APPLIED SECURITY ANALYSIS

IN4253ET

Reproducing POWER-SUPPLaY:

Leaking Data from Air-Gapped Systems by Turning the Power-Supplies Into Speakers

Dex Bleeker *s1460366*

Mihai Macarie *s1919016*

Utz Nisslmueller *s2635895*

April 9th 2021

Contents

1 Introduction

In this report, we present our attempts to reproduce results from a previous paper by Guri et al. [1]. In section 1, we present the overall background and used equipment for the measurement. Then, we discuss how the signals are generated in section 2. Successful exfiltration also required modulation of the signal before the reception, the details of which we dive into in section 2.3. We will then discuss our measurements of the modulated signal in section 3 and how we tackled the issues during demodulation in section 4 and the thread model in 7. We conclude this paper and the insights we received while conducting this project in section 7. All code mentioned is made available in a zipfile. The audio file and generated graphs are available in our repository[1].

1.1 Background

In this project, we aimed to reproduce the results which were achieved during the original work by Guri et al. [1] concerning covert data exfiltration using the frequencies emitted by desktop power supply units (PSUs). During our initial scoping of the project, we identified the following questions which we deemed to be of particular interest in regards to a) further exploration of the possibilities discovered in [1] b) adaptation to a further range of systems and c) possible mitigations:

- Is the approach taken in the paper the only road to successful data exfiltration? Which parameters can be tweaked and what effect will they have on the resulting measurements?
- Can we create a universally portable version of the code responsible for data exfiltration (e.g., using Python)?
- Would the results be reproducible on mobile computers as well? If no, how and why are the results different from the original work?
- Is this technique overall less or more effective on mobile computers and do we need to adapt the threat model to reflect this change in the underlying infrastructure? If yes, how would this impact possible mitigations?

1.2 Used equipment

We conducted the tests and measurements throughout this project on various, mostly mobile systems. The exact specifications are listed in table 1.

Target reference	Device	CPU	PSU	Motherboard	Recording Device
1	Desktop PC	Intel i7 4790K	High Power EP-700 BR-II	ASUS ASRock Z97M OC	OnePlus Nord
2	Lenovo P51	Intel i7 7700HQ	OEM	OEM	OnePlus Nord
3	Raspberry Pi 3 Model B+	ARM Cortex-A53	OEM	OEM	OnePlus Nord
4	Dell XPS 13 9300	Intel i7 1065-G7	45W OEM	OEM	iPhone 11
5	Raspberry Pi 4 Model B	ARM Cortex-A72	OEM	OEM	iPhone 11
6	SFF Desktop PC	Intel i3 4370	BeQuiet SFX POWER 2	Gigabyte H81N	iPhone SE 2020 / Blue Snowball ICE
7	MacBook Air 2021	Apple M1	OEM	OEM	iPhone 11

Table 1: Target specification.

[1] `https://gitlab.utwente.nl/uni/asa_powersupplay/`

2 Signal generation

2.1 Introduction

In this section, we will present how the transmission signal is generated. First, we tried to code a PoC in Python, but due to the slow code execution of Python, we had to write a PoC code in C++. In both cases, the signal is generated by calling a function, which loads the CPU for fraction `LOAD_FACTOR` of the time and idles it for $1 - $ `LOAD_FACTOR` of the time. Depending on how often this function is called within a given time interval, it generates a correlating switching frequency within the PSU's internal capacitors. If we define 2 sufficiently distinct switching frequencies, we can use the resulting signals to code logical 1s and 0s.

2.2 Implementations

2.2.1 Python

The Python implementation is based on the *multiprocessing* (used to create the threads) and *psutils* (to control the affinity of the threads) libraries. The implementation is fairly straightforward. A separate `CpuLoader`-class to handle the loading of the CPU is introduced and different signal generation methods (which depend on the aforementioned class) reside in the `generate_signal.py`-file. Both files are available in the attached zip file.

Correlating switching frequency For this method, the function used to generate a signal is called `generate_signal`, which receives as parameters the duration of the signal, an array with the load levels for each part of the signal, and the granularity factor. The frequency of the signal is determined by the duration of the signal and the number of elements from the load factor array. Then, the CPU is loaded and paused with the time interval determined earlier.

Correlating CPU load directly In this case, the function `generate_signal` gets as parameters the frequency and the duration of the signal. Then, it spawns a thread for each CPU core available by using a *pool* from the multiprocessing library. Each thread makes a call to a function called `spawn_child`, which generates a process, sets the affinity of the process, and calls the function named `busy_wait`, which make CPU busy with random variable assignments until the provided moment (`datetime`-object).

Issues with the approach Given that the CPU was loaded and unloaded properly and we used a low frequency of 1 Hz, one would expect that implementation to work properly, but it did not. After doing some more research and a bit of tinkering and experimenting, we concluded that it was because it was not the *loading* of the CPU, but rather *oscillating* between loading and unloading that made the PSU 'squeak'. We had to change our approach altogether.

When developing the Python code, one of the aims was to keep it as portable and concise as possible. The small investment of writing proper code would be significantly outweighed by the development speedup provided by working with neat code. Python seemed like a great fit at first. There were libraries, all group members were familiar with them and the learning curve is gentle. However, there were problems with this approach. Oscillating the frequency by using Python is practically impossible, as Python is a relatively slow language. It became painfully clear that Python was not going to cut it. This meant we had to go back and come up with a new approach.

Back to the drawing board Given that a new approach had to be significantly faster, using a more low-level language seemed warranted. We quickly decided on C++ and a new prototype saw light. More on this in the next subsection.

2.2.2 C++

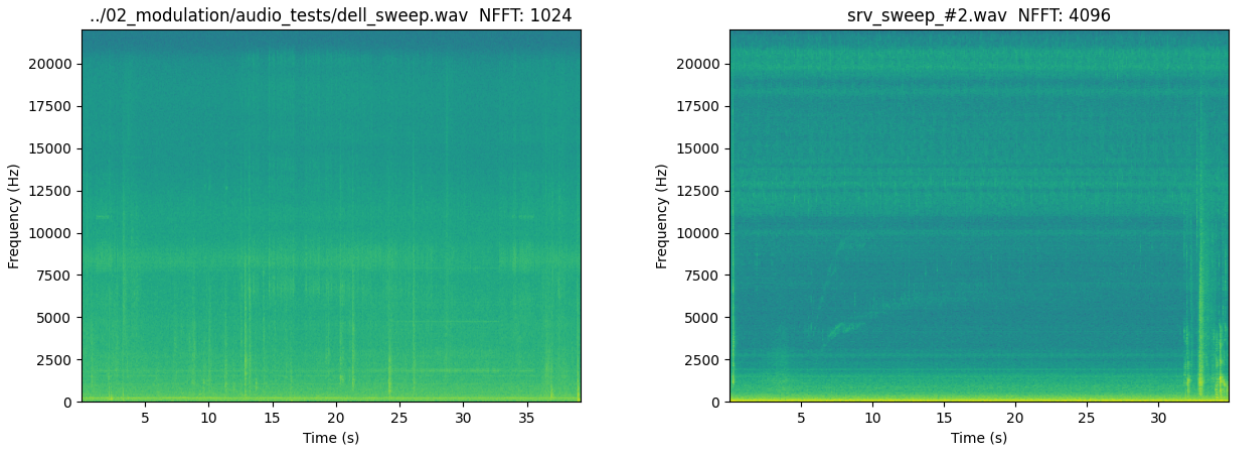
The C++ variant is based on *pthreads* (POSIX Threads[2]) and therefore it runs only under POSIX[3] systems (loosely speaking; proprietary UNIX systems, *BSD and Linux). In terms of algorithm, its fundamental idea is unchanged from the Python implementation: Each available CPU core oscillates at a certain frequency between full load being idle. All related code can be found in the attached `linux. cpp`-file.

The function `generate_signal` is responsible for generating the signal with a certain duration, frequency, and loading ratio. It then generates the frequency on all available CPU cores by calling `generate_frequency_on_all_cores`. The later function creates a thread and sets the affinity for each available CPU core. Each thread calls a function named `generate_frequency_on_core` which keeps the thread busy for a certain period inversely proportional with the load factor and frequency and then put the thread to "sleep" for the same amount of time.

2.3 Modulation

2.3.1 Sweep signal

Unfortunately, we were unable to emulate the sweep signal as seen in figures 16 and 17 in [1] on a laptop PSU, as demonstrated by Figure 1a.



(a) Failure to reproduce a clear sweep signal (target 4).

(b) A successful sweep signal (target 6).

Figure 1: Generation of a sweep signal (both successfully and unsuccessfully on two targets.

We attribute this to the differences in the design of the more portable, lower power PSUs which are used in most modern laptops. The effects of our sweep signal generator function on CPU load can be seen in Figure 2.

[2] `https://en.wikipedia.org/wiki/POSIX_Threads`

[3] `https://en.wikipedia.org/wiki/POSIX`

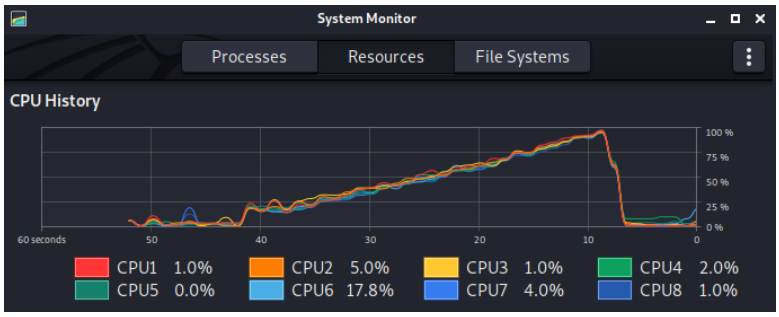
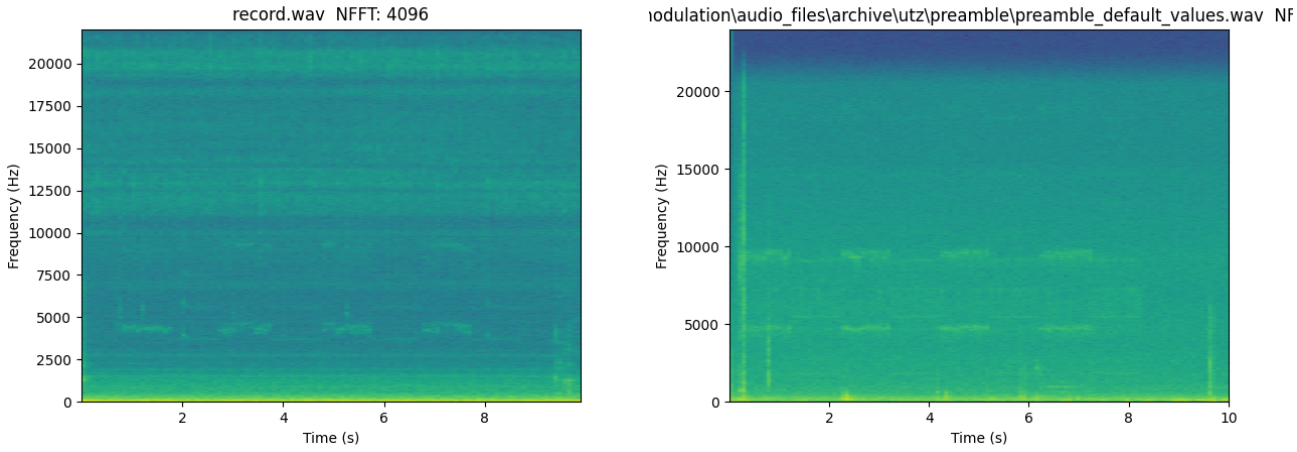


Figure 2: Progressively increasing CPU load in an attempt to produce a sweep signal.

2.3.2 BFSK signal

Despite the setbacks mentioned in section 2.3.1, we were able to produce a BFSK signal, transmitting the preamble used in [1], "10101010". The question was, how to code the binary data. Naturally, we chose to represent 1s with a high CPU load (100%, for the best SNR[4]) and 0s with a lower CPU load, respectively. The question was, which CPU load should we pick for the lower threshold? We have found that roughly 75% load produces the best results on the PSUs which were at our disposal.

However, this approach to BFSK turned out to be *completely* wrong, as mentioned before in section 2.2.1. With the new C++ approach, results were significantly better. This is illustrated by Figure 3. BFSK signal generation in the C++ implementation works roughly as follows. We define four variables: 1. `HIGH_FREQ` 2. `LOW_FREQ` 3. `BFSK_FREQ` 4. `LOAD_FACTOR`. These variables are used to determine the range of oscillation, the time spent on a single bit and the percentage of load generated, respectively. More on the BFSK measurements in Section 3.4.



(a) Target 6

(b) Target X

Figure 3: BFSK signal with the new approach (oscillating) on different targets.

CRC As the paper by Guri et al [1] describes, we implemented a 8-bit Cyclic Redundant Check (CRC[5]). We chose the CRC-8/SMBUS standard.

[4] `https://en.wikipedia.org/wiki/Signal-to-noise_ratio`

[5] `https://en.wikipedia.org/wiki/Cyclic_redundancy_check`

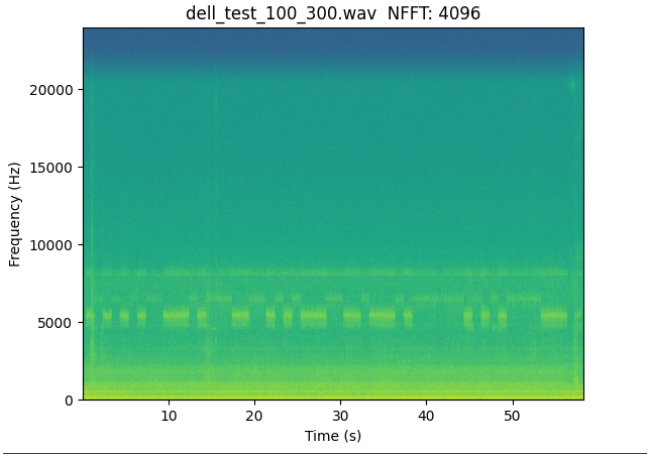


Figure 4: Calculating CRC 11000011 on ASCII payload “test”.

2.3.3 OFDM signal

In the paper by Guri et al [1], OFDM (*orthogonal frequency-division multiplexing*) transmission was also presented. OFDM is used for broadcasting TV and audio signals. We tried to implement it using our Python code and we got some results, presented in Figure 14. We achieve this by loading each CPU core times two for 2 seconds. Based on the (lack of good) results, we decided to not proceed further with this.

3 Measurements

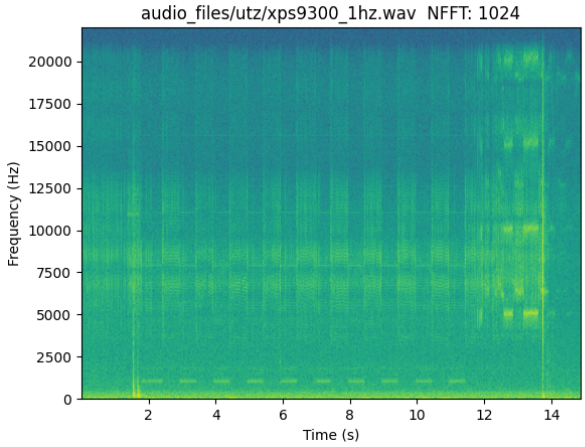
In this section, we present our measuring methodology and our results for different (sub-)types of signals: sweep, BFSK, and OFDM.

3.1 Measurement method

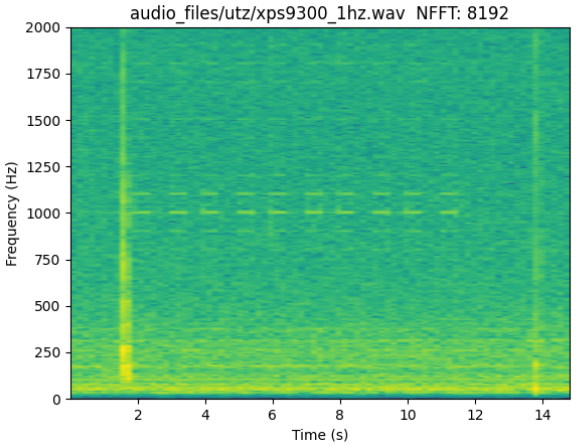
We recorded the signal by initially putting the smartphones/microphones as close as possible to our targets to get the highest possible SNR from our targets. We then recorded the signal for its’ entire duration, and then analyzed the resulting .wav files with signal_analyzer.py. In this script, we first transform the target .wav file to a mono-channeled .wav file (as spectrograms are naturally only defined for single-channel audio). In the code, we have implemented a bandpass filter, but in practice, we only ever used it as a highpass filter at max. 1 kHz, as we saw a lot of background noise in this low-frequency range, diminishing visual prominence of the signal within the spectrograms.

3.2 Initial measurements

After having finished coding our initial prototype application in Python, we were keen on exploring how far we can push PSU switching frequency while still producing a sufficiently distinct signal. It is important to note that while this doesn’t represent our final modus operandi, as coded in C++, it still provides useful measurements in regards to switching frequency and spectrum of the emitted noises. In our first round of measurements, we ran the CPU loading script utilizing all 8 virtual cores at a frequency of 1 Hz on target 4. During execution, we made a `.wav`-recording and analyzed the audio file by creating a spectrogram (Figure 5).



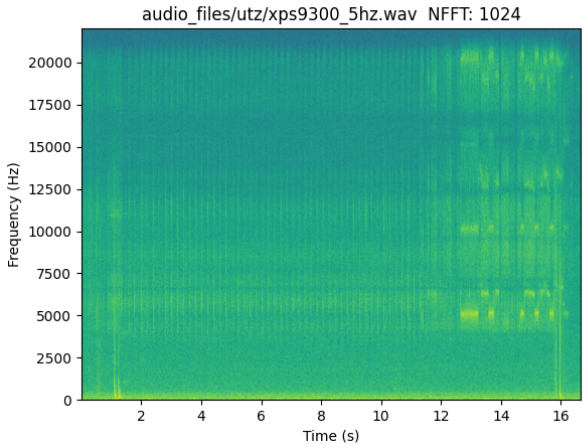
(a) Initial spectogram.



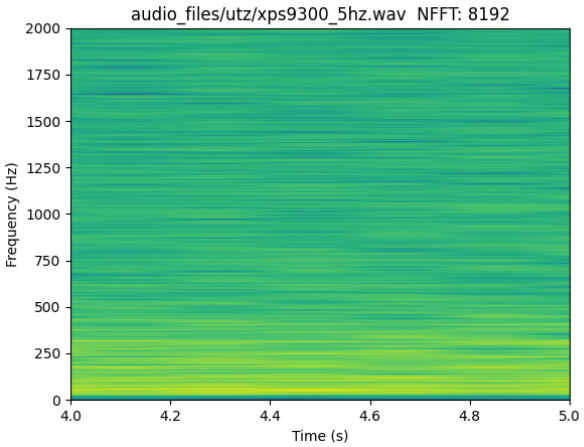
(b) A closer look at the signal peaks at around 1 KHz.

Figure 5: Running the CPU loading script on all cores for 10 seconds @ 1Hz, recording from roughly 5 cm away.

These results are interesting in regards to the results seen in [1], since they occurred at a much lower frequency (~1 kHz as compared to ~10-20 kHz). Next, we attempted to increase the frequency to 5 Hz on the same target (Figure 6).



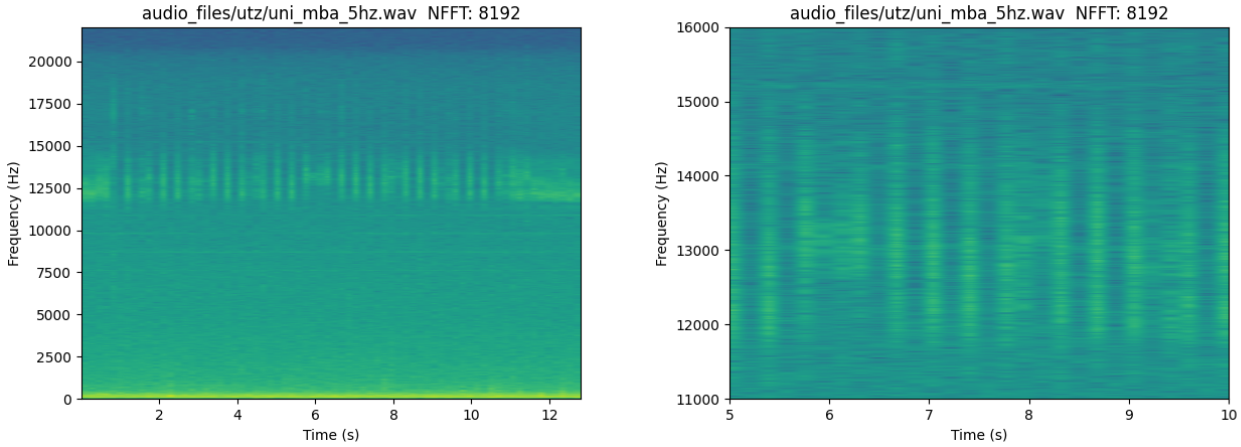
(a) Initial spectogram.



(b) A closer look at the signal peaks at around 1 KHz.

Figure 6: Running the CPU loading script on all cores for 10 seconds @ 5 Hz, recording from roughly 5cm away.

Despite being able to see some harmonics in Figure 6a at 6 (suspected main frequency), 12 and 18 kHz, the close-up look in Figure 6b doesn't reveal any clear carrier frequencies, as in Figure 7. Due to the unusually low carrier frequency of 6 kHz, we suspected this to be a peculiarity of target 4's PSU. We conducted the measurements again on a 2021 MacBook Air PSU and were now able to see a much more distinct carrier signal between 12 and 15 kHz (Figure 7).



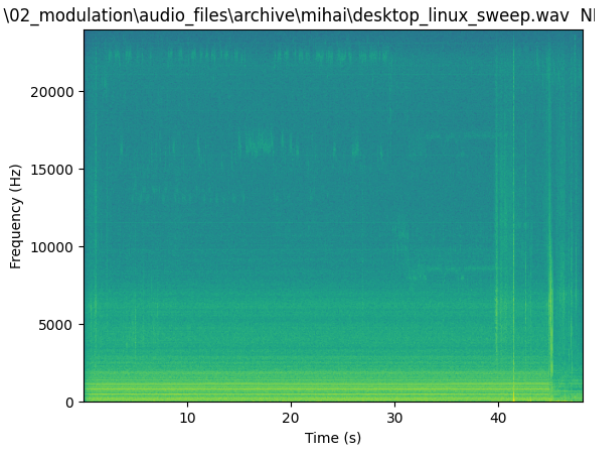
(a) Initial spectogram.

(b) A closer look at the signal peaks between 11 and 16 KHz.

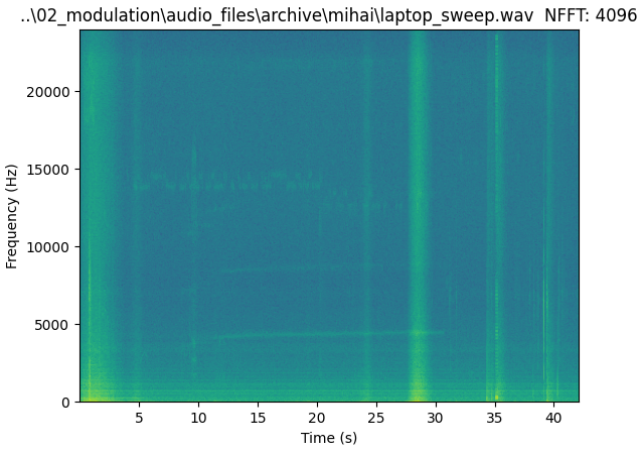
Figure 7: Running the CPU loading script on all cores for 10 seconds @ 5Hz, recording from roughly 5 cm away.

Especially in Figure 7b however, we can see that the amount of actual peaks per second is closer to 3-4 rather than 5. We could verify this in attempts to generate 10, 20, and 50 Hz signals, where the actual switching rate remained consistently between 3 and 4 Hz. We thus concluded that with our initial Python code, this was the maximum reachable switching frequency.

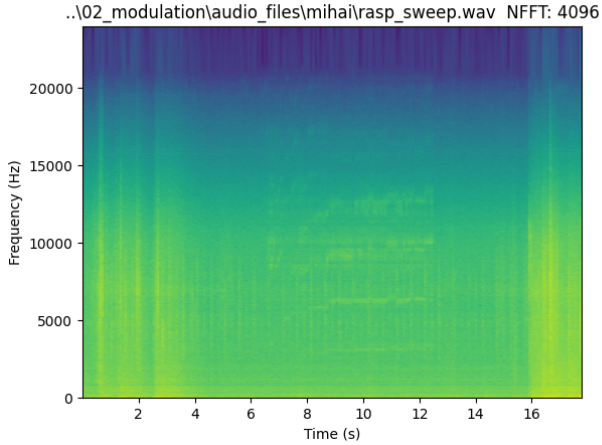
3.3 Sweep signal



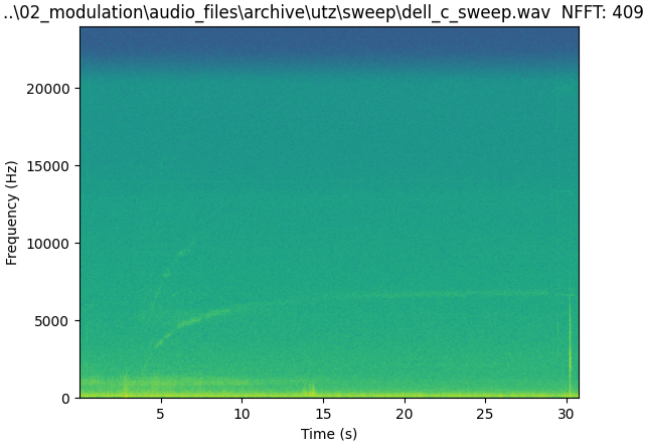
(a) Target 1



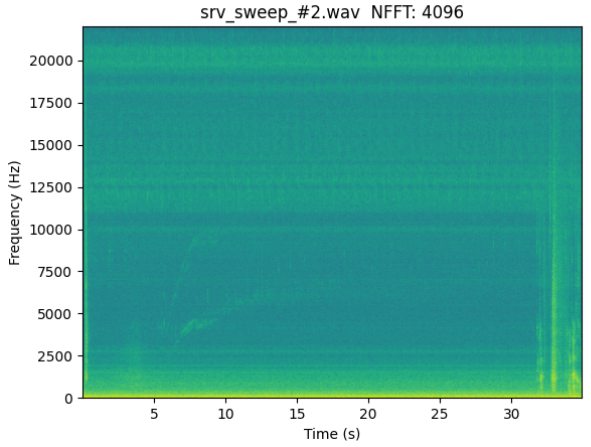
(b) Target 2



(c) Target 3



(d) Target 4



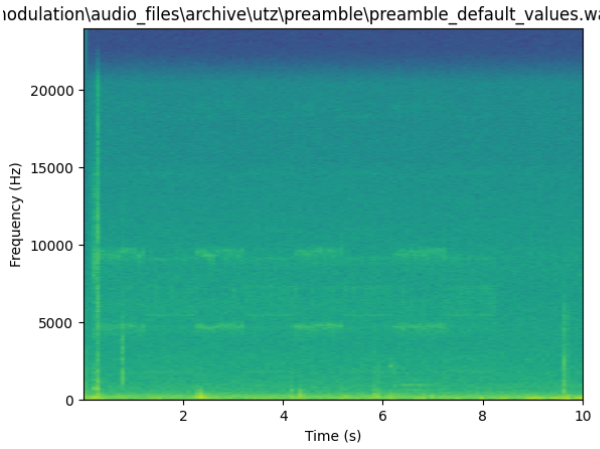
(e) Target 6

Figure 8: Sweep generation tests on several devices

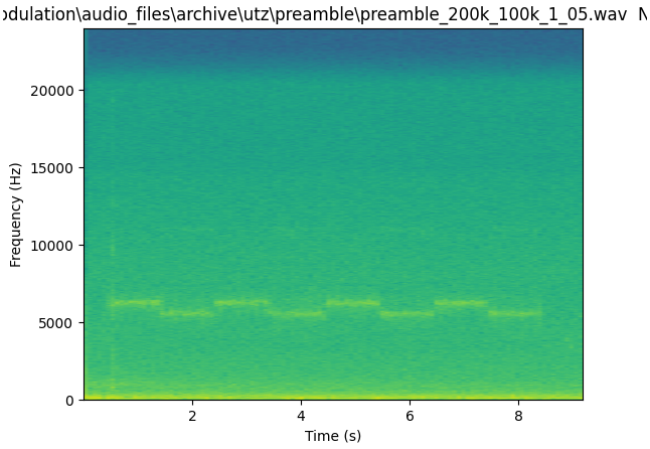
To determine the validity of our PoC code, we followed the methodology in [1] and proceeded with generating a sweep signal, by gradually increasing load/idle switching frequency over time. The goal of this step was to determine whether a) an increased switching frequency leads to a higher emitted frequency by the PSU and b) what the total range of the frequency band was. As seen in Figure 8, this was done on several targets. For targets 1 and 3, we didn't get any useful data. For target 2, there seems to have been a linear increase over the duration of the sweep signal, however only slightly. Targets 4 (Figure 8d) and 5 (Figure 8e showed a sweep pattern resembling a root function with some higher band harmonics at the start. We can see that the range around 3-5 kHz for both targets 4 and 5.

3.4 BFSK preamble

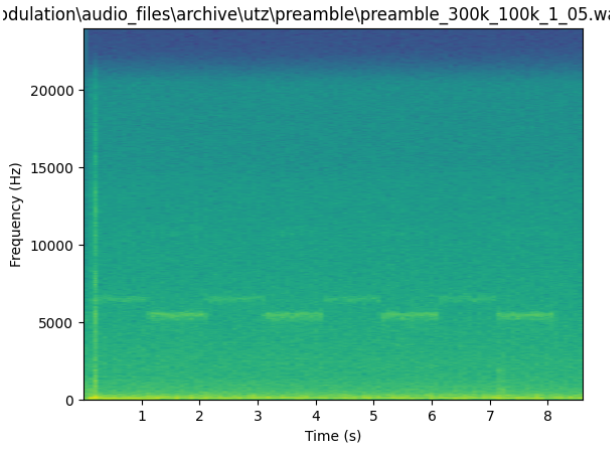
In this subsection, we present and discuss tests for the BFSK preamble transmission conducted on several targets, at different switching frequencies.



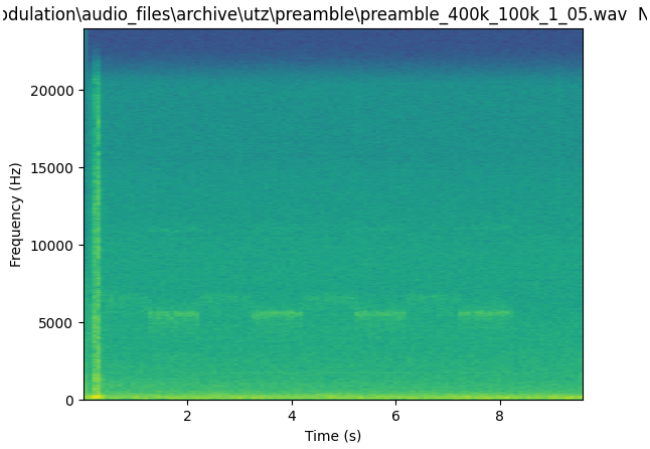
(a) Default value



(b) BFSK preamble with 200 KHz and 100 KHz frequencies



(c) BFSK preamble with 300 KHz and 100 KHz frequencies

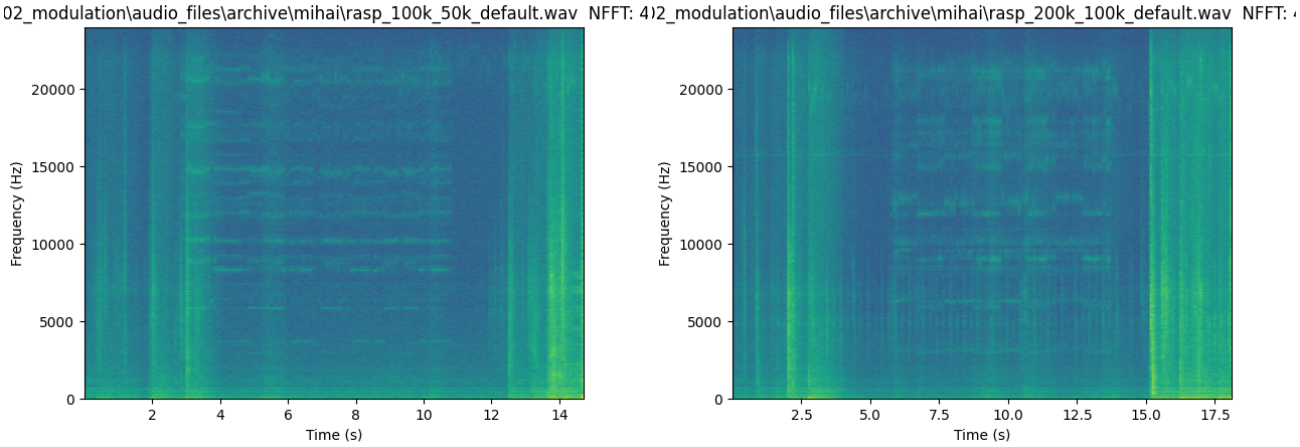


(d) BFSK preamble with 400 KHz and 100 KHz frequencies

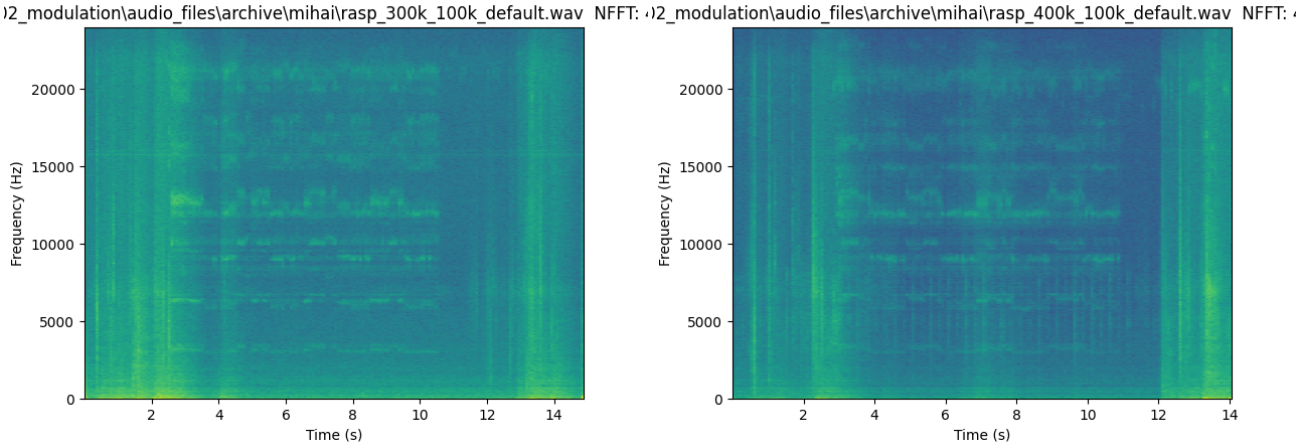
Figure 9: BFSK preamble on Dell notebook at different frequency settings

In Figure 9, we show 4 experiments conducted on target 4 at different switching frequencies. We notice that the best results are achieved in Figures 9b and 9c, using 200/300 kHz (resulting in a roughly 6.5 kHz signal) for coding binary 1s and 100 kHz (resulting in a roughly 5.5 kHz signal) for coding binary 0s, respectively.

In Figure 9a, we observe some strong and distinct harmonics. We used the default values for this measurements (FREQ_HIGH = 100 KHz and FREQ_LOW = 50 KHz). Harmonics like this might cause problems during demodulation, especially within the `get_dominant_frequency()` function (see Section 4). In Figure 9d, we notice these harmonics as well, although at a much lesser extent.



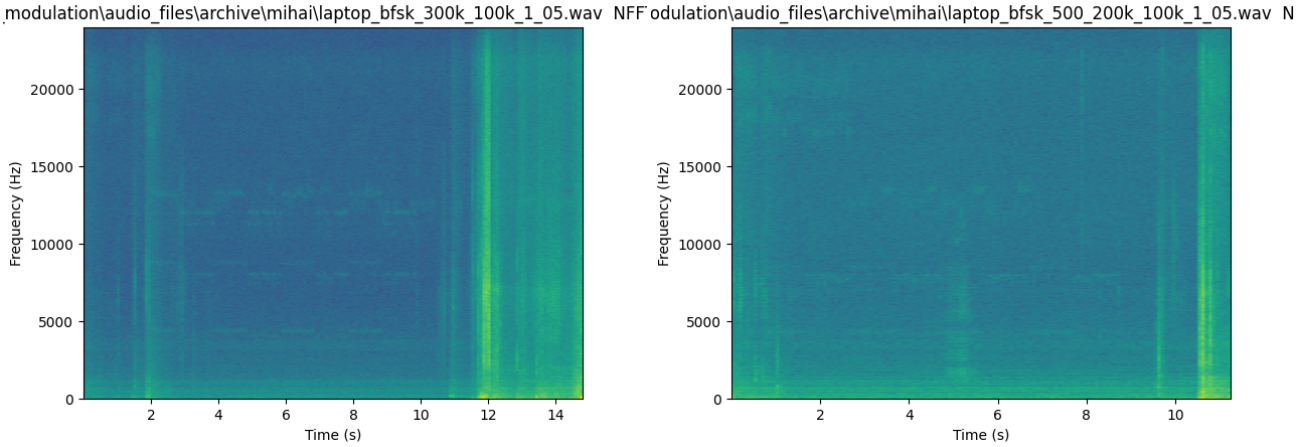
(a) BFSK preamble with 50 KHz and 100 KHz frequencies (b) BFSK preamble with 200 KHz and 100 KHz frequencies



(c) BFSK preamble with 300 KHz and 100 KHz frequencies (d) BFSK preamble with 400 KHz and 100 KHz frequencies

Figure 10: BFSK preamble on Raspberry Pi 3 B+ at different frequency settings

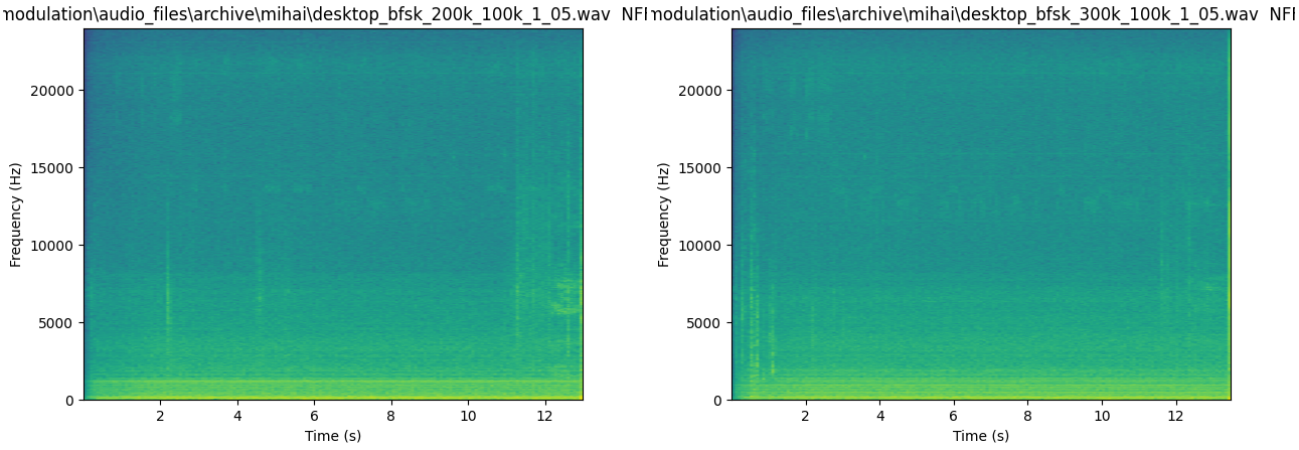
In Figure 10, we show 4 tests using target 3 at different frequencies. We seem to get the clearest pattern in 10a, however, we also receive strong interfering harmonics in multiple frequency bands.



(a) BFSK preamble with 300 KHz and 100 KHz frequencies. (b) BFSK preamble with 200 KHz and 100 KHz frequencies.

Figure 11: BFSK preamble on target 2 at different frequency settings.

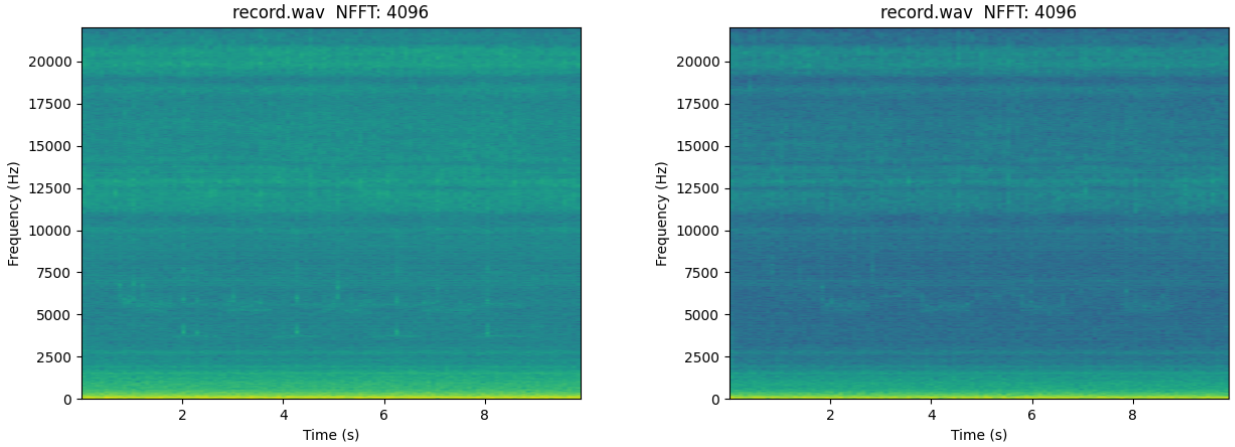
In Figure 11, we present 2 tests ran using target 2 at different frequencies. We notice in Figure 11a that 300 kHz and 100 kHz work the best on this device. However, the SNR ratio looks very low, as the signal is very diminished in terms of strength.



(a) BFSK preamble with 200 KHz and 100 KHz frequencies. (b) BFSK preamble with 300 KHz and 100 KHz frequencies

Figure 12: BFSK preamble on target 1 at different frequency settings.

In the case presented in Figure 12, we show 4 tests using target 1 at different frequencies. In all cases, we notice a lot of interference and no usable signal.



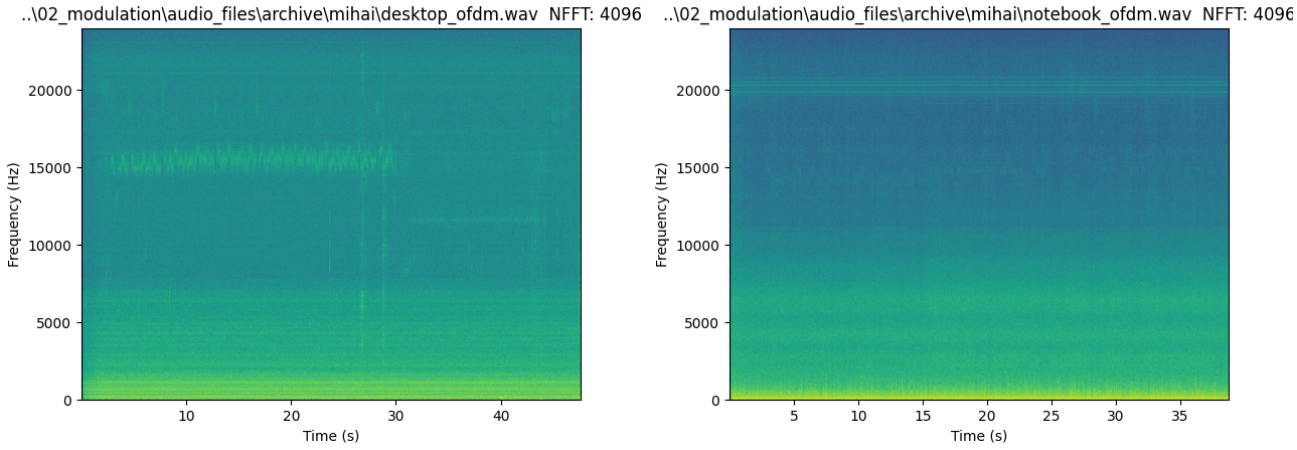
(a) BFSK preamble with 10KHz and 100 KHz frequencies (b) BFSK preamble with 100 KHz and 300 KHz frequencies

Figure 13: BFSK preamble on server at different frequency settings

In Figure 12, we present the result of 2 tests using the server at different frequencies. In all cases, we notice a lot of interference and only partial signal traces at best.

3.5 OFDM

We ran some tests using OFDM on some devices, but the results were not satisfactory, as we didn't achieve sufficiently distinct signals, and were thus unable to attempt frequency-division demultiplexing. The spectrograms for our attempts can be seen in Figure 14.

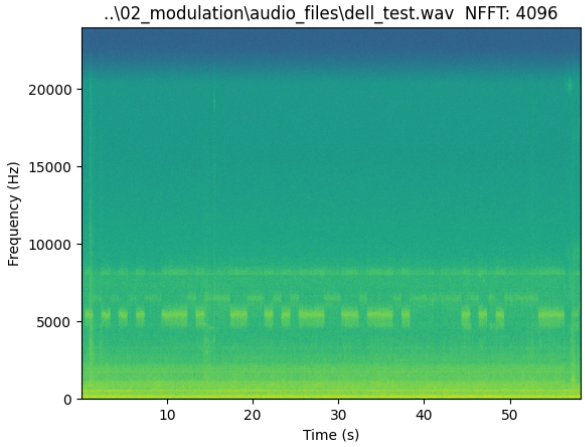


(a) OFDM test on target 1 (b) OFDM test on target 2

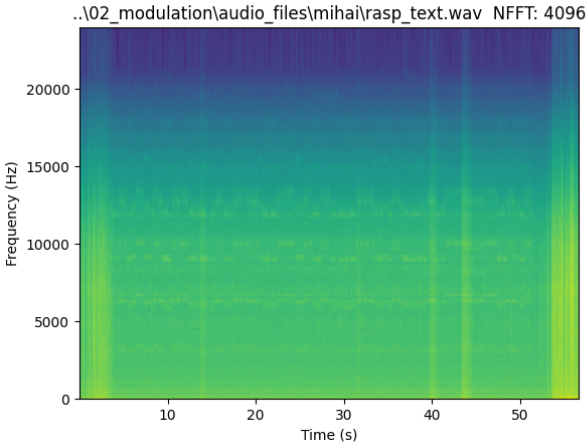
Figure 14: OFDM test on several devices

3.6 Data transmission

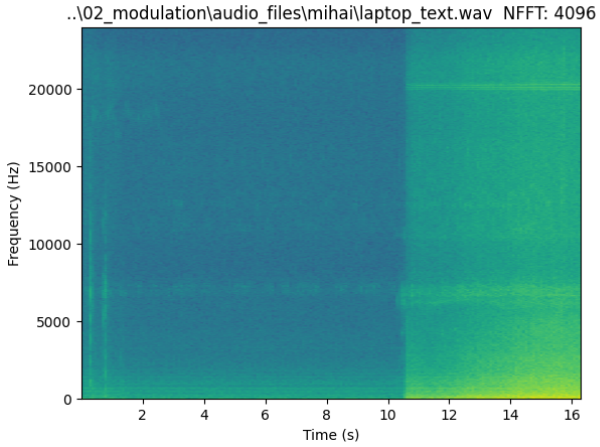
We did some testing transmitting text data from a file using BFSK. We tried to transmit preamble, actual data, and CRC checks. The results for most of the targets can be found in Figure 15, where the best results have been achieved using target 4 in Figure 15a. In all other cases, we can see some traces of the signal and noise.



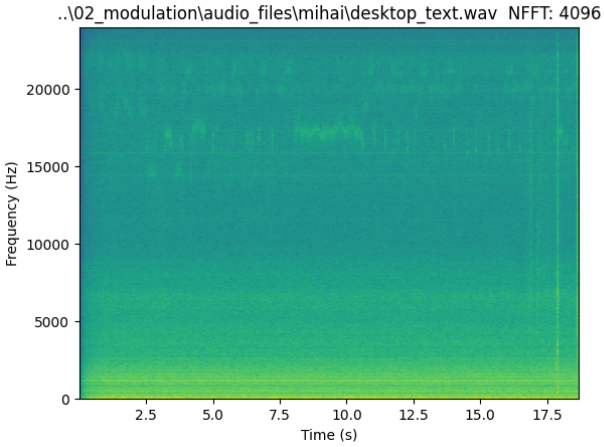
(a) BFSK text transmission on target 4



(b) OFDM test on target 2



(c) OFDM test on target 2



(d) OFDM test on target 2

Figure 15: OFDM test on several devices

3.7 Comparison with paper

Compared to the paper, our measurements contain significantly more noise. Our thesis is that on one hand, while not necessarily having an overall louder noise level, the signal generated by the PSUs of our (useful) targets was significantly weaker due to the wattage being lower by a full order of magnitude. Secondly, we have observed that the generated frequencies were also only fractions of the ones generated in [**1**], which we attribute to the same underlying reason.

4 Demodulation

4.1 Algorithm walkthrough

Having received our strongest signal from target 4, we then started working on a Python prototype for demodulating the signal. For accuracy, the script requires the frequencies, in which 1s and 0s are coded in BFSK, as an input (for target 4, around 5500 and 6500 Hz, respectively). The script would then execute the following steps:

1. **Filter**: We started by applying a bandpass filter around the expected frequencies to reduce noise and errors stemming from unrelated frequencies.
2. **Divide et impera**: We decided that splitting the `.wav`-file data array into chunks of length `TIME_INTERVAL` (default $= 0.1s$) would be the easiest approach.
3. **Frequency retrieval**: Determine the dominant frequency for each chunk by applying FFT[6] on the signal within.
4. **Count and transform**: Counting for how many consecutive chunks 1s and 0s last and removing the first 8 elements (preamble). Based on this resulting array, generate an array of 1s and 0s (data + payload).
5. **Error correction**: Since we still had to deal with non-negligible amounts of noise, we decided to trade flexibility in regards to the signals which we could process for robustness, by flipping single bits whose immediate neighbors were of complementary value. This effectively put a lower bound of $2 * \text{TIME\_INTERVAL}$ on the symbol time of the transmitted data. After this step, we rounded the symbol lengths to the nearest multiple of the symbol time of the transmitted data ($\frac{1}{BFSK\_FREQ}$) as a further error detection step. Despite introducing a risk of "over smoothing", this method seemed to work well in practice and produced the actual bitstream which was observed from the receiver.
6. **Pruning**: After error correction, we pruned the end of the remaining bitstream until we had an array of bits whose length modulo 8 yielded zero, as we weren't interested in anything beyond the CRC, regardless of frequency.
7. **CRC calculation and data display**: Finally, we calculated the CRC for all but the last 8 elements of the array (transmitted CRC) and compared both. The user was informed whether they matched (in which case the binary array was automatically converted to ASCII text and displayed) or not (in which case he was asked to inspect the data manually).

As before, we would like to invite the reader to look at [2] to inspect the code themselves. The source code can be found in `03_demodulation/demodulate_bfsk.py`.

4.2 Limitations and discussion

The code above is to be understood more as a proof-of-concept rather than a production code. Already during coding and also during the following evaluation of results we have identified several deficiencies, the most significant of which are:

- **Rudimentary preamble detection**: The Python script does not detect a preamble, that is, a dedicated 10101010 sequence - rather, it detects the first 8 alternating symbols, which could also be 01010101. This leads to a superfluous 0 bit at the beginning of the payload, causing problems when 1s were coded at a lower frequency than 0s.
- **Hardcoded parameters**: While we initially wanted to make the demodulation as dynamic and flexible as possible, we soon ran into the following problem: How do we know the BFSK frequency beforehand? And if we pass it by hand, how do we deal with incrementally longer or shorter symbol durations? We circumvented these issues by declaring a `TIME_INTERVAL` parameter. We then sliced the NumPy array

[6] `https://en.wikipedia.org/wiki/Fast_Fourier_transform`

resulting from the wav file into slices of length `TIME_INTERVAL` (since we knew the sampling frequency, we knew that one slice had to include `SAMLPING_FREQ` $*$ `TIME_INTERVAL` samples). The default value is 0.1, yielding 100ms sound slices, from which we then determined the dominant frequencies via FFT (see section 4.1). This introduced a limit on the minimum symbol time and thus on the transmission data rate. We have tested it on a `BFSK_FREQ` value of 1 (equaling 1 transmitted symbol per second), and it should hold up well to values of up to 4 since every symbol is guaranteed to last longer than 0.2s (2 slices for the default TIME_INTERVAL). For higher `BFSK_FREQ` values, the code would have to be refactored to some extent.

- **Possible aliasing**: Due to posing hard barriers on 1s and 0s, we were susceptible to aliasing, since a) the signal was very unlikely to start at the beginning of a slice s $P(\text{signal starts at s[0]}, \text{default params}) = \frac{1}{\texttt{SAMLPING\_FREQ}*\texttt{TIME\_INTERVAL}} = \frac{1}{4800}$ and b) especially for smaller `TIME_INTERVAL` values, the symbol would not exactly last for $\frac{\text{BFSK\_FREQ}}{\texttt{TIME\_INTERVAL}}$ slices. We implemented the error correction function as seen in section 4.1 as a countermeasure, and achieved good results for the default values.
- **Rudimentary pruning**: When the signal stops (no more 1s and 0s at the correlating frequencies are detected), the remainder of the signal is pruned away. However, if recording includes other time slices after, which includes one of these frequencies, the symbol is read and appended to the payload + CRC, thus extending the payload by 1 and bit-shifting the CRC one place to the right.

5 Demonstration

Our goal here is to read a file, transform the contents into an ASCII bitstream, modulate the data using the technique laid out in [1], and demodulate a recording we made from a phone sitting right next to the power supply. We will be using target 4 for this demo.

5.1 Modulation and transmission

We have placed a `test.txt` file in the program directory containing the string “test”. We have set the switching frequency within our transmission program to 100 kHz for logical 1s and 300 kHz for logical zeroes. The symbol duration has been set to 1 second and the load factor has been set to 0.5 (loading the CPU for half of the time and idling it for half of the time). We then started recording on the phone and started the program (Figure 16).

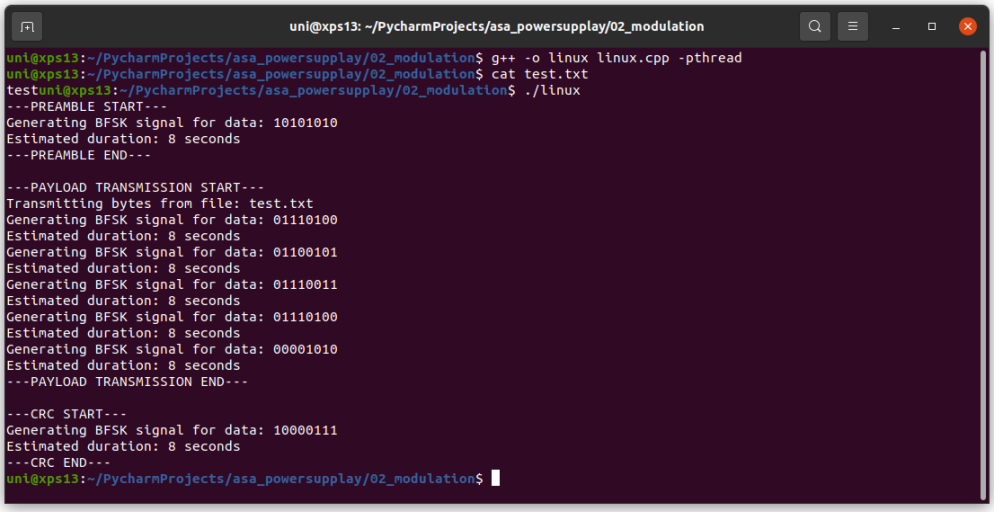


Figure 16: Data transmission

As we can see, the transmitted data was `01110100 0110101 0111011 0110100 0001010 10000111`. The first 4 bytes are ASCII encoding for the letters `t`, `e`, `s` and `t`, the fifth byte is a termination character and the final byte represents the CRC.

5.2 Reception and demodulation

Figure 17 shows a spectrogram of the recording. We can see that 1s have been coded at around 5500 Hz and 0s at around 6500 Hz (these frequencies will be determined exactly later). We can also correlate the received data with the bytes the transmitter sent and confirm visually that no errors have occurred.

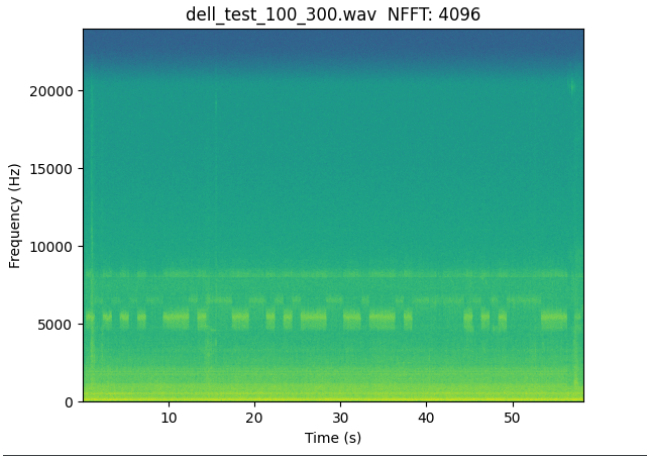


Figure 17: Spectrogram of data transmission

Having gotten a first impression on what to expect, we can now run the demodulation script (Figure 18). We can see that `ZERO_FREQ` and `ONE_FREQ` have been adjusted to 6487 and 5404 Hz respectively. Now knowing the accurate frequencies, the program continues with error correction (flipping single, out-of-order bits) and payload processing. However, after checking the CRC, we can see that the received and calculated values don't

quite match up. The program emits a warning and refrains from automatically transforming the binary values to ASCII since an error in the CRC might be indicative of further bit flips in the payload. We are simply presented with the bitstream which the program extracted from the .wav file for manual analysis.

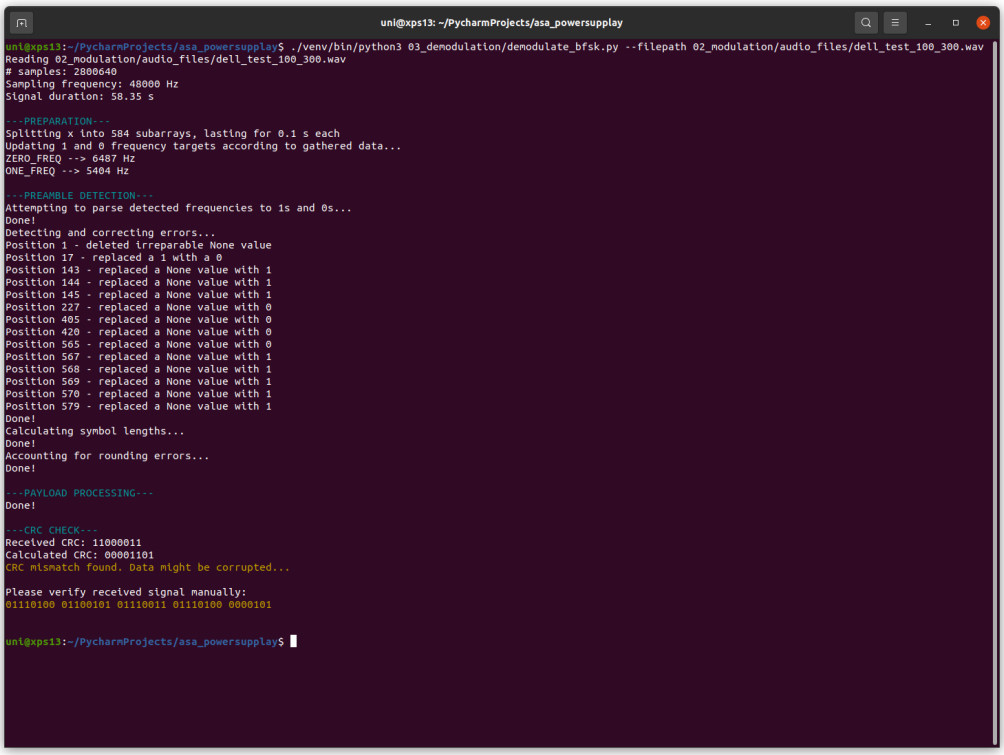


Figure 18: Demodulation script

Plugging `01110100 01100101 01110011 01110100 0000101` into an online binary converter, we get `test?`. Looking back at Figure 16, we can see that the last byte value was supposed to be 6, but we received a 5 instead, due to channel noise. Decimal 5 / binary 101 in ASCII equals the ENQ character, which doesn't have a display representation. The LF (line feed) character (decimal value 6, binary value 1010) makes much more sense in this scenario.

6 Threat model

A classical scenario where the methods laid out by Guri [1] would show great effect would be wide-scale OT systems such as power plants. An employee carrying a phone, which has been infected via a zero-day exploit could enter a room with an air-gapped computer, which has been infected by another employee through insufficient DLP measures (such as sticking a gifted USB stick into it and simultaneously not having disabled automatic execution from external media). The computer would emit the signal via its' PSU, and the phone would record it and send it to a location within the adversary's sovereignty.

Since we tested our implementation on predominantly mobile hardware (laptops & IoT devices), one could argue that for our route, the probability of attack has increased throughout all threat groups, since the range of available targets is greater. People are far more likely to have access to a work laptop rather than to a desktop PC / server or any air-gapped system. However, it is important to mention that due to the extremely low SNR which

we encountered during our measurements, recording devices with comparable sensitivity to that of the mobile phones we used would have to be placed within several centimeters of the PSU. Except for targets 1 and 6, all PSUs output comfortably below 50W, which is likely the reason for this increased sensitivity. Thus, despite an increased probability of attack, there is also a decreased probability of a successful attack due to the higher noise levels compared to the signal.

Both threat radius and stealth are decreased compared to the observations made in [1]. Lower SNR requires the listening device to be in the immediate vicinity of the transmitter, or to be extremely sensitive and thus bulky, which in turn reduces stealthiness substantially.

Regarding the countermeasures laid out in [1], a) zoning remains equally effective. b) signal detection is made harder by the fact that the SNR and thus the signal "prominence" is lower for mobile / IoT targets. Due to this, c) signal jamming and d) signal blocking are equally more effective against our targets since the signal is easier to "overcome".

7 Conclusion

During our project, we attempted to reproduce the research by Guri et al., and were successful to a good extent. Initially, we tried to create a portable version using Python, experimenting with different loading mechanisms and modulation techniques, before switching to C++ due to the need for a faster, low level language in order to generate the switching frequencies which were required to produce a BFSK signal via the "singing capacitors" of our targets' PSUs.

One complication which we encountered constantly were the relatively low signal-to-noise ratios (SNR) observed during our measurements, forcing us to record in immediate vicinity of the PSUs ($<$5cm) and still receiving only a relatively weak signal, which was highly susceptible to background noise.

Our overall proof of concept, as seen in Section 5 produced a valid result however and we thus consider this project to have been a success. One aspect in particular which is interesting from our research was the slightly changed threat model compared to the original paper. Given possible improvements -particularly in regards to different targets, and more "intelligent" demodulation which is able to differentiate better between signal and noise-, it would indeed be interesting to see further measurements. Demodulation is also the area which we identify as the most "improvable" aspect of our research, with possible additions including machine learning, smarter error correction and time interval-independent frequency detection.

References

[1] Mordechai Guri. *POWER-SUPPLaY: Leaking Data from Air-Gapped Systems by Turning the Power-Supplies Into Speakers*. 2020. arXiv: 2005.00395 [cs.CR].

[2] Dex Bleeker, Mihai Macarie, and Utz Nisslmueller. *Project Repository on GitLab*. https://gitlab.utwente.nl/uni/asa_powersupplay. [Online; accessed 09-04-2021]. 2021.