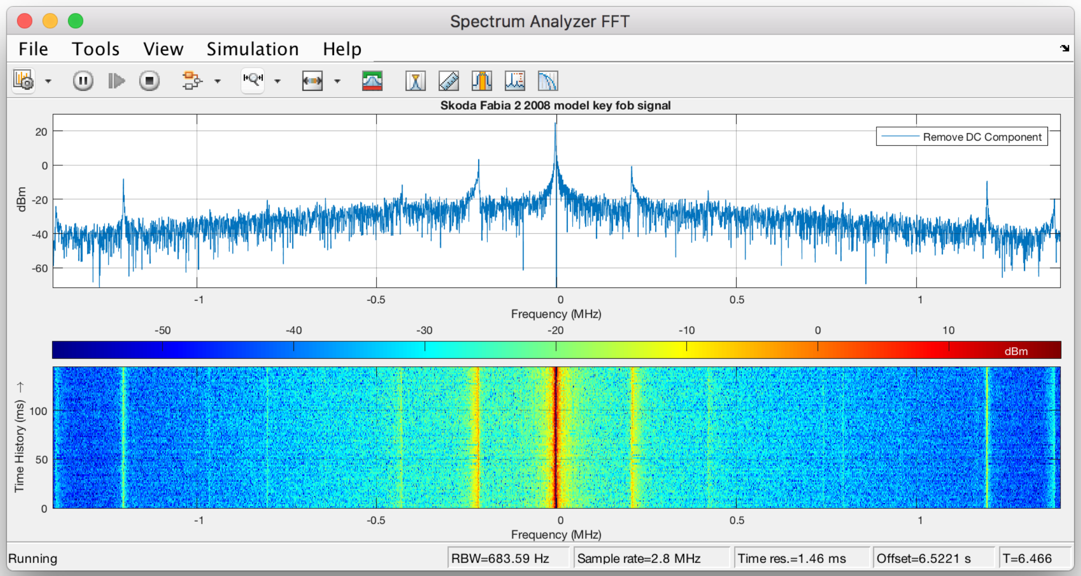


Figure 3.2.: NooElec NESDR Nano 2+ with MCX antenna plugged into a MacBook Pro mid-2014 model.

While it is not possible to retransmit any signals with this simple RTL-SDR device, it is an excellent tool to explore the frequency spectrum, observe what the signal of interest looks like in a waterfall diagram (also referred to as spectrogram), and get an idea about what is possible with SDRs in general. I used this device a lot to improve my understanding of digital signals by tweaking the various parameters in tools such as GQRX, GNU Radio, and MATLAB Simulink.
Figure 3.3 shows the signal of the keyfob from my Škoda Fabia II 2008 model in both the frequency spectrum and the spectrogram. The center frequency is 434420000 Hz, which is quite common for European cars from Volkswagen Group manufactured around that time, the radio frequency (RF) tuner

gain[2] is set to 25 dB, and the "Sample rate"[3] is 2.8 MHz or 2800000 Hz.



Running | RBW=683.59 Hz | Sample rate=2.8 MHz | Time res.=1.46 ms | Offset=6.5221 s | T=6.466

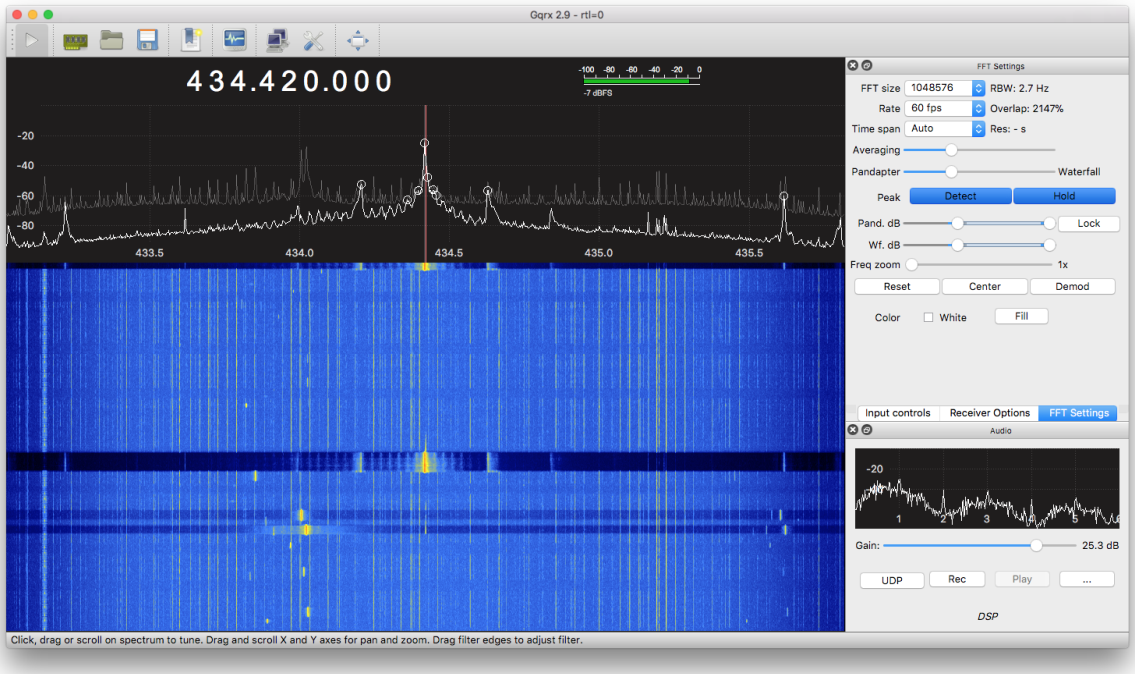
Figure 3.3.: Škoda Fabia II 2008 model keyfob signal at 434.42 MHz shown in the frequency spectrum and in a waterfall diagram created with MATLAB Simulink.

Probably the most interesting aspect to notice in Figure 3.3 is the main spike exactly at the set f_c, as well as the four minor spikes, two of which are occurring with only a minimal offset to either side of f_c while the other two are quite close to $f_c - f_s/2$ and $f_c + f_s/2$.

Observing the keyfob signal in GQRX using the same configuration parameters previously applied to MATLAB Simulink results in quite a similar picture, as can be seen in Figure 3.4. In the waterfall diagram the signal is clearly recognizable as a yellow and red data burst exactly at f_c.

[2]Changing the tuner gain directly affects the quality of signals received by the RTL-SDR device. Increasing the gain will add energy to a signal, thus enhancing its quality, however, it should be noted that this process also affects noise and unwanted signals within an observed bandwidth [11]. Gain is usually measured in decibel (dB) or decibel-milliwatts (dBm), depending on the amount of power.

[3]The sampling rate specifies the bandwidth that can be observed in a spectrum analyzer in f_s MHz. It ranges from $f_c - f_s/2$ to $f_c + f_s/2$ with both ends being equally distant to the center frequency f_c. Sometimes also referred to as sampling frequency, it indicates the average number of samples obtained in 1 second. For a sampling rate of 2.8 MHz that would mean that the SDR device is set to output 2.8 million samples per second. [11]



FFT size 1048576 RBW: 2.7 Hz
Rate 60 fps Overlap: 2147%
Time span Auto Res: - s

Figure 3.4.: Škoda Fabia II 2008 model keyfob signal at 434.42 MHz shown in the frequency spectrum and in a waterfall diagram using GQRX.

The NooElec NESDR Nano 2+ was also used to observe the signals of all the following transmitter devices in the spectrogram and the frequency spectrum.

3.2.2. Rad1o Badge

The Rad1o Badge was created as a giveaway for attendees of the 2015 Chaos Communication Camp [14] and was meant to inspire people to get into Software Defined Radio hacking. The hardware of the badge is based on Michael Ossmann's HackRF One [15], which is a half-duplex transceiver with an operating frequency from 1 MHz to 6 GHz. The advantage of that is that both devices are software-compatible, meaning that the same tools can be used with both of them. However, as some parts of rad1o had to be redesigned around parts donated by some chip vendors, HackRF One and rad1o use different firmwares. Also, the latter is not quite as powerful as a transceiver, operating in a range from 50 MHz to only 4 GHz. An ARM Cortex M4 CPU is used to process the I/Q samples it receives from the Wimax transceiver. What is unique to the rad1o is the Nokia 6100 LCD that displays information about the device status.
"Half-duplex transceiver" means that the device can be used both as a receiver and a transmitter but not simultaneously. Data can either be received or transmitted at a time.

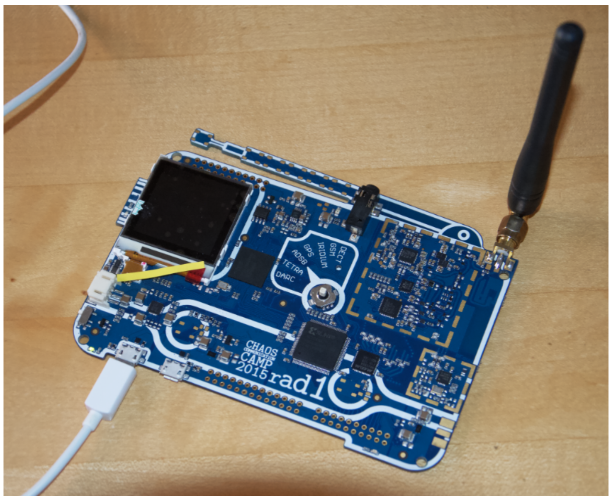
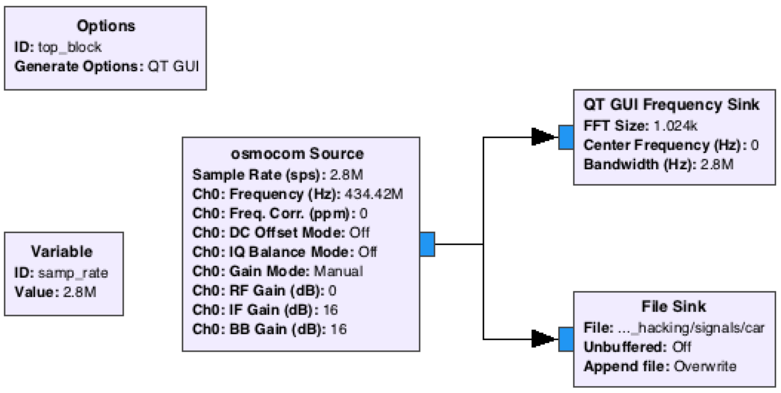


Figure 3.5.: Rad1o Badge that was handed out to attendees of the Chaos Communication Camp 2015.

With the rad1o it was fairly easy to perform a successful capture and replay attack on the test car, *i.e.*, the Škoda Fabia II 2008 model. All that was required was creating two flow graphs in GNU Radio, one for capturing a signal (`repaly_RX.grc`), and another for replaying it (`replay_TX.grc`) [16]. The settings were the same as when using MATLAB Simulink and GQRX with the RTL-SDR: 434.42 MHz as f_c and 2.8 MHz as sampling rate.

To make GNU Radio recognize a connected rad1o or HackRF device and allow it to listen to signals on the frequency spectrum, the osmocom Source block was used. In order to be able to actually see what is happening on the spectrum and to confirm that a signal is received by rad1o, a QT GUI Frequency Sink block that displays the data it receives from the osmocom Source was added to the model, too. Capturing a signal for later replay attacks was done by using the File Sink block that saves a captured signal in the specified output file. The full "receive" model can be seen in Figure 3.6.


Options
ID: top_block
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 2.8M

osmocom Source
Sample Rate (sps): 2.8M
Ch0: Frequency (Hz): 434.42M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 0
Ch0: IF Gain (dB): 16
Ch0: BB Gain (dB): 16

QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 2.8M

File Sink
File: ..._hacking/signals/car
Unbuffered: Off
Append file: Overwrite

Figure 3.6.: GNU Radio flow graph for observing the spectrum at 434.42 MHz and saving a captured signal to a specified output file.

The GNU Radio flow graph for transmitting the previously captured signal, which can be seen in Figure 3.8, is only slightly more sophisticated. This time, a File Source block was used to provide the model with data input, specifically the file that holds the captured signal. Before the signal is passed to the osmocom Sink and thus to the device itself, a Multiply Const block amplifies the signal in the digital domain, increasing its power just a bit.
Visualizing the signal, again, was accomplished by adding a QT GUI Frequency Sink block and a QT GUI Time Sink block. While the former displays signals in the frequency domain, the latter does the same in the time domain, producing an entirely different view of the very same signal. Ultimately, it comes down to the questions "at what frequency is a signal received exactly?", versus "at what times do amplitudes occur in that signal?", each of which allows to draw important conclusions about an observed signal. The difference between frequency domain and time domain is illustrated in Figure 3.7.
Throttling the data throughput simply limits it to the specified sampling rate, ensuring that GNU Radio does not consume all CPU resources when executing the model.

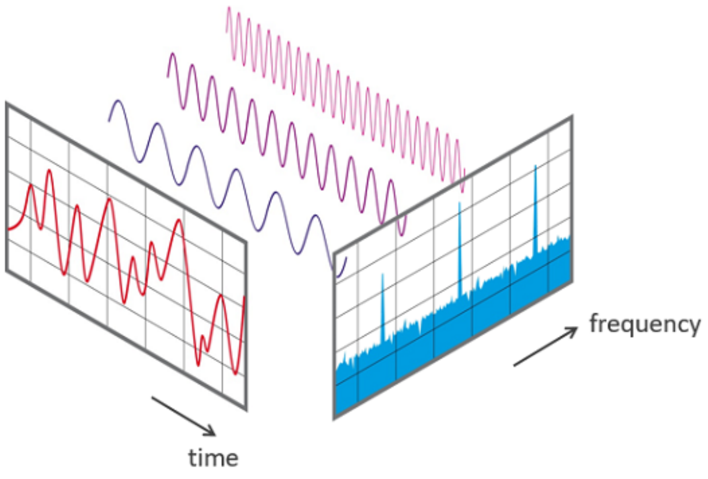
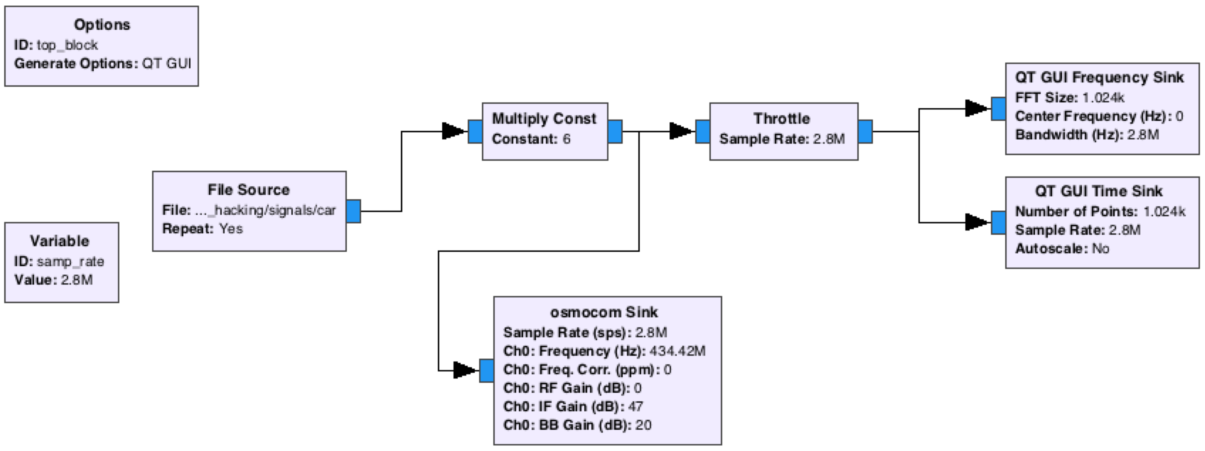


Figure 3.7.: Comparison of a signal observed in the time domain and the frequency domain [17].



Options
ID: top_block
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 2.8M

File Source
File: ..._hacking/signals/car
Repeat: Yes

Multiply Const
Constant: 6

Throttle
Sample Rate: 2.8M

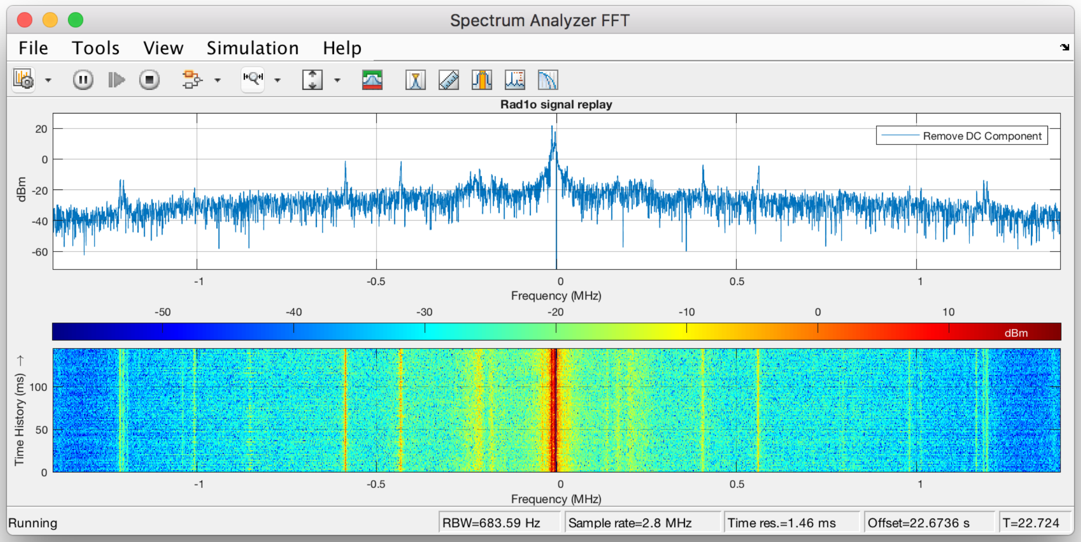
QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 2.8M

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 2.8M
Autoscale: No

osmocom Sink
Sample Rate (sps): 2.8M
Ch0: Frequency (Hz): 434.42M
Ch0: Freq. Corr. (ppm): 0
Ch0: RF Gain (dB): 0
Ch0: IF Gain (dB): 47
Ch0: BB Gain (dB): 20

Figure 3.8.: GNU Radio flow graph for observing the spectrum at 434.42 MHz and replaying a captured signal after retrieving it from a specified input file.

When the flow graph depicted in Figure 3.8 is executed, rad1o transmits the data of the captured signal on the set frequency. If the transmission is observed in the spectrum and the waterfall diagram created by running the MATLAB Simulink model, another signal graph that looks quite similar to the one produced by the keyfob at about f_c can be seen. However, throughout numerous transmissions, rad1o kept producing a different number of minor spikes during signal transmission that could be observed at equal offsets to either side of f_c.

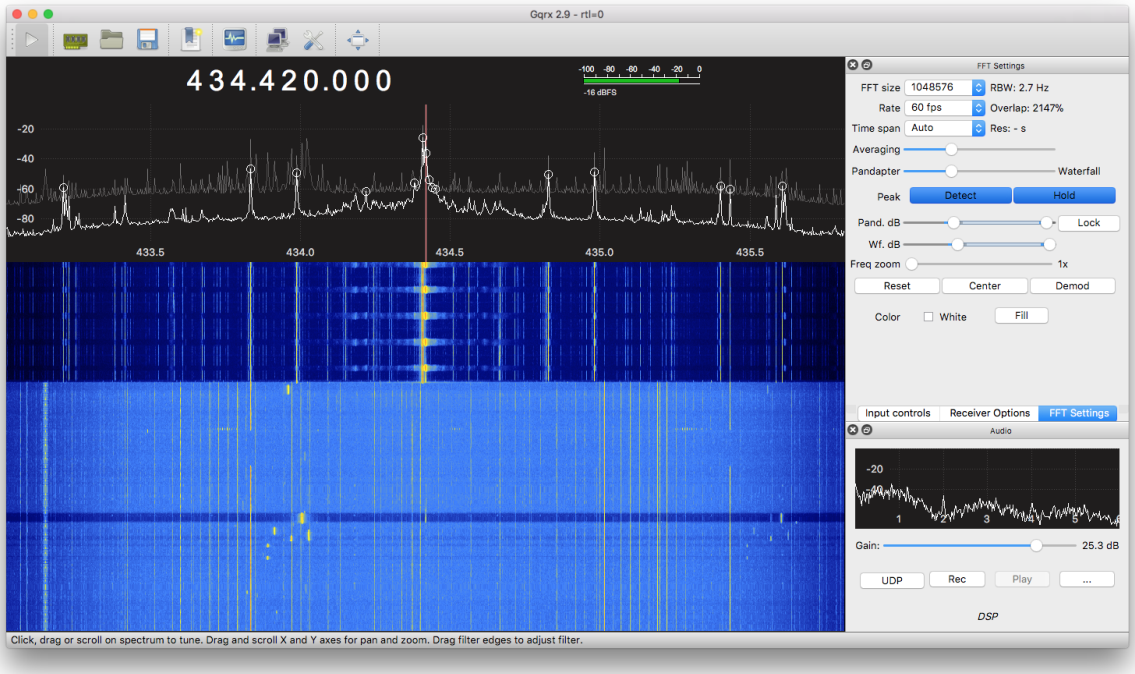


RBW=683.59 Hz | Sample rate=2.8 MHz | Time res.=1.46 ms | Offset=22.6736 s | T=22.724

Figure 3.9.: Signal replay with rad1o at 434.42 MHz shown in the frequency spectrum and in a waterfall diagram created with MATLAB Simulink.

Looking at the signal graph in GQRX verifies the observations made in MATLAB, as can be seen in

Figure 3.10. This is interesting especially when directly comparing the signal graph of each device to that of the others. The continuous appearance of the yellow and red signal in the waterfall diagram hints at the GNU Radio transmit flow graph repeatedly sending the signal at equal time intervals.



FFT size 1048576
RBW: 2.7 Hz
Rate 60 fps
Overlap: 2147%
Time span Auto
Res: - s
Gain: 25.3 dB

Click, drag or scroll on spectrum to tune. Drag and scroll X and Y axes for pan and zoom. Drag filter edges to adjust filter.

Figure 3.10.: Signal replay with rad1o at 434.42 MHz shown in the frequency spectrum and in a waterfall diagram using GQRX.

With rad1o and the two GNU Radio flow graphs I was able to perform a successful capture and replay attack on my Škoda Fabia, opening the car with a MacBook Pro and the connected transceiver device instead of the keyfob. As opposed to the RollJam attack, no jamming of the legit signal originating from the keyfob was performed, which would have prevented the car from receiving (and invalidating) it. Instead, to achieve the same, the signal was simply captured outside of the receiving range of the car. There was no particular reason for doing so other than intentionally keeping the whole setup simple and rather focus on observing what the transmission looks like in the spectrum during the attack. At this point it also became clear that all these wireless experiments with connected transceiver devices draw a lot of power from a notebook and deplete the battery fast if it is not connected to a power source.

3.2.3. YARD Stick One

From a technical perspective, YARD Stick One, which is short for "Yet Another Radio Dongle", is not a SDR, unlike, *e.g.*, the HackRF One. On Great Scott Gadgets, Michael Ossmann’s webstore, it is described as a sub-1 GHz wireless test tool controlled by a computer [18]. The USB dongle with

the yellow PCB is based on a `Texas Instruments` CC1111, a low-power sub-1 GHz system-on-chip (SoC) designed for low-power wireless applications [19]. YARD Stick One, too, is a half-duplex transceiver that supports the following modulation modes: ASK, OOK, GFSK, 2-FSK, 4-FSK, and MSK. Its official operating frequencies are 300-348 MHz, 391-464 MHz, and 782-928 MHz, however, some dongles have been found to also work slightly outside these ranges [20].
What makes YARD Stick One a very powerful tool is the RfCat firmware and client written by GitHub user atlas0fd00m [21], which basically provides interactive Python access to the sub-Ghz frequency spectrum.

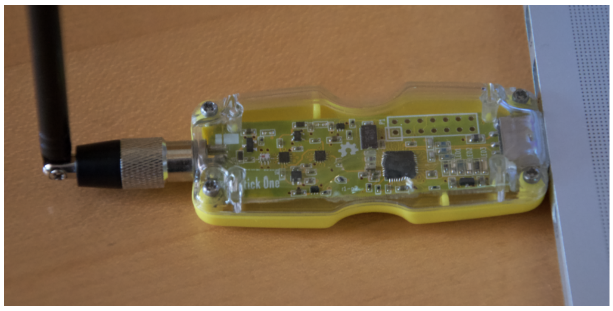


Figure 3.11.: YARD Stick One with enclosure and ANT500 antenna.

As the YARD Stick One is a rather popular device in the hacker community, various Python scripts for capture and replay attacks as well as the actual RollJam attack have already been created (and used) by hackers and hobbyists and can be found on their public GitHub accounts. I tried the ones that looked most promising [22, 23] (although without the jamming part due to having had only one YARD Stick One available to me) and even wrote a Python script myself based on those.
Unfortunately, though, YARD Stick One would neither open my car, nor another even older Škoda Fabia I that was manufactured in 2007. Someone else, however, seems to have been able to open their Toyota RAV4 2011 model using [23] and two YARD Stick One, as they prove by demonstrating the attack in a Youtube video [24].
Their success was the reason for including the YARD Stick One in my experiments as it clearly can be used as a code grabber and replayer in RollJam attacks. Figure 3.12 shows that the signal that was captured and retransmitted with the device does look quite similar to the real keyfob signal, even though it was not accepted by the cars I tried the attack on. In the visualized result of this transmission, numerous small spikes can be seen to both sides of f_c while the main spike and the two spikes closest to it appear to be just the same as in the keyfob signal.

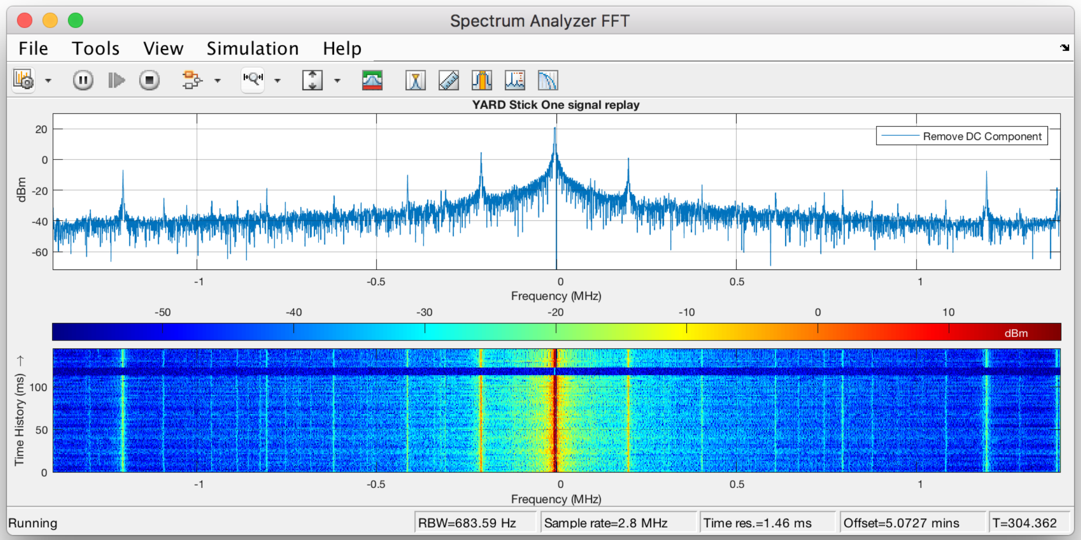


Figure 3.12.: Signal replay with YARD Stick One at 434.42 MHz shown in the frequency spectrum and in a waterfall diagram created with MATLAB Simulink.

The GQRX spectrum viewer with its ability to detect and hold peaks illustrates this even better, as can be seen in Figure 3.13.

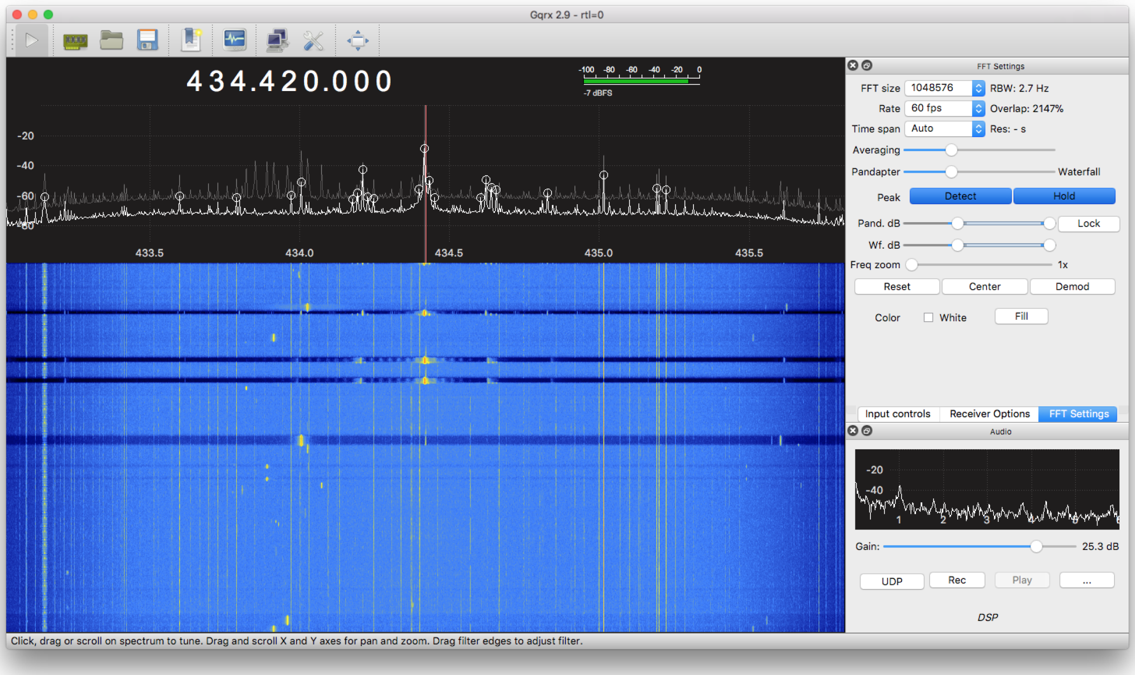


Figure 3.13.: Signal replay with YARD Stick One at 434.42 MHz shown in the frequency spectrum and in a waterfall diagram using GQRX.

3.2.4. Raspberry Pi 3 Model B

Usually, the well-known credit card-sized mini-computer Raspberry Pi 3 [25], which is based on a 1.2 GHz Broadcom BCM2837 64-bit quad-core CPU, is not the first thing that comes to mind when thinking about receiving and transmitting radio signals in the 433 MHz band that European automobiles use in their remote keyless entry systems. However, thanks to the RPiTX library developed by Evariste Courjaud aka F5OEO [26, 27], a Rasperrby Pi can now be turned into a general purpose transmitter for any frequency in the range from 5 KHz to 500 MHz [28]. This is achieved by using square waves to modulate a signal on the GPIO pins of the device. All that is needed is a male to female jumper wire attached to the used GPIO pin where it serves as antenna for the Raspberry Pi. This way, the mini-computer is capable of broadcasting signals using FM, AM, SSB, SSTV, or the FSQ modulation mode.

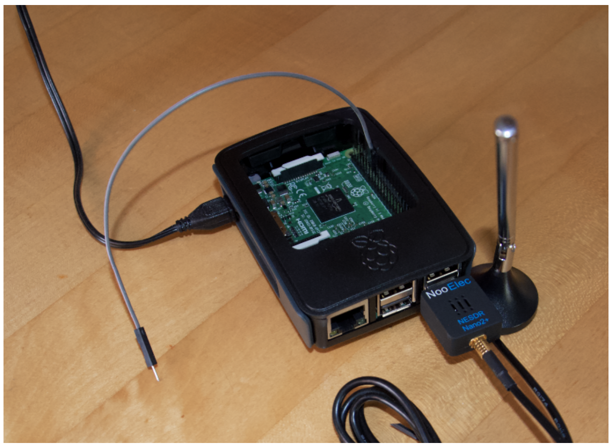


Figure 3.14.: Raspberry Pi 3 Model B with a male-to-female jumper wire connected to GPIO pin 18 as antenna and the NooElec NESDR Nano 2+ plugged into one of its USB ports.

The most interesting approach to use the Raspberry Pi for capture and replay attacks that I have found is the subarufobrob toolset [29] created by Tom Wimmenhove. It consists of several tools such as fobrob, which allows the capture of signals, and rpitxify for converting the signals to a file format compatible to rpitx. Once that is done, the converted signal can be replayed using rpitx, as he describes in the "Operation" section of the repository readme. The toolset works on the Subaru Forester 2009 model, as Wimmenhove demonstrates in a Youtube video [30], and according to him is also likely to work on at least five other Subary models from 2004 to 2011 as those use the same keyfob.
For some reason, fobrob did not capture any signals from my Škoda keyfob, despite the fact that the correct frequency of 434.42 MHz had been specified in the source code, *i.e.*, the fobrob.c file, before building the project. Therefore, the Raspberry Pi could not be used for the experiments in this research

work, however, I did find it worth noting that even with this device people have successfully unlocked cars.

4. Research Methodology

The various sections in this chapter contain detailed explanations of the individual steps required to gain the necessary insight into digital signals and create the foundation upon which it was possible to implement a proof-of-concept for a Signal Intrusion Detection System. Various Python scripts are introduced that allowed to create visual representations of captured signals and extract features from them, as well as accumulate data-points consisting of those features in a dataset and perform machine learning on it. While the complete source code can be found in the Appendix, the most significant parts of each script are presented in this chapter.

4.1. Feature Extraction for Signal Classification

In regards to the first research question, it is clear that the first part of this research work had to be visualizing the signals of all transmitting devices in a way such that it would be possible to compare them and look for characteristics that could be used to distinguish between them. For that purpose, I decided to utilize the Python programming language and some of its powerful libraries such as *NumPy* and *Matplotlib*. The *Seaborn* visualization library was used to enhance the appearance of the signal plots, and custom libraries such as *Pyrtlsdr* and *RfCat* allowed me to interact with the RTL-SDR dongle and the YARD Stick One. The full source code of my Python scripts can be found in the Appendix. I would like to stress, though, that all scripts were written explicitly for this research project and the devices used in it. Using them for other purposes might not yield any meaningful results at all. Great care was, however, taken in writing clearly structured code and using variable names that should make sense to anyone with an understanding of what the scripts aim to accomplish. Thus, it should be fairly easy to adapt them to suite the needs of similar projects.

In this Section I explain what the various parts of the two scripts used to compare signals, *i.e.*, `replay.py` and `plot_diffs.py`, do, what results I got from them, and how those may be interpreted.

4.1.1. Script for Replay Attack: `replay.py`

This Python script was used to capture exactly two signals using the YARD Stick One and then replay them, made possible by the RfCat firmware that every new YARD Stick One comes flashed with, and the RfCat client that can be obtained from [21]. After importing the library with

```python
from rflib import *
```

an `RfCat()` object can be created and configure with all parameters required to receive signals from a specific source, and then later replay them on the same frequency and using the same modulation the original source is using.

In the `main()` function, consecutive raw signals are captured and converted to hexadecimal representation. Once the user decides to switch from receiving to transmitting signals, which they are asked to do after every capture, the signals are first converted to Python bytes containing the raw data and are then transmitted. This happens for as long as the user does not end the script execution, or until all captured signals stored in the `raw_signals` Python list have been transmitted.

4.1.2. Script for Signal Plotting and Feature Extraction: `plot_diffs.py`

After creating the GNU Radio programs depicted in Figure 3.6 and Figure 3.8 and writing the `replay.py` Python script, I finally had the means to capture and replay signals transmitted from the Škoda Fabia II keyfob using both the Rad1o and the YARD Stick One. In order to make the various transmitted signals visually comparable, I decided to write another Python script that would do the following: plot the power spectral density of the first received signal, which always originates from the keyfob as the reference device, do the same for the second received signal, which may originate from either one of the hacking devices, and, lastly, create an overlay plot of both signals. All this is done in one figure using three subplots, to be precise.

Just like in `replay.py`, all the relevant Python libraries are imported, of which *rtlsdr*, *peak* from *peakutils*, *pyplot* from *Matplotlib*, and *Seaborn* are especially noteworthy.

```python
from rtlsdr import *
# ...
import peakutils.peak
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

rtlsdr is used to create a `RtlSdr()` object that serves the same purpose as the `RfCat()` object mentioned in the previous subsection, only that this time the parameters required to make the RTL-SDR dongle listen on the frequency of the keyfob, grab received raw samples and store them in a Python list are provided. From those samples, the corresponding power levels are calculated using the common logarithm (logarithm to base 10). Both these operations take place in an infinite loop that is only exited when two signals meeting a certain requirement to be recognized as such are captured and plotted.
During the observation of the frequency spectrum at 434.42 MHz in GQRX, I noticed that the noise level, *i.e.*, when no distinct signal is present, would always remain below a power level of -10 dB, which led me to the assumption that whenever that specific power level is exceeded, a signal of some sort has to be present. Thus, in each loop iteration the code checks whether the power level calculated from whatever raw data the RTL-SDR reads exceeds the threshold of -10 dB.

```
raw_samples = sdr.read_samples(1024*1024)
raw_power_lvls = 10*log10(var(raw_samples))
# ...

if raw_power_lvls >= -10:
    # ...
```

Once two valid signals have been captured, the first figure consisting of three subplots, as previously mentioned, is created by using *Matplotlib*, which is a Python 2D plotting library. To achieve a better visual appearance of the plots that looks more modern and aesthetic, they are enhanced through the use of *Seaborn*. This library is based on *Matplotlib* and its main purpose is to provide a high-level interface for statistical data visualization.
Two captured signals from the keyfob can be seen in Figure 4.1.

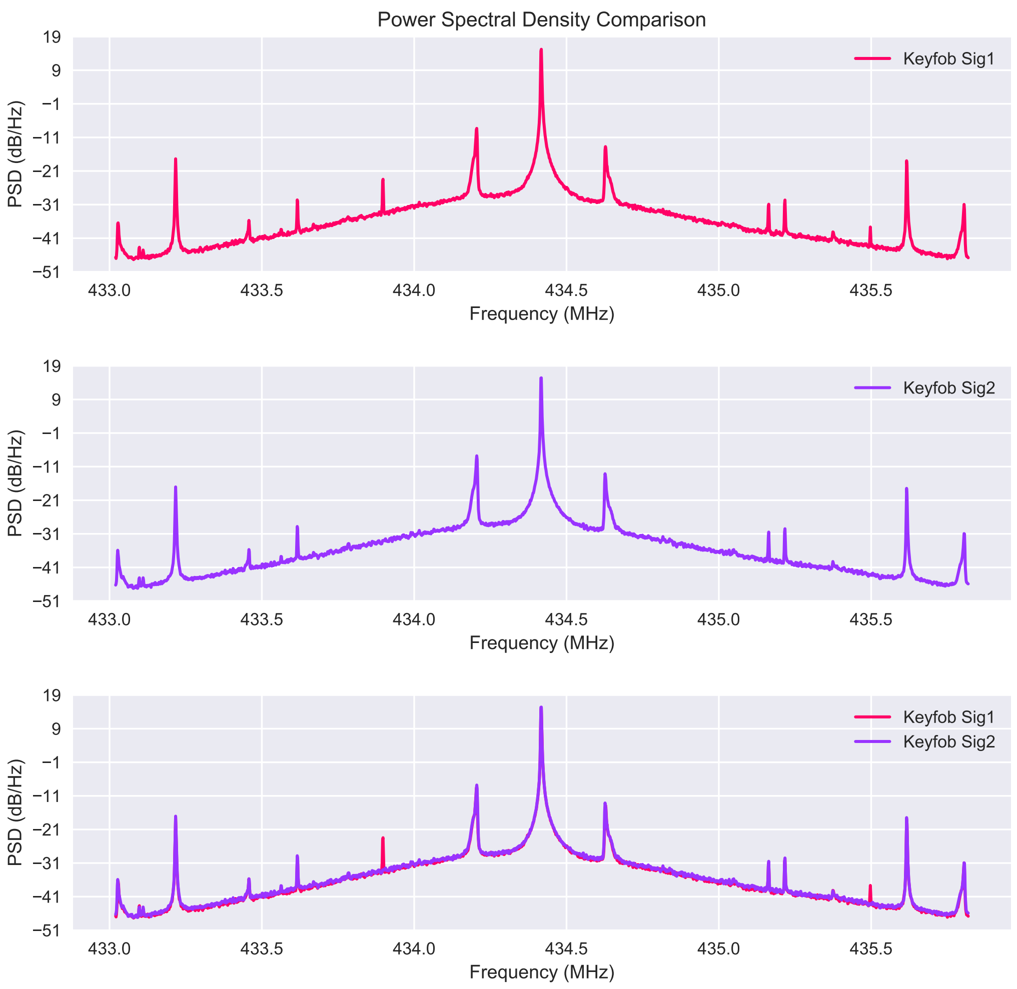


Figure 4.1.: Two signals transmitted by the Škoda Fabia II keyfob.

As expected, the two signals are not absolutely identical but very similar, and if the first signal—plotted in pink—was not marginally stronger than the second in regards to its power level, it would not be possible for the human eye to tell any difference from the overlay plot at all. Both signals feature almost the exact same number of recognizable peaks at seemingly the same positions, and the overall shape appears to be the same as well.

Also, the fact that the keyfob signal spreads over a rather high bandwidth ($f_c - f_s/2$ to $f_c + f_s/2$) was begging for further investigation by "zooming in" on the peaks and checking whether they truly converge on the exact same frequencies. There is already a number of algorithms for detecting peaks in digital signals implemented in Python, which have been listed and described by GitHub user *MonsieurV* on [31]. Of all the various choices, I decided to take a closer look at those two that work in a similar way as the `findpeaks()` function from MATLAB and produce comparable results. Eventually, I chose the `peakutils.peak.indexes()` method as I found it to be the easiest and most adaptable to work with for my specific requirements. The following lines of Python code show how the `plot_diffs.py` script calculates, first, the correct power levels from the ones retrieved by the `plt.psd()` method and, second, uses `peakutils` to get the indices of all the peaks it is able to detect by applying the parameters provided to the `indexes()` method.

```python
Pxx_1, freqs_1 = plt.psd(signals[0], NFFT=2048,
 ↪  Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6,
 ↪  scale_by_freq=True, color=DEF_PINK, label=DEF_SIG_1)

# ...

power_lvls_1 = 10*log10(Pxx_1/(sdr.sample_rate/1e6))+10*log10(8/3)
# ...
indexes_1 = peakutils.indexes(power_lvls_1, thres=0.25, min_dist=10)
```

Dividing the sample rate by 1e6 before dividing the power levels by the sample rate ensured that the result was in MHz rather than Hz. Adding 10 times the common logarithm of $8/3$ was necessary since the `plt.psd()` method does plot the corrected power levels itself but returns the non-corrected ones. The energy correction factor for the signals turned out to be $10 * log10(8/3)$, which equals 4.3 dB. What made the correction necessary in the first place was the window function applied to the fast Fourier transformation of the signal in the power spectral density calculation. By default, the `plt.psd()` method applies the Hanning window, which lowers the overall power level and shapes the signal in a specific way, as the plots show.
While the `indexes()` method performs rather well in detecting peaks and retrieving accurate indices, it can be challenging to choose threshold and distance parameters such that only significant peaks are detected, and that the algorithm does not leave out too many of them due to a too high distance value. Looking for peaks above the lower 25% of the plot that are separated by 10 neighboring peaks worked decently for my signals, although I did experience several executions of my script in which the center peak was left out as it happened to be one of those 10 peaks between two recognized peaks. Rerunning the script several times eventually solved this.
Now, with the indices of all recognized peaks available, the script goes through them and filters out the ones that are above a power level of -20 dB.

```python
power_lvls_1_max = [i for i in power_lvls_1[indexes_1] if i >= -20]
```

```
# ...
check_1 = np.isin(power_lvls_1, power_lvls_1_max)
# ...
indexes_1_max = np.where(check_1)
```

As can be seen in Figures 4.3 to 4.6, signals from different transmitters may exhibit different numbers of recognizable peaks. For obvious reasons, it only makes sense to compare the peaks present in the reference signal with whatever the test signal exhibits at the exact same positions, hence it was necessary to store not only the coordinates of peaks in the test signal but also its power levels at the frequencies of the reference peaks. Since it is rather uncomfortable to work with two separate entities that actually belong together, I utilized the `column_stack()` method from the *NumPy* library to put the corresponding frequencies and power levels together to form actual two-dimensional vectors. The Python list `vectors_sig_1` stores the peak coordinates of the reference signal (S_r), `vectors_sig_2_cmp` does the same for the power levels of the test signal at the identical frequencies, and `vectors_sig_2_real` stores the peak coordinates of the test signal, which is needed for accurate plotting.

```
vectors_sig_1 = np.column_stack((freqs_1[indexes_1_max],
 ↪  power_lvls_1[indexes_1_max]))
vectors_sig_2_cmp = np.column_stack((freqs_2[indexes_1_max],
 ↪  power_lvls_2[indexes_1_max]))
vectors_sig_2_real = np.column_stack((freqs_2[indexes_2_max],
 ↪  power_lvls_2[indexes_2_max]))
```

Zooming in on the peaks was accomplished by, first, specifying the peak frequencies of S_r as centers on the x axis and calculating the mean values of the y coordinates in both signals to get a good center point for each subplot, and, second, by adding small positive and negative offset values on both axis. From the overlay plot in Figure 4.1 it was already possible to tell that most comparable peaks are very close to each other, so it was fairly reasonable to use very small offset values like +/-5 Hz (0.005 MHz) for the x axis and +/-10 dB for the y axis.

```
power_mean = np.mean(np.column_stack((vectors_sig_1[i][1],
 ↪  vectors_sig_2_cmp[i][1])))
plt.xlim(vectors_sig_1[i][0]-DEF_X_OFFSET,
 ↪  vectors_sig_1[i][0]+DEF_X_OFFSET)
```

```
plt.ylim(power_mean-DEF_Y_OFFSET, power_mean+DEF_Y_OFFSET)
```

At this point there was just one last thing to do before plotting of the individual peaks could commence: When comparing two signals originating from different transmitters it is quite possible that one of them will exhibit a significantly larger number of peaks than the other, which makes it a pointless endeavor to take the first few peaks from each signal and try to find any meaningful relation between them. However, if the data in transit is the same and if the transmission parameters such as frequency and sample rate were chosen identically, there will inevitably be a number of spikes at very similar positions with peaks at comparable coordinates as well. Therefore, I had to implement a small window for each peak in the signal exhibiting the smaller number of peaks (S_s), with the exact frequency, *i.e.*, the x coordinate, of the peak at its center. And then iterate over the Python list storing the peak coordinates of the other signal (S_l) and check for each one whether it would lie within the specified range or not. If so, a peak that occurred in both captured signals at comparable positions was found. Otherwise, I would have to accept that for a particular peak in S_s there was no corresponding peak at any similar position in S_l . In Python code this looks as follows:

```
j=i
if (min(len(vectors_sig_1), len(vectors_sig_2_real)) ==
↪  len(vectors_sig_1)): # vector_sig_1 is the smaller list
    while (j < len(vectors_sig_2_real)-1 and (vectors_sig_1[i][0] <
    ↪  vectors_sig_2_real[j][0]-DEF_X_OFFSET*2 or
    ↪  vectors_sig_1[i][0] >
    ↪  vectors_sig_2_real[j][0]+DEF_X_OFFSET*2)):
        j+=1
        plt.axvline(x=vectors_sig_1[i][0], linewidth=1,
        ↪  color=DEF_BLACK)
        plt.axvline(x=vectors_sig_2_real[j][0], linewidth=1,
        ↪  color=DEF_BLACK)
    # ...
elif (min(len(vectors_sig_1), len(vectors_sig_2_real)) ==
↪  len(vectors_sig_2_real)): # vector_sig_2_real is the smaller
↪  signal
```

```
    while (j < len(vectors_sig_1)-1 and (vectors_sig_2_real[i][0] <
    ↪  vectors_sig_1[j][0]-DEF_X_OFFSET*2 or
    ↪  vectors_sig_2_real[i][0] >
    ↪  vectors_sig_1[j][0]+DEF_X_OFFSET*2)):
        j+=1
    plt.axvline(x=vectors_sig_1[j][0], linewidth=1, color=DEF_BLACK)
    plt.axvline(x=vectors_sig_2_real[i][0], linewidth=1,
    ↪  color=DEF_BLACK)
```

The remaining part of the `plot_diffs.py` script produces a second figure with five subplots as that is the number of peaks above the power level of -20 dB in the reference signal. Figure 4.2 shows the peaks from the two keyfob signals plotted in Figure 4.1.

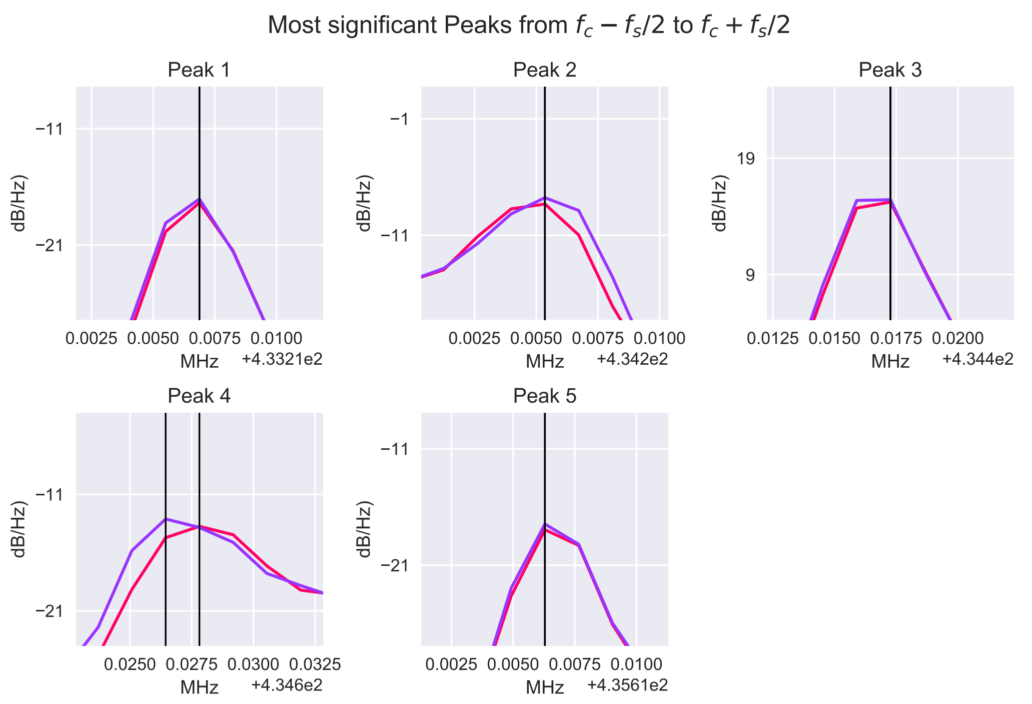


Figure 4.2.: Comparing the peaks of two signals transmitted by the Škoda Fabia II keyfob.

From looking at those five subplots, two things can be told. First, and that is the most obvious, the `peakutils` algorithm was able to detect all five dominant peaks correctly, which can be verified by comparing the center frequencies with the full signal plots in Figure 4.1. Second, those five peaks converge almost perfectly on the very same frequencies. Except for peak 4, there is no offset whatsoever within any meaningful frequency range.

Next, a comparison between a signal originating, again, from the keyfob and one that was transmitted by the YARD Stick One was plotted. The result can be seen in Figure 4.3.

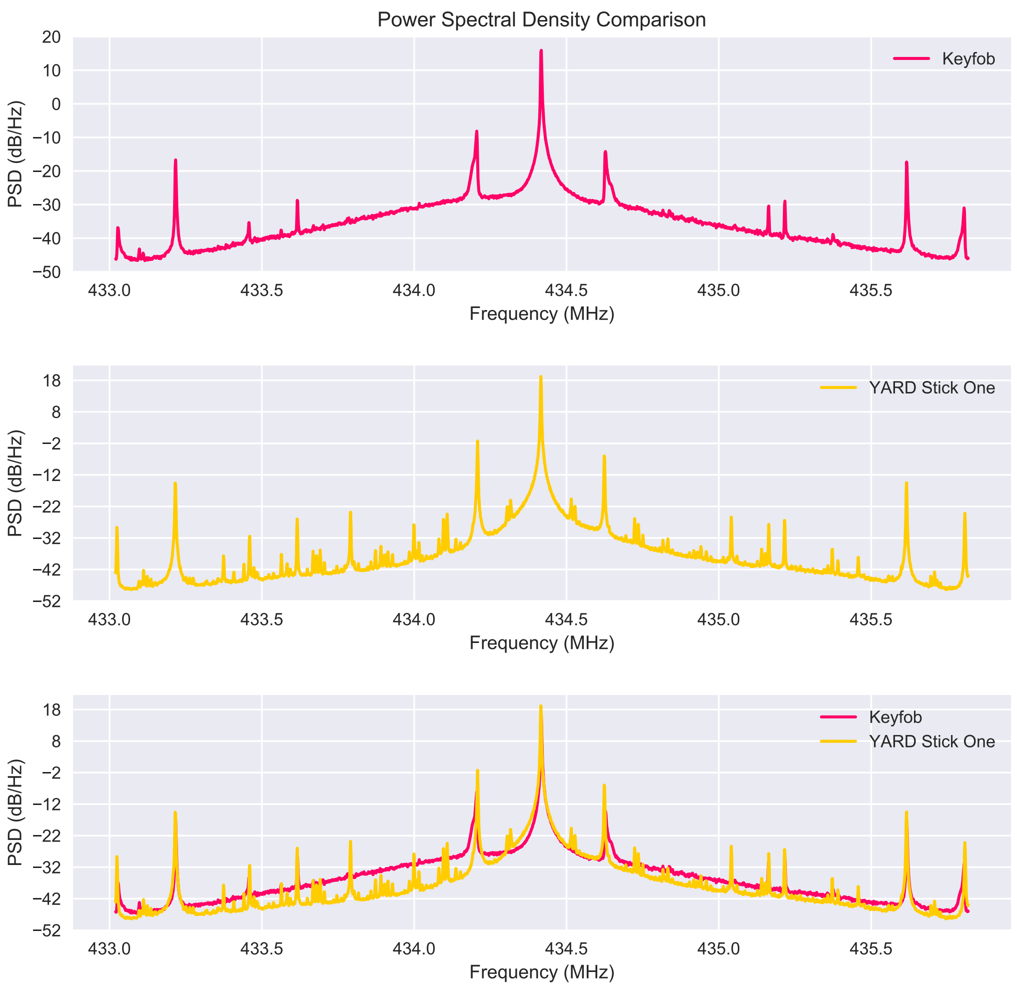


Figure 4.3.: Comparison of a signal transmitted by the Škoda Fabia II keyfob and the YARD Stick One, respectively.

It becomes clear that there is far less similarity when a signal originates from different transmitters than when the same one is used. Not only does the shape of the received signals vary greatly, the one from the YARD Stick One also exhibits a far greater number of peaks. To exclude the possibility that all those minor spikes are caused by noise, the signal capture was repeated in different geographic locations at different times of the day but the result was always pretty much the same. It is thus save to assume that this truly is what a keyfob signal looks like when transmitted by a YARD Stick One.
From the overlay plot it is hard to tell whether the peaks converge, so a closer look at them is required again and provided in Figure 4.4.

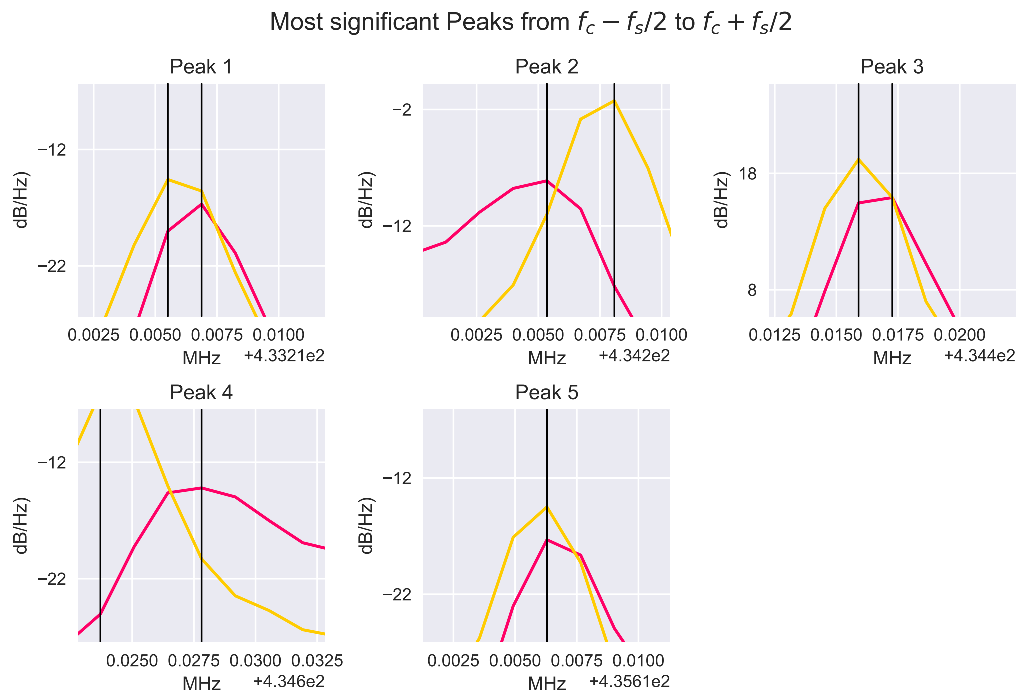


Figure 4.4.: Comparing the peaks of a signal transmitted by the Škoda Fabia II keyfob and the YARD Stick One, respectively.

From looking at the found peaks, it can be deduced that the dominant peaks that both signals have in common seem to occur rather close to each other, however, they almost never converge at the exact same frequencies. By looking at the frequencies it is possible to tell that subplot 3 shows the center peak of both signals, while subplot 1 and subplot 5 show the leftmost and the rightmost dominant peak, respectively. The yellow YARD Stick One peaks of the two massive spikes that occur next to the center spike are depicted in subplot 2 and subplot 4, reaching power levels far higher than their counterparts in the pink keyfob signal. This results in the yellow peak 4 not even being shown inside the subplot.

Lastly, signals transmitted by the keyfob and the Rad1o were compared. Figure 4.5 shows the respective plots and the overlay plot.

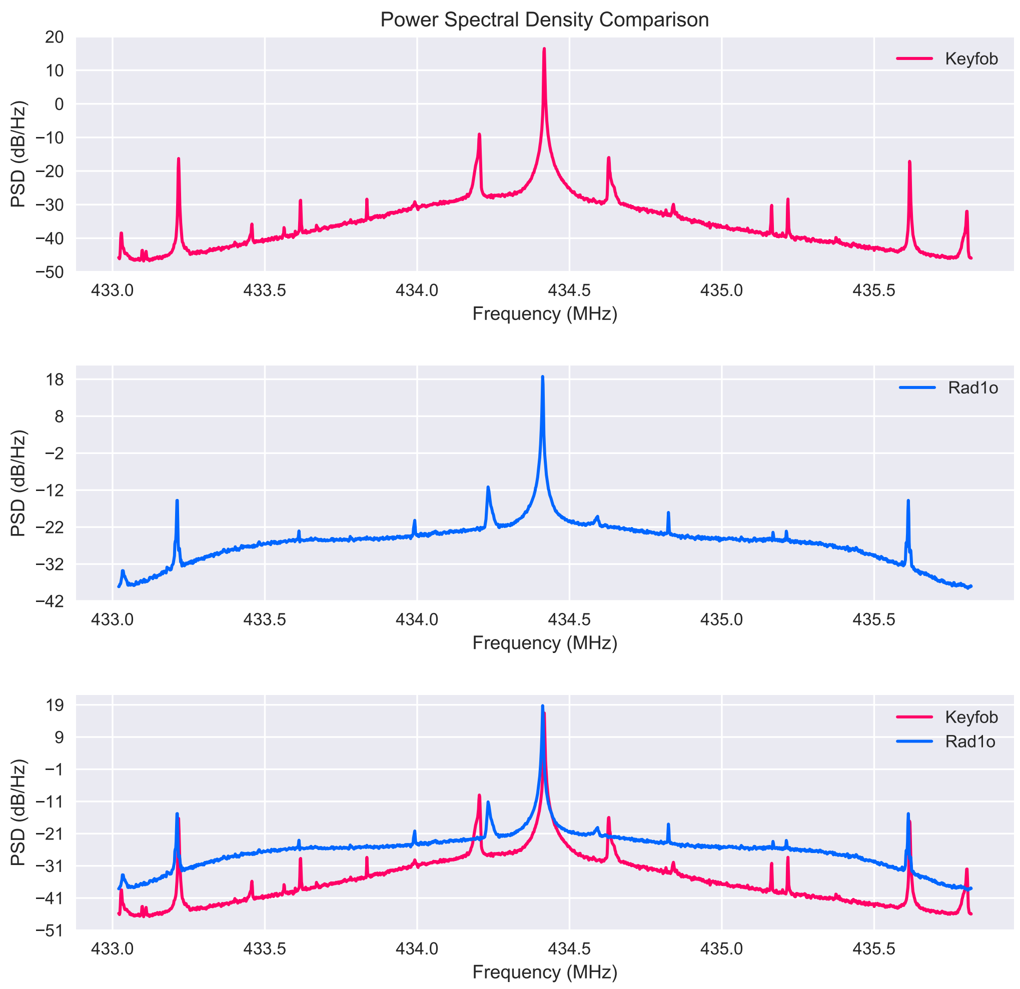


Figure 4.5.: Comparison of a signal transmitted by the Škoda Fabia II keyfob and the Rad1o, respectively.

This comparison exhibits the least number of features that both signals have in common. The shape of both plotted signals appears clearly distinguishable. While the pink signal from the keyfob features a constant rise to the center frequency on both sides of it, the blue signal from the Rad1o appears to have a flattening rise towards the main spike at the center frequency. Also, there is only a very small number of significant peaks in the signal originating from the Rad1o, three of which seem to be truly dominant. The spike to the left of the center spike in the blue signal did not always occur in multiple transmissions from the Rad1o, which excludes it as a reproducible characteristic. What is visible for the human eye in the overlay plot already is that the three dominant peaks do not occur at the exact same frequencies. All peaks seem to feature a small offset towards lower frequencies as compared to their counterparts in the pink reference signal from the keyfob. This characteristic becomes all the more obvious when zooming in on the peaks again, which is depicted in Figure 4.6.

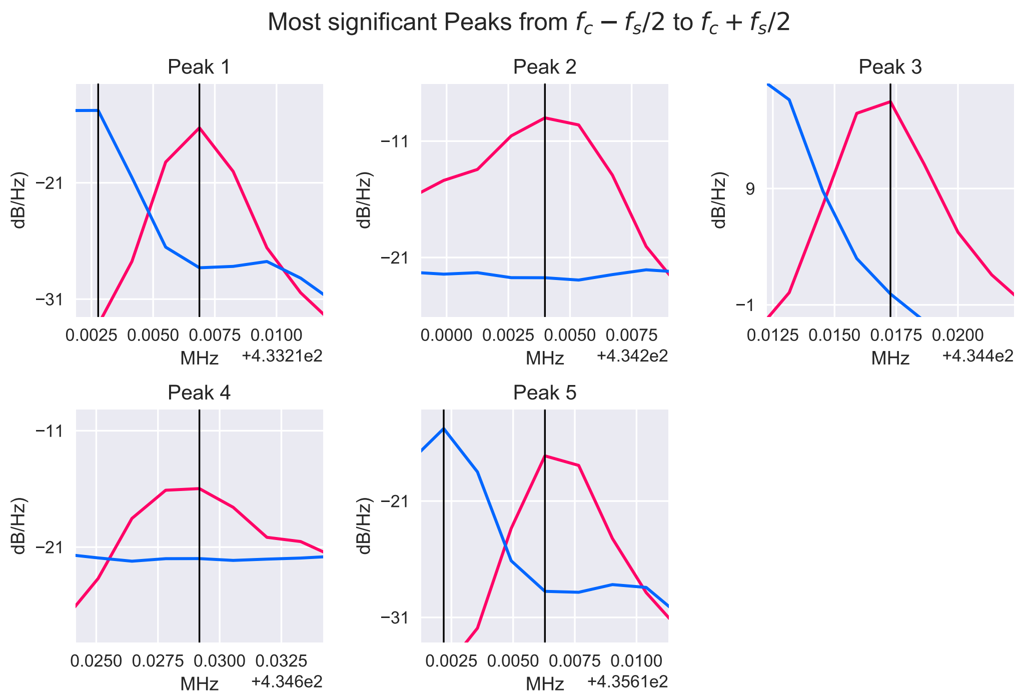


Figure 4.6.: Comparing the peaks of a signal transmitted by the Škoda Fabia II keyfob and the Rad1o, respectively.

The results are three meaningful peaks none of which converge at any frequency. Subplot 3 shows the center peaks of both signals with the blue peak being off by more than ~5000 Hz (0.005 MHz). What makes this offset especially interesting is that it hints at the presence of some tolerance range implemented in the receiver unit in the car. This can be told by the fact that I have been able to successfully unlock my Škoda Fabia II using the Rad1o and the two GNU Radio programs presented in the Background Chapter. Subplot 1 and subplot 5 show the leftmost and the rightmost dominant peak, respectively.

To summarize what can be learned from this Section: It is definitely possible to find characteristics in signals from various origins that distinguish them from each other and allow for signal classification, as I show in the next Section. Furthermore, it could be seen that even when transmitting the very same signal, different transmitting devices produce different signal shapes with various numbers of peaks. Depending on the transmitting device, the peaks may even be clearly off by measurable frequency offsets from the peaks in the signal of the reference transmitter. This leaves me with three characteristics for measurement of differences in digital signals: shape, number of dominant peaks, and position of the peaks within the specified frequency range.

4.2. IDS Approach - Machine Learning for Label Prediction

Considering the fact that this research project aims at classifying signals based on certain characteristics, and my need for a highly dynamic solution that would be able to handle large amounts of data and increase its accuracy through making predictions on constantly added new data, I decided to try machine learning algorithms for classification of labeled data. Specifically, I was working with two Python scripts, the first of which was applying the *k-nearest Neighbors* algorithm (*k*-NN) to a previously prepared dataset. *k*-NN was the obvious choice as it is based on the simple idea of predicting the label of unknown values by matching them with the most similar known, *i.e.*, labeled, values.
The second script was applying a specific *Support Vector Machine* (SVM), namely the *Support Vector Classification* (SVC) *with linear kernel* to the same dataset. Both the *k*-NN and the SVC are well-suited algorithms for this research problem, however, it was also of importance to find out if one of them was going to perform significantly better than the other in terms of accuracy and performance, hence the comparison.

4.2.1. Script for Data Accumulation: `build_dataset.py`

The first step was to build the dataset, which had to be sufficiently large for the machine learning algorithms to produce meaningful results, and it had to reflect what the signal receiving unit inside the car might "see" if it became the target of hacking attempts. Therefore, I wrote the `build_datset.py` Python script that allowed me to capture however many signals I needed and writes them to a CSV file. 300 captured signals, combining 100 for each of the three transmitters, seemed like a reasonable amount of data for a proof-of-concept. If more signals were needed, the following few lines of the script would have to be adjusted accordingly to get the labelling of the data right:

```python
# i1 = reference signal (not written as data row)
DEF_SIG = "Keyfob" # i2 to i101 = 100
if (i >= 102):
    DEF_SIG = "YARD Stick One" # i102 to i201 = 100
if (i >= 202):
    DEF_SIG = "Rad1o" # i202 to i301 = 100
if (i == 302):
    break
```

The script expects the user to transmit the provided number of signals from each device consecutively, without interchanging transmitters during the capture. Otherwise, the labelling would get mixed up.

Also, it should be noted that the keyfob, being the reference transmitter, has to come first and that the counting of captured signals starts at 2 since the very first captured signal is not stored as a data row but instead used to write the header of the CSV file with the reference frequencies, *i.e.*, where peaks occurred in the reference signal.

After that code block, for the most part the `build_dataset.py` Python script does the same as the `plot_diffs.py` script introduced in Subsection 4.1.1, except for the plotting. Vectors containing the frequency and power level values of peaks in the reference signal are created, and the same is done for the equivalent positions in the signal that is to be compared with the previously mentioned one.

Once all the required data has been accumulated and processed, the script creates the header for the CSV dataset file and the data row that will be appended to the dataset.

```python
data = []
header = []
header.append("Label")
header.append("Total Peaks")
data.append(DEF_SIG)
data.append(len(vectors_sig_2_real))
for j in range(0, len(vectors_sig_2_cmp)):
    header.append(str(vectors_sig_2_cmp[j][0]))
    data.append(vectors_sig_2_cmp[j][1])
data_row = dict(zip(header, data))
write_to_csv(header, data_row)
```

Both the header and the created data_row are then passed to the method I wrote for writing data to a specified CSV file:

```python
def write_to_csv(header, data):
    file_exists = os.path.isfile("signal-data-000.csv")

    with open("signal-data-000.csv", "a") as csv_file:
        writer = csv.DictWriter(csv_file, fieldnames=header)
        if not file_exists:
            writer.writeheader()
        writer.writerow(data)
```

To prevent the header from being written to the dataset again and again, the script checks whether the CSV file already exists, and if that is the case, it can safely assume that the header was written already at the time the file was created, thus not appending it needlessly.

Now, with the `build_dataset.py` script ready, building of the dataset could begin. While I did implement user interaction-based execution control steps in the involved scripts to make this process as simple as possible, there is still a number of things to be aware of to get it right. Basically, the following steps were required in the given order:

1. Keyfob data rows:
   1.1. Before the `build_dataset.py` script can be started, a sufficient number of signals has to be captured by the YARD Stick One using the `replay.py` script, because once the former script runs, it will capture every signal it sees, even if a signal is meant to be captured by another script only. Thus, I run the `replay.py` script inside a terminal window and kept ignoring the prompt to switch to transmission mode by just pressing `<enter>`. Due to my experiences made during this part of the research work, if n is the desired number of data rows in the dataset for this device, I recommend capturing $n+n/3$ signals to be on the safe side as some signals will inevitably get lost during transmission in step 2.
   1.2. The following step involves building of the dataset and it is important to note: the reference transmitter, regardless of whether it is the keyfob in this research project or another device in a similar project using my scripts, has to come first. I started running the `build_dataset.py` script inside a terminal window and also started transmitting signals from the reference transmitter until the script said that it had written the desired number of signal data to the CSV file for this particular device. *E.g.*, if the desired number is 100, ideally, this means pressing the "unlock" button of the keyfob 101 times (the first signal does not count as it is only used to create the header), however, it is highly unlikely for any of the devices I used to succeed in transmitting a signal strong enough to exceed a power level of -10 dB every single time. This is the threshold a signal has to exceed to be recognized and captured. Therefore, one needs to continue pressing the "unlock" button until 100 signal data rows have been written, even if it means transmitting signals 110 or 120 times. Also, the `build_dataset.py` script must not be stopped until the last device has transmitted its last signal.
2. YARD Stick One data rows:
   2.1. At this point, the keyfob was no longer needed. Instead, I started transmitting the captured

signals from YARD Stick One one by one until the `build_dataset.py` script said that it had written signal 200 (signals 101 to 200), as that was the number of signals specified for this device in the code. Figure 4.7 illustrates this. In the left terminal window the `build_dataset.py` script is running without any user interaction, while in the right terminal window the `replay.py` script constantly asks the user whether they want to stop transmitting from YARD Stick One, which, if ignored, leads to another signal being transmitted.

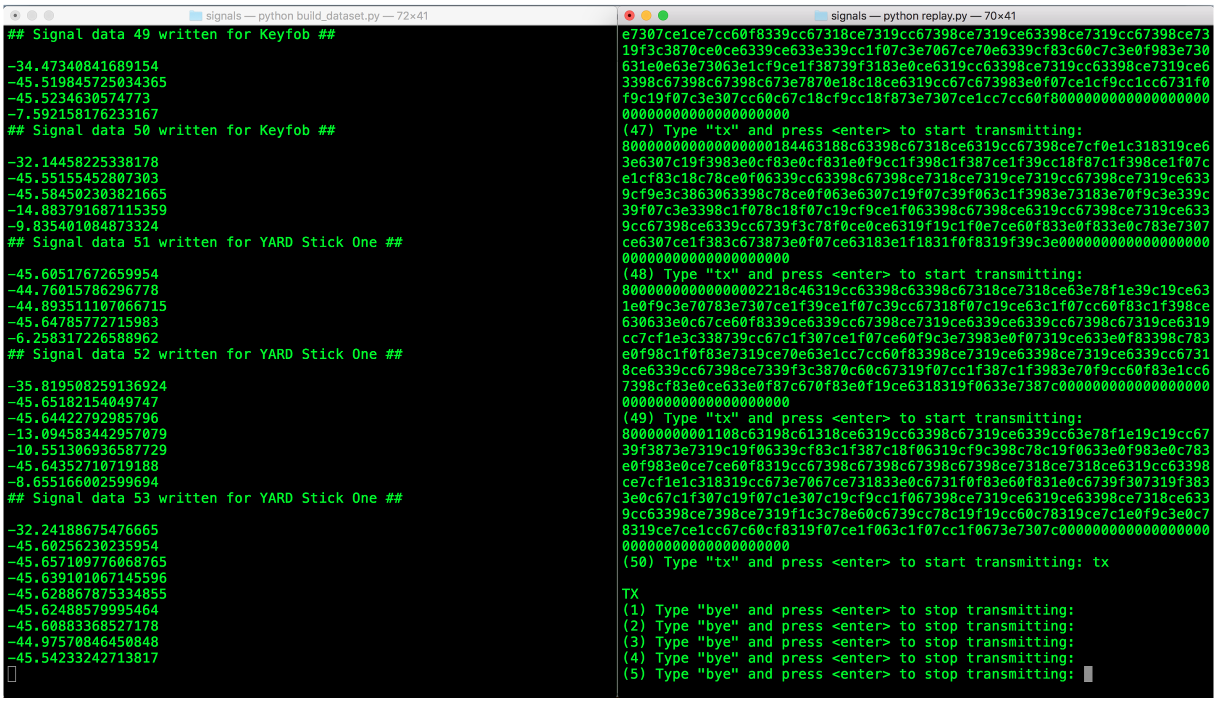
2.2. Afterwards, while the `build_dataset.py` script should still be left running as there is one more transmitter device left, the `replay.py` script may be stopped and the terminal window it was running in closed. YARD Stick One may be ejected from the USB port of the machine.

3. Rad1o data rows:

3.1. For transmitting signals from the Rad1o, GNU Radio and the `replay_TX.grc` program introduced in Chapter 3, Subsection 3.2.2 could be used. As the program would run in a loop anyway, all that was required was to wait for the `build_dataset.py` script to tell me that it had written signal data 150, *i.e.*, the last data row, for this device and stop program execution right afterwards. Figure 4.8 is showing a transmission in the GNU Radio window[1] at the front and the Python script "listening" in the terminal window at the back.

3.2. Since no more data was going to be written to the CSV file, I could now stop building of the dataset by pressing `<ctrl+C>`.

[1]Due to a bug in the QT GUI of the GNU Radio installation on macOS, the axis and legend are missing in the window produced by the Frequency Sink block. However, this did not affect the displaying of signals in the frequency spectrum.

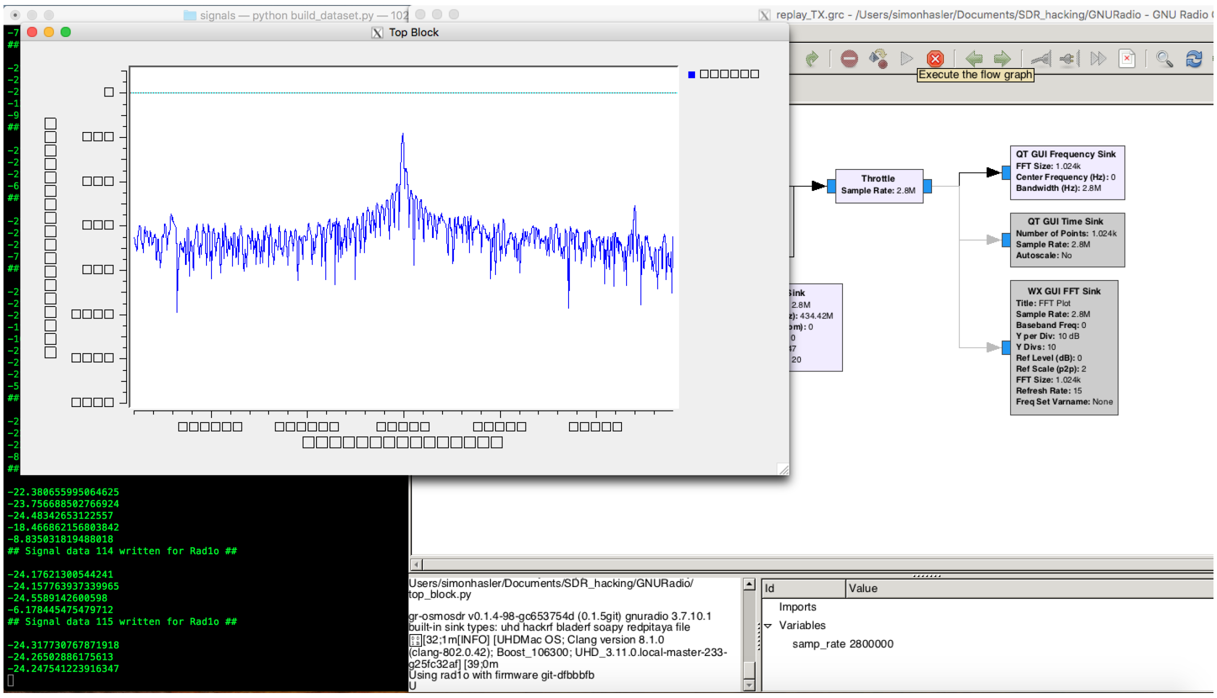


signals — python build_dataset.py — 72×41
signals — python replay.py — 70×41

```
## Signal data 49 written for Keyfob ##

-34.47340841689154
-45.519845725034365
-45.5234630574773
-7.592158176233167
## Signal data 50 written for Keyfob ##

-32.14458225338178
-45.55155452807303
-45.584502303821665
-14.883791687115359
-9.835401084873324
## Signal data 51 written for YARD Stick One ##

-45.60517672659954
-44.76015786296778
-44.893511107066715
-45.64785772715983
-6.258317226588962
## Signal data 52 written for YARD Stick One ##

-35.819508259136924
-45.65182154049747
-45.64422792985796
-13.094583442957079
-10.551306936587729
-45.64352710719188
-8.655166002599694
## Signal data 53 written for YARD Stick One ##

-32.24188675476665
-45.60256230235954
-45.657109776068765
-45.639101067145596
-45.628867875334855
-45.62488579995464
-45.60883368527178
-44.97570846450848
-45.54233242713817
```

```
e7307ce1ce7cc60f8339cc67318ce7319cc67398ce7319ce63398ce7319cc67398ce73
19f3c3870ce0ce6339ce633e339cc1f07c3e7067ce70e6339cf83c60c7c3e0f983e730
631e0e63e73063e1cf9ce1f38739f3183e0ce6319cc63398ce7319cc63398ce7319ce6
3398c67398c67398c673e7870e18c18ce6319cc67c673983e0f07ce1cf9cc1cc6731f0
f9c19f07c3e307cc60c67c18cf9cc18f873e7307ce1cc7cc60f8000000000000000000
00000000000000000000
(47) Type "tx" and press <enter> to start transmitting:
80000000000000000184463188c63398c67318ce6319cc67398ce7cf0e1c318319ce6
3e6307c19f3983e0cf83e0cf831e0f9cc1f398c1f387ce1f39cc18f87c1f398ce1f07c
e1cf83c18c78ce0f06339cc63398c67398ce7318ce7319ce7319cc67398ce7319ce633
9cf9e3c3863063398c78ce0f063e6307c19f07c39f063c1f3983e73183e70f9c3e339c
39f07c3e3398c1f078c18f07c19cf9ce1f063398c67398ce6319cc67398ce7319ce633
9cc67398ce6339cc6739f3c78f0ce0ce6319f19c1f0e7ce60f833e0f833e0c783e7307
ce6307ce1f383c673873e0f07ce63183e1f1831f0f8319f39c3e000000000000000000
00000000000000000000
(48) Type "tx" and press <enter> to start transmitting:
80000000000000002218c46319cc63398c63398c67318ce7318ce63e78f1e39c19ce63
1e0f9c3e70783e7307ce1f39ce1f07c39cc67318f07c19ce63c1f07cc60f83c1f398ce
630633e0c67ce60f8339ce6339cc67398ce7319ce6339ce6339cc67398c67319ce6319
cc7cf1e3c338739cc67c1f307ce1f07ce60f9c3e73983e0f07319ce633e0f83398c783
e0f98c1f0f83e7319ce70e63e1cc7cc60f83398ce7319ce63398ce7319ce6339cc6731
8ce6339cc67398ce7339f3c3870c60c67319f07cc1f387c1f3983e70f9cc60f83e1cc6
7398cf83e0ce633e0f87c670f83e0f19ce6318319f0633e7387c000000000000000000
00000000000000000000
(49) Type "tx" and press <enter> to start transmitting:
8000000001108c63198c61318ce6319cc63398c67319ce6339cc63e78f1e19c19cc67
39f3873e7319c19f06339cf83c1f387c18f06319cf9c398c78c19f0633e0f983e0c783
e0f983e0ce7ce60f8319cc67398c67398c67398c67398ce7318ce7318ce6319cc63398
ce7cf1e1c318319cc673e7067ce731833e0c6731f0f83e60f831e0c6739f307319f383
3e0c67c1f307c19f07c1e307c19cf9cc1f067398ce7319ce6319ce63398ce7318ce633
9cc63398ce7398ce7319f1c3c78e60c6739cc78c19f19cc60c78319ce7c1e0f9c3e0c7
8319ce7ce1cc67c60cf8319f07ce1f063c1f07cc1f0673e7307c000000000000000000
00000000000000000000
(50) Type "tx" and press <enter> to start transmitting: tx

TX
(1) Type "bye" and press <enter> to stop transmitting:
(2) Type "bye" and press <enter> to stop transmitting:
(3) Type "bye" and press <enter> to stop transmitting:
(4) Type "bye" and press <enter> to stop transmitting:
(5) Type "bye" and press <enter> to stop transmitting:
```

Figure 4.7.: Accumulating signal captures from YARD Stick One using both the `build_dataset.py` and the `replay.py` scripts.



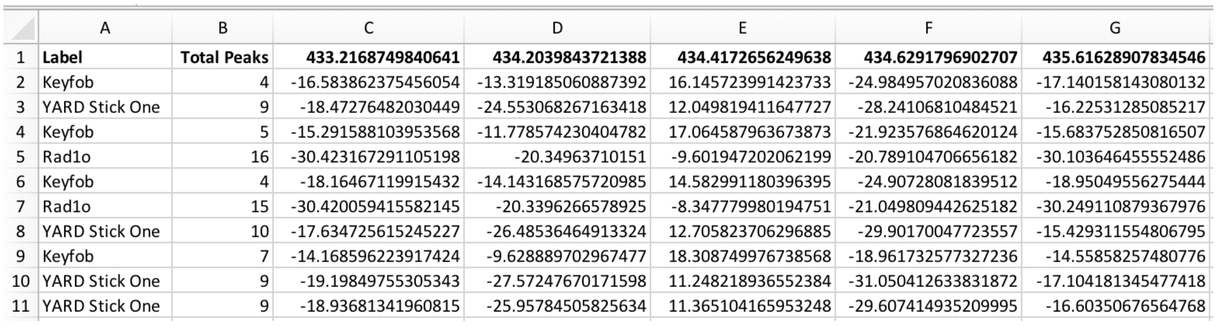
```
-22.380655995064625
-23.756688502766924
-24.48342653122557
-18.466862156803842
-8.835031819488018
## Signal data 114 written for Rad1o ##

-24.17621300544241
-24.157763937339965
-24.5589142600598
-6.178445475479712
## Signal data 115 written for Rad1o ##

-24.317730767871918
-24.26502886175613
-24.247541223916347
```

```
Users/simonhasler/Documents/SDR_hacking/GNURadio/
top_block.py

gr-osmosdr v0.1.4-98-gc653754d (0.1.5git) gnuradio 3.7.10.1
built-in sink types: uhd hackrf bladerf soapy redpitaya file
[32;1m[INFO] [UHDMac OS; Clang version 8.1.0
(clang-802.0.42); Boost_106300; UHD_3.11.0.local-master-233-
g25fc32af] [39;0m
Using rad1o with firmware git-dfbbbfb
U
```

Id	Value
Imports	
Variables	
samp_rate	2800000

Figure 4.8.: Accumulating signal captures from Rad1o using the `build_datset.py` script and the `replay_TX.grc` GNU Radio program.

Building the dataset the way previously described in this subsection results in a CSV file that contains the individual data rows ordered by device and in the order they were written. This is exactly what the `build_dataset.py` script is supposed to produce, however, it does not reflect a realistic scenario in which a car is "under attack" from various transmitting devices and receives their signals in a mixed and seemingly random order. Thus, there was one last thing to do before the then final dataset could be fed to machine learning algorithms: shuffle the rows inside the dataset for random ordering. The desired result was achieved by entering the code lines listed in Appendix 5.1 into an interactive IPython terminal Figure 4.9 displays the first few rows of the final dataset as they appear in Microsoft Excel.



	A	B	C	D	E	F	G
1	**Label**	**Total Peaks**	**433.2168749840641**	**434.2039843721388**	**434.4172656249638**	**434.6291796902707**	**435.61628907834546**
2	Keyfob	4	-16.583862375456054	-13.319185060887392	16.145723991423733	-24.984957020836088	-17.140158143080132
3	YARD Stick One	9	-18.47276482030449	-24.553068267163418	12.049819411647727	-28.24106810484521	-16.22531285085217
4	Keyfob	5	-15.291588103953568	-11.778574230404782	17.064587963673873	-21.923576864620124	-15.683752850816507
5	Rad1o	16	-30.423167291105198	-20.34963710151	-9.601947202062199	-20.789104706656182	-30.103646455552486
6	Keyfob	4	-18.16467119915432	-14.143168575720985	14.582991180396395	-24.90728081839512	-18.95049556275444
7	Rad1o	15	-30.420059415582145	-20.3396266578925	-8.347779980194751	-21.049809442625182	-30.249110879367976
8	YARD Stick One	10	-17.634725615245227	-26.48536464913324	12.705823706296885	-29.90170047723557	-15.429311554806795
9	Keyfob	7	-14.168596223917424	-9.628889702967477	18.308749976738568	-18.961732577327236	-14.55858257480776
10	YARD Stick One	9	-19.19849755305343	-27.57247670171598	11.248218936552384	-31.050412633831872	-17.104181345477418
11	YARD Stick One	9	-18.93681341960815	-25.95784505825634	11.365104165953248	-29.607414935209995	-16.60350676564768

Figure 4.9.: The first 10 rows of signal data in the randomly ordered CSV dataset displayed in Microsoft Excel.

What can be told from those few rows is, first, that the header contains a label indicating the transmitter and the total number of peaks for each signal in the first two columns, and the exact frequency that a peak occurred at in the reference signal in each subsequent column. And, second, that the data rows starting at row 2 in the Excel sheet contain, besides the device label and total number of peaks, the exact power levels that each recorded signal exhibited at the frequencies written in the header.

4.2.2. Script for *k*-nearest Neighbors Algorithm: `kNN_prediction.py`

The Scikit-learn framework of the Python programming language provides various classes for all well-known machine learning algorithms. Consequently, the first part of the `kNN_prediction.py` script consists of imports of all the classes needed for the *k*-nearest Neighbors algorithm:

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from sklearn.cross_validation import cross_val_score, LeaveOneOut
# ...
```

In the `main()` function, first, the CSV dataset is imported through utilization of the Pandas library and its `read_csv()` method. The next few lines of the script are essential in interpreting and understanding the result that the *k*-NN algorithm will produce for the provided 6-dimensional dataset ($n = 6$), as I show in Chapter 5, Section 5.1. What they do is creating a figure that shows $n * n$ subplots, each of which is either representing the distribution of data points for a particular feature from the dataset, or for two features if that was the number of features considered by the machine learning algorithm.

```
sns.pairplot(df, hue="Label", palette={"Keyfob": DEF_PINK, "YARD
↪  Stick One": DEF_YELLOW, "Rad1o": DEF_BLUE})
plt.savefig("dataset.pdf", dpi="figure", format="pdf")
plt.show()
```

The next step was creating the feature matrix `X` consisting of a specified number of samples (n_S) and features (n_F), where "samples" basically refers to the number of data rows and "features" to the number of feature columns that shall be used by the *k*-NN algorithm. As opposed to the usually 2-dimensional feature matrix, the target vector `y` is a 1-dimensional array of length n_S that stores the quantity that shall be predicted from the data, *i.e.*, the label in this case.
Since training and test data is required for any machine learning algorithm to learn from, the whole dataset is split into two parts in a ratio $2/3$ to $1/3$, the smaller part being the test data.

```
X = np.array(df.ix[:, 1:7]) # 1:2 for the first feature only
y = np.array(df["Label"])

X_train, X_test, y_train, y_test = train_test_split(X, y,
↪  test_size=0.33, random_state=42)
```

Sometimes, depending on the type of data that is used in machine learning, it can happen that a particular, dominant feature contributes far more to the result than the other features. This is especially true if one feature has a particularly broad range of values. To prevent this feature from dominating the algorithm and ensure that all features contribute approximately proportionally to the final result, a scaler can be used to normalize the features so that all of them are evaluated uniformly. The next few lines of code show this:

```
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

From here on, the code follows the standard procedure in machine learning: instantiate the class of the desired model with all necessary hyperparameters, fit the model to the data, and predict the labels of the test data points, which are then printed to allow for a comparison with the true labels.

```python
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
print("X_test:\n{}\n".format(knn.predict(X_test)))
print("y_test:\n{}\n".format(y_test))
```

Once a machine learning model has done its job, it is always interesting to see how well it performed on the provided data in regards to accuracy. Therefore, the simple accuracy score is calculated using the respective method from the `metrics` class, and cross validation is performed. The latter is done by splitting the whole dataset into five equally sized parts, continuously fit the model to two of them until all parts have been used at least once, and print the five respective accuracy results. It is also possible to calculate the mean of them to get one final cross validation accuracy score. Another cross validation method that is applied is called "LeaveOneOut" (LOOCV). This method essentially does the same as the one previously explained, with the difference that it does not split the dataset into just five parts, but takes all rows except for one as training data and the single left out row as test data. It then continues this process until each single row has been used as test data once.

```python
print("Accuracy Score:
 ↪  {:.6f}%\n".format(metrics.accuracy_score(y_test,
 ↪  predicted)*100))
x_val_score = cross_val_score(knn, X, y, cv=5)
for i in range(0, len(x_val_score)):
    print("Cross Validation Score ({}): {:.6f}%\n".format(i+1,
     ↪  x_val_score[i]*100))
l1o_score = cross_val_score(knn, X, y, cv=LeaveOneOut(len(X)))
print("\'Leave One Out\' Cross Validation Score:
 ↪  {:.6f}%\n".format(l1o_score.mean()*100))
```

The last part of the code prints the classification report and creates the confusion matrix, which I present and explain in Chapter 5, Section 5.1 for two selected features of the data I am working with.

4.2.3. Script for Support Vector Machine Algorithm: `svm_prediction.py`

The `svm_prediction.py` script that utilizes the Support Vector Classifier for predicting signal sources via machine learning is, for the most part, identical to the previously described `kNN_prediction.py` script. However, there is a small number of significant differences that are noteworthy, such as the import of the required `svm` class from the Scikit-learn framework and its instantiation with the appropriate hyperparameters.

```python
from sklearn import svm

# ...

svc = svm.SVC(C=1, kernel="linear", gamma="auto")
```

After trying a broad range of values for the individual hyperparameters, a linear kernel turned out to be the best choice for the provided dataset as the different classes proved to be very heterogenous and the belonging data-points are distributed well enough to separate them by drawing straight lines. This is clearly visible in Figure 5.2 in Chapter 5.

The γ parameter defines how far the influence of a single training sample reaches, with low values indicating "far" and high values the opposite. In essence, this means that if the value for γ is set rather high, the SVC will attempt to draw the class boundaries tightly around all data-points it believes to be in a certain class. Overfitting problems may ensue due to a too high γ value. The penalty parameter C of the error term controls the trade-off between classifying the individual data-points correctly and creating smooth decision boundaries. A higher C value allows the model to select more samples as support vectors for improved prediction precision [32].

4.3. Implementation of the Signal IDS on a nVidia Jetson TX2 Module

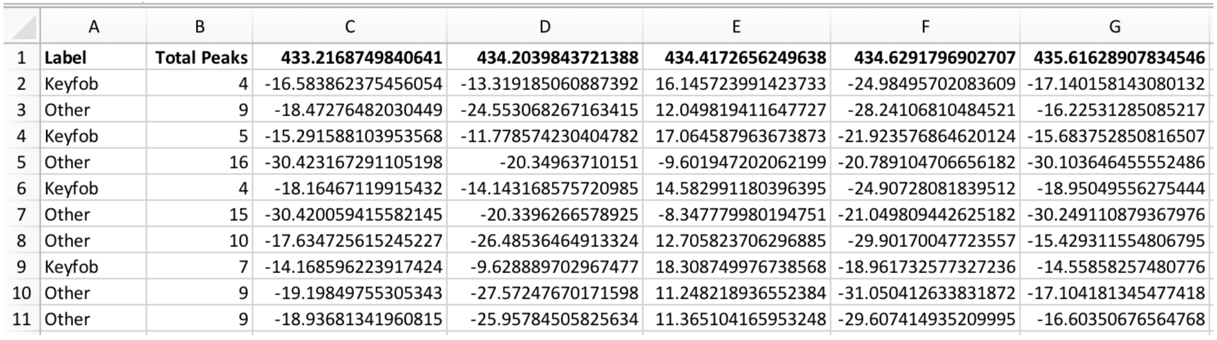
4.3.1. Script for Signal Intrusion Detection: `signal_ids.py`

This Python script is a combination of all the relevant parts from the previously presented scripts and is necessary for the final result of this research work. It also represents the software part of the proof-of-concept and, in the completed system, runs on a nVidia Jetson TX2 module, as described in the following subsection.

The `signal_ids.py` script first reads in the data from the shuffled dataset and then replaces all labels other than "Keyfob" with "Other", the result of which can be seen in Figure 4.10.

```
df = pd.read_csv("signal-data-000_shuffled.csv")
df["Label"] = df["Label"].replace(["YARD Stick One", "Rad1o"],
 ↪  "Other")
df.to_csv("signal-data-000_real.csv", index=False)
df = pd.read_csv("signal-data-000_real.csv")
ref_freqs = np.array(df.columns.values)
```

The reason for doing this is that it is just not possible to know all devices that can be used for a capture and replay attack beforehand, or even obtain signal data from them that could be added to the training data for the Signal IDS. Therefore, the only feasible thing for the Signal IDS to do is to determine whether a received signal resembles the known keyfob signals such that it may be labelled accordingly, or if it rather resembles a signal that is already known to originate from a different transmitter, in which case the unknown signal should be labelled "Other". I elaborate on this issue in Chapter 5, Section 5.2. For a proof-of-concept, however, I deem this solution sufficient, considering that for a car it is irrelevant which hacking device may have transmitted an unlock signal. It is only of importance to determine correctly whether it was the keyfob or not, and perform the appropriate action.



	A	B	C	D	E	F	G
1	**Label**	**Total Peaks**	**433.2168749840641**	**434.2039843721388**	**434.4172656249638**	**434.6291796902707**	**435.61628907834546**
2	Keyfob	4	-16.583862375456054	-13.319185060887392	16.145723991423733	-24.98495702083609	-17.140158143080132
3	Other	9	-18.47276482030449	-24.553068267163415	12.049819411647727	-28.24106810484521	-16.22531285085217
4	Keyfob	5	-15.291588103953568	-11.778574230404782	17.064587963673873	-21.923576864620124	-15.683752850816507
5	Other	16	-30.423167291105198	-20.34963710151	-9.601947202062199	-20.789104706656182	-30.103646455552486
6	Keyfob	4	-18.16467119915432	-14.143168575720985	14.582991180396395	-24.90728081839512	-18.95049556275444
7	Other	15	-30.420059415582145	-20.3396266578925	-8.347779980194751	-21.049809442625182	-30.249110879367976
8	Other	10	-17.634725615245227	-26.48536464913324	12.705823706296885	-29.90170047723557	-15.429311554806795
9	Keyfob	7	-14.168596223917424	-9.628889702967477	18.308749976738568	-18.961732577327236	-14.55858257480776
10	Other	9	-19.19849755305343	-27.57247670171598	11.248218936552384	-31.050412633831872	-17.104181345477418
11	Other	9	-18.93681341960815	-25.95784505825634	11.365104165953248	-29.607414935209995	-16.60350676564768

Figure 4.10.: The first 10 rows of signal data with only two different labels displayed in Microsoft Excel.

Since the Signal IDS, once started, should run until it is explicitly stopped by the user, the main part of the script consists of an infinite loop that is only exited when a keyboard interrupt from the pressed `<ctrl+C>` key combination is received. Inside the loop, this Python script, too, listens for signals but as opposed to the other scripts introduced in this thesis it already recognizes them if they exceed a lower power level of -15 dB. This was necessary as otherwise too many signals would be missed, and it would be far from desirable behavior if a car owner had to press the unlock button on their keyfob multiple times until they succeed in transmitting a strong enough signal.

Next, the script does two distinct things: first, it tries to find the indexes of all peaks of the current signal above a power level of -20 dB. And, second, it determines the power levels of the current signal at the frequencies where peaks occurred in the reference signal. The former is necessary to get the total number of peaks in a signal, which is the first feature in the dataset, whereas the latter is required to obtain values for the remaining five features, *i.e.*, the reference peak frequencies.

```
power_lvls = 10*log10(Pxx/(sdr.sample_rate/1e6))+10*log10(8/3)
indexes = peakutils.indexes(power_lvls, thres=0.25, min_dist=10)
power_lvls_max = [i for i in power_lvls[indexes] if i >= -20]
check_pw_lvls = np.isin(power_lvls, power_lvls_max)
check_freqs = np.isin(freqs, np.float64(ref_freqs[2:7]))
p_indexes_max = np.where(check_pw_lvls)
f_indexes_max = np.where(check_freqs)

vectors_sig_real = np.column_stack((freqs[p_indexes_max],
↪   power_lvls[p_indexes_max]))
```

With all the required data acquired, the `signal_ids.py` continues to build a data row in the same form as those already present in the dataset, the only difference being that the new row of the unknown signal does not yet get a label assigned. That remains to be done by the machine learning algorithm.

```
data = []
header = []
header.append("Total Peaks")
data.append(len(vectors_sig_real))
for j in range(0, len(power_lvls[f_indexes_max])):
    header.append(str(ref_freqs[2:7][j]))
    data.append(power_lvls[f_indexes_max][j])
data_row = dict(zip(header, data))
```

The remaining part of the `for` loop takes the entire dataset as training data for the machine learning algorithm and only the single data row of the unknown signal as test data, scales the features to ensure uniform evaluation, and ultimately fits the instantiated model to the data. Most of this process has already been illustrated with explained code samples in Chapter 4, Subsection 4.2.2, which is why only the first part is shown below:

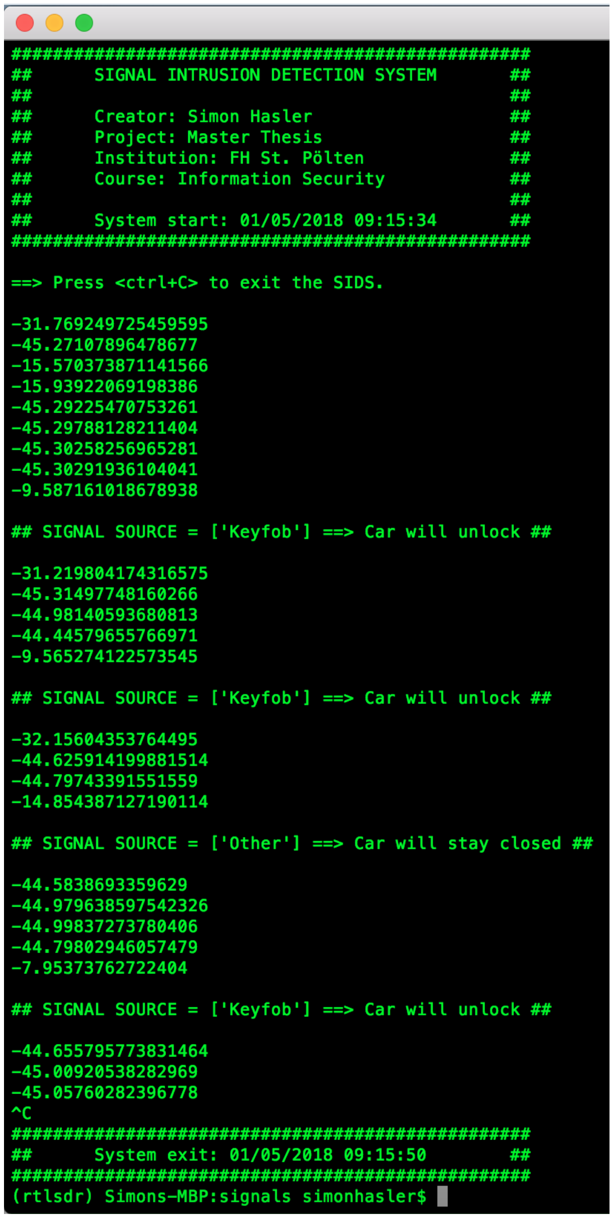
```
X_train = np.array(df.ix[:, 1:7])
y_train = np.array(df["Label"])
X_test  = np.array([list(data_row.values())])
```

After finding that the Support Vector Classifier performs significantly better than the *k*-nearest Neighbor algorithm in regards to accuracy if features with hardly distinguishable data-points are used, which can be seen in Chapter 5, Section 5.1, it was the obvious choice for the implementation of the Signal Intrusion Detection System.

Lastly, once the SVC algorithm has determined the class of a newly received signal, it outputs in the terminal window what action the car should perform based on the classification result:

```
if (predicted == "Keyfob"):
    car_do = "unlock"
else:
    car_do = "stay closed"
print("\n## SIGNAL SOURCE = {} ==> Car will {}
 ↪   ##\n".format(predicted, car_do))
```

Figure 4.11 is showing the final Signal IDS in action. Everything the user is required to do is to start the system, and eventually stop it.



```
##################################################
##      SIGNAL INTRUSION DETECTION SYSTEM       ##
##                                              ##
##      Creator: Simon Hasler                   ##
##      Project: Master Thesis                  ##
##      Institution: FH St. Pölten              ##
##      Course: Information Security            ##
##                                              ##
##      System start: 01/05/2018 09:15:34       ##
##################################################

==> Press <ctrl+C> to exit the SIDS.

-31.769249725459595
-45.27107896478677
-15.570373871141566
-15.93922069198386
-45.29225470753261
-45.29788128211404
-45.30258256965281
-45.30291936104041
-9.587161018678938

## SIGNAL SOURCE = ['Keyfob'] ==> Car will unlock ##

-31.219804174316575
-45.31497748160266
-44.98140593680813
-44.44579655766971
-9.565274122573545

## SIGNAL SOURCE = ['Keyfob'] ==> Car will unlock ##

-32.15604353764495
-44.625914199881514
-44.79743391551559
-14.854387127190114

## SIGNAL SOURCE = ['Other'] ==> Car will stay closed ##

-44.5838693359629
-44.979638597542326
-44.99837273780406
-44.79802946057479
-7.95373762722404

## SIGNAL SOURCE = ['Keyfob'] ==> Car will unlock ##

-44.655795773831464
-45.00920538282969
-45.05760282396778
^C
##################################################
##      System exit: 01/05/2018 09:15:50        ##
##################################################
(rtlsdr) Simons-MBP:signals simonhasler$ █
```

Figure 4.11.: The Signal IDS running in a macOS terminal window on a MacBook Pro mid-2014 model.

4.3.2. Hardware Setup and Final Implementation

The final Signal IDS proof-of-concept had to be as close as possible to something that could actually be implemented in a real car. Therefore, I researched on what hardware platforms machine learning algorithms, *e.g.*, for pattern recognition or computer vision, are implemented in modern cars from manufacturers such as Tesla, Volvo, Audi, and Mercedes-Benz. Those manufacturers, and many more, have all partnered with nVidia to use the Tegra-based Drive PX platform for features such as visually stunning cockpits and infotainment systems, as well as for the technology that will be driving the first SAE Level 3 autonomous cars [33, 34].

Due to not having any of the Drive PX modules available for this research project, a nVidia Jetson TX2 module from Taiwanese company Aetina that manufactures industrial grade graphics cards and GPGPU solutions for embedded applications was chosen as alternative hardware platform. From a technical perspective, the Jetson TX2 is very well suited for AI computing tasks supposed to be performed on an embedded device, which also makes it a perfect fit for running the Signal IDS. Basically, it is a highly power-efficient supercomputer on a credit card-sized board, running at only 7.5 watt and featuring a nVidia Pascal-family GPU with 256 CUDA cores. Memory-wise it is equipped with 8 GB 128 bit LPDDR4 that allow for a memory bandwidth of 59.7 GB/s [35].

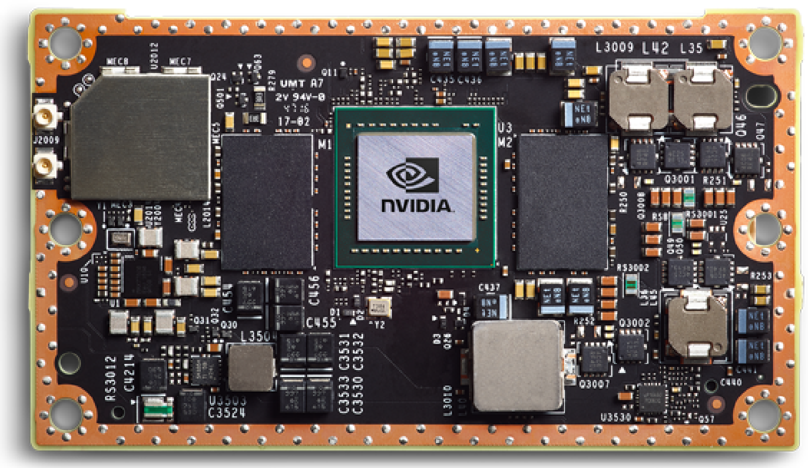
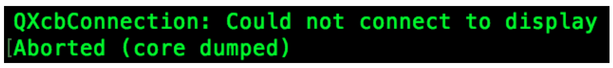


Figure 4.12.: PCB of the credit card-sized nVidia Jetson TX2 module [36].

The Jetson TX2 Board comes with a pre-installed 64-bit Ubuntu 16.04 LTS operating system—nVidia calls the most recent version as of this writing *Linux for Tegra* 28.2 or in short L4T 28.2—that makes it easy to establish an SSH session to the module and control it from another machine. However, as the Tegra TX2 is a 64-bit ARMv8 processor and there was no Anaconda/Miniconda distribution with support for that architecture available during this research project, the difficult part was the set-up of a Python environment with all packages required to run the Signal IDS. I was able to get it working by manually executing the commands listed in Appendix 5.2.

The first command installs necessary Ubuntu libraries that the systems needs to interact with a RTL-SDR device connected via USB, and the second one installs the dependencies that the Python frameworks Matplotlib and Pandas rely on and without which an installation is not even possible. Then, the Python3 setuptools are used to install pip, the package manager of Python. The next command installs the frameworks and libraries that are imported in the `signal_ids.py` Python script. In this step I tried installing Pandas via pip too, but the installation would not succeed and always ended with error messages. Thus, in the next command the APT package managing tool of Ubuntu is used to install Pandas for Python3.

At this point, all the software required by the Signal IDS had been installed but running it resulted in the error that can be seen in Figure 4.13.



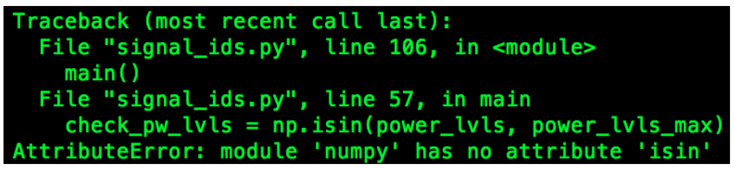
```
QXcbConnection: Could not connect to display
Aborted (core dumped)
```

Figure 4.13.: QXcbConnection error caused by Matplotlib not beeing able to connect to any active X server.

The cause of this error was that I was trying to run the Signal IDS on a remote machine, *i.e.*, the Jetson TX2 module, via an SSH connection, which resulted in Matplotlib not being able to connect to any active X server. One possible solution would have been to run a local X server and enable X11-forwarding for the SSH client, so that any plotting output would have been displayed on the local machine. However, another, simpler solution seemed preferable: adding two lines of code to the script that specify a different, non-interactive back-end for Matplotlib [37]. It is important to note that those lines have to be placed before the import of pylab.

```python
import matplotlib as mpl
mpl.use("Agg")
```

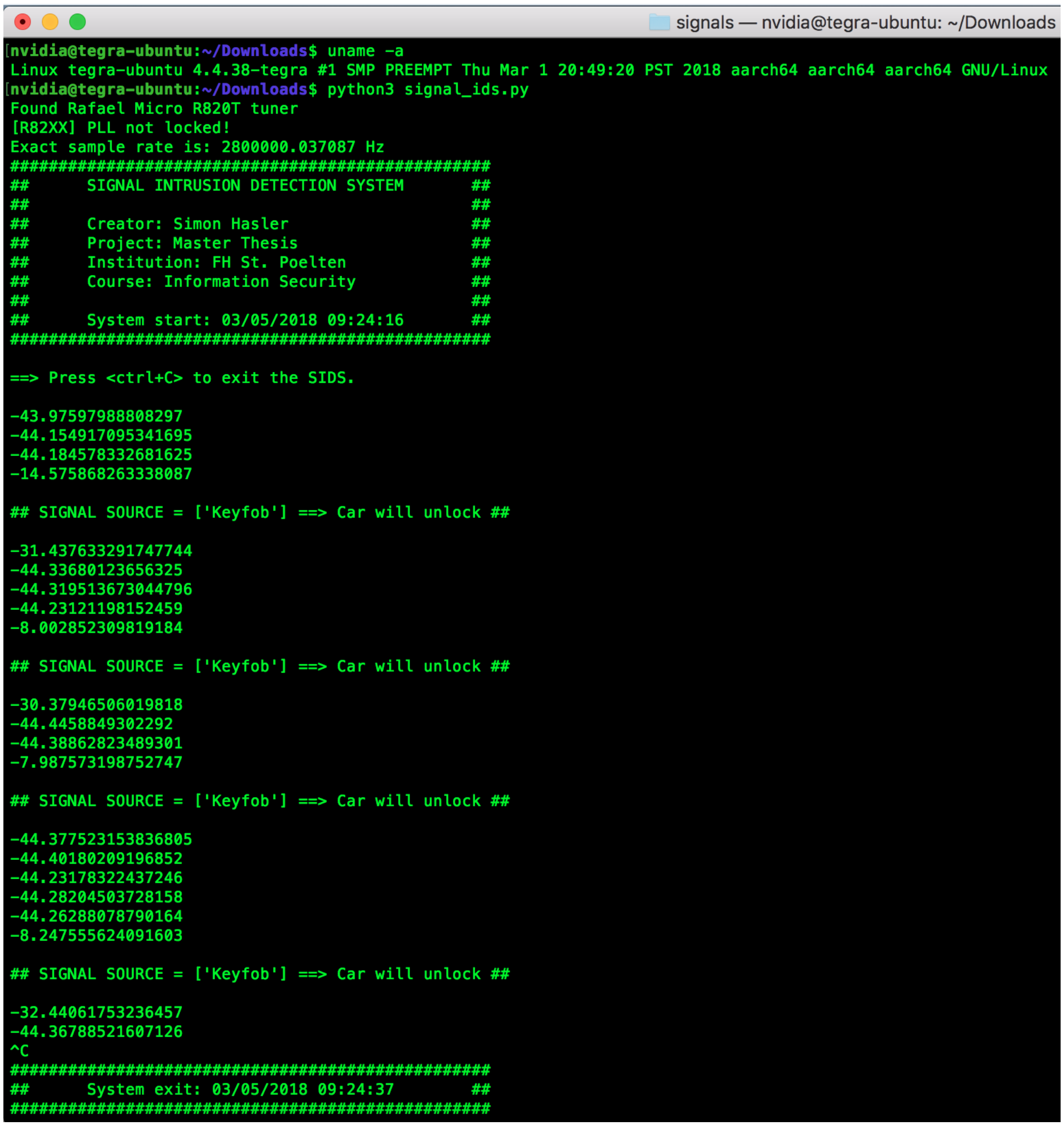
Another attempt to run the Signal IDS resulted in the error depicted in Figure 4.14.



```
Traceback (most recent call last):
  File "signal_ids.py", line 106, in <module>
    main()
  File "signal_ids.py", line 57, in main
    check_pw_lvls = np.isin(power_lvls, power_lvls_max)
AttributeError: module 'numpy' has no attribute 'isin'
```

Figure 4.14.: NumPy attribute error due to an outdated package version.

This one was easy to solve once I had figured out that pip had installed the outdated version 1.11.0 of the NumPy package, which was a problem insofar as that the `isin` function was added to NumPy in version 1.13.0. After upgrading the package to the latest version, which is 1.14.3 by the time of this writing, the Signal IDS could finally be run on the Tegra CPU of the nVidia Jetson TX2 module, as can be seen in Figure 4.15.



```
nvidia@tegra-ubuntu:~/Downloads$ uname -a
Linux tegra-ubuntu 4.4.38-tegra #1 SMP PREEMPT Thu Mar 1 20:49:20 PST 2018 aarch64 aarch64 aarch64 GNU/Linux
nvidia@tegra-ubuntu:~/Downloads$ python3 signal_ids.py
Found Rafael Micro R820T tuner
[R82XX] PLL not locked!
Exact sample rate is: 2800000.037087 Hz
##################################################
##      SIGNAL INTRUSION DETECTION SYSTEM       ##
##                                              ##
##      Creator: Simon Hasler                   ##
##      Project: Master Thesis                  ##
##      Institution: FH St. Poelten             ##
##      Course: Information Security            ##
##                                              ##
##      System start: 03/05/2018 09:24:16       ##
##################################################

==> Press <ctrl+C> to exit the SIDS.

-43.97597988808297
-44.154917095341695
-44.184578332681625
-14.575868263338087

## SIGNAL SOURCE = ['Keyfob'] ==> Car will unlock ##

-31.437633291747744
-44.33680123656325
-44.319513673044796
-44.23121198152459
-8.002852309819184

## SIGNAL SOURCE = ['Keyfob'] ==> Car will unlock ##

-30.37946506019818
-44.4458849302292
-44.38862823489301
-7.987573198752747

## SIGNAL SOURCE = ['Keyfob'] ==> Car will unlock ##

-44.377523153836805
-44.40180209196852
-44.23178322437246
-44.28204503728158
-44.26288078790164
-8.247555624091603

## SIGNAL SOURCE = ['Keyfob'] ==> Car will unlock ##

-32.44061753236457
-44.36788521607126
^C
##################################################
##      System exit: 03/05/2018 09:24:37        ##
##################################################
```

Figure 4.15.: The Signal IDS running on the Tegra CPU of the nVidia Jetson TX2 module.

Figure 4.16 is showing the final hardware setup consisting of the Aetina carrier board equipped with the nVidia Jetson TX2 module and a passive custom cooling solution right on top of it. The NooElec NESDR Nano 2+ RTL-SDR is plugged into one of the USB ports of the board to allow the Signal IDS to listen for signals and capture them.

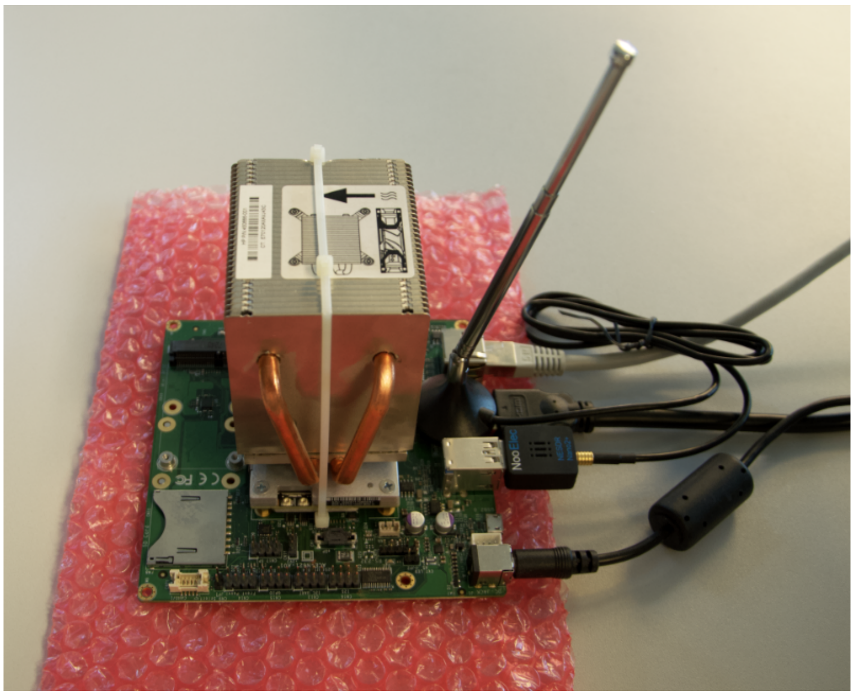


Figure 4.16.: The Aetina Jetson TX2 Carrier Board equipped with a custom cooling solution and the RTL-SDR plugged in.

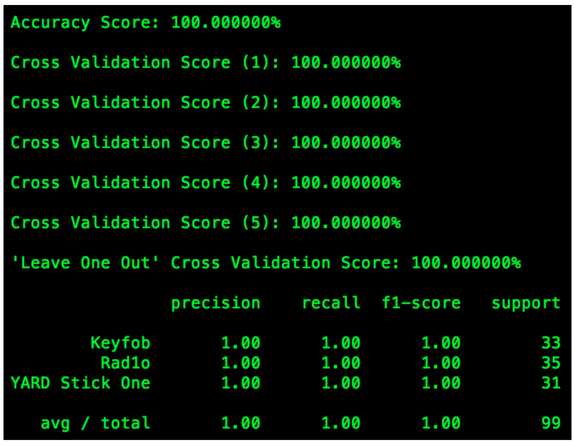
5. Results and Findings

The following section provides a detailed overview of the results obtained from the machine learning experiments in the previous chapter. It is explained how differing levels of accuracy can be achieved through careful feature selection, and how each of the two selected machine learning algorithms performed in regards to resource utilization. In the second part of this chapter an overview is given of various limitations to the machine learning approach that might occur under certain circumstances.

5.1. Prediction Accuracy and Performance

In the field of machine learning, a prediction result with an accuracy above 90% is usually considered fairly decent for classification tasks. If an algorithm is able to predict labels with an accuracy of 96-98%, it performs extremely well, although, ultimately, this always depends on the kind of data a machine learning model is applied to.

It was quite surprising to find that both the *k*-NN and the SVC algorithms resulted in a prediction precision of 100% for the provided dataset consisting of 300 data-points, two thirds of which were used for training and the other third for testing. Figure 5.1 shows the validation scores and the classification report for *k*-NN.



```
Accuracy Score: 100.000000%

Cross Validation Score (1): 100.000000%

Cross Validation Score (2): 100.000000%

Cross Validation Score (3): 100.000000%

Cross Validation Score (4): 100.000000%

Cross Validation Score (5): 100.000000%

'Leave One Out' Cross Validation Score: 100.000000%

                precision    recall  f1-score   support

        Keyfob       1.00      1.00      1.00        33
         Rad1o       1.00      1.00      1.00        35
YARD Stick One       1.00      1.00      1.00        31

   avg / total       1.00      1.00      1.00        99
```

Figure 5.1.: *k*-nearest Neighbor algorithm predicting labels with an accuracy of 100% if all 6 features of the data are used.

A result such as this always requires further investigation into whether it is truly possible for a given dataset, and what factors may contribute to achieving such exact predictions. Therefore, the first step was to create a pair-plot of all features present in the dataset and visualize how the data-points would scatter if only two selected features were used for classification, all of which can be seen in Figure 5.2. The bar charts visualize the occurrence of data samples for just one particular feature, and whether samples belonging to different classes overlap or not.

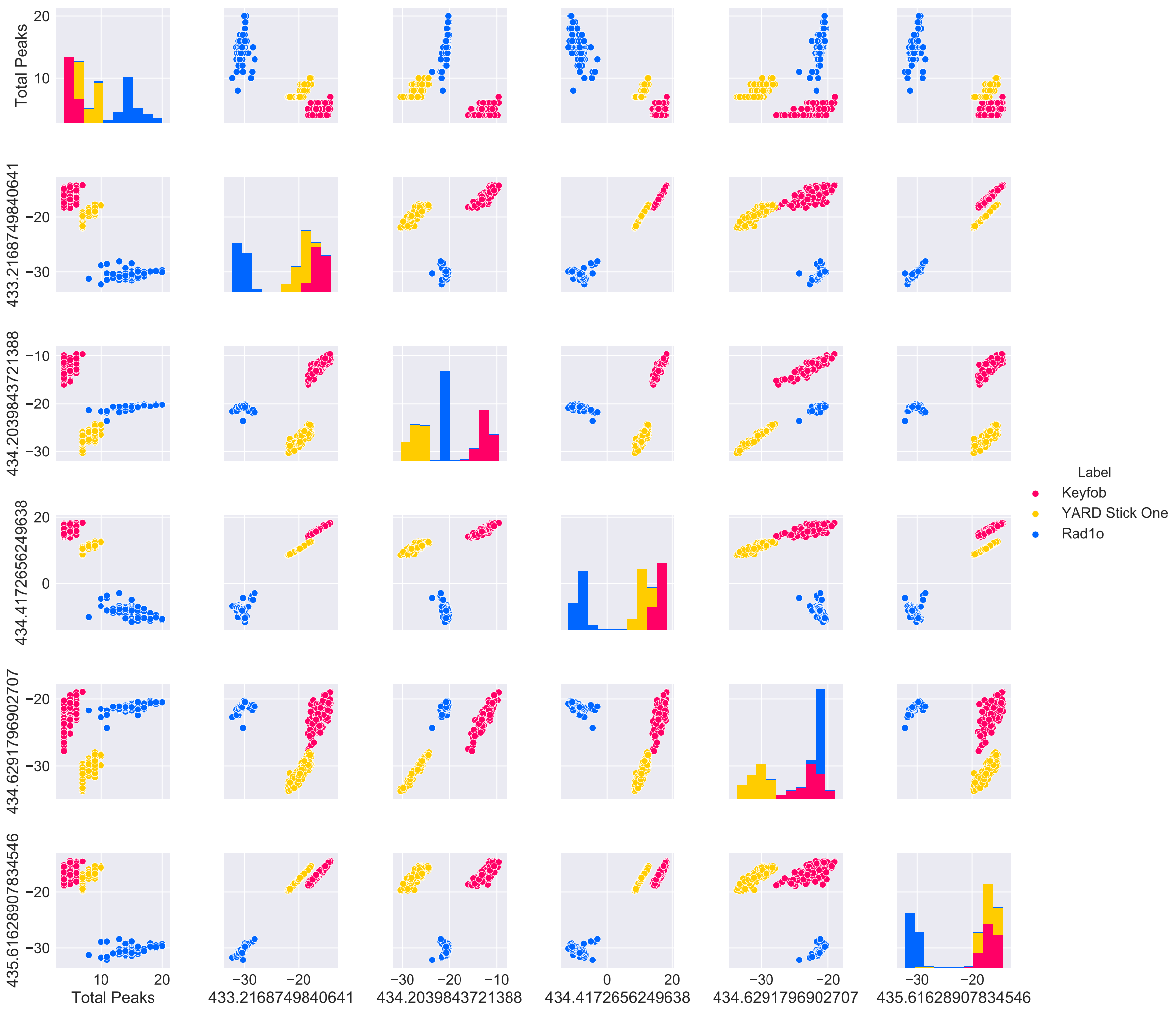


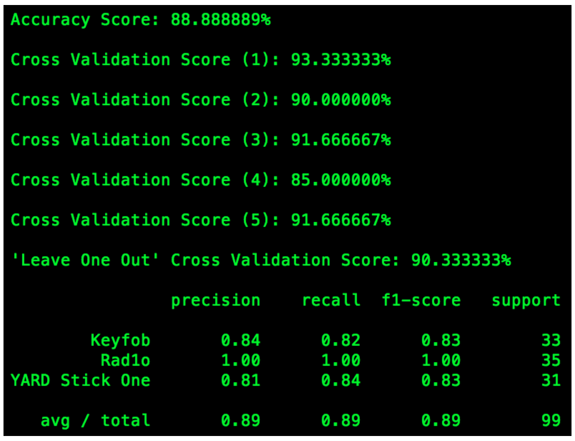
Figure 5.2.: Distribution of data from the dataset depending on which two features are put in relation to each other.

What becomes clear immediately is that the data-points of different classes (Keyfob in pink, YARD Stick One in yellow, and Rad1o in blue) are separated extremely well if the relationship of at least two different features is taken into account. Only for a small number of individual features, such as "Total Peaks" (bar chart in the first row/column) and the reference peak frequency of "433.2168749840641" MHz (bar chart in the second row/column), does overlapping of data-points from different classes seem to occur.

These findings were already giving strong hints as to why the result might have been a prediction precision of 100% for the data samples in the dataset, however, it was yet unconfirmed that there was no error in how I was using the machine learning algorithms in the `kNN_prediction.py` and `svm_prediction.py` scripts. If everything was correct, the prediction precision of both algorithms had to drop significantly if only one of those features with overlapping data-points was used for the classification, which was achieved by adjusting the following code line accordingly:

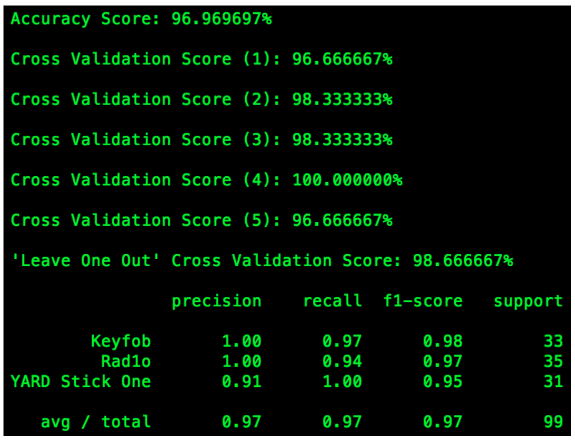
```
X = np.array(df.ix[:, 1:2]) # 1:7 for all 6 feature columns
```

And indeed, as Figure 5.3 is showing, the prediction precision drops to roughly 89% for *k*-NN if only the "433.2168749840641" feature is specified in the source code of the `kNN_prediction.py` script. The same is true for the SVC, only that this algorithm still produces a very good accuracy score of approximately 97%.



```
Accuracy Score: 88.888889%

Cross Validation Score (1): 93.333333%

Cross Validation Score (2): 90.000000%

Cross Validation Score (3): 91.666667%

Cross Validation Score (4): 85.000000%

Cross Validation Score (5): 91.666667%

'Leave One Out' Cross Validation Score: 90.333333%

                precision    recall  f1-score   support

        Keyfob       0.84      0.82      0.83        33
         Rad1o       1.00      1.00      1.00        35
YARD Stick One       0.81      0.84      0.83        31

   avg / total       0.89      0.89      0.89        99
```

(a) Accuracy and validation scores of *k*-NN

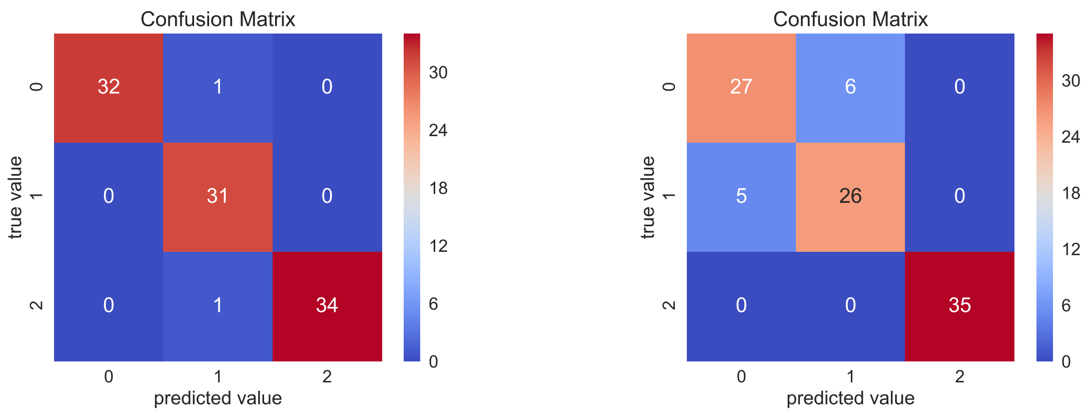


```
Accuracy Score: 96.969697%

Cross Validation Score (1): 96.666667%

Cross Validation Score (2): 98.333333%

Cross Validation Score (3): 98.333333%

Cross Validation Score (4): 100.000000%

Cross Validation Score (5): 96.666667%

'Leave One Out' Cross Validation Score: 98.666667%

                precision    recall  f1-score   support

        Keyfob       1.00      0.97      0.98        33
         Rad1o       1.00      0.94      0.97        35
YARD Stick One       0.91      1.00      0.95        31

   avg / total       0.97      0.97      0.97        99
```

(b) Accuracy and validation scores of SVC

Figure 5.3.: The prediction precision drops significantly in both algorithms if a single feature with overlapping data-points is selected for classification, but much less so for the SVC.

One last things that is very interesting to take a closer look at in regards to accuracy and prediction precision is which features, if used alone, cause what labels to be predicted wrongly. A heat-map of each classification result is the best way to visualize this and can be seen in Figure 5.4. The index `0` of the rows and columns marks the keyfob, `1` the YARD Stick One, and `2` the Rad1o.

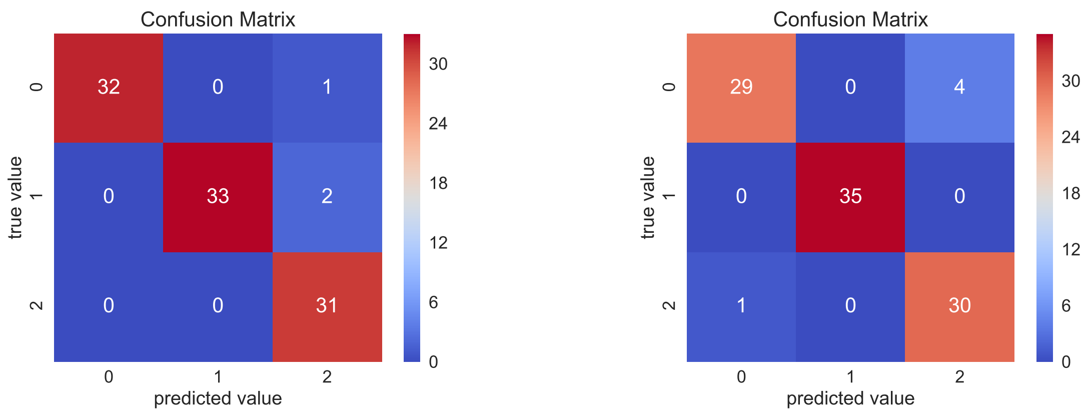


(a) Feature 1: "Total Peaks" (b) Feature 2: "433.2168749840641"

Figure 5.4.: Confusion matrices of the *k*-nearest Neighbors algorithm when applied to the "Total Peaks" and "433.2168749840641" features.

If the *k*-NN algorithm is allowed to use only the "Total Peaks" feature for its classifications of the data-points, 32 keyfob signals get classified correctly and only 1 is misclassified as YARD Stick One signal. All 31 YARD Stick One signals get classified correctly, and while 34 Rad1o signals get classified correctly too, 1 was wrongly classified as YARD Stick One signal. While this is a fairly good result, it does not hold if the "433.2168749840641" feature is the only one used for classification.

In this case, only 27 keyfob signals get labelled correctly, while 6 get labelled wrongly as YARD Stick One signal. It is almost the same for the latter device, with only 26 signals labelled correctly, and 5 labelled as keyfob signals. However, this feature does seem to allow all 35 Rad1o signals to be labelled correctly.



(a) Feature 1: "Total Peaks" (b) Feature 2: "433.2168749840641"

Figure 5.5.: Confusion matrices of the Support Vector Classifier with linear kernel when applied to the "Total Peaks" and "433.2168749840641" features.

Even with only the "Total Peaks" feature specified, the SVC is able to classify 32 keyfob and 33 YARD Stick One signals correctly, as can be seen in Figure 5.5. Only 1 keyfob signal gets misclassified as Rad1o signal, which is also the case for 2 signals from YARD Stick One. The "433.2168749840641" feature does increase the number of confusions for the SVC too, although not as much as for the *k*-NN algorithm. When this feature is used, the SVC classifies only 4 keyfob signals wrongly as Rad1o signals, and 1 signal of the latter device gets misclassified as keyfob signal.

What is interesting about these results is that the *k*-nearest Neighbors algorithm seems to get confused more easily by signals from the keyfob and the YARD Stick One, while the Support Vector Classifier, on the contrary, seems to have a harder time distinguishing correctly between signals from the keyfob and the Rad1o. In general, however, the SVC quite obviously produces more accurate results for the provided dataset than the *k*-NN algorithm does (8 versus 13 wrong predictions). I would like to stress again, though, that both algorithms are fully capable of a 100% prediction precision if they are allowed to use all 6 features of the data for the classification of unknown data samples. Even if one particular feature would cause confusion, looking at another or, better yet, several others that have clearly distributed datapoints, would result in accurately predicted labels.

While this concluded the research into the validation scores and the classification report, additionally it was also necessary to examine the performance of both machine learning algorithm since the impact on available resources was going to influence the choice of hardware for the final proof-of-concept. On Unix systems, measuring the processor time that the `kNN_prediction.py` or the `svm_prediction.py` script takes to import the dataset and perform all its computations up to the point where the respective algorithm makes its predictions could be done easily by adding the following lines to the source code:

```python
from timeit import time

# ...

start_time = time.clock() # put at the beginning of the main()
↪  function

# ...

end_time = time.clock() - start_time # put after 'predicted'
↪  variable is assigned the prediction result
print("k-NN duration: {}\n".format(end_time))
```

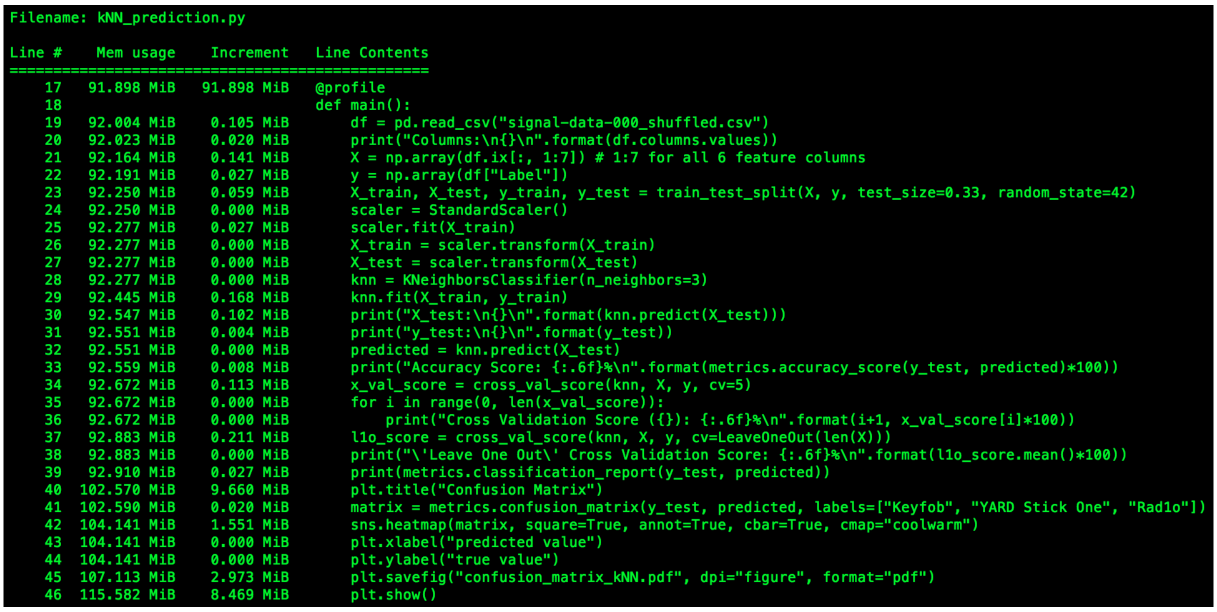
The processor times on the 2.2 GHz Intel Core i7-4770HQ processor of the MacBook Pro mid-2014 model are 0.005842000000000014 seconds and 0.005716000000000054 seconds for k-NN and SVC, respectively. It is clear that for a dataset the size of the one used in this research project a powerful CPU is not a necessity. This might, however, change fast once the dataset starts growing through the addition of newly captured signals.

Next, the Memory Profiler Python module revealed the memory consumption of each machine learning script. To use it, the function which shall be profiled needs to be decorated with the `profile` keyword:

```python
@profile
def main():
    # ...
```

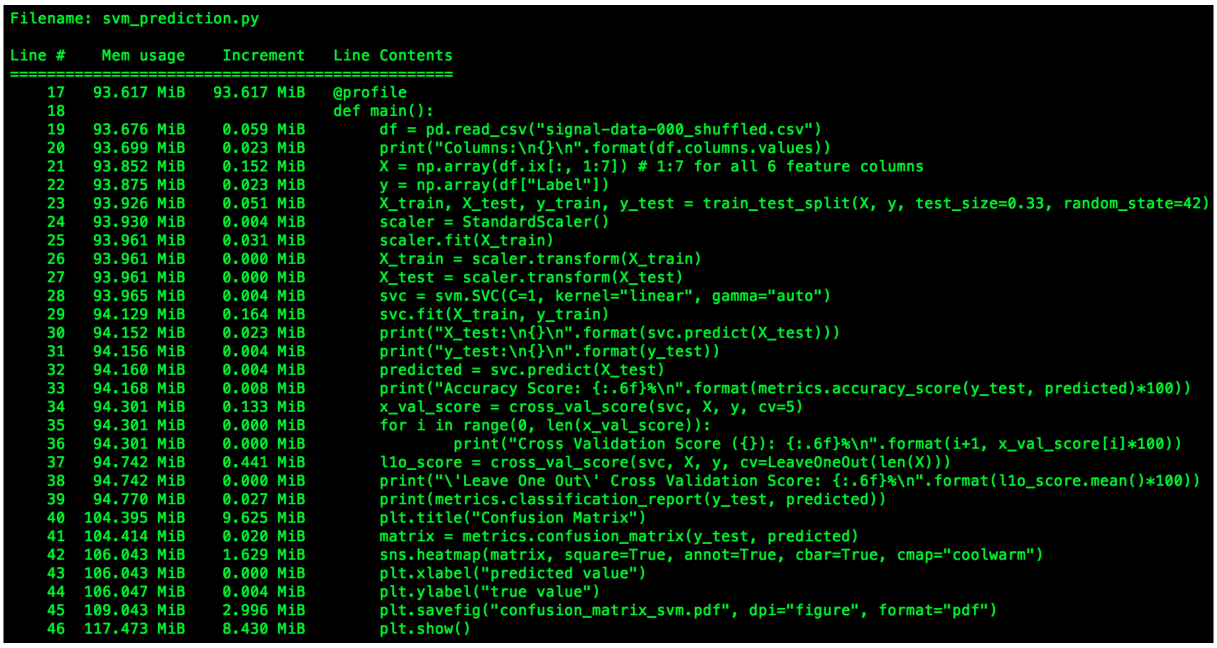
Then, the script could be executed using the following command in the terminal window:

```
$ python -m memory_profiler kNN_predict.py
```



```
Filename: kNN_prediction.py

Line #    Mem usage    Increment   Line Contents
================================================
    17   91.898 MiB   91.898 MiB   @profile
    18                             def main():
    19   92.004 MiB    0.105 MiB       df = pd.read_csv("signal-data-000_shuffled.csv")
    20   92.023 MiB    0.020 MiB       print("Columns:\n{}\n".format(df.columns.values))
    21   92.164 MiB    0.141 MiB       X = np.array(df.ix[:, 1:7]) # 1:7 for all 6 feature columns
    22   92.191 MiB    0.027 MiB       y = np.array(df["Label"])
    23   92.250 MiB    0.059 MiB       X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
    24   92.250 MiB    0.000 MiB       scaler = StandardScaler()
    25   92.277 MiB    0.027 MiB       scaler.fit(X_train)
    26   92.277 MiB    0.000 MiB       X_train = scaler.transform(X_train)
    27   92.277 MiB    0.000 MiB       X_test = scaler.transform(X_test)
    28   92.277 MiB    0.000 MiB       knn = KNeighborsClassifier(n_neighbors=3)
    29   92.445 MiB    0.168 MiB       knn.fit(X_train, y_train)
    30   92.547 MiB    0.102 MiB       print("X_test:\n{}\n".format(knn.predict(X_test)))
    31   92.551 MiB    0.004 MiB       print("y_test:\n{}\n".format(y_test))
    32   92.551 MiB    0.000 MiB       predicted = knn.predict(X_test)
    33   92.559 MiB    0.008 MiB       print("Accuracy Score: {:.6f}%\n".format(metrics.accuracy_score(y_test, predicted)*100))
    34   92.672 MiB    0.113 MiB       x_val_score = cross_val_score(knn, X, y, cv=5)
    35   92.672 MiB    0.000 MiB       for i in range(0, len(x_val_score)):
    36   92.672 MiB    0.000 MiB           print("Cross Validation Score ({}): {:.6f}%\n".format(i+1, x_val_score[i]*100))
    37   92.883 MiB    0.211 MiB       l1o_score = cross_val_score(knn, X, y, cv=LeaveOneOut(len(X)))
    38   92.883 MiB    0.000 MiB       print("\'Leave One Out\' Cross Validation Score: {:.6f}%\n".format(l1o_score.mean()*100))
    39   92.910 MiB    0.027 MiB       print(metrics.classification_report(y_test, predicted))
    40  102.570 MiB    9.660 MiB       plt.title("Confusion Matrix")
    41  102.590 MiB    0.020 MiB       matrix = metrics.confusion_matrix(y_test, predicted, labels=["Keyfob", "YARD Stick One", "Rad1o"])
    42  104.141 MiB    1.551 MiB       sns.heatmap(matrix, square=True, annot=True, cbar=True, cmap="coolwarm")
    43  104.141 MiB    0.000 MiB       plt.xlabel("predicted value")
    44  104.141 MiB    0.000 MiB       plt.ylabel("true value")
    45  107.113 MiB    2.973 MiB       plt.savefig("confusion_matrix_kNN.pdf", dpi="figure", format="pdf")
    46  115.582 MiB    8.469 MiB       plt.show()
```

Figure 5.6.: Memory consumption of the k-NN algorithm when classifying signal data using all six features.



```
Filename: svm_prediction.py

Line #    Mem usage    Increment   Line Contents
================================================
    17   93.617 MiB   93.617 MiB   @profile
    18                             def main():
    19   93.676 MiB    0.059 MiB       df = pd.read_csv("signal-data-000_shuffled.csv")
    20   93.699 MiB    0.023 MiB       print("Columns:\n{}\n".format(df.columns.values))
    21   93.852 MiB    0.152 MiB       X = np.array(df.ix[:, 1:7]) # 1:7 for all 6 feature columns
    22   93.875 MiB    0.023 MiB       y = np.array(df["Label"])
    23   93.926 MiB    0.051 MiB       X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
    24   93.930 MiB    0.004 MiB       scaler = StandardScaler()
    25   93.961 MiB    0.031 MiB       scaler.fit(X_train)
    26   93.961 MiB    0.000 MiB       X_train = scaler.transform(X_train)
    27   93.961 MiB    0.000 MiB       X_test = scaler.transform(X_test)
    28   93.965 MiB    0.004 MiB       svc = svm.SVC(C=1, kernel="linear", gamma="auto")
    29   94.129 MiB    0.164 MiB       svc.fit(X_train, y_train)
    30   94.152 MiB    0.023 MiB       print("X_test:\n{}\n".format(svc.predict(X_test)))
    31   94.156 MiB    0.004 MiB       print("y_test:\n{}\n".format(y_test))
    32   94.160 MiB    0.004 MiB       predicted = svc.predict(X_test)
    33   94.168 MiB    0.008 MiB       print("Accuracy Score: {:.6f}%\n".format(metrics.accuracy_score(y_test, predicted)*100))
    34   94.301 MiB    0.133 MiB       x_val_score = cross_val_score(svc, X, y, cv=5)
    35   94.301 MiB    0.000 MiB       for i in range(0, len(x_val_score)):
    36   94.301 MiB    0.000 MiB           print("Cross Validation Score ({}): {:.6f}%\n".format(i+1, x_val_score[i]*100))
    37   94.742 MiB    0.441 MiB       l1o_score = cross_val_score(svc, X, y, cv=LeaveOneOut(len(X)))
    38   94.742 MiB    0.000 MiB       print("\'Leave One Out\' Cross Validation Score: {:.6f}%\n".format(l1o_score.mean()*100))
    39   94.770 MiB    0.027 MiB       print(metrics.classification_report(y_test, predicted))
    40  104.395 MiB    9.625 MiB       plt.title("Confusion Matrix")
    41  104.414 MiB    0.020 MiB       matrix = metrics.confusion_matrix(y_test, predicted)
    42  106.043 MiB    1.629 MiB       sns.heatmap(matrix, square=True, annot=True, cbar=True, cmap="coolwarm")
    43  106.043 MiB    0.000 MiB       plt.xlabel("predicted value")
    44  106.047 MiB    0.004 MiB       plt.ylabel("true value")
    45  109.043 MiB    2.996 MiB       plt.savefig("confusion_matrix_svm.pdf", dpi="figure", format="pdf")
    46  117.473 MiB    8.430 MiB       plt.show()
```

Figure 5.7.: Memory consumption of the SVC algorithm when classifying signal data using all six features.

The memory consumption of each line of the machine learning scripts, as well as the total consumption can be seen in Figure 5.6 and Figure 5.7. While both algorithms seem to handle memory quite sparingly when working on the provided dataset, the *k*-NN manages with a slightly lesser amount of mebibyte (115.582 MiB versus 117.473 MiB). As with processor time, memory consumption may rise significantly with a growing dataset. This will need to be closely monitored when capturing large amounts of new signals and adding them to the dataset in a field-test or in follow-up projects.

5.2. Limitations to the Machine Learning Approach

The machine learning approach for the Signal IDS that I propose in Chapter 4 works surprisingly well for a proof-of-concept, however, it does come with certain limitations that need to be taken into consideration when adapting this approach for a real-world scenario.

For instance, if the Signal IDS observes new signals of devices other than the keyfob, the YARD Stick One, and the Rad1o, *i.e.*, those currently represented in the dataset, it is highly likely that eventually a signal from an unknown transmitter will exhibit characteristics far more similar to those of the keyfob than those of the hacking devices used in this work, thus resulting in a significant increase in false classification results. I therefore suggest to add captured signals of hacking devices other than the ones I used as soon as they become known and available, allowing the Signal IDS to learn their characteristics. Ultimately, the success of the system heavily depends on the maintenance and updating of the dataset it uses to train the machine learning algorithm on. In a way this is quite similar to what anti-virus software developers have to do by keeping the virus signatures up-to-date.

A problem that may arise when signal data of new transmitters is added to the dataset is that this new data may be very similar or even overlap with the data-points of signals from other transmitters already present in the dataset. As a consequence, the prediction accuracy of the machine learning algorithm may drop significantly. It is impossible to make a reliable prediction about this. The behavior of the machine learning algorithm as the dataset grows should be monitored closely, not just for this reason.

With a growing dataset it may even turn out that at some point the Support Vector Classifier is no longer the best-suited algorithm for the purpose of the Signal IDS. If its performance or accuracy should drop dramatically, I would suggest to, first, try the SVC with different kernels and other hyperparameters. And, second, start considering alternative classification algorithms such as the Stochastic Gradient Descent (SGD) Classifier, or attempt Kernel Approximation.

So far, the CPU performance of the proof-of-concept presented in this research work is excellent and even capturing signals observed at almost the same time does not present an issue. However, with a much larger dataset this might change. Eventually, it could become difficult for the Signal IDS to receive several signals at the same time and still make accurate predictions. This, too, needs to be monitored.

Overall, I would expect an extensive field-test to reveal several flaws or at least some room for improvements, as is natural in any research project, but I am convinced that my proof-of-concept provides a solid basis to build upon, and that is its true value.

6. Conclusion and Future Work

This chapter concludes the research work by giving an overview of all the steps that were required to implement a proof-of-concept for a Signal Intrusion Detection System. Additionally, suggestions are made on how future work could build upon the basis created in this research work to improve the final result, or attempt a different approach towards the same goal.

6.1. Conclusion

In this research work I have presented a machine learning approach for classifying digital signals based on their individual characteristics. I started out by using various transmitters in an attempt to replicate a capture and replay attack on signals from car keyfobs that has become known as "RollJam". An attack made possible by significant flaws in the KeeLoq rolling code scheme that is implemented in a vast number of cars manufactured in the last two decades.

To achieve the goal of creating a proof-of-concept for a Signal Intrusion Detection System capable of distinguishing between signals originating from different transmitters, first, a visual representation of the signals had to be produced to gain a better understanding of their individual characteristics. After inspecting a number of different possibilities, the Python programming language quickly turned out to be the best choice for this research project as it seemed best-suited for the task at hand due to its powerful libraries and scientific frameworks. In Section 4.1 of the Research Methodology I could already show that it is indeed possible to detect distinct features in a signal depending on the transmitter it originates from, thus answering the first research question.

With the knowledge gained by visualizing the signals it was possible to choose suitable features and build a dataset consisting of equal numbers of captured signals from each transmitter used in this research project. This dataset is then utilized to train two different machine learning algorithms for classification, namely the *k*-nearest Neighbors algorithm and the Support Vector Classifier, which I decided to try both to compare them and ensure that the best performing and most accurate one would be used in the final implementation. It turned out that the data-points of the signals in the dataset are distributed so well that both classification algorithms are able to predict labels of unknown signals with an astonishing accuracy of 100%. In Chapter 5, this is proven by presenting a visual representation of the individual features in the dataset, and it is furthermore shown that the accuracy would drop significantly if only features with overlapping data-points were used. Also, my findings show that the accuracy drop is far less dramatic in such scenarios with the Support Vector Classifier, thus qualifying it for the Signal Intrusion Detection System.

The final software implementation combines reading of the dataset, signal capture, feature extraction, data assembly, machine learning, and label prediction into one Python script that builds upon the insights gained into these procedures while working on the scripts I used during the research phase of this project. At the end of the Research Methodology I explain why the nVidia Jetson TX2 module is the perfect hardware choice for the final proof-of-concept and show that the aforementioned Python script can be run on it. The second and third research question are answered at the end of Chapter 4 as well by showing that the final Signal Intrusion Detection System is serving its purpose as desired, with minimal required user interaction.

In Section 5.2 of Chapter 5, several limitations to the machine learning approach, which it nevertheless has, are mentioned. On a final note, subsequently I propose another, quite different machine learning approach to the same research problem for future work.

6.2. Future Work

6.2.1. Machine Learning for Image Recognition with TensorFlow

The core concept of the machine learning approach presented in this work is to have a classification algorithm determine whether the values of pre-defined features of new data match those of the same features from known data. While the features have been carefully selected based on observations made during various signal comparisons, there is a chance that a computer would classify signals based on different features, or even a far greater number of features than a human programmer could specify. Therefore, I propose another, entirely different approach to a Signal IDS for future work that would allow classifications without pre-defined specifications: image recognition with *TensorFlow*, Google's open-source machine learning framework that is especially useful in the field of language and image processing.

One possible way to do this would be to, first, build a dataset of images by capturing signals, plotting every single one of them, and assigning labels to the plots. Then, a neural network can be modelled with TensorFlow, and through training on the dataset it will learn highly complex relationships between the specified labels and what it determines as features of the plotted signals. Once the model is trained, its performance and accuracy should be evaluated and, if need be, improved by fine-tuning the hyperparameters and increasing the size of the dataset. Ultimately, the process of the Signal IDS should be very similar to what is described in Section 4.2 of Chapter 4 in this research work: the dataset should be used to train the machine learning model, while newly created plots of unknown captured signals serve as test data on which the algorithm performs its classification. An excellent tutorial that uses the well-known Iris flowers dataset can be found in the "Develop" section of the TensorFlow website [38]. Another easy-to-follow tutorial for image recognition with TensorFlow that I would recommend is provided in the "Tutorials" section of the DataCamp community website [39] and focuses on classifying Belgian traffic signs.

One thing that may require special consideration is the visual representation of the signals, which should probably be reduced to nothing more than the signal itself so as to not bias the classification algorithm with additional numerical data. It is probably also necessary to plot all signals in the same color to keep the color from becoming a misleading feature for the classification algorithm. I leave it to other researchers and future work to figure out all that is required to implement a working proof-of-concept for an image recognition approach.

6.2.2. GPU Accelerated Computing with Python

While the ARMv8 processor of the Jetson TX2 module is achieving reasonable performance in machine learning on the small dataset used in this research project, the true power of the module lies in the CUDA capability of its 256-core Pascal GPU. If in future work using the same hardware platform performance should become an issue, which is likely going to happen when the size of the dataset starts to grow, I would recommend having a look at GPU usage in Python. As of this writing, there are two promising open-source projects aimed at GPU accelerated computing in Python: the *Numba* project [40, 41], which is supported by Anaconda, Inc., and *PyCUDA* [42]. Both may help to considerably reduce the training time of the machine learning model through massively parallelizing computations on the GPU to a degree that is not possible on any CPU, regardless of the number of its physical and virtual cores. However, this will also require rewriting parts of the scripts used in this research projects.

A. Python Source Code

A.1. Scripts for Capture, Replay, and Plotting

A.1.1. replay.py

```python
#!/Users/simonhasler/miniconda3/bin/python

from rflib import *
import bitstring
import sys

def conf_device(d):
    d.setMdmModulation(MOD_ASK_OOK)
    d.setFreq(434420000)
    d.setMdmSyncMode(0)
    d.setMdmDRate(4800)
    d.setMdmChanBW(60000)
    d.setMdmChanSpc(24000)
    d.setChannel(0)
    d.setMaxPower()
    d.lowball(1)

    #d.printRadioConfig()

def main():
    i = 0
    d = RfCat(idx=0)
    conf_device(d)

    print "RX"
    raw_signal = []
    while True:
        try:
            y, t = d.RFrecv(timeout=1, blocksize=255) # original
             ↪  blocksize is 400
            signal = y.encode("hex")
            print str(signal)
            raw_signal.append(signal)
            i+=1

            check = raw_input("({}) Type \"tx\" and press <enter>
             ↪  to start transmitting: ".format(i))
            if (check != ""):
                break

        except ChipconUsbTimeoutException:
            pass

        except KeyboardInterrupt:
            d.setModeIDLE()
            print "Bye!"
            sys.exit()
            break

    print "\nTX"
    for i in range(0, len(raw_signal)):
        check = raw_input("({}) Type \"bye\" and press <enter> to
         ↪  stop transmitting: ".format(i+1))
        if (check != ""):
            break
        key_packed = bitstring.BitArray(hex=raw_signal[i]).tobytes()
        d.makePktFLEN(len(key_packed))
        d.RFxmit(key_packed)

    d.setModeIDLE()


if __name__ == "__main__":
    main()
```

A.1.2. `plot_diffs.py`

```
#!/Users/simonhasler/miniconda3/bin/python

from rtlsdr import *
from pylab import *
import peakutils.peak
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

DEF_SIG_1     = "Keyfob Sig1"
DEF_SIG_2     = "Keyfob Sig2"
DEF_X_OFFSET = 0.005
DEF_Y_OFFSET = 10
DEF_PINK      = "#ff0066"
DEF_LILAC     = "#9933ff"
DEF_BLUE      = "#0066ff"
DEF_YELLOW    = "#ffcc00"
DEF_BLACK     = "#000000"

def main():
    i = 0
    j = 0
    sdr = RtlSdr()

    # Configure RTL-SDR device
    sdr.sample_rate = 2.8e6         # Hz
    sdr.center_freq = 434.42e6    # Hz
    sdr.gain = 25

    signals = []
    while True:
        raw_samples = sdr.read_samples(1024*1024)
        raw_power_lvls = 10*log10(var(raw_samples))
        print(raw_power_lvls)

        if raw_power_lvls >= -10:
            signals.append(raw_samples)
            i += 1

            if i == 2:
                # Set size of plot in inches and space between
                 ↪  subplots
                plt.subplots(figsize=[9.84, 9.45])
                plt.subplots_adjust(hspace=0.4)

                # Plot signal of first transmitter
                plt.subplot(3, 1, 1)
                plt.title("Power Spectral Density Comparison")
                Pxx_1, freqs_1 = plt.psd(signals[0], NFFT=2048,
                 ↪  Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6,
                 ↪  scale_by_freq=True, color=DEF_PINK,
                 ↪  label=DEF_SIG_1)
                plt.legend()
                plt.xlabel("Frequency (MHz)")
                plt.ylabel("PSD (dB/Hz)")

                # Plot signal of second transmitter
                plt.subplot(3, 1, 2)
                Pxx_2, freqs_2 = plt.psd(signals[1], NFFT=2048,
                 ↪  Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6,
                 ↪  scale_by_freq=True, color=DEF_LILAC,
                 ↪  label=DEF_SIG_2)
                plt.legend()
                plt.xlabel("Frequency (MHz)")
                plt.ylabel("PSD (dB/Hz)")

                # Plot overlay of both received signals
                plt.subplot(3, 1, 3)
                plt.psd(signals[0], NFFT=2048,
                 ↪  Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6,
                 ↪  scale_by_freq=True, color=DEF_PINK,
                 ↪  label=DEF_SIG_1)
                plt.psd(signals[1], NFFT=2048,
                 ↪  Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6,
                 ↪  scale_by_freq=True, color=DEF_LILAC,
                 ↪  label=DEF_SIG_2)
                plt.legend()
                plt.xlabel("Frequency (MHz)")
                plt.ylabel("PSD (dB/Hz)")

                plt.savefig("sig_comparison.pdf", dpi="figure",
                 ↪  format="pdf")
                plt.show()

                # Detect peaks in signals and print their
                 ↪  coordinates
                power_lvls_1 = 10*log10(Pxx_1/(sdr.sample_rate/1e6)⌋
                 ↪  )+10*log10(8/3)
                power_lvls_2 = 10*log10(Pxx_2/(sdr.sample_rate/1e6)⌋
                 ↪  )+10*log10(8/3)
```

```
                indexes_1 = peakutils.indexes(power_lvls_1,
                 ↪   thres=0.25, min_dist=10)
                indexes_2 = peakutils.indexes(power_lvls_2,
                 ↪   thres=0.25, min_dist=10)
                print("\nSIGNAL PEAKS\n")
                print("Peaks in Signal 1\nX: {}\n\nY:
                 ↪   {}\n".format(freqs_1[indexes_1],
                 ↪   power_lvls_1[indexes_1]))
                print("Peaks in Signal 2\nX: {}\n\nY:
                 ↪   {}\n".format(freqs_2[indexes_2],
                 ↪   power_lvls_2[indexes_2]))
                power_lvls_1_max = [i for i in
                 ↪   power_lvls_1[indexes_1] if i >= -20]
                power_lvls_2_max = [i for i in
                 ↪   power_lvls_2[indexes_2] if i >= -20]
                check_1 = np.isin(power_lvls_1, power_lvls_1_max)
                check_2 = np.isin(power_lvls_2, power_lvls_2_max)
                indexes_1_max = np.where(check_1)
                indexes_2_max = np.where(check_2)
                print("Highest Peaks in Signal 1:\nX: {}\n\nY:
                 ↪   {}\n".format(freqs_1[indexes_1_max],
                 ↪   power_lvls_1[indexes_1_max]))
                print("Highest Peaks in Signal 2:\nX: {}\n\nY:
                 ↪   {}\n".format(freqs_2[indexes_2_max],
                 ↪   power_lvls_2[indexes_2_max]))

                # Create vectors from single coordinates
                vectors_sig_1 =
                 ↪   np.column_stack((freqs_1[indexes_1_max],
                 ↪   power_lvls_1[indexes_1_max]))
                vectors_sig_2_cmp =
                 ↪   np.column_stack((freqs_2[indexes_1_max],
                 ↪   power_lvls_2[indexes_1_max]))
```

```python
    vectors_sig_2_real = np.column_stack((freqs_2[indexes_2_max], power_lvls_2[indexes_2_max]))
    print("Vector 2 reference frequency comparison:\n{}\n".format(vectors_sig_2_cmp))
    print("Vector 2 real peak frequencies:\n{}\n".format(vectors_sig_2_real))

    plt.subplots(figsize=[9.84, 5.90])
    plt.subplots_adjust(hspace=0.4, wspace=0.4)

    # Zoom in on overlay of most significant peaks
    for i in range(0, min(len(vectors_sig_1), len(vectors_sig_2_real))):
        plt.subplot(2, 3, i+1)
        plt.suptitle(r"Most significant Peaks from $f_c-f_s/2$ to $f_c+f_s/2$")
        plt.title("Peak {}".format(i+1))
        plt.psd(signals[0], NFFT=2048, Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6, scale_by_freq=True, color=DEF_PINK)
        plt.psd(signals[1], NFFT=2048, Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6, scale_by_freq=True, color=DEF_LILAC)
        power_mean = np.mean(np.column_stack((vectors_sig_1[i][1], vectors_sig_2_cmp[i][1])))
        plt.xlim(vectors_sig_1[i][0]-DEF_X_OFFSET, vectors_sig_1[i][0]+DEF_X_OFFSET)
        plt.ylim(power_mean-DEF_Y_OFFSET, power_mean+DEF_Y_OFFSET)
```

```
                j=i
                if (min(len(vectors_sig_1),
                ↪   len(vectors_sig_2_real)) ==
                ↪   len(vectors_sig_1)): # vector_sig_1 is the
                ↪   smaller list
                    while (j < len(vectors_sig_2_real)-1 and
                    ↪   (vectors_sig_1[i][0] <
                    ↪   vectors_sig_2_real[j][0]-DEF_X_OFFSET*2
                    ↪   or vectors_sig_1[i][0] > vectors_sig_2_⌋
                    ↪   real[j][0]+DEF_X_OFFSET*2)):
                        j+=1
                    plt.axvline(x=vectors_sig_1[i][0],
                    ↪   linewidth=1, color=DEF_BLACK)
                    plt.axvline(x=vectors_sig_2_real[j][0],
                    ↪   linewidth=1, color=DEF_BLACK)
                    print("i = {}\nj = {}\nFREQ_1: {}\nFREQ_2:
                    ↪   {}\n\n".format(i, j,
                    ↪   vectors_sig_1[i][0],
                    ↪   vectors_sig_2_real[j][0])) # DEBUG
                    ↪   OUTPUT
                elif (min(len(vectors_sig_1),
                ↪   len(vectors_sig_2_real)) ==
                ↪   len(vectors_sig_2_real)): #
                ↪   vector_sig_2_real is the smaller signal
                    while (j < len(vectors_sig_1)-1 and
                    ↪   (vectors_sig_2_real[i][0] <
                    ↪   vectors_sig_1[j][0]-DEF_X_OFFSET*2 or
                    ↪   vectors_sig_2_real[i][0] >
                    ↪   vectors_sig_1[j][0]+DEF_X_OFFSET*2)):
                        j+=1
                    plt.axvline(x=vectors_sig_1[j][0],
                    ↪   linewidth=1, color=DEF_BLACK)
```

```
                        plt.axvline(x=vectors_sig_2_real[i][0],
                         ↪  linewidth=1, color=DEF_BLACK)
                        print("FREQ_1: {}\nFREQ_2:
                         ↪  {}\n\n".format(vectors_sig_1[j][0],
                         ↪  vectors_sig_2_real[i][0])) # DEBUG
                         ↪  OUTPUT
                    plt.xlabel("MHz")
                    plt.ylabel("dB/Hz)")

                # Show plot and save result as PDF
                plt.savefig("peak_comparison.pdf", dpi="figure",
                 ↪  format="pdf")
                plt.show()

                break


if __name__ == "__main__":
    main()
```

A.2. Script for Data Accumulation

A.2.1. build_dataset.py

```
#!/Users/simonhasler/miniconda3/bin/python

from rtlsdr import *
from pylab import *
import sys
import os.path
import csv
import peakutils.peak
import matplotlib.pyplot as plt

```

```
DEF_SIG = "" # "Keyfob", "YARD Stick One", "Rad1o"

def write_to_csv(header, data):
    file_exists = os.path.isfile("signal-data-000.csv")

    with open("signal-data-000.csv", "a") as csv_file:
        writer = csv.DictWriter(csv_file, fieldnames=header)
        if not file_exists:
            writer.writeheader()
        writer.writerow(data)


def main():
    i = 0
    j = 0
    sdr = RtlSdr()

    # Configure RTL-SDR device
    sdr.sample_rate = 2.8e6         # Hz
    sdr.center_freq = 434.42e6     # Hz
    sdr.gain = 25

    signals = []
    while True:
        try:
            raw_samples = sdr.read_samples(1024*1024)
            raw_power_lvls = 10*log10(var(raw_samples))
            print(raw_power_lvls)

            if raw_power_lvls >= -10:
                signals.append(raw_samples)
                i+=1

```

```python
            # i1 = reference signal (not written as data row)
            DEF_SIG = "Keyfob" # i2 to i51 = 50
            if (i >= 52):
                DEF_SIG = "YARD Stick One" # i52 to i101 = 50
            if (i >= 102):
                DEF_SIG = "Rad1o" # i102 to i151 = 50
            if (i == 152):
                break

            if (i == 2): # Sig0, Sig1
                Pxx_1, freqs_1 = plt.psd(signals[i-2],
                 ↪  NFFT=2048, Fs=sdr.sample_rate/1e6,
                 ↪  Fc=sdr.center_freq/1e6, scale_by_freq=True)
                Pxx_2, freqs_2 = plt.psd(signals[i-1],
                 ↪  NFFT=2048, Fs=sdr.sample_rate/1e6,
                 ↪  Fc=sdr.center_freq/1e6, scale_by_freq=True)
            elif (i > 2): # Sig3 and following
                Pxx_1, freqs_1 = plt.psd(signals[0], NFFT=2048,
                 ↪  Fs=sdr.sample_rate/1e6,
                 ↪  Fc=sdr.center_freq/1e6, scale_by_freq=True)
                Pxx_2, freqs_2 = plt.psd(signals[i-1],
                 ↪  NFFT=2048, Fs=sdr.sample_rate/1e6,
                 ↪  Fc=sdr.center_freq/1e6, scale_by_freq=True)

            if (i >= 2):
                # Detect peaks in signals
                power_lvls_1 = 10*log10(Pxx_1/(sdr.sample_rate/⌟
                 ↪  1e6))+10*log10(8/3)
                power_lvls_2 = 10*log10(Pxx_2/(sdr.sample_rate/⌟
                 ↪  1e6))+10*log10(8/3)
                indexes_1 = peakutils.indexes(power_lvls_1,
                 ↪  thres=0.25, min_dist=10)
```

```
                    indexes_2 = peakutils.indexes(power_lvls_2,
                    ↪  thres=0.25, min_dist=10)
                    power_lvls_1_max = [i for i in
                    ↪  power_lvls_1[indexes_1] if i >= -20]
                    power_lvls_2_max = [i for i in
                    ↪  power_lvls_2[indexes_2] if i >= -20]
                    check_1 = np.isin(power_lvls_1,
                    ↪  power_lvls_1_max)
                    check_2 = np.isin(power_lvls_2,
                    ↪  power_lvls_2_max)
                    indexes_1_max = np.where(check_1)
                    indexes_2_max = np.where(check_2)

                    # Create vectors from single coordinates
                    vectors_sig_1 =
                    ↪  np.column_stack((freqs_1[indexes_1_max],
                    ↪  power_lvls_1[indexes_1_max]))
                    vectors_sig_2_cmp =
                    ↪  np.column_stack((freqs_2[indexes_1_max],
                    ↪  power_lvls_2[indexes_1_max]))
                    vectors_sig_2_real =
                    ↪  np.column_stack((freqs_2[indexes_2_max],
                    ↪  power_lvls_2[indexes_2_max]))

                    # Build data row and write it to CSV file
                    data = []
                    header = []
                    header.append("Label")
                    header.append("Total Peaks")
                    data.append(DEF_SIG)
                    data.append(len(vectors_sig_2_real))
                    for j in range(0, len(vectors_sig_2_cmp)):
                        header.append(str(vectors_sig_2_cmp[j][0]))
                        data.append(vectors_sig_2_cmp[j][1])
                    data_row = dict(zip(header, data))
                    write_to_csv(header, data_row)

                    print("## Signal data {} written for {}
                     ↪  ##\n".format(i-1, DEF_SIG)) # DEBUG OUTPUT

        except KeyboardInterrupt:
            print("Bye!")
            sys.exit()
            break


if __name__ == "__main__":
    main()
```

A.3. Scripts for Classification

A.3.1. kNN_prediction.py

```
#!/Users/simonhasler/miniconda3/bin/python

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from sklearn.cross_validation import cross_val_score, LeaveOneOut
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set(font_scale=1.4)

DEF_PINK      = "#ff0066"
DEF_BLUE      = "#0066ff"
```

```
DEF_YELLOW    = "#ffcc00"

def main():
    # Load data
        df = pd.read_csv("signal-data-000_shuffled.csv")
        print("Columns:\n{}\n".format(df.columns.values))
        #print("First 10 rows of the
         ↪  dataset:\n{}\n".format(df.head(10))) # DEBUG OUTPUT

        # Plot dataset and save result as PDF
       sns.pairplot(df, hue="Label", palette={"Keyfob": DEF_PINK,
        ↪  "YARD Stick One": DEF_YELLOW, "Rad1o": DEF_BLUE})
        plt.savefig("dataset.pdf", dpi="figure", format="pdf")
        plt.show()

        # Create feature matrix X [n_samples, n_features] and
         ↪  target array y [n_samples]
        X = np.array(df.ix[:, 2:3]) # 1:7 for all 6 feature columns
        y = np.array(df["Label"])

        # Split into train and test data
        X_train, X_test, y_train, y_test = train_test_split(X, y,
         ↪  test_size=0.33, random_state=42)

        # Scale features so that all of them can be uniformly
         ↪  evaluated
        scaler = StandardScaler()
        scaler.fit(X_train)
        X_train = scaler.transform(X_train)
        X_test = scaler.transform(X_test)

        # Instantiate class of learning model (k = 3)
        knn = KNeighborsClassifier(n_neighbors=3)

        # Fit the model to the data
        knn.fit(X_train, y_train)

        predicted = knn.predict(X_test)

        # Predict the label of X_test
        print("X_test:\n{}\n".format(predicted))

        # Print y_test to check the results
        print("y_test:\n{}\n".format(y_test))

        # Print accuracy and cross validation scores
        print("Accuracy Score:
         ↪  {:.6f}%\n".format(metrics.accuracy_score(y_test,
         ↪  predicted)*100))
        x_val_score = cross_val_score(knn, X, y, cv=5)
        for i in range(0, len(x_val_score)):
                print("Cross Validation Score ({}):
                 ↪  {:.6f}%\n".format(i+1, x_val_score[i]*100))
        l1o_score = cross_val_score(knn, X, y,
         ↪  cv=LeaveOneOut(len(X)))
        print("\'Leave One Out\' Cross Validation Score:
         ↪  {:.6f}%\n".format(l1o_score.mean()*100))

        # Print the classification report of y_test and predicted
        print(metrics.classification_report(y_test, predicted))

        # Print the confusion matrix
        plt.title("Confusion Matrix")
        matrix = metrics.confusion_matrix(y_test, predicted,
         ↪  labels=["Keyfob", "YARD Stick One", "Rad1o"])
        sns.heatmap(matrix, square=True, annot=True, cbar=True,
         ↪  cmap="coolwarm")
        plt.xlabel("predicted value")
        plt.ylabel("true value")
        plt.savefig("confusion_matrix_kNN.pdf", dpi="figure",
         ↪  format="pdf")
        plt.show()


if __name__ == "__main__":
    main()
```

A.3.2. svm_prediction.py

```
#!/Users/simonhasler/miniconda3/bin/python

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn import svm
from sklearn import metrics
from sklearn.cross_validation import cross_val_score, LeaveOneOut
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set(font_scale=1.4)

DEF_PINK     = "#ff0066"
DEF_BLUE     = "#0066ff"
DEF_YELLOW   = "#ffcc00"

def main():
    # Load dataset
    df = pd.read_csv("signal-data-000_shuffled.csv")
    print("Columns:\n{}\n".format(df.columns.values))
    #print("First 10 rows of the
    ↪  dataset:\n{}\n".format(df.head(10))) # DEBUG OUTPUT

    # Plot dataset and save result as PDF
    sns.pairplot(df, hue="Label", palette={"Keyfob": DEF_PINK,
    ↪  "YARD Stick One": DEF_YELLOW, "Rad1o": DEF_BLUE})
    plt.savefig("dataset.pdf", dpi="figure", format="pdf")
    plt.show()

    # Create design matrix X and target vector y
    X = np.array(df.ix[:, 2:3]) # 1:7 for all 6 feature columns
    y = np.array(df["Label"])

    # Split into train and test data
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    ↪  test_size=0.33, random_state=42)

    # Scale features so that all of them can be uniformly evaluated
    scaler = StandardScaler()
    scaler.fit(X_train)
    X_train = scaler.transform(X_train)
    X_test = scaler.transform(X_test)

    # Instantiate class of learning model
    svc = svm.SVC(C=1, kernel="linear", gamma="auto")

    # Fit the model to the data
    svc.fit(X_train, y_train)

    predicted = svc.predict(X_test)

    # Predict the label of X_test
    print("X_test:\n{}\n".format(predicted))
```

```python

    # Print y_test to check the results
    print("y_test:\n{}\n".format(y_test))

    # Print accuracy and cross validation scores
    print("Accuracy Score:
    ↪  {:.6f}%\n".format(metrics.accuracy_score(y_test,
    ↪  predicted)*100))
    x_val_score = cross_val_score(svc, X, y, cv=5)
    for i in range(0, len(x_val_score)):
        print("Cross Validation Score ({}): {:.6f}%\n".format(i+1,
        ↪  x_val_score[i]*100))
    l1o_score = cross_val_score(svc, X, y, cv=LeaveOneOut(len(X)))
    print("\'Leave One Out\' Cross Validation Score:
    ↪  {:.6f}%\n".format(l1o_score.mean()*100))

    # Print the classification report of y_test and predicted
    print(metrics.classification_report(y_test, predicted))

    # Print the confusion matrix
    plt.title("Confusion Matrix")
    matrix = metrics.confusion_matrix(y_test, predicted)
    sns.heatmap(matrix, square=True, annot=True, cbar=True,
    ↪  cmap="coolwarm")
    plt.xlabel("predicted value")
    plt.ylabel("true value")
    plt.savefig("confusion_matrix_svm.pdf", dpi="figure",
    ↪  format="pdf")
    plt.show()


if __name__ == "__main__":
    main()
```

A.4. Script for the Signal Intrusion Detection System

A.4.1. signal_ids.py

```
#!/Users/simonhasler/miniconda3/bin/python

from rtlsdr import *
from pylab import *
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn import svm
import peakutils.peak
import time
import pandas as pd
import numpy as np

def main():
    sdr = RtlSdr()

    # Configure RTL-SDR device
    sdr.sample_rate = 2.8e6         # Hz
    sdr.center_freq = 434.42e6    # Hz
    sdr.gain = 25

    # Load data, replace device specific labels, and retrieve peak
     ↪  frequencies of reference signal
    df = pd.read_csv("signal-data-000_shuffled.csv")
    df["Label"] = df["Label"].replace(["YARD Stick One", "Rad1o"],
     ↪  "Other")
    df.to_csv("signal-data-000_real.csv", index=False)
    df = pd.read_csv("signal-data-000_real.csv")
    ref_freqs = np.array(df.columns.values)

    print("###################################################\n"
        "##\tSIGNAL INTRUSION DETECTION SYSTEM\t##\n"
        "##\t\t\t\t\t\t##\n"
        "##\tCreator: Simon Hasler\t\t\t##\n"
        "##\tProject: Master Thesis\t\t\t##\n"
        "##\tInstitution: FH St. Poelten\t\t##\n"
        "##\tCourse: Information Security\t\t##\n"
        "##\t\t\t\t\t\t##\n"
        "##\tSystem start: {} {}\t##\n"
        "##################################################\n\n"
        "==> Press <ctrl+C> to exit the
            ↪ SIDS.\n".format(time.strftime("%d/%m/%Y"),
            ↪ time.strftime("%I:%M:%S")))

    while True:
        try:
            raw_sample = sdr.read_samples(1024*1024)
            raw_power_lvls = 10*log10(var(raw_sample))
            print(raw_power_lvls)

            if raw_power_lvls >= -15:
                signal = raw_sample

                Pxx, freqs = plt.psd(signal, NFFT=2048,
                    ↪ Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6,
                    ↪ scale_by_freq=True)

                # Retrieve power levels at x coordinates of
                 ↪ reference peaks
                power_lvls = 10*log10(Pxx/(sdr.sample_rate/1e6))+10
                 ↪ *log10(8/3)
                indexes = peakutils.indexes(power_lvls, thres=0.25,
                 ↪ min_dist=10)
```

```python
                power_lvls_max = [i for i in power_lvls[indexes] if i >= -20]
                check_pw_lvls = np.isin(power_lvls, power_lvls_max)
                check_freqs = np.isin(freqs, np.float64(ref_freqs[2:7]))
                p_indexes_max = np.where(check_pw_lvls)
                f_indexes_max = np.where(check_freqs)

                # Create vectors from single coordinates
                vectors_sig_real = np.column_stack((freqs[p_indexes_max], power_lvls[p_indexes_max]))

                # Build data row for newly cpatured signal
                data = []
                header = []
                header.append("Total Peaks")
                data.append(len(vectors_sig_real))
                for j in range(0, len(power_lvls[f_indexes_max])):
                    header.append(str(ref_freqs[2:7][j]))
                    data.append(power_lvls[f_indexes_max][j])
                data_row = dict(zip(header, data))

                # Create design matrix X and target vector y
                X_train = np.array(df.ix[:, 1:7])
                y_train = np.array(df["Label"])
                X_test = np.array([list(data_row.values())])

                # Scale features so that all of them can be uniformly evaluated
                scaler = StandardScaler()
                scaler.fit(X_train)
                X_train = scaler.transform(X_train)
                X_test = scaler.transform(X_test)

                # Instantiate class of learning model and fit the model to the data
                svc = svm.SVC(C=1, kernel="linear", gamma="auto")
                svc.fit(X_train, y_train)
                predicted = svc.predict(X_test)

                if (predicted == "Keyfob"):
                    car_do = "unlock"
                else:
                    car_do = "stay closed"
                print("\n## SIGNAL SOURCE = {} ==> Car will {} ##\n".format(predicted, car_do))

        except KeyboardInterrupt:
            print("\n##################################################\n"
                "##\tSystem exit: {} {}\t##\n"
                "##################################################".format(time.strftime("%d/%m/%Y"), time.strftime("%I:%M:%S")))
            sys.exit()
            break


if __name__ == "__main__":
    main()
```

A.5. IPython and Terminal Commands

A.5.1. Commands for Shuffling the Rows of the Dataset

```
import pandas as pd
```

```
df = pd.read_csv("signal-data-000.csv")

ds = df.sample(frac=1)

ds.to_csv("signal-data-000_shuffled.csv", index=None)
```

A.5.2. Commands for Installation of the Python Requirements on L4T 28.2

```
$ sudo apt-get install libusb-1.0-0-dev librtlsdr-dev libatlas-base-dev
gfortran

$ sudo apt-get build-dep python-matplotlib python-pandas

$ sudo easy_install3 -U pip

$ pip install numpy scipy matplotlib scikit-learn PeakUtils --user

$ sudo apt-get install python3-pandas

$ pip install numpy --upgrade --user
```

List of Figures

Bibliography

[1] F. D. Garcia, D. Oswald, P. Pavlidès, and T. Kasper, “Lock It and Still Lose It—On the (In)Security of Automotive Remote Keyless Entry Systems,” in *25th USENIX Security Symposium (USENIX Security 16)*. Montreal, Canada: USENIX Association, August 2016.

[2] A. Greenberg, “A New Wireless Hack Can Unlock 100 Million Volkswagens,” August 2016. [Online]. Available: https://www.wired.com/2016/08/oh-good-new-hack-can-unlock-100-million-volkswagens/

[3] R. Verdult, F. D. Garcia, and J. Balasch, “Gone in 360 Seconds - Hijacking with Hitag2,” in *21st USENIX Security Symposium (USENIX Security 12)*. Montreal, Canada: USENIX Association, August 2012.

[4] A. Francillon, B. Danev, and S. Capkun, “Relay Attacks on Passive Keyless Entry and Start Systems in Modern Cars,” in *Network and Distributed System Security Symposium 2011 (NDSS 2011)*. San Diego, California, USA: NDSS, January 2011.

[5] S. Kamkar, “Drive it like you hacked it,” in *DEF CON 23*. Las Vegas: Samy Kamkar, 2015.

[6] DEFCONConference, “DEF CON 23 - Samy Kamkar - Drive it like you Hacked it: New Attacks and Tools to Wireles,” December 2015. [Online]. Available: https://www.youtube.com/watch?v=UNgvShN4USU

[7] A. Greenberg, “This Hacker’s Tiny Device Unlocks Cars And Opens Garages,” June 2015. [Online]. Available: https://www.wired.com/2015/08/hackers-tiny-device-unlocks-cars-opens-garages/

[8] C. News, “High-tech car theft: How to hack a car,” April 2016. [Online]. Available: https://www.youtube.com/watch?v=ARrlhlQiFzM

[9] Microchip, *MCS3142 Dual KEELOQ® Technology Encoder Data Sheet*, MICROCHIP, Microchip Technology Inc., 2355 West Chandler Blvd., Chandler, Arizona, USA 85224-6199, March 2014.

[10] ——, *KEELOQ® Code Hopping Encoder*, MICROCHIP, Microchip Technology Inc., 2355 West Chandler Blvd., Chandler, Arizona, USA 85224-6199, October 2001.

[11] R. W. Stewart, K. W. Barlee, D. S. W. Atkinson, and L. H. Crockett, *Software Defined Radio using MATLAB & Simulink and the RTL-SDR*, 1st ed. Glasgow, Scotland, UK: Strathclyde Academic Media, September 2015.

[12] Realtek Semiconductor Corp., "Rtl2832u," Februray 2018. [Online]. Available: http://www.realtek.com.tw/products/productsView.aspx?Langid=1&PFid=35&Level=4&Conn=3&ProdID=257

[13] NooElec Inc., "NooElec NESDR Nano 2+," February 2018. [Online]. Available: http://www.nooelec.com/store/sdr/sdr-receivers/nesdr/nesdr-nano2-plus.html

[14] "Welcome to the CCCamp 2015 rad1o Badge Wiki," August 2015. [Online]. Available: https://rad1o.badge.events.ccc.de

[15] M. Ossmann, "HackRF One," 2016. [Online]. Available: https://greatscottgadgets.com/hackrf/

[16] ——, "Software Defined Radio with HackRF, Lesson 11." [Online]. Available: https://greatscottgadgets.com/sdr/11/

[17] Phonical, "Fft-time-frequency-view," By Phonical (Own work) [CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)], via Wikimedia Commons, November 2017. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/6/61/FFT-Time-Frequency-View.png

[18] M. Ossmann, "YARD Stick One," 2016. [Online]. Available: https://greatscottgadgets.com/yardstickone/

[19] Texas Instruments, *CC1110Fx / CC1111Fx*, Texas Instuments, 12500 TI Boulevard Dallas, Texas, USA 75243, July 2013.

[20] M. Ossmann, "YARD Stick One," September 2015. [Online]. Available: https://github.com/greatscottgadgets/yardstick/wiki/YARD-Stick-One

[21] atlas0fd00m, "rfcat." [Online]. Available: https://github.com/atlas0fd00m/rfcat

[22] A. Mohawk, "Rfcathelpers," February 2016. [Online]. Available: https://github.com/AndrewMohawk/RfCatHelpers

[23] alextspy, "rolljam," 2016 July. [Online]. Available: https://github.com/alextspy/rolljam

[24] Ghostlulz, “Car Hack Rolljam,” April 2016. [Online]. Available: https://www.youtube.com/watch?v=sqLYpxzEQCw&list=PLmh1Jlaj91uzNDanztB85KL38JJZIJo0G

[25] Raspberry Pi Foundation, “Raspberry Pi 3 Model B.” [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[26] E. Courjaud, “rpitx,” October 2017. [Online]. Available: https://github.com/F5OEO/rpitx

[27] “rpitx-app-note,” October 2015. [Online]. Available: https://github.com/ha7ilm/rpitx-app-note

[28] “Replay Attacks with an RTL-SDR, Raspberry Pi and rpitx,” July 2017. [Online]. Available: https://www.rtl-sdr.com/tutorial-replay-attacks-with-an-rtl-sdr-raspberry-pi-and-rpitx/

[29] T. Wimmenhove, “subarufobrob,” October 2017. [Online]. Available: https://github.com/tomwimmenhove/subarufobrob

[30] ——, “Subaru fobrob exploit,” October 2017. [Online]. Available: https://www.youtube.com/watch?v=ewMZCxi8l8A

[31] MonsieurV, “py-findpeaks,” February 2018. [Online]. Available: https://github.com/MonsieurV/py-findpeaks

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[33] nVidia, “Driving Innovation,” 2018. [Online]. Available: https://www.nvidia.com/en-us/self-driving-cars/

[34] F. Lambert, “Look inside Tesla’s onboard Nvidia supercomputer for self-driving,” May 2017. [Online]. Available: https://electrek.co/2017/05/22/tesla-nvidia-supercomputer-self-driving-autopilot/

[35] nVidia, “Nvidia Jetson,” 2018. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/

[36] ——, “Jetson TX2 Module.” [Online]. Available: https://developer.nvidia.com/sites/default/files/akamai/embedded/images/jetsontx2/TX2_Module_170203_0017_TRANSP_2000px.png

[37] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, May-Jun 2007.

[38] "Get Started with Graph Execution," April 2018. [Online]. Available: https://www.tensorflow.org/get_started/get_started_for_beginners

[39] K. Willems, "TensorFlow Tutorial For Beginners," July 2017. [Online]. Available: https://www.datacamp.com/community/tutorials/tensorflow-tutorial

[40] "Numba," 2018. [Online]. Available: https://numba.pydata.org

[41] "GPU Accelerated Computing with Python," 2018. [Online]. Available: https://developer.nvidia.com/how-to-cuda-python

[42] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.