Politecnico di Torino

Cybersecurity for Embedded Systems
01UDNOV

Master's Degree in Computer Engineering

Live Audio Watermarking

Project Report

Candidates:
Lorenzo Marino (317703)
Agostino Saviano (307946)
Niccolò Cacioli (305325)

Referee:
Prof. Paolo Prinetto

Contents

List of Figures

List of Tables

Abstract

With an increasing level of technology and an even higher progress of AI, attacking the credibility of high level positions has become more frequent and therefore assuring the authenticity of recorded conversations has also become way more important.

Here can be read how Watermarking can be properly used to guarantee the integrity of data sent over voice communication as a possible solution for this kind of problem. Watermarking allows to embed data into a signal and by doing this the author can eventually sign the data and certify the ownership of a communication.

Another useful tool descripted in this paper is the usage of high frequencies as a possible improvement to be able to operate and assure the properties of the signal through an inaudible range of the human hearing, thus being able to not interfere with a normal conversation and also to discreetly defend the talker itself.

Both these concepts have been studied and applied in order to propose a demo of an application that can be used to apply a watermark to a nearby recording microphone, implementing an easier and naive version, although still relevant and practical for this project.

Live watermarking can be a solution to prevent malicious attackers from counterfeiting and misusing simple voice commands or personal sensitive chats and this paper provides an all around guide to start studying these topics.

CHAPTER 1

Introduction

The goal of this chapter is to introduce the main technologies that are used to build a modern microphone: MEMS and ECM.

In the next chapter the watermarking procedure will be described in detail and how such technique helped to implement the use case of the project.
Chapter 3 describes the state of the art of modern day attacks and then in the last chapters the implementation and the results of the project will be discussed.

1.1 State of the Art: MEMS and ECM

1.1.1 MEMS Microphone

MEMS (Micro-Electro-Mechanical System) Microphones use a MEMS component placed on a printed circuit board (PCB) and protected with a mechanical cover. A small hole is fabricated in the case to allow sound into the microphone and is either designated as top-ported if the hole is in the top cover or bottom-ported if the hole is in the PCB. The MEMS component is often designed with a mechanical diaphragm and mounting structure created on a semiconductor die.

1.1.2 How it works

The MEMS diaphragm forms a capacitor and sound pressure waves cause movement of the diaphragm. MEMS microphones typically contain a second semiconductor die which functions as an audio preamplifier, converting the changing capacitance of the MEMS to an electrical signal. The output of the audio preamplifier is provided to the user if an analog output signal is desired. If a digital output signal is desired, then an analog-to-digital converter (ADC) is included on the same die as the audio preamplifier. A common format used for the digital encoding in MEMS microphones is pulse density modulation (PDM), which allows for communication with only a clock and a single data line. Decoding of the digital signal at the receiver is simplified due to the single bit encoding of the data. Digital I^2S outputs are a third option that include an internal decimation filter, which allows for processing to be completed in the microphone itself. This means the microphone can connect directly to a digital signal processor (DSP) or microcontroller, eliminating the need for an ADC or codec in many applications.

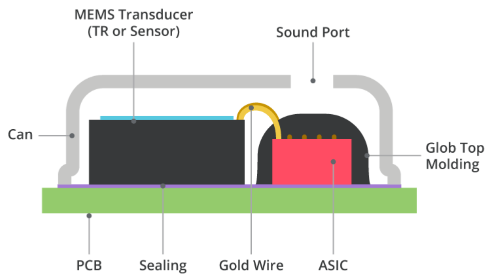


Figure 1.1: : MEMS Structure

1.1.3 ECM

An electret diaphragm (material with a fixed surface charge) is spaced close to a conductive plate, and similar to MEMS microphones, a capacitor is formed with the air gap as the dielectric. Voltage across the capacitor varies as the value of the capacitance changes due to sound pressure waves moving the electret diaphragm, $\Delta V = Q/ \Delta C$. The capacitor voltage variations are amplified and buffered by a JFET internal to the microphone housing. The JFET is typically configured in a common-source configuration, while an external load resistor and dc blocking capacitor are used in the external application circuit. [6]

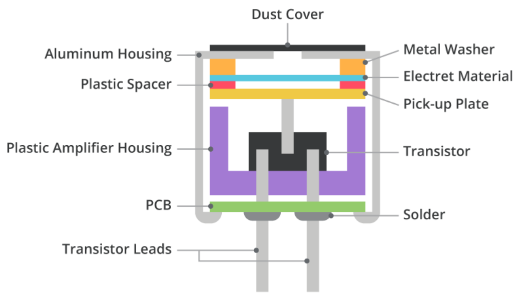


Figure 1.2: : ECM Structure

1.2 Main differences

Due to their small size, electrical noise immunity, and mechanical robustness, MEMS microphones are becoming increasingly popular. However, this does not mean that they are the de-facto best choice for every application. Many legacy applications may benefit from a simple change or upgrade to their current ECM. Apart from their excellent IP ratings, which allow them to perform well in harsh environments, the intrinsic nature of ECMs also makes them an excellent choice for applications that benefit from noise-canceling or unidirectionality.

If the space constraints placed upon a design are particularly acute (such as in smartphones, wearables, hearing aid implants, etc.), or there is likely to be the need to distribute multiple microphones throughout an item of equipment (like in a VR headset for instance) then MEMS may be the better option. Conversely, if elevated performance or resilience to challenging operating conditions are a priority, then ECM could prove to be the more appropriate route to take. Therefore, professional audio equipment, voice-controlled home assistants, voice recognition systems, and a wide range of other applications will continue to rely on these devices. [7]

During the last decade the usage of MEMS in common devices raised significatively (back in 2009 when Apple started to use MEMS microphones for iPhone 4) and since the project focused on such devices like smartphones due to the nature of the attack and the target of the attacks, the solution that has been implemented was on MEMS microphones. As shown in Figure 1.3 it is possible to understand why it is much more useful for the purpose of this project to consider MEMS technology rather than the ECM, it is sufficient to consider Average Growth Rate to take into account the rising of the first technology and the falling of the latter.

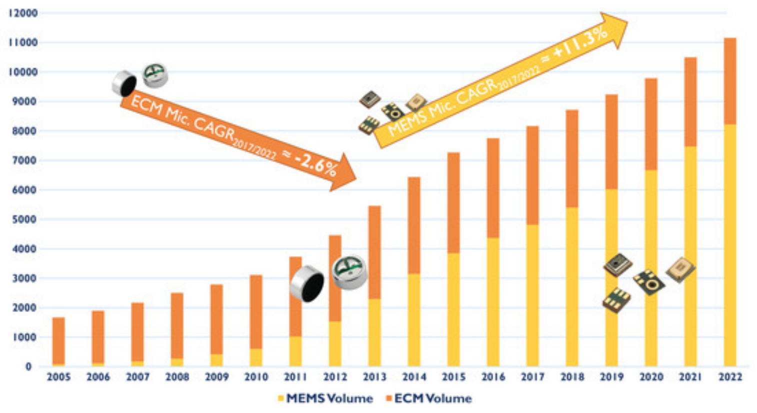


Figure 1.3: : Market Evolution

1.3 Exploiting Non-Linearity in MEMS microphones

Even though this effect is not exploited unless very high frequencies (order of 60kHz) are used it is really important to describe one of the core development of this project: the non-linearity of this technology.

Modules inside a microphone are mostly linear systems, in the case of the pre-amplifier with input S:

$$S_{out} = A_1 S$$

Acoustic amplifiers maintain a strong linearity only in the audible frequency range but outside of this range the response exhibits non linearity. Thus, for f > 25kHz, the net recorded S_{out} can be expressed as:

$$S_{out}\bigg|_{f>25} = \sum_{i=1}^{\infty} A_i S^i = A_1 S + A_2 S^2 + A_3 S^3 + ...$$

With the third and higher terms that are extremely weak and can be ignored.

To operate the microphone in its non-linear range, one can play a sound S composed of two tones $S_1 = 40$ and $S_2 = 50$ kHz. S = $Sin(2\pi 40t) + Sin(2\pi 50t)$. After passing through the diaphragm and pre-amplifier of the microphone, the output S_{out} can be modeled as:

$$S_{out} = A_1(S_1 + S_2) + A_2(S_1 + S_2)^2 =$$

$$= A_1[Sin(\omega_1 t) + Sin(\omega_2 t)] + A_2[Sin^2(\omega_1 t) + Sin^2(\omega_2 t) + 2Sin(\omega_1 t)Sin(\omega_2 t)$$

where $\omega_1 = 2\pi 40$ and $\omega_2 = 2\pi 50$.

The first order terms produce frequencies out of the microphone's cutoff, the second order term instead is a multiplication of signals, resulting in different frequencies for each component such as $2\omega_1$, $2\omega_2$, (ω_1 - ω_2) and (ω_1 + ω_2). Mathematically,

$$A_2(S_1 + S_2)^2 = 1 - \frac{1}{2}Cos(2\omega_1 t) - \frac{1}{2}Cos(2\omega_2 t) + Cos((\omega_1 - \omega_2)t) - Cos((\omega_1 + \omega_2)t)$$

With the cutoff frequency at 24kHz, every frequency gets filtered except for the $\omega_1 - \omega_2$ component, that results in a 10kHz tone, this allows a totally inaudible frequency to be recorded by the microphone. This is the main concept that revolves around this project and some possible future implementations and improvements. [2]

The goal of the project proposed is to show how some high frequency tones can carry some data thanks to Watermarking, anyway if the data can be carried at such high frequency the future improvement can basically carry these information at an even higher frequency thanks to the implementation of an amplifier and work at inaudible frequencies.

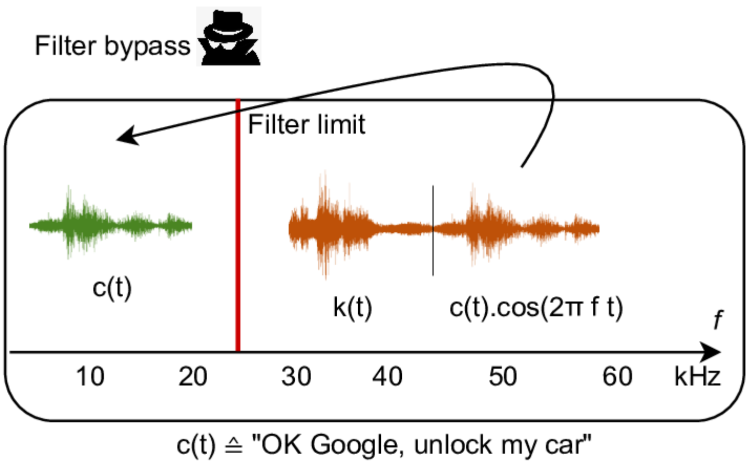


Figure 1.4: : Example of a filter bypass exploiting non linearity

CHAPTER 2

Watermarking

2.1 Introduction to watermarking

Watermarking is a technique used in both analog and digital works alike to embed *usually* hidden information about the origin, the status or recipient of the host data.

The core of the idea is to add a watermark signal to the data, in such a way that it cannot be easily detected by a standard user but can later be recovered from the mixed signal using the right key and algorithm. Since the watermark has to be concealed, its information is usually repeated all throughout the data with light modifications compared to the average amplitude of the original signal, so that it can always be recovered even if small amounts of watermarked data is available [1, page 4, "Basic Watermarking Principles"].

A generic watermark signal usually depends on a key and the watermark information; sometimes it also depends on the data into which it is embedded. The extracted watermark, or relative confidence measure, instead is obtained using the watermarked data and the key, either in conjunction with the original data or not [1, page 5, figs. 1-2, formulas 1-5].

Now, given the concept of a watermark, the needed qualities for a good watermark can be inferred [1] [1, page 3, "Requirements"] :

- Robustness against standard manipulation: one of the major requirements. All manipulation and modification that the data might be subjected to throughout the distribution chain: conversions between different digital formats, digital to analog conversion, editing, printing. An attack in this context is every manipulation done with the purpose of rendering the watermark unrecognizable, destroying it or altering it.
- Imperceptibility: perceptual transparency, or the inability for the user to notice the watermark. The watermarking process should not introduce perceptible artifacts in the data. Nonetheless, it is desirable for a robust watermark's amplitude to be as high as possible, which is in direct contrast with the imperceptibility requirement. The implication is that a threshold must be found that does not cross the fine line of perception for the end user, but the subjectivity of human viewers/listeners makes it very difficult to find. Only they can be the final judges and tell if the presence of the watermark is compromising the fidelity of the original data or not.
- Density: a watermark should convey as much information as possible. Date of delivery, original recipient, source and every additional piece of information apt to find clues on the origin and

[1]Note: all cited papers about watermarking had a dedicated section about requirements, Hartung & Kutter however went more into detail. Some were deemed more important to list than others.

use of the "incriminated" data. This is a soft requirement, because every use case has different demands. Some applications may need only to check if a watermark is present or not, while some others may need to embed large quantities of identifying data.

- Speed in embedding and retrieval: the algorithm should not be too complex or computationally expensive because real-time watermarking is a needed feature, especially in the case of live audio watermarking.

These qualities are what drives the development of different watermarking techniques, some of which will be better analysed and explained in the following chapters, focusing on the ones that have been found to be most useful within the scope of "live audio watermarking" but also giving a fast overview of other usages.

2.2 Common use cases

2.2.1 As means of digital copy protection

Watermarking is used on a wide variety of media, including text, video and audio. In this day and age the whole process, from creation and processing to distribution, is done digitally for all of the aforementioned media types. This provides many advantages, like transmission free of noise at a very cheap cost, ease of use through software and many more. Arguably, one the most important advantages of digital media is ease of reproduction without loss of fidelity. However, content providers do not find it a good quality, because it may hinder their ability to extract profits from their work. Mechanisms of digital copy protection are usually employed for this very reason (DRMs, or Digital Rights Management systems), but a paying user must have access to cleartext whom can then reproduce and illegally distribute copies. DRMs are usually successful in their purpose, at least for a starting period, but eventually always get circumvented. One last method for protecting IP rights is the insertion of a digital watermark into the data. Indeed its most frequent usage is copyright enforcement/protection. While it does not actively contrast illegal copying, it can be used at least as a last resort to identify source and destination of the multimedia data, allowing producers to engage in legal action if case should emerge [1, page 1, "Introduction"].

Let us make an example for clarity's sake. Let us imagine a producer that wants to do screening tests for their latest movie. They would need to send the product to untrusted parties that may leak it without the owner's approval. Even if they were trusted, a leak may always happen regardless (by means of hacking or lack of attention). How can a producer protect his or her work? One way would be to disincentivize sharing the protected work by threatening to cut ties, but this introduces the problem of identifying who did actually leak the work. This is where watermarking comes into use. A hidden piece of data, different for every subject whose work is sent to, is introduced into the movie itself. It is difficult if not impossible to notice but also robust to modifications. After obtaining the leaked data, an algorithm is used by the producing party to extract the watermark and identify who leaked the data.

This is of course not the only use case.

2.2.2 As an authentication method

Specifically, it is of interest the ability of a watermark to authenticate a recording.

A typical scenario is composed of a recording device and the people to be recorded (whether knowingly or not). The recording may be presented before a court of law and the person speaking may claim that the recording has been forged and so it is not valid as evidence. This puts the people that have the burden of proof in a difficult position because proving that an audio is forged is easy (audio tampering is easily spotted by looking at sudden cuts, differences in spectral signatures in different parts of the audio and abnormal peaks and valleys) but proving that it is not is really hard. In addition, in a court of law any alteration of the data after recording is considered tampering and consequently invalidating for evidence's purposes [3, pp. 2-3, "Live audio watermarking for forensics"].

By embedding a watermark into a live recording (of a private conversation for example, but it may also be used in public speeches) there is a guarantee that if a malicious agent tries to modify it, they will modify the watermark too. In that case, the person talking can prove that recording is fake by extracting the watermark from the original recording and comparing it to the faked one, showing that they differ.

CHAPTER 3

Attack research

What follows is a series of brief descriptions of the watermarking techniques that have been found through research on various papers. Each paper is given a short introduction and a rapid overview on the method employed.

3.1 BackDoor: Making Microphones Hear Inaudible Sounds

Roy et al. [2] describe a method to introduce disturbances in the recording of an audio signal in such a way that they cannot be heard by human beings, but do appear in the final data.
Being composed of frequency ranges outside the human hearing capabilities, a sound is designed specifically to exploit the non-linearities of the recording equipment, microphones found mainly in smartphones and IoT devices, so that a "shadow" sound appears in the audible range of the recording. Considering that this signal can be modulated, this enables the creation of a communication channel that can be used to transmit arbitrary data, on the order of 4kbps reported by the paper in certain conditions.
The idea of exploiting non-linearities in the recording equipment and of using ultrasonic signals for the live watermarking implementation has been inspired by this paper.

3.2 Tic-Tac, forgery time has run-up! Live acoustic watermarking for integrity check in forensic applications

Niță & Ciobanu [3] propose an interesting solution for the live audio watermarking problem. Their solution consists in the usage of a loud and recognizable sound, the ticking of a clock, to convey covert information. This brings a few advantages to usual watermarking methods:

- It is a sound ubiquitous in its presence all throughout the world, meaning that even if it is loud it does not bring disturbance to the people listening because it is familiar.
- Being impulse-like in nature, the bandwidth of its signal is wide, making it very hard to be extracted from a recording and reused in another one to make it seem original.

The Tic-Tac signal, with a period of 1s, is injected in-between periods with small but imperceptible delays that can be modulated to an arbitrary message.

Another method for watermark injection is also provided due to the low SNR of the one mentioned above. It consists in generating a watermarking signal made by mixing a ticking sound with a chirp signal. A matched filter is obtained for the chirp signal as the decoding key. Finally, a modified clock positioned inside a room will broadcast an audio signal with a predetermined pattern that repeats every x seconds as long as a recording is taking place. Only an authorized person that knows the adapted filter will be able to extract the watermark.

3.3 Robust audio watermarking in the time domain

Bassia and Pitas [4] in Robust audio watermarking employ a simple method for watermark injection into audio data. However only on the digital side, namely MPEG Layer III and II files.

The scheme proposed in the paper acts on the individual samples contained in the audio files, represented on either 16 or 8 bits, by changing the least significant bit of each one. The change is small enough in amplitude that it does not produce any perceptual difference. The key is the sequence of bits, which is randomly generated.

The detector does not use the original signal and provides a confidence measure normalized between 0 (no watermark detected) and 1 (fully detected watermark).

This method is solid enough to resist MPEG compression and both moving average and low pass filtering, with a worst case detection success of 99.8%.

3.4 Sonic watermarking

Ryuki Tachibana [5] describes a method based on frequency subdivision of the original signal with a multi-bit message embedded into it together with a synchronization signal.

The method is explained as follows. A sequence of power spectrums, calculated using short-term DFTs, is put together as a segmented area in a time-frequency plane called pattern block. Each pattern block is divided into tiles and the tiles in a row are called a sub-band. A pseudorandom number is used to assign values +1 and -1 to tiles and indicates their change in magnitudes. This pseudorandom number is the key given to the embedder and the detector.

The detection is performed by computing the magnitudes of the content for all the tiles, which in turn get correlated with the pseudorandom array used as a key.

The paper shows that this method is fast enough to compute to be used for live watermarking applications. Specifically, the paper tests different scenarios involving playing of solo, orchestral and popular music. The results showed very high detection rates for popular and orchestral music but acted very poorly on instrumental solos, because they often presented long intervals of silence. This implies preferred usage to be live shows and concerts, leaving speeches to other methods.

CHAPTER 4

Implementation Overview

4.1 Goal of the project

The goal of this project is to implement an inaudible live audio watermarking solution that makes it possible to authenticate a live recording, enabling a person to speak and simultaneously guarantee that a recording of their speech will not be either forged or altered.

The scope of the project is limited to recording devices that are available to most people, namely cheap and widespread ones: microphones embedded into smartphones, laptops, IoT devices and the like.

4.2 Issues to be solved

During the planning phase of the project, many issues surfaced on how to implement a solution with the required characteristics.

First and foremost, emphasis was put on providing a solution that would be easy to use and immediately available to end users. The choice fell on a smartphone app that could be readily downloaded from app stores [1].

After that, the question has been posed of which data to transmit that might be useful for authenticity, but especially *which* data to transmit, even though the proposed solution enables transmission of arbitrary packets of data (within limits). The application provides an easy way to embed GPS coordinates into the watermark, but the user is encouraged to include highly identifying info like e-mail addresses and ID numbers.

In chapter 2 the desirable qualities of a watermark have been addressed. Of those, it is indispensable for the watermark to be inaudible. The solution had to take into consideration a transmission method that would not affect the quality of the recording, ideally to not be perceivable at all.

This has been a highly limiting factor for the choice of the frequencies (for the chosen method employs frequency manipulation) in which to transmit the watermark data, together with the sampling rate of the voice recording equipment and the tendency for microphones and recording software to be tuned to the low-frequency bandwidth [2].

The suggested solution will be explained in further detail in the next section.

[1]Since our project delivers a Proof of Concept, the app is distributed as an *apk* file for Android smartphones only.

[2]Frequencies in the audible range go from a minimum of 20Hz to a maximum of 20kHz, depending on sex and age.

4.3 Solution overview

The implemented solution is split in two parts.

The first is the provided application, intended to be used the most and by frequent (such as people active in PR) and occasional users alike. It is developed in such a way to be reasonably configurable, portable by means of being installable on every smartphone and tuned for the average user. The message inside the watermark, whether to encrypt it or not, options for the frequency, inclusion of the GPS coordinates: every one of these can be changed inside the app in a dedicated menu.

The other half of the solution is a set of two tools to be used in conjunction that constitute the post-processing analysis capacity to check for the presence of a watermark inside a recording and, possibly, to extract the data hidden inside it. A MATLAB LiveScript for data detection and extraction and a CLI tool written in Kotlin that enables the MATLAB script, provided with the corresponding key, to decrypt the incoming string of bits.

An in-depth description of the solution, with its implementation details is presented in the next chapter.

CHAPTER 5

Implementation Details

In this chapter is present a description of the implementation of the project, motivating the design choices.

5.1 Preliminary evaluation of the target devices

As previously mentioned, the target of this work are the most common available recorders nowadays: smartphones. Notably, Samsung and Apple smartphones, owing to their widespread usage, have been evaluated for the project purpose. Specifically, the analysis focused on the audio signal captured by the default recording app on each device. Additionally, an Asus Windows PC was included in the evaluation process. To evaluate the feasibility of the watermark, an initial analysis of the frequency response of the recording system is needed. Firstly, the analysis has been conducted by recording audio samples from the smartphone devices, inside a regular room, without a specific noise, or sound. The power density spectrum of a not-marked recorded samples is depicted in figure5.1. In all the three cases the examined file is a MPEG-4 file (.m4a extension), with a sampling frequency $Fs = 44100Hz$, therefore the resulting hypothetical watermarking signal cannot exceed the limiting maximum frequency of about $22kHz$. The spectrum is extracted using Matlab's Audio Toolbox Spectrum Analyzer[1].

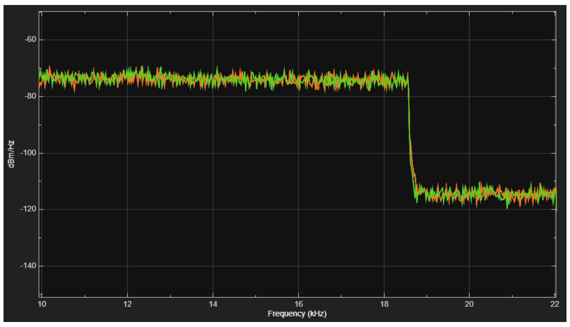
From the results, it can be inferred that a low-pass filter is applied to the audio data, indeed the frequency response shows a the steep reduction at a certain frequency: 16kHz for iPhone, 19kHz for windows and 20kHz for Samsung. The attenuation at high frequency, in this case, may stem from both the hardware itself and the software used for recording. As an additional test, the response to a generated signal[2] was analyzed, varying the frequency starting from 22 kHz and descending until the signal was clearly audible. The test confirmed that signals with frequency above the previously mentioned ones, are not detected, even at minimum distance and high intensity.

As a premise, it must been said that most adults have a hearing range below the 16kHz, but this value could change with respect to age (the lower the age the higher the frequency) and gender. This facts were proven by doing interviews, even though to a limited number of people. On average the best inaudibility results were obtained with frequency higher than 18 kHz, even if the generated tone remains audible to young female subjects.

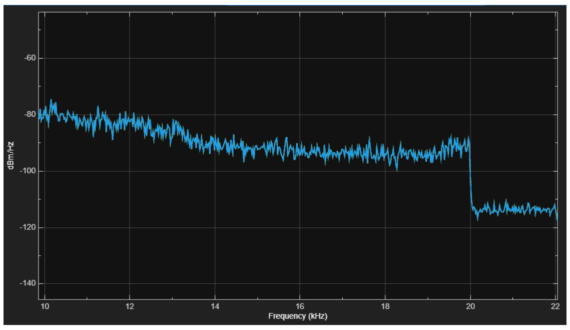
Consequently, considering watermark signal frequencies above this threshold results unfeasible with the apple products, while there is still some usable design range in Samsung's and Windows' devices.

[1]The spectrum can be seen by opening Matlab "Audio Test Bench" App, setting the file reader as input, clicking the button "Spectrum Analyzer" and then running the simulation.

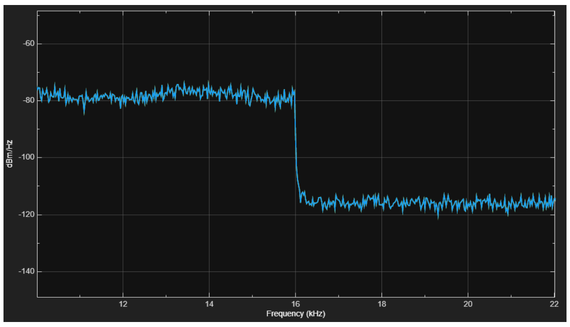
[2]The signal was generated using an Android application on a Samsung s20+ 5G. The application's operation was validated using the computer's microphone as the input source, in "Spectrum Analyzer".



(a) Density power spectrum of the signal recorded from Asus X507UF



(b) Density power spectrum of the signal recorded from Galaxy Tab S6 Lite



(c) Density power spectrum of the signal recorded from Iphone 14

Figure 5.1: Frequency analysis of the signals

5.2 Implementation of the channel of transmission

Considering the analysis performed by Roy et al. [2, page 5, BackDoor: Making Microphones Hear Inaudible Sounds] the solution for creating a communication channel consist in a FM modulated transmission. Differently from the paper approach, it is out of the scope of this project to result in a final frequency lower than the previously discussed range (e.g. 10 kHz), because it would lead to a perceivable sound in the final recording and therefore in a audible watermark. The channel used in this work is the data embedded in the recording with signals of which frequencies are greater or equal than 18 kHz, in order to occupy the higher end of the available spectrum.

5.2.1 Evaluation of the method for transmitting data

Digital data can be encoded and transmitted with different types of frequency modulation techniques, among the most common we selected ASK (Amplitude Shift Keying) and FSK (Frequency Shift Keying). In FSK the frequency of the signal can assume two values: f_1 for transmitting bit '1' and f_0 for transmitting bit '0'. The binary data is therefore transmitted by means of a sine wave, which change frequency accordingly, as illustrated in figure 5.2

The FSK modulation can be easily transformed into a binary type of ASK, OOK (On-Off Keying), by setting one of the two frequencies at 0 Hz. To easily detect the start and end of the transmission, two additional symbols (other than bit0 and bit1) have been added to the alphabet, therefore in a specific time interval the transmitted tone can assume 4 different frequencies (f_0, f_1, f_{start} and f_{stop}). An example frame of one of the possible transmission of the character 'b' is shown in figure 5.3, from left to right represented symbols are bit start, bit0, 2 times bit1, 3 times bit0, bit1, bit0 and finally

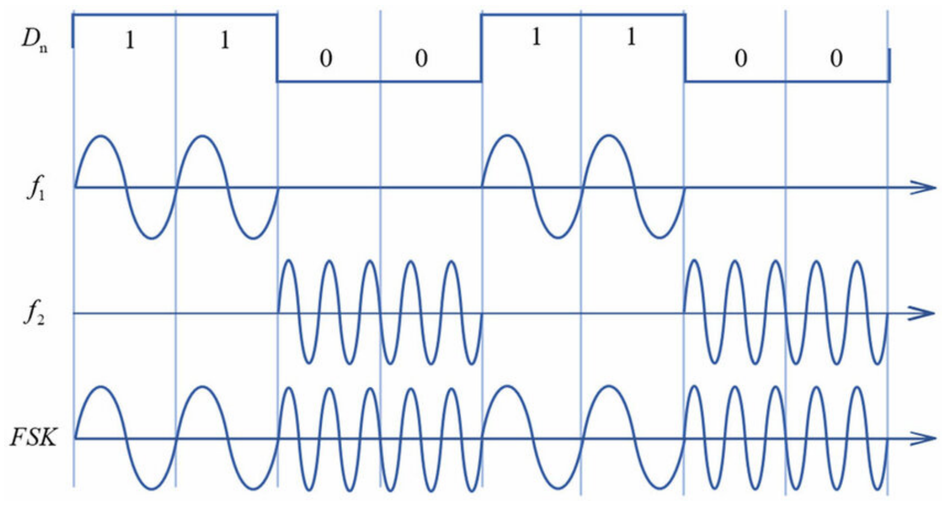


Figure 5.2: Visual representation of the construction of an FSK modulated signal

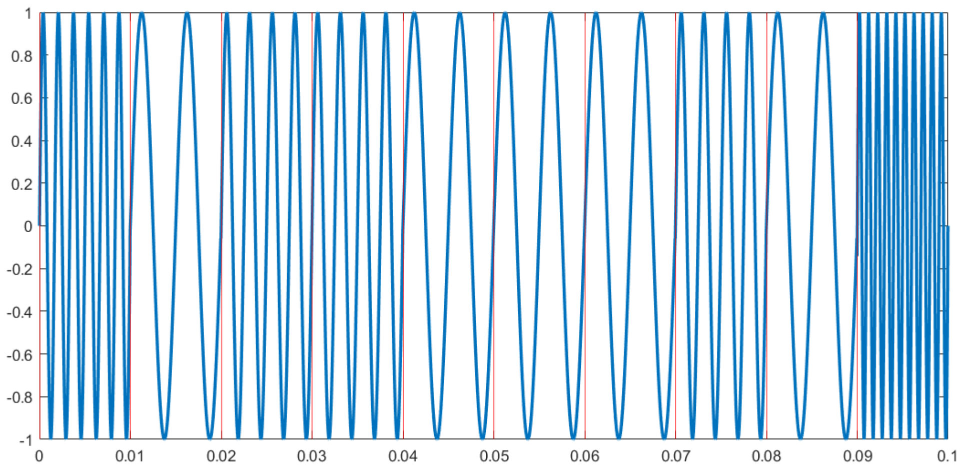
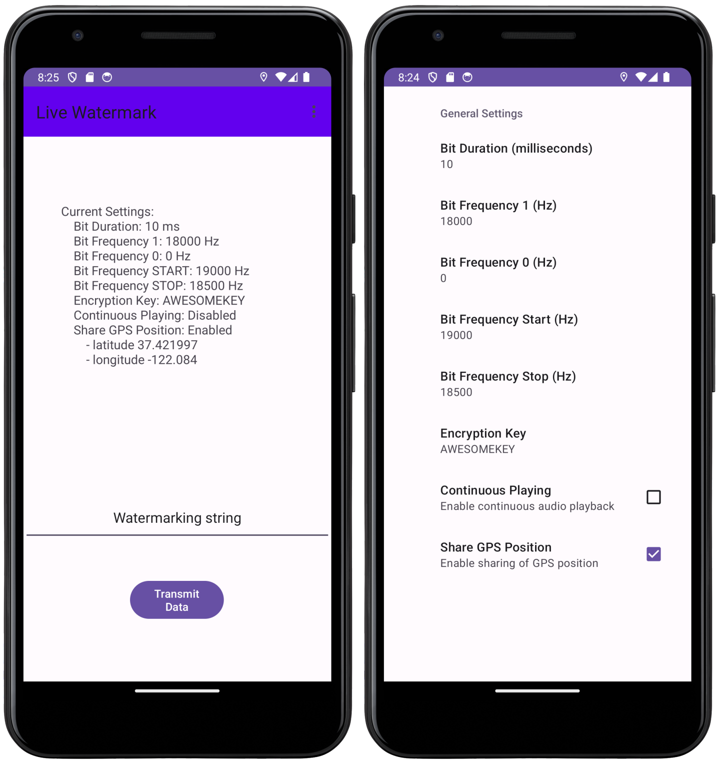


Figure 5.3: Example of transmitted signal for a message composed of the character 'b'

bit stop.

5.2.2 Implementation of the platform used for transmitting the signal

As previously discussed the most accessible way to use the watermark transmitter is by utilizing a smartphone app. For this purpose we developed a simple prototype application, in which the user can set the data and the parameters of the transmission. Moreover, since the application is installed on the user's smartphone, they can choose to embed meaningful information such as a real-time GPS position into the watermarking signal. The user is also able to set an encryption key that is then used to encrypt all the data to integrate in the signal through an "AES-256" algorithm, applied in Cipher Feedback Block mode, without Padding in order to keep the watermark track as short as possible. This mode can provide an additional protection against the modification of the audio sample, since, ideally, changing the order of the bits results in a message different from the transmitted one. The application has been developed in Kotlin language, with Android Studio IDE. The application is



Live Watermark

Current Settings:
Bit Duration: 10 ms
Bit Frequency 1: 18000 Hz
Bit Frequency 0: 0 Hz
Bit Frequency START: 19000 Hz
Bit Frequency STOP: 18500 Hz
Encryption Key: AWESOMEKEY
Continuous Playing: Disabled
Share GPS Position: Enabled
- latitude 37.421997
- longitude -122.084

Watermarking string

Transmit Data

General Settings

Bit Duration (milliseconds)
10

Bit Frequency 1 (Hz)
18000

Bit Frequency 0 (Hz)
0

Bit Frequency Start (Hz)
19000

Bit Frequency Stop (Hz)
18500

Encryption Key
AWESOMEKEY

Continuous Playing
Enable continuous audio playback ☐

Share GPS Position
Enable sharing of GPS position ☑

Figure 5.4: Main and Settings view of the application.

composed of two views: the home page and the settings.
From the main view the user can insert a string to use as a watermark. The transmission begins on pressing the "Transmit Data" button. In this prototype, there are no constraints on the format of the string or limitations on its size, except for the defaults set by the 'TextBox' object used. In the Settings view the user can change the timing parameters previously described. The following source code shows the implementation of the transmit function, the complete code can be found in Appendix B

```
    private fun transmitData(data: ByteArray) {
        val minBuff = AudioTrack.getMinBufferSize(
            44100,
            AudioFormat.CHANNEL_OUT_MONO,
            AudioFormat.ENCODING_PCM_16BIT
        )
        val audioTrack = AudioTrack(
            AudioManager.STREAM_MUSIC,
            44100,
            AudioFormat.CHANNEL_OUT_MONO,
            AudioFormat.ENCODING_PCM_16BIT,
            max((44100 * BIT_DURATION / 1000 * data.toString().length), minBuff),
            AudioTrack.MODE_STREAM
        )
        audioTrack.play()
        val bitData = dataToBits(data)
        val start = generateTone(FREQ_BIT_START, BIT_DURATION)
        val stop = generateTone(FREQ_BIT_STOP, BIT_DURATION)
        audioTrack.write(start, 0, start.size, AudioTrack.WRITE_BLOCKING)
        for (bit in bitData) {
            val tone = if (bit == '1') generateTone(FREQ_BIT_1, BIT_DURATION)
            else generateTone(FREQ_BIT_0, BIT_DURATION)
            audioTrack.write(tone, 0, tone.size, AudioTrack.WRITE_BLOCKING)
        }
        audioTrack.write(stop, 0, start.size, AudioTrack.WRITE_BLOCKING)
        handler.postDelayed({
            audioTrack.stop()
            audioTrack.release()
        }, 100 * BIT_DURATION.toLong()) //delay added to finish the transmission
    before stop
    }

private fun encrypt(plaintext: ByteArray, input_key: String): ByteArray {

        val key: SecretKey = SecretKeySpec(input_key.padStart(32, '0').toByteArray(),
    "aes")

        val emptyArray = byteArrayOf(
            0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
    ,0x00
        )

        // set cipher and options
        val cipher = Cipher.getInstance("AES/CFB/NOPADDING")
        cipher.init(Cipher.ENCRYPT_MODE, key, IvParameterSpec(emptyArray))

        return cipher.doFinal(plaintext)
    }


```

Listing 5.1: Code of the Transmit and encrypt Function.

The code makes use of other implemented function:

- `dataToBits`: is a function that generates the binary string of the data
- `generateTone`: is the function responsible of generating the vector containing the value of the tone to write on the stramed audioTrack.
- `encrypt`[3]: is the function that apply the encryption algorithm. In this simple prototype, the Initialization Vector is a string of zeros. To enhance uniqueness and unpredictability, alternative approaches can be adopted, such as incorporating additional data or a combination of different data points, such as the device serial number, IMEI or a timestamp.

5.3 Post-processing of the signal to retrieve the data

To simplify the post processing identification of the data, it has been considered OOK-like transmission, with default parameters $f_1 = 18kHz$, $f_{start} = 19kHz$, $f_{stop} = 18.5kHz$ and $tone\_length = 10ms$.
The analysis on the Audio track has been pursued with the help of Matlab. In particular, to automate the process and show the details of the signal, a Matlab Live Script has been coded. The script can be found in Appendix B.
Below, the steps and methods applied to extract the embedded data from the recorded audio file are described.

1. **extraction of the audio data from the MPEG-4 file**
   The data is stored into a vector using the program function "audioread", which extracts both the audio data and the sampling frequency from the file.

[3]Not explicitly called by the function, but conditionally executed on *data* argument before it is called.

2. **preliminary frequency analysis to prove the existence of the watermarking data**
   The analysis is made by performing the fft function on the vector of signal's data and observing the magnitude at the previously mentioned frequencies.

3. **Extraction of the raw watermark signal** The frequencies of interest are simply isolated through a band pass filter, exploiting Matlab's *bandpass* function.

4. **Frequency analysis with efficient DFT algorithm**
   The algorithm used is Goertzel algorithm, which is very efficient for detecting the presence of a particular frequency component in a signal. It is more computationally efficient than the Fast Fourier Transform (FFT) for detecting a single frequency component. The algorithm calculates the discrete Fourier transform (DFT) of a specific frequency directly, without the need to compute the entire spectrum as FFT does. It is often used in applications such as tone detection in telecommunications and audio processing. The resulting magnitudes are used to compute the thresholds used in the next step.

5. **Derivation of the transmitted binary string**
   The segment of the vector containing the watermark is isolated and extracted by searching the start and stop bit. To search for the right index, a cross correlation with a generated bit start tone is computed, the index of the maximum in the resulting vector represent the delay at which the start(stop) tone is found. Knowing the positions of these bits, it is possible to compute the number of tones present in the sample. Consequently each one segment is analyzed with Goertzel algorithm and compared against the threshold, which consists of the previous threshold multiplied by a constant empirically defined.

6. **Decryption and interpretation of the binary string**
   If the transmitted data is encrypted, it can be decrypted using the provided tool, whose decrypting function code is listed here, the full code can be found in Appendix B.

```
    fun decrypt(ciphertext: ByteArray, input_key: String): ByteArray {

    // exactly same code as function "encrypt" but in decryption mode
    val key: SecretKey = SecretKeySpec(input_key.padStart(32, '0').toByteArray(),
     "aes")
    val iv = byteArrayOf(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0
    x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00)
    val cipher = Cipher.getInstance("AES/CFB/NOPADDING")
    cipher.init(Cipher.DECRYPT_MODE, key, IvParameterSpec(iv))

    return cipher.doFinal(ciphertext)
}
```

   The code use the same library and approach that can be found in the application code.

Also a variant of the live script has been developed, it simply record some second of the input data and then performs the same analysis

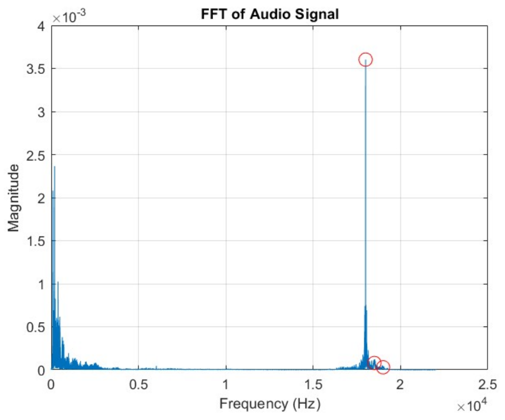
CHAPTER 6

Results

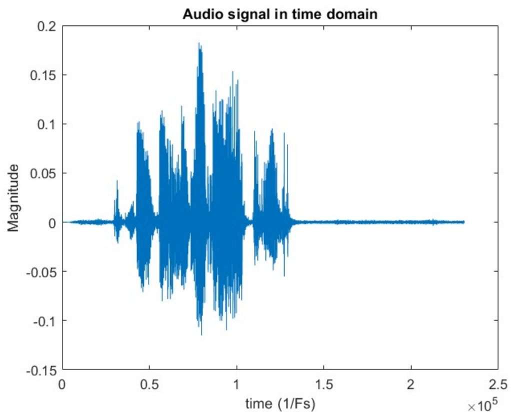
A watermarked audio sample has been recorded[1] and then analyzed with the script listed in Appendix B. Here the results are reported and discussed.

6.1 Analysis of the result of a watermarked audio recording.

As a first step, let's analyze the spectrum and magnitude of the "raw" signal. In figure 6.1 is reported the spectrum of the signal on the left (6.1a) and the plain signal on the right (6.1b). The audio sample comprises a sentence ("example of watermarked audio") that has been marked using the previously described application. The watermark is set to a simple string ('ciao') and a GPS location, both encrypted with the methods previously explained, using as key the string *"AWESOMEKEY"*.



(a)

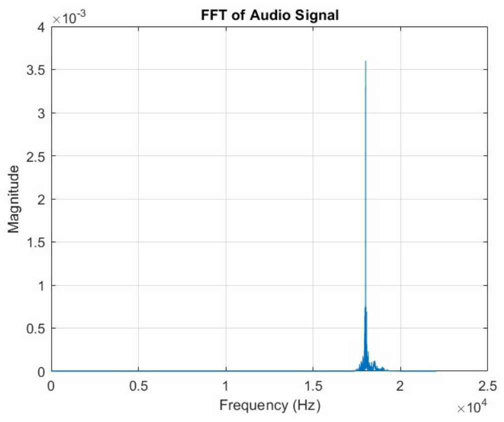


(b)

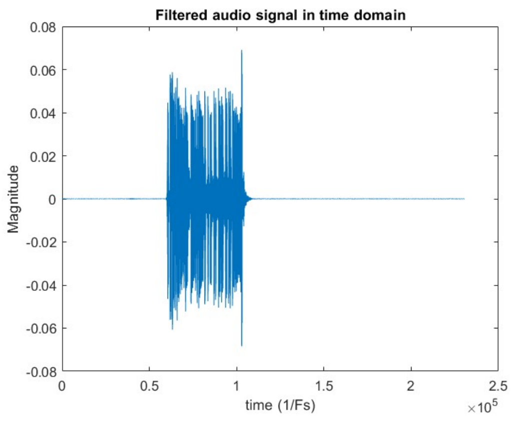
Figure 6.1: Unprocessed audio sample

It can be observed from the figure that there is some content at the desired frequency, indicating that the watermark has been successfully embedded into the audio file. The result of the applied band-pass filter is shown in figure 6.2. The filtered data contains only the audio watermark. When listening to the processed signal, no noise can be heard.

[1]The recording was made with Samsung Tab S6 Lite. The audio watermark, however, is emitted from a Samsung S20+ 5G.



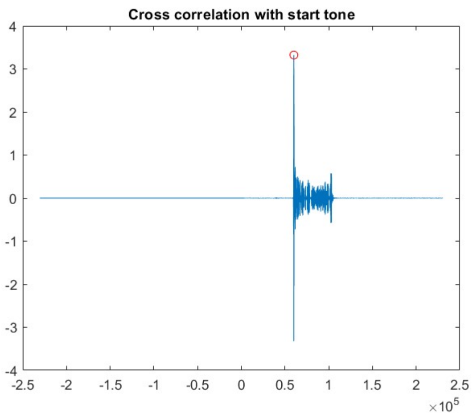
(a)



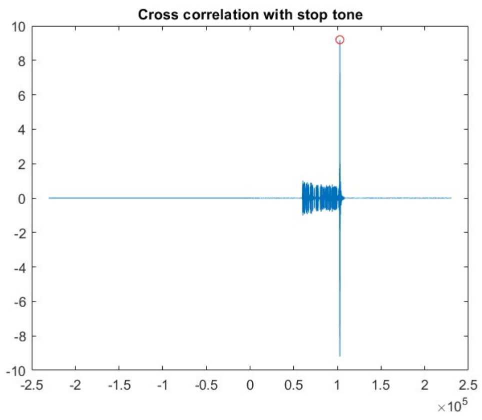
(b)

Figure 6.2: pre-processed audio sample

The subsequent phase involves determining the positions of both the start and stop bits. For this purpose, cross-correlation analysis was conducted using the 'xcorr' command in Matlab between a tone generated in the script and the pre-processed audio track. The results of these analyses are displayed in figure 6.3. The red circle indicates the index (more precisely the *lag* with respect the start of the audio sample) at which the tone begins.



(a)



(b)

Figure 6.3: cross correlation results

After identifying these tones, we can deduce the length of the watermark. By knowing the duration of the tone, we can extract segments for further analysis using additional signal processing techniques. a representation of the segments can be found in figure 6.4

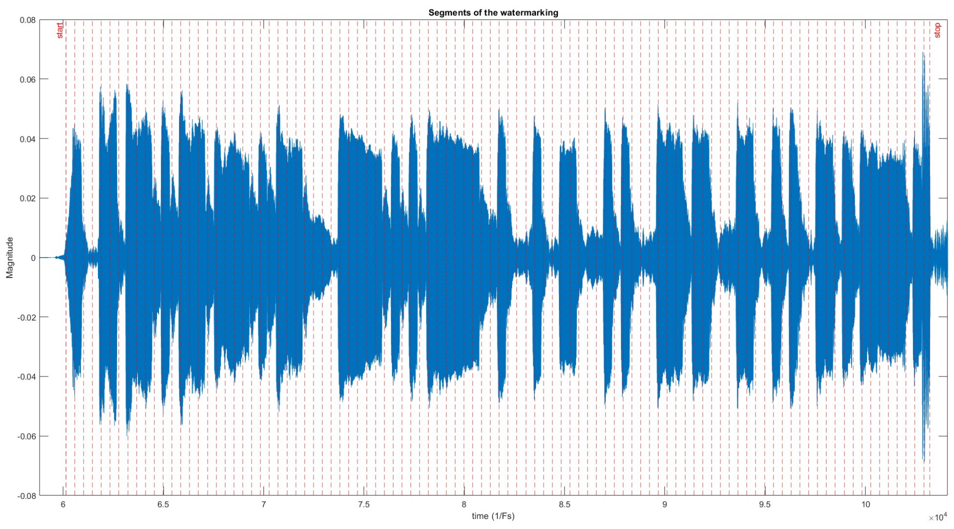


Figure 6.4: Segments representing the tones in the audio sample.

Before assessing the segments, it is essential to establish a threshold that allows to determine whether a frequency (and consequently a symbol) is present within a segment or not. To accomplish this objective, an analysis with Goertzel algorithm is made, the result are showed in figure 6.5.

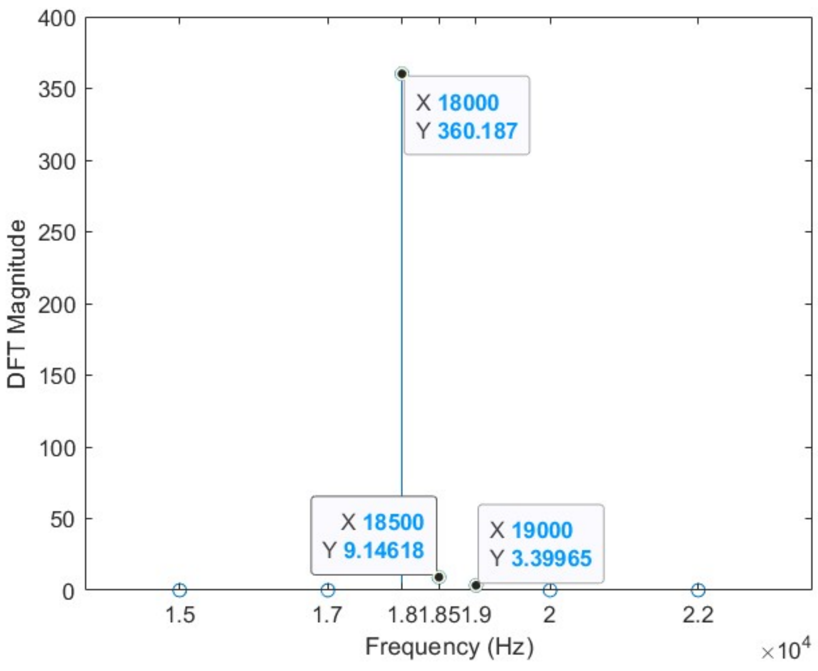
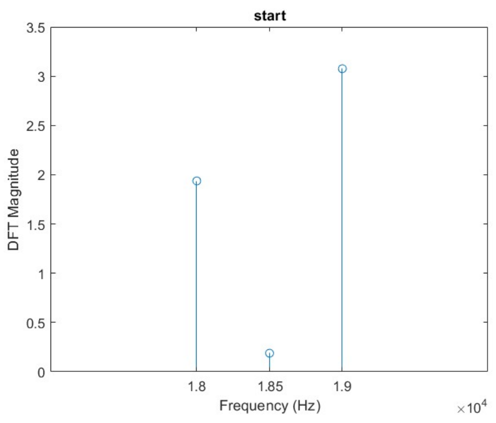


Figure 6.5: Analysis with Goertzel Algorithm

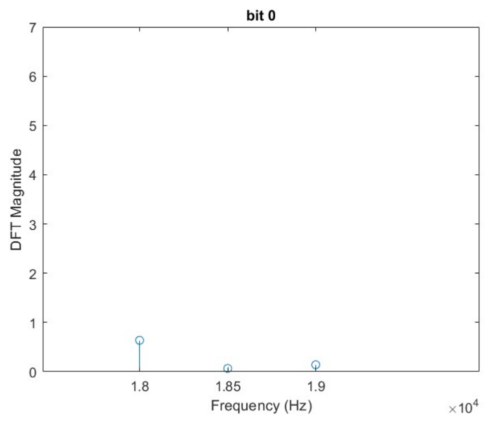
The values highlighted in the figures changes in every recording, therefore the needed threshold depends on the energy of the signal, that is not guaranteed to be constant throughout the sample (i.e. within a segment, one may find a magnitude different from that in another segment representing the same symbol). A simplistic approach has been adopted: Assuming the frequency is distributed across all segments, one can calculate a sort of average energy by dividing the magnitude of the frequency in the entire signal by the total number of segments. This value can then be adjusted as needed (since each track is unique) by simply multiplying it by a constant. In the context of this specific scenario, the number of segments has been computed. Consequently, the mean value is computed by dividing the overall signal magnitude by this quantity.

$$Thr_{symb} = k \times \frac{ABS(goertzel(f_{symb}))}{N_{seg}}$$

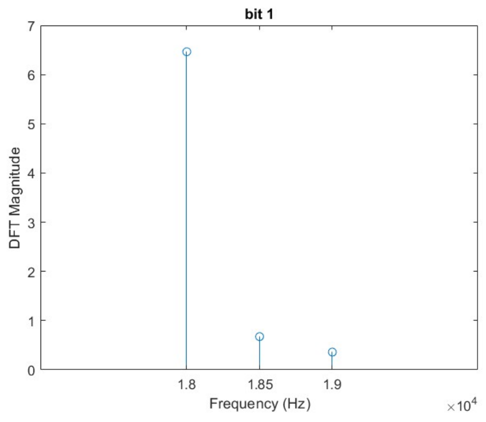
In light of the aforementioned reasons the determination of k, in this project, is empirical, the attributed value is 0.8. For the start and stop bit, since the energy is cointained in a single segment, the chosen threshold is half of the total magnitude. In figure 6.6 the results of the Goertzel algorithm applied to the segments are depicted. Specifically, an analysis per symbol is presented. From the represented values, the recognition mechanism of each one is easily discernible: each magnitude is compared to its threshold and the magnitudes of the other symbols, yielding the result readily.



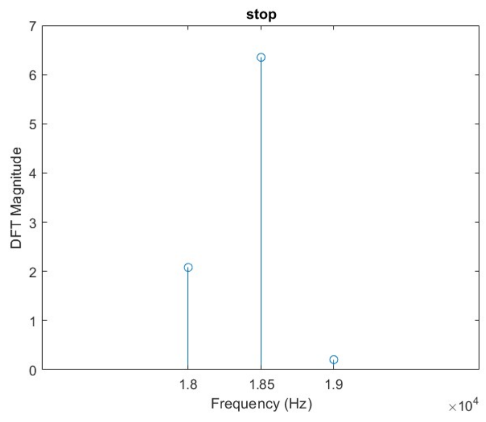
(a)



(b)



(c)



(d)

Figure 6.6: Frequency encoding of the symbols

In this case the received binary string is "9BAEF5C3EAFE898A3B194D7D", that can be represented as a Unicode string, but the encrypted output manifests as a sequence of seemingly random characters, devoid of any meaning. At this stage, the solution to retrieve the data, is to employ the designated decryption tool. The tool is invoked with Matlab's command `system()`, passing as arguments the binary string (in binary representation) and the previously defined encryption key, the output is saved in case further operation are needed.

In this experiment the resulting string still contains unintelligible characters: those are the latitude and longitude values of the GPS location, that needs to be converted into a IEEE 754 float representation. Finally the watermark is successfully extracted and the information are in human readable form.

6.2 Known Issues

The work described in the preceding sections was conducted in a controlled environment, with the following characteristics:

- **The room was not noisy**, even if the audible noise (e.g. air condition, chatter etc), does not have impact on the selected frequency and it is filtered.
- **The watermarking device was situated a few centimeters away from the microphone**, to guarantee a sufficient power to detect the signal
- **The transmission was kept short**, since the watermark is extracted with simple signal processing technique
- **The bitrate was set to an already verified value**, since the tone length was set to 10 ms the bitrate is 100 bit/s, that is sufficient considering recordings of a minimum duration of few seconds, but relatively slow considering the sampling frequency.

This characteristics stem from the issues encountered during the experimental trials, which are:

- **Non-linearity of the smartphone's speaker:** when playing the audio watermark, sudden frequency shifts occurs. In an ideal case, this should not result in any noise coming from the device, but in this implementation, a *"crackling"* sound is audible. This sound is below the designed frequencies, indeed it is removed by the band pass filter. At the current state of the project, the source of this issue is not totally certain but possible improvement are discussed in the *Future Work* section.
- **Limiting range of near-ultrasound frequencies tones:** as previously discussed the implementation has been pursued with devices that are suitable for the selected frequency range, therefore the functionality is not guaranteed on other devices (e.g. Iphone). This issue affects also the recognition of tones with the cross correlation, since the tones have similar frequency.
- **Possible instability of the developed software:** the developed application is a prototype, therefore the performance and correctness of the code is not guaranteed: some delays have been noticed in long transmission (e.g. irregular tone duration) that can lead to misinterpretation of the symbols, because of misalignment of the symbol's window. Furthermore, with the current software, setting a lower duration of the tone may result in a tone skip, or no output at all.
  The output of the first tone has some latency, resulting in having a lower magnitude with respect to the others. The script for decoding the watermark may need *"ad-hoc"* tuning for retrieving sample in some condition (e.g. changing the threshold, failing to detect start/stop tone).

6.3 Future Work

Considering the issues listed in the previous section, we can outline some potential improvements to be addressed as future work. The first improvement consist into a better quality of the watermarked signal: starting from the software, an experience app developer or a more in depth research endeavor, can make application transmit a more sharper signal, solving the issues related to timing performances. Also, alternative approaches to the chosen modulation technique can improve both the density of the watermark and its robustness, for example APSK (Amplitude and phase-shift keying) allows for a lower bit error rate, at the cost of the increased complexity in the development of a transmitting e receiving software. Regarding the hardware, utilizing an external device such as a mini speaker equipped with ultrasonic transducers, connected to the smartphone's audio interface (e.g., audio jack, Bluetooth, USB-C), could significantly enhance the signal. This external device has the capability to reproduce '0' and '1' bits at a higher frequency, ensuring superior performance and potentially amplified power. Additionally, the transmitted signal can be split into stereo channels, allocating the right channel for one bit symbol and the left channel for the other, further optimizing the transmission process. Furthermore, the ultrasonic transducers, being capable of working in ultrasound frequency, open up the possibility of injecting a signal into frequencies near ultrasound, leveraging the intrinsic nonlinearities of the MEMS microphones.

Another area of enhancement of this work is to developed a better post processing analysis, exploiting more complex and advanced technique used in modern digital signal processing. An additional feature that can be developed is live processing, making the less experienced users able to easily check the presence of the watermark and its content.

Regarding the data to be embedded in the signal, there exists a plethora of possibilities. Unique device identification codes, such as IMEI numbers, can be retrieved. Additionally, a wide array of sensors available on any smartphone offers diverse choices for data incorporation.

CHAPTER 7

Conclusions

To sum up the previous chapters and also give a brief recap it is possible to state that this project showed how watermarking and high frequency signals can be used to manipulate data.

This project, through the studying of the state of the art of the current technologies and techniques, took into account the weaknesses of such technologies and how concepts like non linearity could be used to obtain useful and interesting results regarding this study on security over voice communication.

High frequency signals, at first, were pretty useful at least in a theoretical way, with more powerful equipment and a refined code it is possible to eventually inject data into these communications and this can have a double effect: a malicious attacker can modify data and alter the whole meaning of a phrase. Otherwise a "signature" can be embedded in the signal to certificate the source and the content of the signal.

The concept of watermarking, thanks to its properties, came to hand and was put into practice in an early version to be the "signature" cited above. The idea was to use watermarking as an assurance of authenticity and consistency of the information and guarantee the integrity of the information transmitted through common modern day microphones.

To implement all these concepts it became necessary to develop an Android app, this app was designed to be simple and easy to use, also a Matlab script has been wrote to analyze the correctness of the watermarking applied to the signal and to visualize its frequency spectrum.

The long term goal of this project can be to improve the equipment and use this work as a starting point for further and more refined implementations of this approach towards audio watermarking for authentication purposes.

Bibliography

[1] Frank Hartung, Martin Kutter, ”Multimedia Watermarking Techniques”, 1999.

[2] Nirupam Roy, Haitham Hassanieh, Romit Roy Choudhury, ”BackDoor: Making Microphones Hear Inaudible Sounds”, 2017.

[3] V.A. Niță, A. Ciobanu, ”Tic-Tac, Forgery Time Has Run-Up! Live Acoustic Watermarking For Integrity Check in Forensic Applications”, 2018.

[4] P.Bassia, I.Pitas, ”Robust audio watermarking in the time domain”, 2015.

[5] Ryuki Tachibana, ”Sonic Watermarking”, 2004.

[6] Bruce Rose, https://www.cuidevices.com/blog/comparing-mems-and-electret-condenser-microphones

[7] Ryan Smoot, https://www.electronicproducts.com/ecm-vs-mems-microphones-does-new-mean-better/

APPENDIX A

User Manual

A.1 Analysis with Matlab Live Script

To replicate the example, a provided audio sample can be used. In order to get the previously described result the script (*"fsk_decoder.mlx"*), the audio sample(*"Test.m4a"*) and the decryption tool (*"decode.jar"*) must be in the same folder. To run the experiment simply open the script and press run (or observe the already existing data) A variant of the script where you can record directly from the microphone of the hosting device, and then run the analysis is available and its name is "mic_fsk_decoder.mlx". The latter is useful for experimenting with the Live Watermark app, see the relative section for information about the app.

A.2 Installation of the App

An APK file (*"LiveWatermark.apk"*) is available to easily install the application on a Device. To work and experiment with the live scripts the same parameters must be set. The recommended one are:

- FREQ_BIT_START = 19000 Hz
- FREQ_BIT_1 = 18000 Hz
- FREQ_BIT_0 = 0 Hz
- FREQ_BIT_STOP = 18500 Hz
- tone_length = 10 ms;

Remember to modify accordingly the Settings in the application (tree dots menu - Settings) or the parameters in the live script (at the beginning of the script).

A.3 Importing the App source code

Here are reported the steps to import the source code.

1. Install Android Studio from this link.
2. Unzip the Archive "LiveWatermark.zip" in a directory preferably without spaces

3. Open Android Studio
4. Go To File - Open... and Select the extracted folder.
5. Building the code should install automatically the dependencies

A.4 Compiling the decrypt tool

The source file of the app is in the folder ”decrypt tool” and consist only of the file ”main.kt” To compile the code download the command line compiler from here, then from Terminal launch the following command: ”`kotlinc main.kt -include-runtime -d decoder.jar`”.

APPENDIX B

Code

B.1 Live Watermark application code

```
package com.example.myapplication

import android.Manifest
import android.content.Context
import android.content.Intent
import android.content.pm.PackageManager
import android.location.LocationManager
import android.media.AudioFormat
import android.media.AudioManager
import android.media.AudioTrack
import android.os.Bundle
import android.os.Handler
import android.os.Looper
import android.view.Menu
import android.view.MenuItem
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity
import androidx.appcompat.widget.Toolbar
import androidx.core.app.ActivityCompat
import androidx.core.content.ContextCompat
import androidx.navigation.ui.AppBarConfiguration
import androidx.preference.PreferenceManager
import com.example.myapplication.databinding.ActivityMainBinding
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.launch
import java.lang.Integer.max
import android.location.Location
import android.location.LocationListener
import android.widget.Toast
import java.nio.ByteBuffer
import javax.crypto.Cipher
import javax.crypto.SecretKey
import javax.crypto.spec.IvParameterSpec
import javax.crypto.spec.SecretKeySpec


class MainActivity : AppCompatActivity(), LocationListener {


    private var FREQ_BIT_START = 19000  // 19 kHz
    private var FREQ_BIT_STOP = 18500  // 19 kHz
    private var FREQ_BIT_1 = 18000  // 18 kHz
    private var FREQ_BIT_0 = 0  // no tone
    private var BIT_DURATION = 20  // Duration of each bit in milliseconds
    private lateinit var appBarConfiguration: AppBarConfiguration
    private lateinit var binding: ActivityMainBinding
    private lateinit var locationManager: LocationManager
    private val handler: Handler = Handler(Looper.getMainLooper())
    private val locationPermissionCode = 2
    private var latitude = 0f
    private var longitude = 0f
    private var USERKEY = ""

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById<Toolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
        val transmitButton: Button = findViewById(R.id.transmitButton)
        val dataToTransmit: EditText = findViewById(R.id.dataEditText)

        //retrieve the settings
        val sharedPreferences = PreferenceManager.getDefaultSharedPreferences(this)

        //retrieve gps data and update preferences
        if (sharedPreferences.getBoolean("share_gps_position", false))
            getLocation()

        //update parameters
        BIT_DURATION = sharedPreferences.getString("bit_duration", "100")?.toIntOrNull() ?: 10
        FREQ_BIT_0 = sharedPreferences.getString("bit_frequency_0", "0")?.toIntOrNull() ?: 0
        FREQ_BIT_1 = sharedPreferences.getString("bit_frequency_1", "18000")?.toIntOrNull() ?: 18000
        FREQ_BIT_START =
            sharedPreferences.getString("bit_frequency_start", "19000")?.toIntOrNull() ?: 19000
        FREQ_BIT_STOP =
            sharedPreferences.getString("bit_frequency_stop", "18500")?.toIntOrNull() ?: 18500
        USERKEY = sharedPreferences.getString("encryption_key", "") ?: ""


        transmitButton.setOnClickListener {
            GlobalScope.launch(Dispatchers.IO) {
                var transData = byteArrayOf() //data to transmit
                if (!sharedPreferences.getBoolean("share_gps_position", false))
                    transData += dataToTransmit.text.toString().toByteArray() //adding the GPS position to the data to send
                else {
                    transData += floatToByteArray(latitude)
                    transData += floatToByteArray(longitude)
                    transData += dataToTransmit.text.toString().toByteArray()
                }
                if (USERKEY != "") { //if a key s specified the data is encrypted
                    transData = encrypt(transData, USERKEY)
                }

```

```kotlin
                do {
                    transmitData(transData)
                } while (sharedPreferences.getBoolean("continuous_playing", false)) // continuous playing
            }
        }

    }

    private fun floatToByteArray(value: Float): ByteArray {
        val buffer = ByteBuffer.allocate(4)
        buffer.putFloat(value)
        return buffer.array()
    }

    override fun onResume() {
        super.onResume()
        val sharedPreferences = PreferenceManager.getDefaultSharedPreferences(this)
        // Retrieve the TextView
        val settingsTextView = findViewById<TextView>(R.id.settingsTextView)

        // Retrieve and format the current settings
        val bitDuration = sharedPreferences.getString("bit_duration", "100")?.toLongOrNull() ?: 100
        val bitFrequency1 =
            sharedPreferences.getString("bit_frequency_1", "18000")?.toIntOrNull() ?: 18000
        val bitFrequency0 = sharedPreferences.getString("bit_frequency_0", "0")?.toIntOrNull() ?: 0
        val bitFrequencyStart =
            sharedPreferences.getString("bit_frequency_start", "19000")?.toIntOrNull() ?: 19000
        val bitFrequencyStop =
            sharedPreferences.getString("bit_frequency_stop", "18500")?.toIntOrNull() ?: 18500
        val encryptionKey = sharedPreferences.getString("encryption_key", "") ?: ""
        val continuousPlaying = sharedPreferences.getBoolean("continuous_playing", false)
        val shareGpsPosition = sharedPreferences.getBoolean("share_gps_position", false)

        // Create a formatted string with the current settings
        val settingsText = """
            Current Settings:
                Bit Duration: $bitDuration ms
                Bit Frequency 1: $bitFrequency1 Hz
                Bit Frequency 0: $bitFrequency0 Hz
                Bit Frequency START: $bitFrequencyStart Hz
                Bit Frequency STOP: $bitFrequencyStop Hz
                Encryption Key: $encryptionKey
                Continuous Playing: ${if (continuousPlaying) "Enabled" else "Disabled"}
                Share GPS Position: ${if (shareGpsPosition) "Enabled" else "Disabled"}
                    - latitude $latitude
                    - longitude $longitude
        """

        // Update the TextView with the formatted settings
        settingsTextView.text = settingsText
        //update parameters
        BIT_DURATION = bitDuration.toInt()
        FREQ_BIT_0 = bitFrequency0
        FREQ_BIT_1 = bitFrequency1
        FREQ_BIT_START = bitFrequencyStart
        FREQ_BIT_STOP = bitFrequencyStop
        USERKEY = encryptionKey
        if (shareGpsPosition) getLocation()
    }


    private fun getLocation() {
        locationManager = getSystemService(Context.LOCATION_SERVICE) as
    LocationManager
        if ((ContextCompat.checkSelfPermission(
                this,
                Manifest.permission.ACCESS_FINE_LOCATION
            ) != PackageManager.PERMISSION_GRANTED)
        ) {
            ActivityCompat.requestPermissions(
                this,
                arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
                locationPermissionCode
            )
        }
        locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 5000, 5f,
     this)
    }

    override fun onLocationChanged(location: Location) {
        latitude = location.latitude.toFloat()
        longitude = location.longitude.toFloat()
    }

    override fun onRequestPermissionsResult(
        requestCode: Int,
        permissions: Array<out String>,
        grantResults: IntArray
    ) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults)
        if (requestCode == locationPermissionCode) {
            if (grantResults.isNotEmpty() && grantResults[0] == PackageManager.
    PERMISSION_GRANTED) {
                Toast.makeText(this, "Permission Granted", Toast.LENGTH_SHORT).show()
            } else {
                Toast.makeText(this, "Permission Denied", Toast.LENGTH_SHORT).show()
            }
        }
    }

    private fun encrypt(plaintext: ByteArray, input_key: String): ByteArray {

        val key: SecretKey = SecretKeySpec(input_key.padStart(32, '0').toByteArray(),
    "aes")

        val emptyArray = byteArrayOf(
            0x00,
            0x00,
            0x00,
            0x00,
            0x00,
            0x00,
            0x00,
            0x00,
            0x00,
```

```kotlin
            0x00,
            0x00,
            0x00,
            0x00,
            0x00,
            0x00,
            0x00
        )

        // set cipher and options
        val cipher = Cipher.getInstance("AES/CFB/NOPADDING")
        cipher.init(Cipher.ENCRYPT_MODE, key, IvParameterSpec(emptyArray))

        return cipher.doFinal(plaintext)
    }

    override fun onCreateOptionsMenu(menu: Menu?): Boolean {
        menuInflater.inflate(R.menu.menu_main, menu)
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.action_settings -> {
                // Open the settings activity
                val intent = Intent(this, SettingsActivity::class.java)
                startActivity(intent)
                true
            }

            else -> super.onOptionsItemSelected(item)
        }
    }


    private fun generateTone(frequency: Int, durationMs: Int): ShortArray {
        val sampleRate = 44100
        val numSamples = (sampleRate * durationMs / 1000)
        val sample = ShortArray(numSamples)
        val angle = 2.0 * Math.PI * frequency / sampleRate

        for (i in 0 until numSamples) {
            sample[i] = (32767.0 * Math.sin(i * angle)).toInt().toShort()
        }
        return sample
    }

    private fun stringToBits(input: String): String {
        val binaryStringBuilder = StringBuilder()

        for (char in input) {
            val asciiValue = char.toInt()
            val binaryChar = Integer.toBinaryString(asciiValue)

            // Ensure that the binary representation has 8 bits
            val paddedBinaryChar = binaryChar.padStart(8, '0')

            binaryStringBuilder.append(paddedBinaryChar)
        }

        return binaryStringBuilder.toString()
    }

    private fun dataToBits(input: ByteArray): String {
        val binaryStringBuilder = StringBuilder()

        for (byte in input) {
            val binary = Integer.toBinaryString(byte.toInt() and 0xFF)

            // Ensure that the binary representation has 8 bits
            val paddedBinary = binary.padStart(8, '0')

            binaryStringBuilder.append(paddedBinary)
        }

        return binaryStringBuilder.toString()
    }

    private fun transmitData(data: ByteArray) {
        val minBuff = AudioTrack.getMinBufferSize(
            44100,
            AudioFormat.CHANNEL_OUT_MONO,
            AudioFormat.ENCODING_PCM_16BIT
        )
        val audioTrack = AudioTrack(
            AudioManager.STREAM_MUSIC,
            44100,
            AudioFormat.CHANNEL_OUT_MONO,
            AudioFormat.ENCODING_PCM_16BIT,
            max((44100 * BIT_DURATION / 1000 * data.toString().length), minBuff),
            AudioTrack.MODE_STREAM
        )
        audioTrack.play()
        val bitData = dataToBits(data)
        val start = generateTone(FREQ_BIT_START, BIT_DURATION)
        val stop = generateTone(FREQ_BIT_STOP, BIT_DURATION)
        audioTrack.write(start, 0, start.size, AudioTrack.WRITE_BLOCKING)
        for (bit in bitData) {
            val tone = if (bit == '1') generateTone(FREQ_BIT_1, BIT_DURATION)
            else generateTone(FREQ_BIT_0, BIT_DURATION)
            audioTrack.write(tone, 0, tone.size, AudioTrack.WRITE_BLOCKING)
        }
        audioTrack.write(stop, 0, start.size, AudioTrack.WRITE_BLOCKING)
        handler.postDelayed({
            audioTrack.stop()
            audioTrack.release()
        }, 100 * BIT_DURATION.toLong()) //delay added to finish the transmission
    }


}
```

B.2 Matlab Live Script

```
%%
% exctract data from file:

clear all
%%PARAMETERS%%
FREQ_BIT_START = 19000;
FREQ_BIT_1 = 18000;
FREQ_BIT_STOP = 18500;
tone_length=10e-3; %length in seconds
```

```matlab
audioFile = "Test.m4a";
key='AWESOMEKEY';

% Read the audio file.
[x, Fs] = audioread(audioFile);
plot(x);
sound(x, Fs)
%%
% Now "x" contains all the sampled data present in the audio file. we can check
% the spectum of the signal with fft:

N = length(x); % Length of the audio signal
Y = fft(x);    % Compute the FFT
%%
% The prevous is the double side spectrum, we want to compute the single side
% spectrum:

P2 = abs(Y/N);
magnitude = P2(1:floor(N/2+1));
magnitude(2:end-1) = 2*magnitude(2:end-1);

%Plot the FFT results
f = Fs*(0:(N/2))/N; % Frequency vector
figure;
plot(f, magnitude);
title('FFT of Audio Signal');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
grid on;
hold on; % To overlay the line on the existing plot

%%
% We can see that there is some content between the frequency 18kHz and 19kHz


% Mark the point with a red circle
[~, index] = min(abs(f - FREQ_BIT_START));
hold on;
plot(f(index), magnitude(index), 'ro', 'MarkerSize', 10);
[~, index] = min(abs(f - FREQ_BIT_1));
plot(f(index), magnitude(index), 'ro', 'MarkerSize', 10);
[~, index] = min(abs(f - FREQ_BIT_STOP));
plot(f(index), magnitude(index), 'ro', 'MarkerSize', 10);
hold off;
%%
% Now we can filter those frequencies and analyze the content

%filtering with band pass
x = bandpass(x,[0.8 0.9],'Steepness',0.85,'StopbandAttenuation',90);
N = length(x);
plot(x);
sound(x, Fs)
Y = fft(x);
P2 = abs(Y/N);
magnitude = P2(1:floor(N/2+1));
magnitude(2:end-1) = 2*magnitude(2:end-1);

%Plot the FFT results
f = Fs*(0:(N/2))/N; % Frequency vector
figure;
plot(f, magnitude);
title('FFT of Audio Signal');
```

```
xlabel('Frequency (Hz)');
ylabel('Magnitude');
grid on;
%%
%
%%
% let's compare the result using goertzel algorithm

frequencies = [ 100, 1000, 12000, 15000, 17000, FREQ_BIT_1, FREQ_BIT_STOP,
    FREQ_BIT_START,20000, 22000];
freq_indices= round(frequencies/Fs*length(x))+1;
gdft= goertzel(x, freq_indices);
stem(frequencies,abs(gdft))
ax = gca;
ax.XTick = frequencies;
% xlim(ax,[100,22000])
xlabel('Frequency (Hz)')
ylabel('DFT Magnitude')
%%
% The goertzel algorithm is more efficient in calculating the dft at determined
% frequency. From this dft analysis we can extract the maximum values that will
% be used for calculating the  threshold:

thr_start = abs(gdft(8))
thr_stop = abs(gdft(7))
thr_bit1 = abs(gdft(6))
%%
%
%
% The general idea is to divide the signal into intervals and then check the
% presence of one or the one frequency with goertzel algorithm, but before we
% need to find the starting point of the comunication, to do so we can use the
% correlation function to find the position of bit start, by generating a reference
% tone signal

%reference tone signal
bitstart_tone = sin(2 * pi * FREQ_BIT_START * (0:1/Fs:tone_length - 1/Fs));
[r, lags] = xcorr(x, bitstart_tone);
plot(lags,r)
%%
% we can find the maximum correlation that corresponds to the initial time of
% the message

[val, max_idx] = max(abs(r));
start_idx = lags(max_idx);
hold on;
plot(max_idx-N,val, 'ro')
hold off;
%%
% With the same approach we can analyze the signal to find the stop bit

bitstop_tone = sin(2 * pi * FREQ_BIT_STOP * (0:1/Fs:tone_length - 1/Fs));
[r, lags] = xcorr(x, bitstop_tone);
plot(lags,r)
[val, max_idx] = max(abs(r));
stop_idx = lags(max_idx);
hold on;
plot(max_idx-N,val, 'ro')
%%
% Finally we can see the rappresentation of all the symbols in the signal

%compute the number of symbols
```

```
nseg=round((stop_idx-start_idx)/(tone_length*Fs))+1;
delta=round((stop_idx-start_idx)/nseg)
figure
plot(x)
hold on;
xline(start_idx, 'r--', 'LineWidth',  'Label', 'start');
for i=0:nseg-2
    xline(start_idx+i*(tone_length*Fs), 'r--');
end
xline(stop_idx, 'r--', 'LineWidth',  'Label', 'stop');
xlim([start_idx - tone_length*Fs, stop_idx+tone_length*Fs])
hold off;
%%
% Search for frequency in a block starting from the previously found index

frequencies = [18000, 18500, 19000];
data="";
for i=0:nseg-1
    segment=x(start_idx+i*(tone_length*Fs):start_idx+(i+1)*(tone_length*Fs));
freq_indices = round(frequencies/Fs*length(segment))+1;
dft_data = goertzel(segment,freq_indices);
figure
bit_mag = abs(dft_data); %index 3 is start bit, 2 is stop bit, index 1 is bitdata

if(bit_mag(3)> thr_start/2 && bit_mag(3) > bit_mag(1))
    label = "start";
    data=data.append('s');
elseif(bit_mag(1)> thr_bit1/(nseg+1)*0.8 && bit_mag(1) > bit_mag(2) && bit_mag(1) >
    bit_mag(3))
    label = "bit 1";
    data=data.append('1');

elseif(bit_mag(2) > thr_stop/2 && bit_mag(2) > bit_mag(3) && bit_mag(2) > bit_mag(1))
    label = "stop";
    data=data.append('s');
else
    label = "bit 0";
    data=data.append('0');
end

 stem(frequencies,abs(dft_data))
 title(label)
 ax = gca;
 % ylim(ax, [0,50])
 xlim(ax, [17000,20000])
 ax.XTick = frequencies;
 xlabel('Frequency (Hz)')
 ylabel('DFT Magnitude')
end
data
%%
%

to_process=data.extractAfter('s').extractBefore('s');
to_process=char(to_process)
result="";
for i=0:length(to_process)/8-1
dig=char(bin2dec(to_process(i*8+1:(i+1)*8)));
result=result.append(dig);
end
result
binary=to_process
```

```
%%
% If the string is encrypted it can be decrypted with the developed tool

S=to_process
command = ['java -jar cyber.jar ' S ' ' key]
[status, out] = system(command);
result_decrypted="";
for i=0:length(out)/8-1
dig=char(bin2dec(out(i*8+1:(i+1)*8)));
result_decrypted=result_decrypted.append(dig);
end
result_decrypted
%%if location is sent the string is
result_decrypted.extractAfter(8)
%%
%
%
% If the string contains geolocalization data, it can be extracted analyzing
% the first 8 byte,  since latitude and longiude are sent as float numbers

%%Encrypted version
latitude_e = typecast(uint32(bin2dec(out(1:32))), 'single')
longitude_e = typecast(uint32(bin2dec(out(33:64))), 'single')
%%Not encrypted version
latitude = typecast(uint32(bin2dec(binary(1:32))), 'single')
longitude =  typecast(uint32(bin2dec(binary(33:64))), 'single')
```

B.3 decrypt tool code

```
import javax.crypto.Cipher
import javax.crypto.SecretKey
import javax.crypto.spec.IvParameterSpec
import javax.crypto.spec.SecretKeySpec


fun encrypt(plaintext: ByteArray, input_key: String): ByteArray {

    // input key gets padded with character '0' until length 32 (256 bit key length)
    is reached
    val key: SecretKey = SecretKeySpec(input_key.padStart(32, '0').toByteArray(), "aes
    ")

    // empty IV for easiness of implementation, should be changed to random for
    increased security
    val emptyInitializationVector =
        byteArrayOf(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00)

    // set cipher and options (AES256, CFB mode with no padding)
    val cipher = Cipher.getInstance("AES/CFB/NOPADDING")
    cipher.init(Cipher.ENCRYPT_MODE, key, IvParameterSpec(emptyInitializationVector))

    return cipher.doFinal(plaintext)
}

fun decrypt(ciphertext: ByteArray, input_key: String): ByteArray {

    // exactly same code as function "encrypt" but in decryption mode
    val key: SecretKey = SecretKeySpec(input_key.padStart(32, '0').toByteArray(), "aes
    ")
    val iv = byteArrayOf(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0
    x00, 0x00, 0x00, 0x00, 0x00, 0x00)
    val cipher = Cipher.getInstance("AES/CFB/NOPADDING")
    cipher.init(Cipher.DECRYPT_MODE, key, IvParameterSpec(iv))
    //println(cipher.blockSize)


    return cipher.doFinal(ciphertext)
}

fun main(args: Array<String>) {
    System.setProperty("file.encoding","Unicode")
    // check for arguments soundness
    if (args.size != 2)
    {
        println("Correct usage is: java -jar cyber.jar string_to_decrypt key")
    }
    else
    {
        var stringToDecrypt = args[0] // first argument from command line

        // cut the incoming string to decrypt into chunks of 8 characters, then
        // convert each chunk into a byte and append it to an array
        var stringToDecryptAsByteArray: ByteArray = byteArrayOf()
        for (i in 0..stringToDecrypt.length - 1 step 8) {
            //println(exampleString.slice(IntRange(i, i+7))) //.toInt(2).toByte()
            stringToDecryptAsByteArray += stringToDecrypt.slice(IntRange(i, i + 7)).
    toInt(2).toByte()
        }

        // decrypt the string of bits with the key from the second argument from
    command line
        // then print it
        val message=(decrypt(stringToDecryptAsByteArray, args[1]))
        //println(String(message))
        val binaryString = message.joinToString("") { byte -> String.format("%8s",
    Integer.toBinaryString(byte.toInt() and 0xFF)).replace(' ', '0') }
        print(binaryString)

    }
}
```