Documentation

Audio Steganography Techniques

Sneha Mohanty (120799), Jannis Leuther (117050), Frederik Sukop (114723), Sebastian Reichmann (120450)

Contents

1 Abstract

Audio Steganography is a technique used to transmit hidden information by modifying an audio signal in an imperceptible manner. It is the technique of hiding some secret text or audio information in a host message. The host message and stego message have the same characteristics. In our project, we have implemented three key techniques for Audio Steganography, i.e. Echo hiding, Discrete Wavelet Transform and Discrete Spread Spectrum. Some time was also spent looking for benefits of combining some of these methods for improved results. We have also analysed the stegowork generated using each of these techniques using several plots, mathematical visualizations such as MFCC and also using Machine Learning. We have finally presented a short summary and outlook of all our techniques, analyses and inferences as well as possible future work.

2 Watermark vs Stegowork

2.1 Watermark

Watermarking ensures the security of data by embedding a "watermark" for authentication purpose, which is important for copyright protection. The embedded watermark is usually visible and cannot easily be removed from the watermarked message. Watermarks are often considered extended information or attributes of the cover image but more so they could be required to be semi-fragile.

2.2 Stegowork

A steganography system involves inserting a secret message or a marker into a medium that shall be affected as little as possible. When this information is embedded, the steganographic technique ensures that it will not be detected and accessed by any unauthorized person or system. The cover message that hosts or contains the embedded message is called a stego-object.
The basic model of steganography contains two processes. Namely the embedding process (F_e) and the extracting process (F_x). A message M is embedded into a host or cover file C to create a stego-object SO by using F_e and a key K. Afterwards, F_x is processed with SO and K to obtain M.[1]

3 Embedding with Discrete Wavelet Transform

3.1 Introduction

When analyzing signals, we can use the Fourier Transform to transform a signal from its time-domain representation into its representation in frequency-domain. This, however, leaves us without any temporal resolution. We know **which** frequencies are present in the signal, but we do not know **when** these frequencies occur. The Wavelet Transform solves this problem by capturing both frequency and time information. The wavelet transform is a mathematical transform which uses wavelets to decompose signals. At first, we look at the host signal using a large scale and extract large frequencies, then we subsequently decrement the scale to extract smaller features using related low- and high-pass filters respectively. This leaves us with a high frequency resolution at a scale where frequency-dependent features are interesting, and with a high time resolution where time-dependent features are interesting.

There are two main categorizations of the transform: discrete wavelet transforms (DWT) and continuous wavelet transforms (CWT). As the names imply, they differ by using discrete wavelets (e.g. Daubechies wavelets) compared to continuous wavelets (e.g. Meyer wavelets). We focus on DWT, although the analysis of using CWT may be interesting to further investigate our results in the future. There are many types of specialized wavelets throughout literature. An analysis of a selection of different wavelet types and their performance in our embedding scenario can be found in section 3.3.

We will use the DWT for the purpose of embedding messages into an audio

signal. We outline a (primitive) embedding and detection algorithm using the DWT followed by an actual implementation. At first, we decompose the cover audio signal using the DWT. This results in the so called approximation coefficients (low frequencies) and the detail coefficients (high frequencies). The number of each of these coefficients corresponds to half of the amount of samples of the original signal, which is now split into two sets. Subsequently, we can use the detail coefficients to embed our message into. It is also possible to continue decomposing the resulting detail coefficients using DWT until a desired level is reached. After embedding the message into the detail coefficients, the inverse discrete wavelet transform (IDWT) is used to reconstruct the audio signal, now with the message embedded. The detector needs to perform the DWT using the same wavelet type and extract the message out of the detail coefficients.

There are already different versions of watermarking algorithms using wavelet transforms in the literature, i.e. [2], [3].

3.2 Simple Algorithm for DWT embedding

In this chapter, a practical implementation for embedding binary messages into audio signals using the DWT is presented. The algorithm is written in `Python` and can be found at [4]. To perform the wavelet transform, the library `PyWavelets` [5] was used. It allows the usage of single- and multilevel wavelet transforms which can be individually adapted by choosing parameters such as the wavelet type.

In `audio_file.py`, the object `AudioFile` as well as some helpful static methods are defined. An instance of `AudioFile` is created by passing the file path of the original audio file in `.wav` format. The object then collects the raw signal data and sampling rate. It can also be used to create an AudioFile and write it to a file path. Both reading and writing are implemented by functions from the `soundfile` library [6].

In `embedder.py`, the `Embedder` object is defined. By creating an Embedder object and passing the relevant and necessary parameters, one can easily embed a binary message into the first level detail coefficients of the input signal. The embedding is performed by specifying an integer e, which represents the *embedding bit* marking the position of the bit in the binary representation of each detail coefficient sample where the corresponding message bit gets embedded. Consider a binary message $m = (m_1, m_2, \ldots, m_n)$ where each m_i represents a bit $\in \{0, 1\}$ and the set of floating point detail coefficients after one application of the DWT $dc = (dc_1, dc_2, \ldots, dc_n)$. The bit-wise binary representation of a detail coefficient is denominated by $bin(dc_j) = (b_1^j, b_2^j, \ldots, b_n^j)$ where each b_k^j is the k-th bit $\in \{0, 1\}$ of detail coefficient j.

To embed m into dc, for each $i \in \{1, \ldots, n\}$ replace the bit b_e^i with m_i. This can mean that the bit b_e^i flips if it is different from m_i, or the bit does not change if $b_e^i = m_i$. As we embed one message bit per detail coefficient, the embedding capacity is the same as the number of detail coefficients $|dc|$. After one iteration of the DWT, this number is equal to half of the amount of samples in the

unmodified audio file.

In our code [4], this procedure is realized in `embed()` in `embedder.py`:

```
1  def embed(self):
2      self.approx_coeffs, self.detail_coeffs = pywt.dwt(self.
       cover_audio_file.signal_data, self.wavelet_type)
3      self.marked_detail_coeffs = self.detail_coeffs.copy()
4
5      for i in range(len(self.detail_coeffs[0])):
6          val = self.detail_coeffs[0][i]
7          bin_val_list = list(AudioFile.float2bin(val))
8          if self.message[i] == '1':
9              bin_val_list[self.embed_bit] = '1'
10         elif self.message[i] == '0':
11             bin_val_list[self.embed_bit] = '0'
12
13         new_bin_val = "".join(bin_val_list)
14         new_val = AudioFile.bin2float(new_bin_val)
15         self.marked_detail_coeffs[0][i] = new_val
```

Where the DWT is executed using `pywt.dwt()` in line 2 and the corresponding detail coefficients are embedded into in the loop in lines 5-15. The variable `bin_val_list` represents the binary representation of the current detail coefficient that is embedded into as an ordered list of bits and `self.embed_bit` is the position of the bit where the message is embedded into. Afterwards, the method `reconstruct_and_write()` reconstructs the message employing the inverse discrete wavelet transform with the newly created detail coefficients. Finally, the same method writes the output `.wav` file to the disc.

To try to detect a message, a `Detector` object in `detector.py` must be created. The detection is implemented in `detect()`:

```
1      def detect(self):
2          self.approx_coeffs, self.detail_coeffs =
3          pywt.dwt(self.audio_file.signal_data, self.wavelet_type)
4
5          extracted_message = ""
6          for i in range(len(self.detail_coeffs[0])):
7              val = self.detail_coeffs[0][i]
8              bin_val_list = list(AudioFile.float2bin(val))
9              extracted_message =
10             extracted_message + str(bin_val_list[self.embed_bit])
```

Here, the input audio is again decomposed with the DWT. Subsequently, the bits at the specified position `self.embed_bit` of each detail coefficient are extracted and concatenated to receive the extracted message. Apart from the audio file, the only required parameter to instantiate a `Detector` object and receive the extracted message is the `embed_bit`.

The algorithm was thoroughly analyzed and tested. In `test.py`, one can see the testing procedure. The goal was to embed messages over a range of embedding bit positions (`embed_bit`) and analyze the performance of both embedding and detection. In `dwt_plotter.py`, functions are implemented to visualize the analysis and plot useful data for comparison and further analysis. Other documents, such as `percentage.py` and `statistics.py` house experimental analysis tools and functions partly referenced to in the plotting process.

Key	Wavelet Family
haar	Haar Wavelet
db	Daubechies Wavelets
dmey	Discrete Meyer Wavelet
sym	Symlet Wavelets
coif	Coiflet Wavelets
bior	Biorthogonal Wavelets

Table 1: Wavelet families considered in Algorithm Analysis.

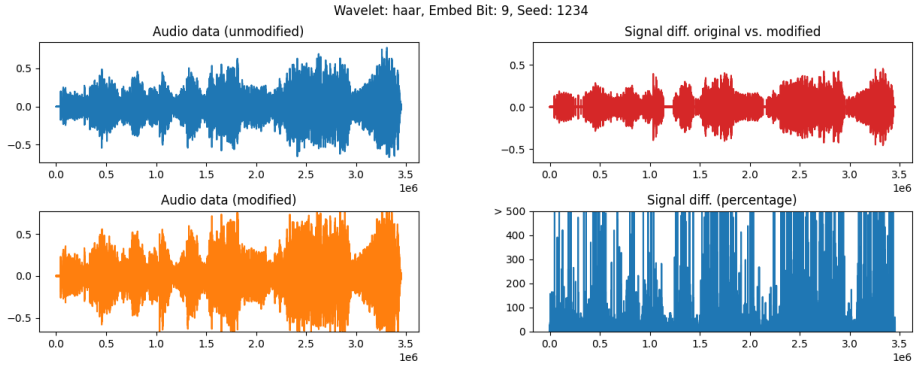


Figure 1: Embedding message generated with seed 1234 into position 9 of the detail coefficients of the audio file generated by the DWT using Haar wavelet.

3.3 Algorithm Analysis

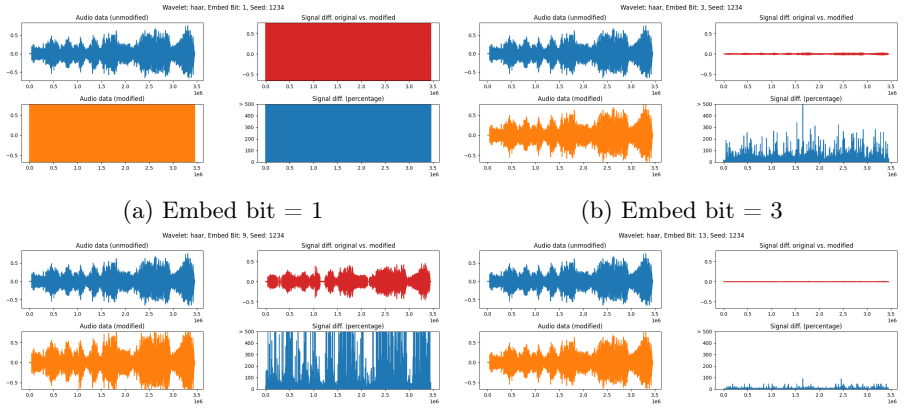
In order to look for an optimal combination of embedding parameters, many configurations were tested and evaluated. The cover audio file stayed the same for each configuration, as well as the seed used to generate a random message. A large list of wavelet types was tested, Table 1 shows the different wavelet families considered. The full list of wavelets is found in `test.py` in [4]:

```
wavelet_list = ['haar', 'db2', 'db3', 'db4', 'db5', 'db6', 'db12',
    'db16', 'db20', 'db30', 'db38', 'dmey', 'sym2', 'sym3', 'sym4',
     'sym5', 'sym6', 'sym12', 'sym16', 'sym20', 'coif1', 'coif2', '
    coif10', 'coif16', 'bior1.1', 'bior1.5', 'bior2.2', 'bior2.8',
    'bior3.1', 'bior3.9', 'bior4.4', 'bior6.8']
```

Additionally, for each wavelet, every embed bit position from 1 to 16 was tested. For each configuration (wavelet type, embed bit) a figure containing four subplots was created. These subplots showed the original audio signal, the modified audio signal, the difference between the original and modified signal and the percentage difference how much each coefficient was altered during the embedding process (capped at 500%). As there were 32 wavelet types tested, each with 16 different embedding positions, a total of $32 \times 16 = 512$ figures, each containing four subplots, were created. For example, the result for the Haar wavelet on embed bit 9 is seen in figure 1. With these visualizations, it is easy to see how much impact the embedding process had on the audio signal. By comparing the figures for each embed bit of one wavelet type, one can see the best configurations that should be used when trying to achieve minimum embedding impact.

In figure 2, a comparison is made between the four embed bit locations 1, 3, 9 and 13 for the haar wavelet. One can easily see that for embed bit position 13, the embedding had the least impact on the signal.

This does not conclude the analysis, however. Even though an embedding has a



(a) Embed bit = 1

(b) Embed bit = 3

(c) Embed bit = 9

(d) Embed bit = 13

Figure 2: Comparison of embedding with haar wavelet in bits 1, 3, 9 and 13.

low impact, this usually comes at the cost of low fidelity. A well hidden message is of no use if only a fraction of the original message can be detected afterwards. To get a good trade-off between fidelity and embedding impact, for each wavelet type, the error rates of detection for each embed bit were calculated and plotted as seen in figure 3. Error rates were computed by iterating through the whole message (as a bit string) and comparing each bit with the detected message. If a bit differs, the sum of errors was incremented by 1. Finally, the sum of all errors was divided by the length of the message. The same was done for double and triple errors. A double [triple] error is defined as an immediate sequence of two [three] wrong bits in the extracted message. These multiple errors are important to consider if error-correcting codes such as 3-repetition-, Reed-Muller- or Reed-Solomon-Codes can or have to be used.

3.4 Results

Now we have enough information to figure out a good trade-off between fidelity and embedding impact. For the Haar wavelet, a good choice for the embed bit position would be between 11, 12 and 13, depending on the valuation of fidelity versus impact. While embed bit 11 has a lower error rate than 13, it also produces a higher impact in the audio file, although the absolute impact is still quite low. While the error rate rises from bits 11 to 16, the embedding impact shrinks continuously.

For all the considered wavelet types, the Haar Wavelet as well as the biorthogonal wavelet family distinctly outperformed the remaining wavelet families. Both had error rates below 0.1 for embedding bits $\in \{2, 3, \ldots, 15\}$ and low embedding

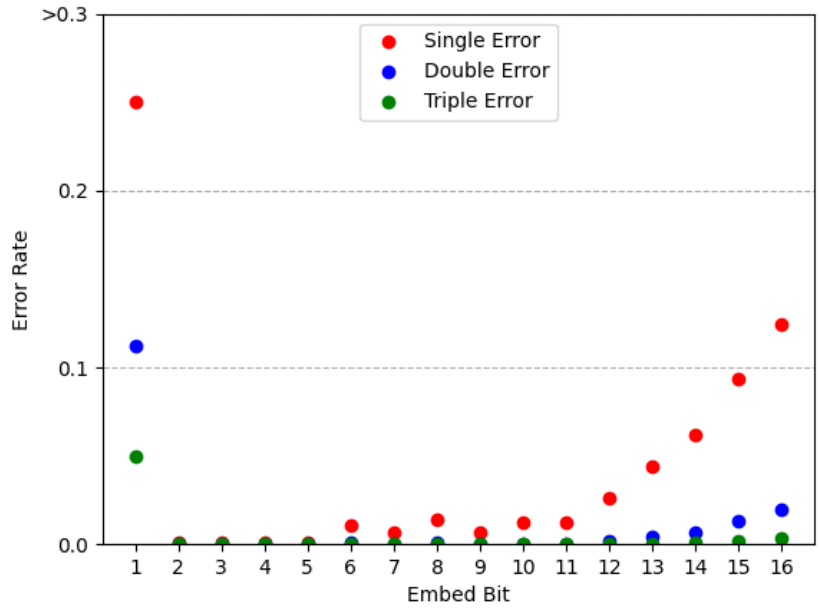


Figure 3: Rate of error occurrence for single, double and triple errors for Haar wavelet embedding.

impact when compared to the other wavelets. Daubechies, Symlet, Coiflet and the discrete Meyer wavelets were all very similar when it comes to embedding impact and error rates. For the discrete Meyer wavelet, Error rates below 0.1 were only achieved for one embedding bit position. When ignoring Haar and biorthogonal wavelets, the best performing wavelet when it comes to error rates were Daubechies and Symlet Wavelets, although an error rate below 0.1 was only achieved at four embedding bit positions. These higher error rates are a trade-off for a low embedding impact, where an embedded message can not be detected by average humans. Nevertheless, Haar/Biorthogonal wavelets can also produce an embedding impact that is low enough to make the embedding inaudible.

For all wavelets, the occurrence of double and triple errors seemed to be randomly distributed. This would mean that there are no structural error clusters, i.e. a larger amount of errors at the beginning or the end of the file. Therefore, if the normal error rate is low enough, one can easily employ error-correcting codes to improve the fidelity of the result.

All of the figures and plots for the wavelet analysis can be found at [7].

3.5 Future Work

In this section, a very primitive embedding algorithm with the discrete wavelet transform as a basis was introduced. A more sophisticated steganography or

watermarking algorithm based on this concept could be envisioned. Although, many of the discrete wavelet families were analyzed, it was only a subset of many wavelet types found in scientific literature [8]. A more thorough, systematic analysis of wavelet types may be necessary to see which wavelets are suitable for embedding with the discrete wavelet transform and which wavelets provide the best trade-off when it comes to embedding impact and fidelity.

Additionally, chapter 3 did only consider the *discrete* wavelet transform, not the *continuous* wavelet transform. Latter might also be a better alternative in a whole or for certain scenarios. As for most audio embedding techniques, the optimal parameters for a good embedding algorithm may depend on the shape of the audio. Possibly, some wavelet families are better for certain ranges of frequencies than others. I.e. for a classical orchestra audio piece, the optimal parameters might differ when compared to a heavy metal piece. An automatic adaptation based on audio input that switches between suitable wavelet types or decomposition levels could be good solution to combat this variance.

4 Echo Hiding

The Echo-Hiding Algorithm is one of the common audio watermarking techniques that use echoes to encode binary messages. One purpose is to modify this algorithm and find a approach for a steganographic system. To reach this goal the algorithm is modified in the encoding and in the decoding process, subsequently the algorithm is testing if it is possible to hiding information, also in terms of attacks.

4.1 Theory/ Basic Idea

One of the focus points is the development of an steganographic algorithm that based on the Echo-Hiding-Technology, a common watermark technique in the spatial domain of signals. As groundwork for the development should be the bachelor thesis 'Analysis and improvement of detection methods of echo-based watermark technology, based on delay and attenuation rates', which mention ideas in the section future works. This part of the elaboration should focus on the extension of that algorithm , to transfer a watermark technique into a stable and robust stego system.
In the work of Frederik Sukop it is recognizable, that the cepstrum analysis is one of the essential function for the detection of embedded messages or in general to calculate echo delays in signals. In his research he mention the 'effective number of data points for extraction' [9], which should be used as foundation for this development.

The author describe noisy signals for small amounts of data points in the cepstrum analysis and much smoother and use-fuller signals for larger sets of data for the echo detection - the larger the data set, the better its detectability. Another property is pictured in the future work of the thesis, which describe the functionality of sine waves in combination with the Echo-Hiding-Approach. The sine is multiplied to the message signal via the given function:

$$x(n) = y(n) + \lambda \cdot y(n - \delta) * s(n) \tag{1}$$

Where x (watermarked signal) is defined by the original signal y the echo signal $y(n - \delta)$ and a sine function, λ is the decay rate. It is also described, the cepstrum must applied to specific areas of the watermarked work to detect the embedded echo and recreate the message.
The idea is to use these two scenarios in combination to disguise the message under the rules of steganography. The starting point for the implementation is based on the Echo-Hiding-Algorithm, which is adjust and extended, of the mentioned thesis. After the implementation the work is tested in terms of functionality and steganography.

4.2 Implementation

As mentioned the algorithm is an watermarking algorithm to embed binary messages into the time or rather discrete representation via echoes, a time delayed copy of the original signal. The extension and changes that we are make, will start with implementation of a simple pseudo-random number generator (PRNG). This PRNG is used as key instead of the sine wave and also to embedded the message at specific areas of of the signal. To implement the generator, the algorithm is expanded by a class - `keyGen`.
The following describes the changes in the Echo-Hiding-Algorithm.

Pseudo-Random Number Generator

To generate the key for the embedding process under specific condition, for example the maximum length of an embedding areas, the class is hand over some values, which can calculate the key. In this case we will work with a maximum area length of hundred data points. For that the algorithm has to receive the length of the original signal and the message size to the class. By these values the key seed length is calculated and generates key seed via the random function of the numpy package. The function generate an array in form of $a = \{n_0, n_1, ..., n_m\}$, where m is the seed length an $n_m \in [0, 1]$, afterwards the array is formt into a binary array with different properties. In one example the array can be distributed equaly, like:

$$a(x) = \begin{cases} 1, & x < 0.5 \\ 0, & x > 0.5 \end{cases} . \tag{2}$$

To reduce the impact of the message signal on the carrier signal, the implementation use following distribution:

$$a(x) = \begin{cases} 1, & x <= 0.3 \\ 0, & x > 0.7 \end{cases} . \tag{3}$$

In the next step is to expand the key seed to the segment size that is define by the Echo-Hiding-Algorithm, to embed each message bit into the signal. This is done by a simple multiplication of the seed array with a given value. In this case is the maximum length of the embedding area - 100 frames. For the embedding

process it is necessary to use a smooth signal, like in the section `Audio`-Class of [9], to avoid discontinuous value changes in the key signal, which can be audible in the audio signal. This is implemented in Listing **Signal Smoothing**.

```
1     h = sc.signal.get_window(triang, 50)
2     fil = sc.signal.convolve(sig, h/h.sum())
```

Listing 1: Signal Smoothing

In accordance to the smoothing and the lengthening corresponding to the size of the original signal, the key signals are stored in the key object, and can be used for the embedding process. Figure 4 shows the key signal and the smoothed signal.

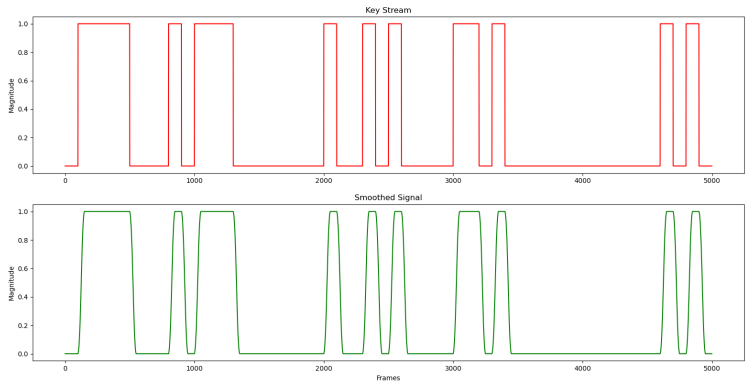


Figure 4: Key signal in comparison to the smoothed signal

Adjustments in the Embedding Process

For the given embedding process are just marginal changes to do. At first the fundamental addition of the signal components is changed. For later testing of functionality etc. the write function is modified to extract the message and the binary key.
So in the audio class is rewritten as:

```
1     stego.y = self.y+(key.fil*EchoSig.y*self.decay)
```

The Stego object holds the resulting marked signal, `self.y` is the original signal. `EchoSig` contains the message in echo representation, as it is described in the thesis, further the key signal is added to the function to use the addition for the area-wise combination of all signals. And thus similar to the above formula 1, where `s(n)` is replaced by the key.

Decoding Algorithm

Because of the changes in the encoding process, the decoding process has to be modified to guarantee, that the disguised message is recovered from the carrier signal, especially in combination with the key, which was used to embbed the

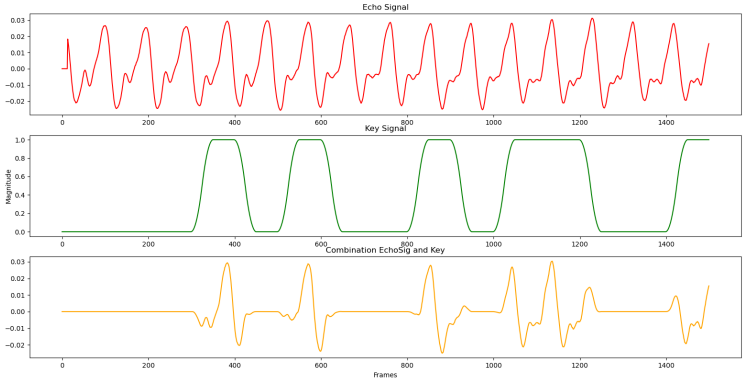


Figure 5: Echo Signal, Key Signal and Combination for the embedding

information.
As in previous section the changes based on the watermarking algorithm of the Echo-Hiding process. At first the extended key has to be recovered from the key seed in the same manner as the key calculation in the encoding algorithm. To recreate the key the algorithm use a text file, which contains the use the key seed to the corresponding marked audio signal. The `embedded`-class holds the `keyBuild()`-method to operate the key reconstruction.

```
def buildKey(self):
    text_file = open(self.keypath, 'r')
    keyseedtxt = text_file.read().split(';')
    keyseed = np.zeros(len(keyseedtxt), dtype=int)

    for i in range(0, len(keyseedtxt)):
      keyseed[i] = keyseedtxt[i]

    id_p = np.where(keyseed == 1)
    pos = id_p[0]

    all_pos = (pos*self.multi)+self.segsize*0

    for i in range(1, self.msg_len):
      all_pos = np.vstack(all_pos,(pos*self.multi)+self.segsize*i)

    return all_pos
```

Listing 2: Key reconstruction

The method reads the text file by constructing of the audio object, get the positions, where the echoes are in encoding algorithm by using the same variables as by the generation of the key, as like the length of the area (`obj.multi = 100`) and also it needs the message size that can be static or defined in the key

After the recreation of the key, which gives information about the location of the embedding areas, all components are combined with the audio signal.

The signal is deconstruct into the segments that is correspond to the message size, also each embedding area is concatenated with a window function to smooth the transition between the areas - it is the hanning window. The array holds all areas of each message segment in rows. Afterwards the cepstrum is calculated to find in the next step the delays of the periodic structures. The delays represents the indicators of an Zero or One in the segment. Now the message can be recreated by finding the right peaks in the cepstrum by the method `Peak()`, `Can()` and `Decode()`.

After presentation the algorithm can be tested with different types of signals, messages and keys. The next chapter deals with the functionality of the algorithm, if it is possible to encode and decode a given message with the written algorithm.

4.3 Functionality

In this section we want to verify the functionality of the developed algorithm. For this task we checked various audio signals in terms of audibility or visibility of artifacts, do we have an embedding in the right areas and is the encoding of the message even possible with this algorithm. Another thing would be the improvement of the embedding, if there artifacts, background noise or something alike. For this we create multiple test data of songs with different keys and messages. To start the first test set is generate with a decay rate of 1 to 1 (see eq. 2), so the echo amplitudes has the same magnitude as the carrier signal. Also we start with an small message size and a small signal length - about 30 seconds. Also the decoding is checked, for that to do we look at the cepstrum graphs, that is necessary for the decoding, also we check the algorithm via an error-rate-test.

Audibility

As described, we use different disintegration (decay) rates for the test data. In the beginning we start with an rate, which has the same magnitude as the carrier signal. Afterwards we discuss the test data, which used lower rates.

1-to-1-Embedding

The first set of data has clearly audible artifacts. These artifacts suggest, that the audio signal was modified or it was damaged during saving, compression etc. It is not possible to use these files for satisfy the listeners claim. On the other hand the high value for the embedding leads an easy detection of the echoes for the encoding process, but also it is easy for an attacker to make sure, that the signal was changed with a simple moving-window-test, which is discuss later. As we noticed an 1 to 1 embedding leads us not to an unsatisfying result for the embedding under the terms of steganography. Figure 6 shows the detection of the delays in the signal. The cepstrum analysis based on the created segments, with contains only the areas that contains the echoes.

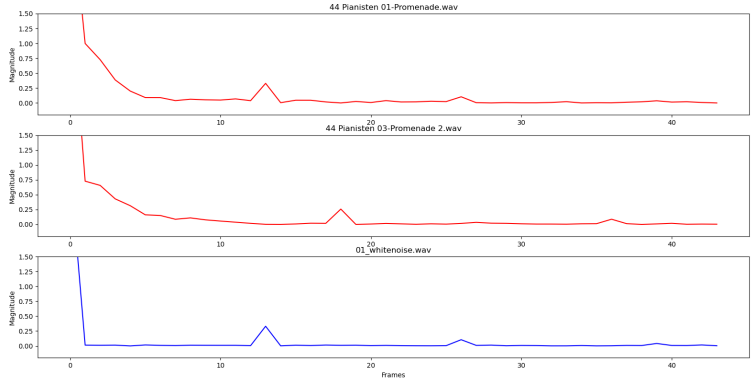


Figure 6: Cepstrum of a signal with a 1:1 embedding

50%-Embedding

In the next step we decrease the embedding rate by 50%, where the message signal is added with 50% magnitude of the carrier signal.
What can be recognized, is that the artifacts are slightly less audible for the listener. In the White-Noise-Signal, are no interference detectable, which is attributed to the structure of the signal. In other music based signal it is possible to recognize interference or artifacts. The sound is not clear and not enjoyable, because of little crack noises. Which also can lead to the impression that the signal is modified in any way. But the impact of the signals is much smaller as in the first embedding test. In direct comparison with the original unmarked signal the differences are obvious. A detection of the message also possible as it can been seen in figure 7.

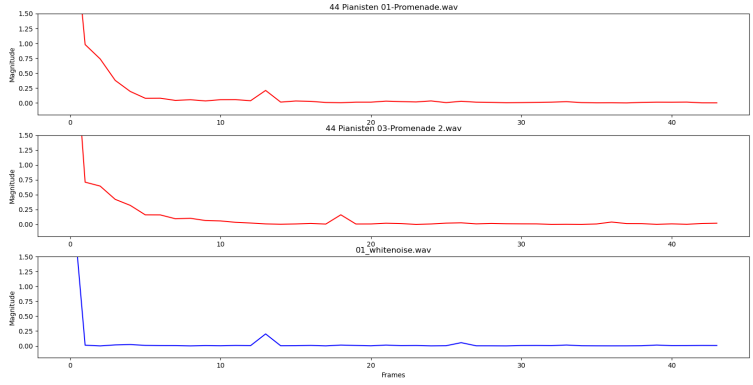


Figure 7: Cepstrum of a signal with a 2:1 embedding

So in the next test we decrease the rate again to check the audibility and also, if a detection under the developed algorithm is possible.

20%-Embedding

Because of the worse results of the previous test sets a lower decay is used for the hiding. So, we use 20% magnitude of the original amplitudes for the message signal to added it to the carrier signal.
This proportion does not seem to have the problems of the previous tests. Artifacts or noise are not or hardly to identify. Even if we put the original and the marked signal in direct comparison and put one signal on one channel (for example the right channel of a headphone) and the marked signal in the left channel, it is hard to detect a difference between the two signal. If the listener is informed about a difference is much easier to identify the marked signal, especially we inform the listener which one was changed. But to use a secure stego-system it is necessary to delete the original medium to prevent direct comparison in any way. But this topic will be discuss in a later section. The following table shows the results of an error rate test for this test set (s. 9). In comparison to the figures 6 and 7, we see a difference in the cepstrum, especially that the scale of the y-axis is adjusted, the the delays of the given embedding areas are still detectable.

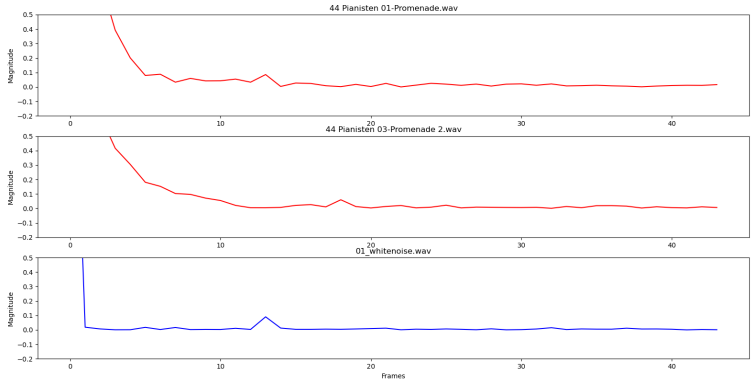


Figure 8: Cepstrum of a signal with a 5:1 embedding

The properties that we used to encode a message:

- Message: 01010101
- Key distribution: $1 \rightarrow 70\%, 0 \rightarrow 30\%$
- Decay: 0.2

This error test can be used to represent other messages that embedded under the same circumstances and properties, that can been seen in the appendix. These error rate test shows that in the most of case the decoding algorithm works with different values for the embedding. Especially the reconstruction of the message of the piano-based pieces of music by the '44 Pianisten' have the low or a zero error-rate, also the the pieces interpreted by 'Sa Chen' show

File	Decoded								error								count_error	error_rate
44 Pianisten 01-Promenade	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
44 Pianisten 02-Der Zwerg	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
44 Pianisten 03-Promenade 2	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
Pond 01-Promenade	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
Pond 02-Gnomus	0	1	0	1	0	-	0	1	0	0	0	0	0	1	0	0	1	0,125
Pond 03-Promenade II	0	1	0	1	0	-	0	-	0	0	0	0	0	1	0	1	2	0,25
Sa Chen 1. Promenade	0	1	0	-	0	1	0	1	0	0	0	1	0	0	0	0	1	0,125
Sa Chen 2. Gnomus	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
Sa Chen- 3. Promenade	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
01_whitenoise	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0

Figure 9: Error-Rate results of decoding process

similar results, which makes the algorithm look promising. But in other song of the artist 'Pond' the rates are quiet high. The results are also can be seen in the appendix of this work.
In the following section we use the error rate to construct some attack to get a idea of the security of the developed algorithm.

4.4 Attacks/Testing

To get an idea of the security of the algorithm we perform some different types of tests. At first we perform visual test to analyze the resulting graphs if it is possible to receive information of the carrier signals and make conclusions of the data. In the second section we perform different decoding test with and without knowledge of properties of the algorithm. But we start with a simple moving window approach.
The moving or sliding window algorithm[1] should be a useful approach. The overlapping areas are transformed into into the cepstral representation (Cepstrum) with different properties as like window size and overlap size. With that we simulate a blind attack. The attack has no information about the functions of the algorithm, only that the carrier signal is manipulated via periodic structures.
The testing process starts with analysis of the whole stego work as described in [9] in the section **Future Work**.

[1] An algorithm, that scans a set of data from the beginning to an end area-wise. Within the areas have an overlap to have a smooth transition between the areas.

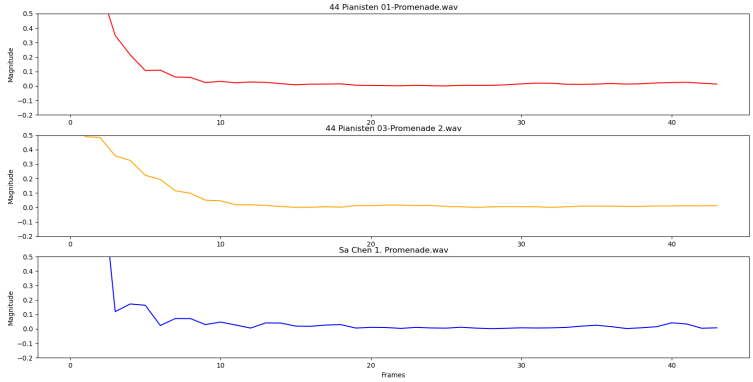
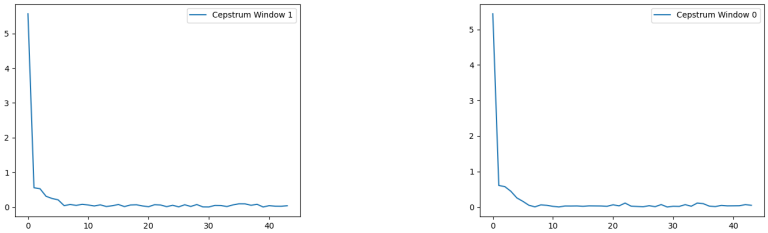


Figure 10: Cepstrum of three signals (all data points)

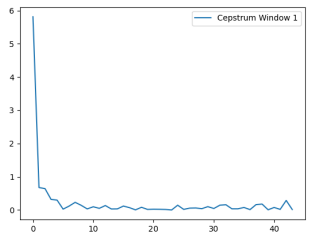
As we can see in figure 10, the periodic structures of the echoes are not visual detectable, which depends on the amount of embedding areas. The areas which not contains echoes overlays areas with message signal. We try to finding specific embedding areas, so we use the sliding window approach with window size of 1000 frames (approx 22.68 ms) and a step size (overlap) of 50 frames (approx 1 ms).



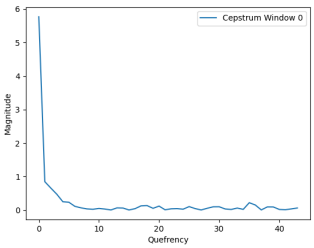
(a) Example of the first segment (b) Example of the second segment

Figure 11: Sliding-Window with 1000 Data points

The example graphs (s. figures 11a and 11b) shows the cepstrum in two segments where should a zero and a one is embedded and around areas, where the echo signal is combined with the carrier signal. As we see the graphs shows no peculiar peaks at any frames to find evidences of an encoded message via echoes. The next step is decrease the window size to find these evidence. The algorithm is adjust to a window size of 500 frame. As in the previous test we analyse areas around embedding areas. The example figures (see 12a and 12b) shows more noisy cepstra, which shows peaks at different frames and leads to a not specific statement about the delays. We can identify multiple peaks in different graphs, that can be interpreted as the delays, which were used to create the periodic structures, that corresponds to the message.



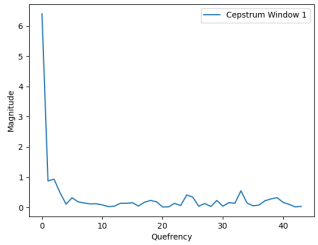
(a) Example of the first segment



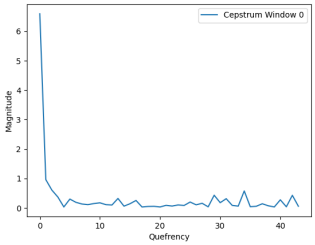
(b) Example of the second segment

Figure 12: Sliding-Window with 500 Data points

Maybe the reduction of the window size gives as a clearer picture of the delays. So the imaginary attacker use a windows size of 100 frames (approx 2 ms), which is exact the same size as the length of the embedding areas. Maybe it is possible to find via the moving window algorithm the exact areas, where an echo is encoded. So in figures 13a and 13b we see the results of the test set. As expected the cepstra seems much noisier, which is also described in [9]. The figures show that is not possible to find characteristic peaks in the areas where an echo should be in embedded. In this case the highest peaks are the wrong delay, that were not used for the encoding process. The circumstances makes it not possible to reconstruct the message, because in other areas we find other peaks at different quefrencies, which were not use as echo delays. The next step is to check how the decoding is algorithm is working, wo do this by using an error rate test.



(a) Example of the first segment



(b) Example of the second segment

Figure 13: Sliding-Window with 100 Data points

Error Rate

The previous tests were based on the detection only using the cepstrum without the knowledge about the Algorithm. Another way would be a blind detection with knowledge about the embedding algorithm, but not about the key properties or the encoding values like decay or delay. So the idea is to implement a algorithm that analyse the signal area-wise, calculate the cepstrum and compare the results the actually embedded message via an error rate test. To do so, we simulate a key signal that have a distribution of 50:50, also shown function 2. This signal is used to get areas from the carrier signal randomly, which possibly

contains echoes. The implementation of decoding is known by the attacker, so we concatenate the signal areas and examine the data via the cepstrum. For the experiment we use the same data as described in 4.3 with the error rate shown in figure 9.

So, we have a 8-bit message embedded in the carrier signal we increase the expected message size by the attacker algorithm, which one use following properties for the decoding:

- Message size: 16
- Key distribution: $1 \rightarrow 50\%, 0 \rightarrow 50\%$

This attribute leads to the results in figure 14.

File	Decoded Peaks	Decoded Bits
44 Pianisten 01-Promenade	21 11 14 21 13 13 12 14 23 14 16 14 11 13 14 30	0 0 1
44 Pianisten 02-Der Zwerg	13 13 11 14 16 13 18 12 13 11 21 19 16 13 19 21	0 0 0 0 1 0 1
44 Pianisten 03-Promenade 2	11 13 11 32 37 13 18 18 12 29 30 32 13 13 11 18	0 0 1 1 0 1
Pond 01-Promenade	11 12 11 19 17 15 12 12 17 12 18 18 11 15 14 15	0 0 1 0 1 1
Pond 02-Gnomus	13 11 16 14 15 13 15 18 13 14 16 14 12 17 14 16	0 1 0 0 1 1
Pond 03-Promenade II	13 33 14 11 12 13 15 18 12 13 12 12 17 12 19 14	0 1 0 0 1
Sa Chen 1. Promenade	13 13 18 11 17 17 14 14 16 13 14 13 13 13 16 14	0 0 1 1 0 1 0 0 0 1
Sa Chen 2. Gnomus	13 13 11 13 16 13 11 13 12 13 19 14 11 13 14 18	0 0 0 0 0 0 0 1
Sa Chen- 3. Promenade	13 13 11 11 13 16 18 18 13 13 22 36 27 11 17 15	0 0 0 0 0 1
01_whitenoise	13 13 18 18 22 13 18 18 13 13 18 18 13 15 18 18	0 0 1 1 0 1 1 0 0 1 1 0 1 1

Figure 14: Error-Rate results of decoding process by the attacker

The figure shows the same pieces of music as in figure 9 but in this case we shown the detected peaks, because the encoding from peaks to bits does not find clearly matches between the peaks. As the peaks shown that in nearly all signals the found peaks have no correlation to each other to encode the message. The special case, the white noise, is an exception. In this case it is possible to reconstruct the embedded massage, because the embedded bits are just doubled and leads to the statement that white noise signals are not suitable for the embedding via echo hiding. In other data sets with other attacker scenarios we are getting similar results.

Another scenario would be, that the hiding process uses fix message sizes, which is known by the attacker. So we can examine the explicit embedding segements of a `One` or a `Zero`. We also use the random selection process as in the previous test set and the same embedding properties, but only with eight 'search bits'. The figure 15 shows us the results of this test.

File	Message								Decoded								error								count_error	error_rate
44 Pianisten 01-Promenade	0	1	0	1	0	1	0	1	-	1	0	-	0	-	0	1	1	0	0	1	0	1	0	0	3	0,375
44 Pianisten 02-Der Zwerg	0	1	0	1	0	1	0	1	0	-	-	-	-	-	0	-	0	1	1	1	1	1	0	1	6	0,75
44 Pianisten 03-Promenade 2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
Pond 01-Promenade	0	1	0	1	0	1	0	1	-	-	-	-	-	1	-	1	1	1	1	1	0	1	0	6	0,75
Pond 02-Gnomus	0	1	0	1	0	1	0	1	0	-	0	-	-	-	-	-	0	1	0	1	1	1	1	1	6	0,75
Pond 03-Promenade II	0	1	0	1	0	1	0	1	-	-	-	-	0	-	0	-	1	1	1	1	0	1	0	1	6	0,75
Sa Chen 1. Promenade	0	1	0	1	0	1	0	1	-	1	0	-	0	-	0	-	1	0	0	1	0	1	0	1	4	0,5
Sa Chen 2. Gnomus	0	1	0	1	0	1	0	1	0	1	0	-	0	-	0	-	0	0	0	1	0	1	0	1	3	0,375
Sa Chen- 3. Promenade	0	1	0	1	0	1	0	1	0	-	-	1	0	-	0	1	0	1	1	0	0	1	0	0	3	0,375
01_whitenoise	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0

Figure 15: Error-Rate results of the attacker, with knowledge of the message size

The table shows the error rate of the blind detection. In the column 'Decoded' we can see the bits that are detected by the attacker, where the `Ones` and `Zeros` indicate the right peaks, which were use during the encoding. The '-' are false peaks that were detected by the attacker, which leads to the error rate. Because of different peaks that are detect it is not possible to find a majority to encode to peaks into bits. As in other scenarios it is possible to recover the embedded bits for the marked signal in the signal `01_whitenoise`. The other test data, except the signal `44 Pianisten 02-Der Zwerg`, create the a high error rate between 37.5% an 75.0%, which makes an encoding for larger messages difficult. The majority of the test data shows a high error rate, what makes it difficult to reconstruct the whol message from the carrier signal.

4.5 Conclusion

The test sets with the an adaptive version of the Echo-Hiding-Algorithm shows that it possible to embedded information by adding a key stream under specific circumstances, like embedding length, decay rate and key distribution. The test (visual, auditory, error rate etc.) leads to the option to use this algorithm for the veiling the information more than in the algorithm mentioned in [9], but to called a steganography approach is hard to say. These tests are a few of many test to make for the detection of information, maybe it is possible to find major specifica in other audio feature, that leads to the statement of a manipulated signal. In the end it should be hard to make a decision of a manipulated or an unchanged audio signal. But this algorithm is a nice approach to work further on.
One idea is shown below (see section 5), where two watermarking algorithm are merged in to one single algorithm, to manipulate the signal in such a way, that is hard or impossible to detect any bit information.

5 Combining DWT and Echo Hiding

5.1 General Idea

Using a multilevel decomposition of the DWT, we can extract the detail coefficients for a specific level (or depth). This allows us to extract specific bands of frequencies from our audio file. These bands can then be used to embed messages into, as shown in the algorithm in section 3.2. There are, however, many steganography and watermarking algorithms that are already well established

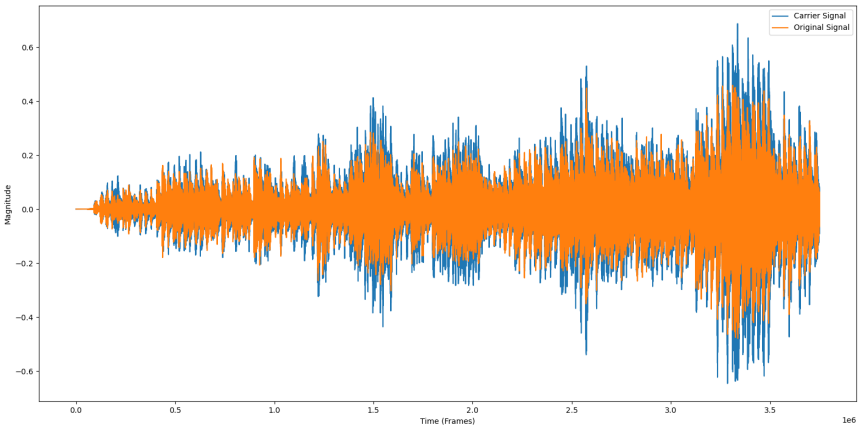


Figure 16: Difference of original and carrier audio signal.

and studied. To see if we can enhance such an algorithm by using the DWT, we looked at a combination of the DWT and the *echo hiding* approach for audio steganography.

5.2 Embedding

By employing multilevel decomposition, a desired band of detail coefficients was extracted from the cover audio file and used as the input for an echo hiding algorithm. Echo hiding, as described in section 4, would use these detail coefficients as it would use a normal audio signal to embed a message with inaudible echoes. Afterwards, the inverse DWT reconstructs the signal with the original non-altered coefficients, and the modified detail coefficient band. The reconstructed signal is written to the disc as a `.wav` file.

For example, a randomly generated bit string was embedded into the level 2 detail coefficients under the Haar wavelet DWT. The delays used for the echo hiding algorithm were 20 (bit 0) and 25 (bit 1) and the decay was set to 1 (see section 4.1). Figure 16 shows the difference of the original audio file and the modified, reconstructed audio file after embedding. Since an echo is getting added to the original signal, the modified audio will always differ from the original. In this case, however, the difference is rather large and clearly audible. In figure 17, one can see a magnified section of the previous figure, with the same signals. Here, the stark contrast between original and carrier signal is clearly visible. While the original signal (orange) is rather smooth, the carrier signal seems jagged with many local inflection points. These unnatural structures are very much perceptible by the human auditory system and lead to a distorted sound in the embedded audio signal.

This leads to the conclusion that the echo embedded into the relatively small band of the original audio influences the whole signal in the inverse DWT. This can presumably be explained mathematically, but this project did not dive into the mathematical formalities of the DWT.

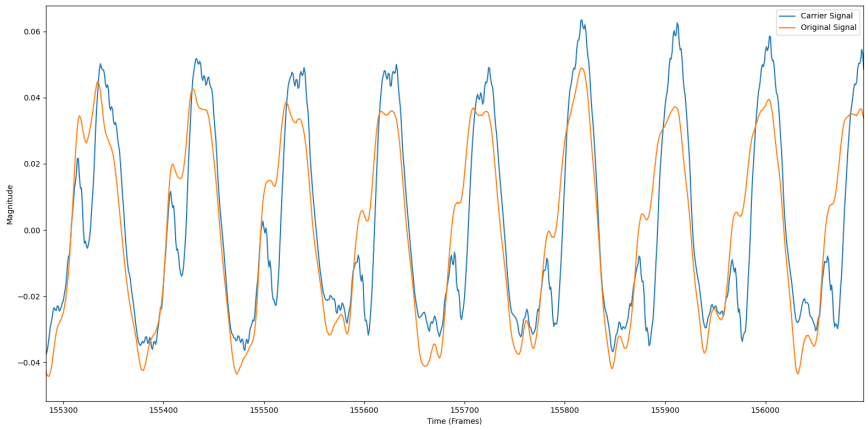


Figure 17: Magnified section of the audio signal in figure 16.

Now that we have analyzed this specific example, it is necessary to explore how this problem behaves when considering detail coefficients of other levels of the multilevel DWT. For this, we ran tests with the same parameters and audio file as above, but incrementing the level or depth in each test by one. As we reached level 10, it was clear that a non-audible signal can be embedded by choosing a level that is deep enough in the multilevel DWT. This comes at a very high cost, though: The amount of detail coefficients you can embed into is a function of the specific *level* chosen. Each time you perform the DWT (once for each level), the amount of detail coefficients is halved. If $|dc_x|$ is the amount of detail coefficients at level x, and $|oc|$ the amount of coefficients for the whole file without any transformation applied, this leads to the following formula:

$$|dc_x| = \frac{1}{2^x} \cdot |oc|.$$

Therefore, while the embedding impact could be described as shrinking linearly when incrementing the DWT level, the embedding capacity shrinks exponentially, which leaves little room for messages to be embedded when reaching a level with a satisfying embedding impact.

5.3 Detection

The detection of a message embedded with echo hiding in a specific band of detail coefficients is rather straightforward. At first, the multilevel discrete wavelet transform is applied onto the audio signal containing the message. Then, knowing the level at which the embedding took place, the detail coefficients at this level of the DWT are taken and analyzed using the cepstrum of the signal. Figure 18 demonstrates a cepstrum analysis on the detail coefficients at level 2 of the signal created in the previous subsection. There are two peaks clearly visible at quefrency values 20 and 25. These correspond to the delays employed in the embedding process and therefore show that a decoding of the original message is possible.

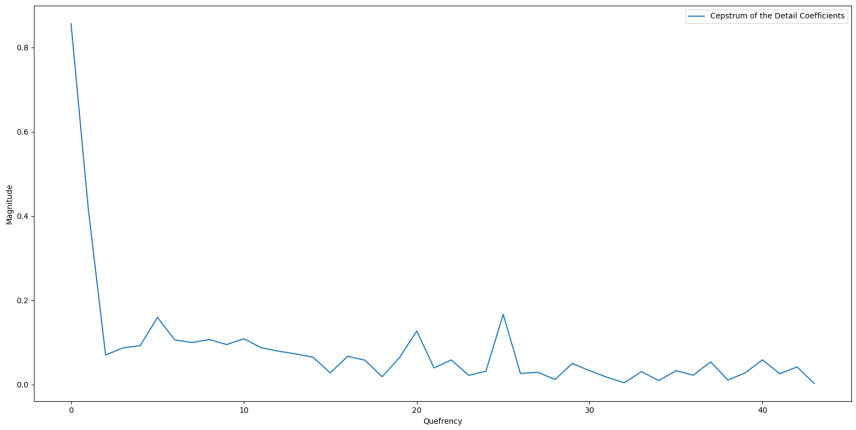


Figure 18: Cepstrum analysis of the detail coefficients at level 2 from the signal in figures 16 and 17. Delay values are clearly visible.

5.4 Results

The results from our tests seem to suggest that this combination of DWT and echo hiding is not suitable for audio steganalysis. For once, the embedding impact is actually very high and while the audio can still be linked to the original, it is heavily distorted on specific frequency levels depending on the chosen level of the detail coefficients, as can be seen in figures 16 and 17. Although, the embedding impact can be lowered by choosing a higher depth/level of DWT decomposition, at the same time the embedding capacity shrinks dramatically, such that it seems very inefficient to embed messages in this way. A small change in few detail coefficients is propagated through the whole file by the inverse DWT.

Furthermore, we performed some shallow analysis on how our algorithm performs on different input sounds. For this, we chose three instruments (cello, trumpet, tuba) and acquired sound samples of a given melody for each instrument [10]. As the length of each sample was only 12 seconds, we looped them to get a `.wav` file with a length of 3 minutes for each instrument. On each of these files, the algorithm was applied and each level of the DWT with subsequent embedding through echo hiding was analyzed. As it turns out, there is a difference, albeit a small one, between the instruments. For the cello and trumpet, level 10 (and subsequent levels) ensured an embedding with an impact that was not audible by our ears. For the tuba, however, by listening very closely, some artifacts could be made out even at levels 10, 11 and 12. As the frequencies of a tuba are rather low, this leads to the rough conclusion that our algorithm is more effective for sounds with an abundance of high frequencies.

Although it seems that the mix of DWT and echo hiding might only be a tool for very niche cases, it does not mean that the combination of DWT decomposition and other steganography or watermarking algorithms is generally doomed. For example, a working concept with spread spectrum audio watermarking can be

found at [11].

5.5 Future Work

At first, all points from section 3.5 apply to this section as well. Of course, the combination of echo hiding with the DWT also benefits from new insights to the DWT. Additionally, the usage of other wavelet types than the Haar wavelet should be evaluated as well.

To prove our assumption about high-frequency inputs being a better host for an inaudible embedded message, it may be worth to analyze a larger set of sounds, tracks or music pieces with distinct frequency distributions. In our case, lone instruments were tested. However, different music genres and audio files with a mixture of instruments might be worth looking into as well.

The algorithm explained in this section is a combination of the *discrete* wavelet transform and echo-hiding. As also explained in section 3.5, the *continuous* wavelet transform has not yet been investigated in the context of the project but could potentially improve (or downgrade) the results.

6 Discrete Spread Spectrum Technique

6.1 Introduction

DSSS stands for Direct Sequence Spread Spectrum. Data to be transmitted is divided into small pieces and each piece is allocated to a frequency channel across the spectrum. Transmitter utilizes a phase varying modulation technique to modulate each piece of data with a higher data rate bit sequence. A key is needed to embed messages into the noise, this key is used to generate a pseudo-noise wave. The information to be embedded must first be modulated using the pseudorandom generator . Spread Spectrum method is known to be very robust, but as a consequence the cost is high and the information capacity is limited.

Spread spectrum technique spreads hidden signal data through the frequency spectrum. Spread Spectrum (SS) is a concept developed in communications to ensure a proper recovery of a signal sent over a noisy channel by producing redundant copies of the data signal.Basically, Data is multiplied by an M-sequence code known to sender and receiver, then embedded in the cover audio. Thus, If noise corrupts some values, there will still be copies of each value left to recover the embedded message.In conventional direct sequence spread spectrum (DSSS) technique was applied to hide confidential information in WAV audio digital signals.

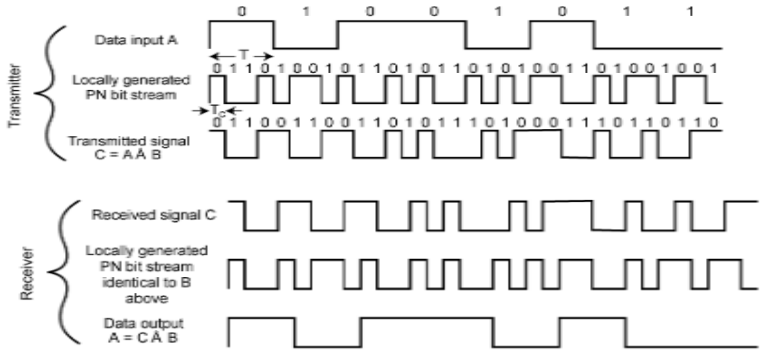


Figure 19: Example of Discrete Spread Spectrum

6.2 Embedded message

The embedding step embeds the message (data array) such as the one below[12] -

```
# generate random integer values
from numpy.random import seed
from numpy.random import randint
# seed random number generator
seed(1)
# generate some integers
data_array = np.random.randint(0, 2, 20)
print(data_array)
```

Notice that if we calculate the data array without the use of seed then it generates different values for the array on each iteration of the above code snippet.

6.3 Encoding

In the encoding step, we take the original audio signal and input the data array into it to generate the stegowork. An array of encoding bits is taken and a mixer signal is obtained by reshaping the encoding bits array with an array of ones. The mixer signal is then convolved with a hanning window to produce a smoothed out mixer signal.

```
    encbit = np.transpose(np.floorxx](np.reshape(bits,N*1,order="C"
    )))
    mix_sig = np.reshape(np.ones((L,1))*encbit,N*L,order="C")
    c = np.convolve(mix_sig,hanning(K))
    w_norm = c/max(abs(c))
```

A pseudo-random number generator is also taken in the next step to generate the key for the embedding process under specific conditions. The key values are calculated and generated via the random.randint function of the numpy package.

```
def prng2( key, L ):
    passwd = np.random.randint(key*L)
    seed(passwd)
    pwd = 2*(np.random.rand(L,1)>0.5)-1
    pwd1 = pwd.ravel()
    return(pwd1)
```

The final stegowork is generated by combining the carrier signal with the mixer signal and pseudo-random number generated above.

```
    stego = data[1:N*L+1,] + alpha*mix*prng2(10,352800)
```

6.4 Decoding

In the Decoding step, we take the extracted signal (obtained in the last step of the encoder) and correlate that with each of the values from the pseudorandom number generator array.

```
    r = prng2(10,352800)
    dat = np.zeros((N,1))
    corr = numpy.correlate((sig_extract),(r),'full')/L
    for k in range(1,N):
        if(corr[k]<0):
            dat[k] = 0
        else:
            dat[k] = 1
```

A new binary array $\{0, 1\}$ is then created out of these correlation values, depending on whether they are less than , greater than or equal to zero. These binary values are then output, to generate the decoded message.

The other decoding or detection methods are included in the Steganalysis section , where the peak calculation technique , the visual detection method using Mel-Frequency Cepstral Coefficients as well as the Machine Learning approach , where key features such as; Chroma Short-term Fourier Transform , Root Mean Square Error, The Spectral Centroid and Zero-Crossing Rate for the original versus each of the stega works are covered in detail.

6.5 Error calculation

The following function calculates the bit error rate by comparing the bit differences between the hidden message and the extracted (decoded) message.
Bit-Error Rate calculation :

```
def bit_error(hidden,extracted):
    y = hidden
    x = extracted
    l = min(len(x),len(y))
    ber = 0
    for i in range(1,l):
        if(y[i] != x[i]):
            ber += 1

    return(ber/l)
```

6.6 Test Cases

The below test cases take into consideration two specific modes for calculation of correlation between the pseudo-random number and the stegowork; i.e, 'full' mode (by default, mode is 'full'. This returns the convolution at each point of overlap, with an output shape of (N+M-1,)) and 'valid' mode (returns output of length max(M, N) - min(M, N) + 1. The convolution product is only given

for points where the signals overlap completely.).

We observe that both of these modes give us comparable bit error rates for the inputs of the hidden data array and the decoded output.
Using the 'full' mode for correlation:

```
err9 = bit_error(data_array2,msg3)

print(err9)


#[0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1] dec
#[1 0 0 1 1 1 0 1 1 1 1 0 1 1 0 1 0 1 0 1] orig
```

```
0.25
```

```
err7 = bit_error(data_array2,msg3)

print(err7)


#[1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]  dec

#[1 1 1 0 0 0 1 0 0 1 1 0 1 0 1 1 0 1 1 1] orig
```

```
0.45
```

Figure 20: Error-rates using 'full' mode

Using the 'valid' mode for correlation:

```
err14 = bit_error(data_array2,msg3)

print(err14)


#[0 1 0 0 1 0 0 0 1 0 0 0 1 1 1 1 1 0 0 0] orig
#[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] dec
```

```
0.4
```

```
err = bit_error(data_array2,msg3)

print(err)


#[1 1 0 0 0 1 1 0 0 0 0 1 0 1 0 1 1 1 0 1] orig
#[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] dec
```

```
0.45
```

Figure 21: Error-rates using 'valid' mode

6.7 Conclusion

The Discrete Spread Spectrum Technique makes use of data that is transmitted after dividing it into small pieces. Each piece is then allocated to a frequency channel across the spectrum. The data to be embedded is modulated with a pseudo-random number and in turn multiplied with a mixer signal . It is then added to the carrier signal to generate the Stegowork. The Decoding step makes use of correlation, to correlate the extracted signal (stegowork) with the pseudo-random number. Thus can be done in two modes , the 'full' and 'valid' , with the former returning the convolution at each point of overlap between the stegowork and pseudo-random number. The latter returns the convolution product only for the points where the signals overlap completely. In our project , the bit error rate

for both the above modes was found to be between 0.25 and 0.45. To improve over the conventional decoding technique and better the accuracy of prediction for stego vs original work, other unconventional techniques of hidden message detection were then employed such as using the sequential model in Machine Learning, that could distinguish the stegowork from the original work with an accuracy of 1. Other visual techniques such as the Mel-Frequency cepstral coefficients as well as Peak and Prominence analysis have been employed to check for visual distinction between the two works , i.e; Stegowork vs Original work.

6.8 Future Work

A possible future work to consider would be the combination of the Discrete spread spectrum technique with the Echo hiding approach. While a direct combination of the two approaches is difficult, but going by the following source[13], it is possible to switch between these two techniques for embedding and extraction of hidden message bits.

Primary Key
In producing the proposed key method, the sender and the receiver should first agree on the primary key; then a matrix with fixed numbers, whose rows and columns are equal to the columns of the primary key, is provided for both parties of steganography.

Subkeys with examples
For producing the sub-keys , we then act by the following algorithm -

- The primary key is multiplied in constant matrix, this is a decimal multiplication.
- The result is multiplied in binary number , the result of this process is the ith subkey.
- We shift the primary key and the constant matrix keys to the right.
- To produce the next keys, we repeat stages 1 to 3 until the total number of bytes equals the number of bytes of the hidden message

If the primary key is according to string 2531 and constant matrix in the first line 1260, the second line 3001, the third line 1021, and the last line 2021,the shifted key will be equal to 1253 and shift to the right for each line in matrix is 0126, 1300, 1102, and 1202, respectively. In the process of multiplication, at first 2531 is multiplied by 0111,which is the first column, then the same number is multiplied by 6022 in the second column. We continue up to the last column,and finally we sum up the results. The sum is equal to 96839327, whose binary equivalent is 10111000101101001101001111 1

Proposed Embedding Method
If the first byte of the first subkeys equals 1,then we apply the algorithm of spread spectrum for the first frame of sound. And if the byte is equal to 0, the algorithm of echo hiding is used for the first frame of sound. Then,the second

byte of the first subkey and the second frame of the sound are checked. We continue the process for the next bytes until the hidden message or the first subkey ends.

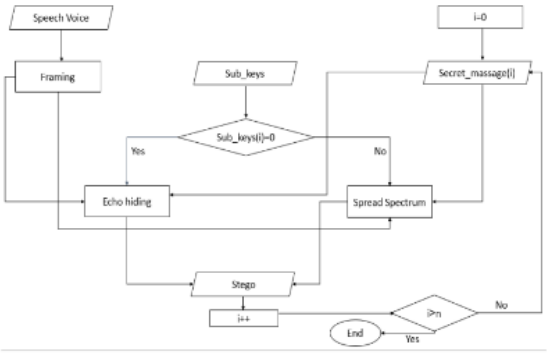


Figure 22: The embedding process according to proposed method

Proposed Extraction Method
At the extraction stage, we will do what we did at the integration stage. First, the receiver who is aware of steganography,produces a subkey, because the primary key and the algorithm of sub-key have been shared with the sender and the receiver. Since every byte of subkey is 0 or 1, the first or second extraction algorithm is used to extract the hidden message. We continue this process until we get the number of byte and subkey which were also hidden using the first algorithm.

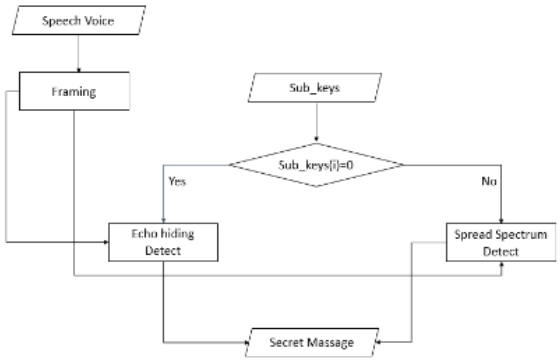


Figure 23: The Extraction procedure according to the proposed method

7 Comparison of two Signals

7.1 Naive algorithm

To verify the state of our embedding and the hidden echos we implemented an algorithm which compares two given signals. For this comparison we used the

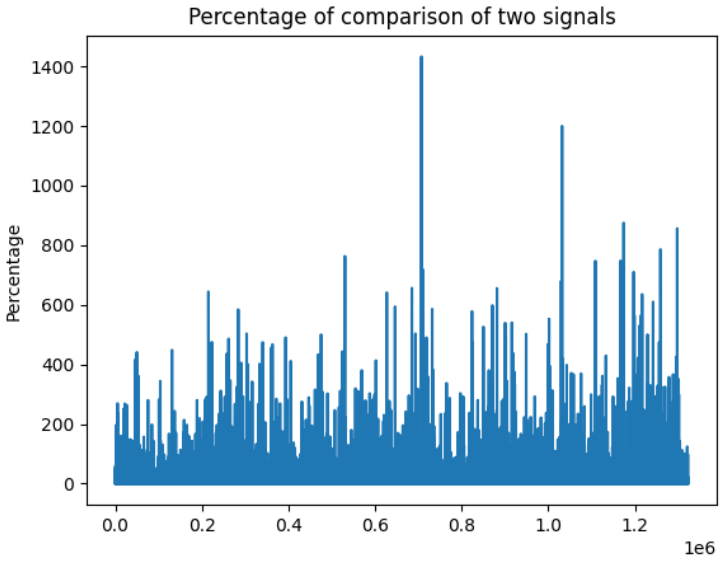
y-values out of the signals and transferred them into two new arrays, which should be similar regarding the number of x-values. Afterwards the algorithm will compare every single y-value of those two arrays and calculate a difference in percentage. This will result in a new array which is plotted for further inspections and interpretations. This algorithm also provides the opportunity to search for a specific interval in those given signals. To ensure the functionality of this procedure we designed a dynamical user argument parser, which will trigger different conditions. If an user won't provide any arguments then a default calculation will be run, which calculates the complete percentage of the difference of both signals.

```
    try:
        start = int(sys.argv[1])
        end = int(sys.argv[2])

    except:
        print("You provided no area for closer inspection. Default
    plot will be created!")
        start = 0
        end = len(data_sound)

    try:
        index = int(sys.argv[3])
        width = int(sys.argv[4])
    except:
        index = None
        width = None
        print("No index and frame provided")
```

Also there is a way to inspect an index and calculate an average value based on the given width. This can come in handy, if there is a possible peak which could be a hint for some embedding.
It's important to state, that this algorithm only works, if both signals are present, which is the original and the modified one. Based on the results of those calculations it's possible to verify the functionality as well as the embedded echo of our used echo-hiding-algorithms.
The following code snippet will show you the core concept of our calculation as well as the optional parameters, which makes our algorithm so extreme flexible.

```
# index is the wanted point where an average should be calculated
# width represents the range to the left and right of this given
    point
# both parameters are optional, but if you provide one you have to
    pass the other one as well.
def percentage_one(audio_org, audio_mod, start=None, end=None,
    index=None, width=None):
    rounds = len(audio_org)
    result = []

    if index is None:
        print("First")
        for i in range(rounds):
            if audio_org[i] == 0:
                value = 0.0
            elif audio_mod[i] == 0:
                value = 1
            else:
```

```
16                 value = abs(((audio_org[i] - audio_mod[i]) /
    audio_org[i]) * 100)
17             result.append(value)
```



As seen based on this image the difference could be quite huge, but this results in cases, where the values are so small that we can't hear a difference, even if the values differ that much. For example, if a value is $1.2e^{-8}$ compared to $1.2e^{-5}$ then the difference is enormous, but still not audible. For those cases we, as mentioned earlier, expanded our algorithm to be able to let us look closer at those specific points to verify that the embedding is not audible.

7.2 Average Algorithm

For further inspections regarding some visible or audible peaks we modified the already explained algorithm even further. In this case the the algorithm will trigger an additional condition based on the parsed arguments. As mentioned in the naive algorithm section those additional arguments are the index and the width of the interval, where the average calculation should take place. Based on the index i and the width w of the interval a frame will be created, which looks like $[i - w, i + w]$. Followed by this knowledge the algorithm will add up every value in the original and modified signal and divide it by the number of x values used for the interval, which will result in $\frac{\sum_n^{i+w} n}{2w+1}$, where $n = i - w$. The result will then be inserted into the existing array and plotted, as soon as the algorithm determines. This method of using the algorithm best fits for small parts of the signal, because otherwise the number of x values is to huge to see a difference with this calculation compared to the naive one.

7.3 Conclusion

We used this algorithm to compare our signals after the process of embedding to make sure, that everything works as intended or mentioned. Based on this usage

we where able to detect changes, even if they where not audible. This algorithm is not recommended for detecting potential embeddings, because the signals have to have the exact same length. If the signal gets eavesdropped, then the original signal is needed for comparison, but the quality of the downloaded content can differ, without being audible. So we clearly recommend this algorithm for internal testing only. For later purposes it could be improved by returning the index, which seems to have something embedded to make it easier for the user to detect it or to take a closer look at. Also it can be even more dynamic for other types of signal files. Additionally an improvement regarding unusual behaviour, where a new frame is created of the part of the signal for further inspections can be done.

8 Steganalysis

8.1 Attacking Echo Hiding through Interpolation

To test the robustness of the echo hiding approach in section 4, we derived a steganalysis approach based on interpolating byte values of the sample coefficients in an audio file. After an echo has been embedded into an audio file, the resulting audio was broken down into an ordered list of its sample values. For an in-depth look into the binary structure of `.wav`-files, see [14]. For the PCM16-subtype of `.wav`, each sample is represented as a two-byte pair in 2's complement unsigned integers, ranging from 0 to 65535. The list of samples of any `.wav` audio file can therefore be expressed as a list of integers.

To try and erase the watermark without impacting the audio file, at first we interpolated each integer i based on its previous sample value p and subsequent sample value s. This was done by calculating the average of both p and s and assigning it as the new value for i: $i \leftarrow \frac{p+s}{2}$. This technique was equally simple as effective, since it erased the trace of the echo sufficiently and did not audibly impact the audio.

Next, we narrowed the interpolation down to samples we deemed 'outliers'. An outlier was defined as an integer, which is either bigger or smaller than both of its previous and it subsequent sample, i.e. for a sequence of samples p, i, s, sample i was deemed an outlier if ($i > p$ AND $i > s$) or if ($i < p$ AND $i < s$). Every outlier was then assigned a new value based on the average of p and s, as seen above. In this method, however, there was a negligible amount of outliers present in our test signals, such that the interpolation did not change a sufficient number of samples to have an impact on the detection process of the echo.

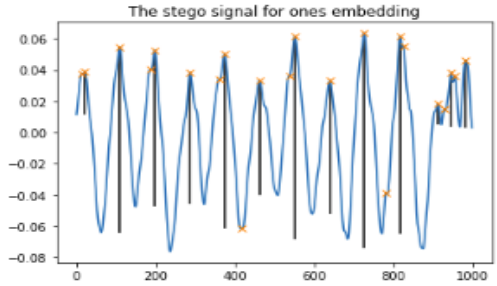
Between these two methods, there might exist a middle-ground, where you can find a percentage value of samples that have to be interpolated for the effect to be sufficiently large to destroy a watermark. Keep in mind that these methods were tested on a rather small set of sample files and parameters, lacking the diversity to confidently derive a general conclusion. Although, it seems to be a low-effort approach to tamper with a watermark or steganographic message effectively.

8.2 General Comparisons

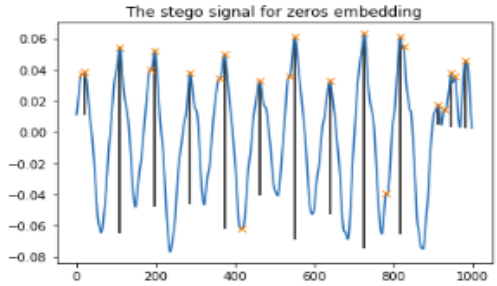
For the Discrete Spread spectrum technique, the peak comparisons were conducted to clearly visualize the peak value variations between the stego-work (with random 0s and 1s) vs the original work.

```
data, samplerate = sf.read('./stegoworksdsss/original/demo.wav')
crop_data = data[13000:14000]
peaks2, _ = find_peaks(crop_data)
prominences2 = peak_prominences(crop_data, peaks2)[0]
prominences2
contour_heights2 = crop_data[peaks2] - prominences2
plt.plot(crop_data)
plt.plot(peaks2, crop_data[peaks2] , "x")
plt.vlines(x=peaks2, ymin=contour_heights2, ymax=crop_data[peaks])
plt.title('The original signal')
plt.show()
print(peaks2)
```

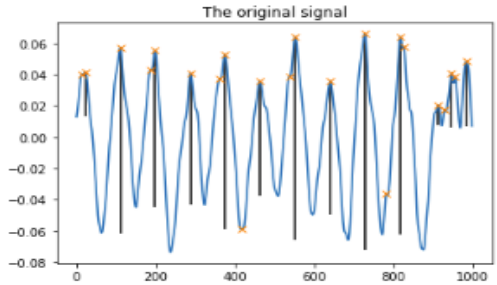
The Peaks and Prominences for the original work as well as the stegoworks with 0s and 1s embedding are as shown below -




```
[ 14  21 110 186 197 287 361 374 417 462 537 551 640 727 782 817 826 913
 928 946 955 983]
```




```
[ 14  21 110 186 197 287 361 374 417 462 537 551 640 727 782 817 826 913
 928 946 955 983]
```




```
[ 15  22 111 187 198 288 362 375 418 463 538 552 641 728 783 818 827 914
 929 947 956 984]
```

Figure 24: Peaks and Prominences from Dataset

As can be seen in the above plottings , there is a corresponding difference in the array values between the original signal versus the stegoworks , where the digits often vary by a single decimal value, such as; 14 in the stegoworks to 15 in the original signal , 20 in the stegoworks in comparison to 21 in the original and so on. This gives us a visual method of concluding that there is a difference between the original and the stegowork signals.

Another method for visually comparing between the original vs the stegowork signal is the Mel-Frequency Cepstral Coefficients (MFCCs) . This is covered in the subsection below.

8.3 Mel-Frequency Cepstral Coeffcients

The Mel-frequency cepstral coefficients prove to be an essential visual detection tool to find differences between the Original work versus the stegoworks in the Discrete Spread Spectrum (DSSS) technique.

Mel-frequency cepstral coefficients (MFCCs) are coefficients that collectively make up an MFC. They are derived from a type of cepstral representation of the audio clip (a nonlinear "spectrum-of-a-spectrum").[15]

Process of acquiring MFCC from a spectrogram[16]:

1. The process of acquiring MFCCs from a spectrogram is via a triangular filterbank placed at linear steps on the mel-frequency scale.

2. When each window of the spectrogram of a speech segent is multiplied with the triangular filterbank, we obtain the mel-weighted spectrum.Here we see that the gross-shape of the spectrogram is retained, but the fine-structure has been smoothed out. In essence, this process thus removes the details related to the harmonic structure. Since the identity of phonemes such as vowels is determined based on macro-shapes in the spectrum, the MFCCs thus preserve that type of information and remove "unrelated" information such as the pitch.

3. The final outcome is obtained once the mel-weighted spectrogram is multiplied with a DCT to generate the final MFCCs. Where the mel-weighted spectrogram does retain the original shape of the spectrum, the MFCCs do not offer such easy interpretations. It is an abstract domain, which contains information about the spectral envelope of the speech signal.

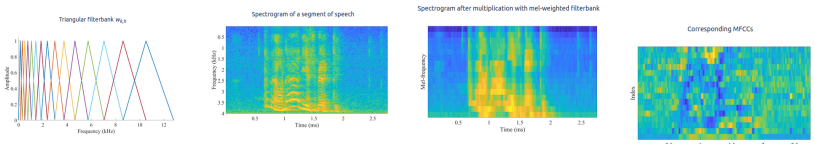
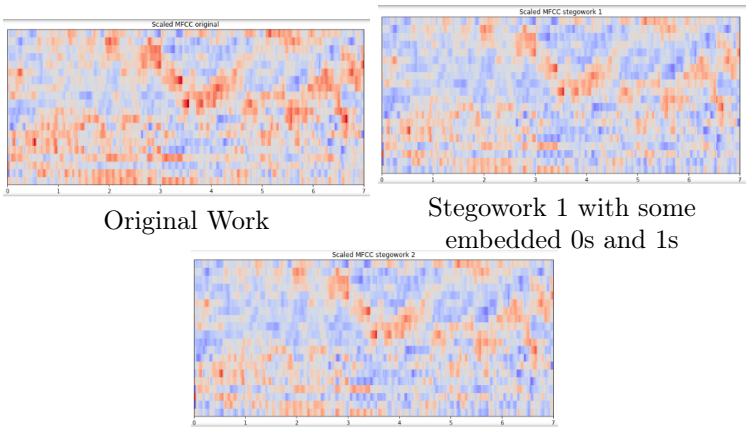


Figure 25: A visual summary of MFCC features

In the case of Discrete Spread Spectrum, we have taken the dataset to be 1)The original MFCC ,2) The MFCC for the stegowork with random 0s and 1s 3) The MFCC for stegowork with another random sequence of zeros and ones. The MFCC index vs Time figures generated for our dataset are as follows -



Original Work

Stegowork 1 with some embedded 0s and 1s

Stegowork 2 with some embedded 0s and 1s

Figure 26: Scaled MFCCs from Dataset

As can be clearly seen in the above figures from our dataset , the Scaled MFCC visualisation of the original work has deeper intensities, especially in the red values. There is a significant reduction in this intensity when it comes to the stegoworks with the random 0s and 1s embedding. This could prove instrumental in making a conclusion as to which is the original work vs the stegoworks.

8.4 Machine Learning Approach

In the Machine Learning Approach for the DSSS technique , we take the three files, i.e ; the Original wave file , Stegowork1 (with random zeros and ones) as well as Stegowork2 (with another set of random zeros and ones). We extract the following information out of all the three files - chroma short-term fourier transform, root mean square error, spectral centroid, spectral bandwidth, rolloff and zero-crossing rate.

The training and testing set is taken out of these three files in the ratio of 80:20 . The model used to train is the Sequential model with 400 epochs and a batch-size of 128.It is observed that the model made correct predictions after just 3 epochs, in identifying the original work as well as each of the stego works separately.[17]

Chroma Short-term Fourier Transform :

To obtain back the hidden time information, Dennis Gabor introduced in the year 1946 the modified Fourier transform, now known as short-time Fourier transform or simply STFT. This transform is a compromise between a time- and a frequency-based representation by determining the sinusoidal frequency and phase content of local sections of a signal as it changes over time. In this way, the STFT does not only tell which frequencies are 'contained' in the signal but also at which points of times or, to be more precise, in which time intervals these frequencies appear.

Root Mean Square Error :

Root Mean Square Error (RMSE) is the standard deviation of the residuals (prediction errors). Residuals are a measure of how far from the regression line data points are; RMSE is a measure of how spread out these residuals are. In other words, it tells you how concentrated the data is around the line of best fit. Root mean square error is commonly used in climatology, forecasting, and regression analysis to verify experimental results.

The formula is:

$$\mathrm{RMSE} = \sqrt{\overline{(f-o)^2}}$$

Where:
f = forecasts (expected values or unknown results), o = observed values (known results).

The Spectral Centroid :

The spectral centroid is a measure used in digital signal processing to characterise a spectrum. It indicates where the center of mass of the spectrum is located.

It is calculated as the weighted mean of the frequencies present in the signal, determined using a Fourier transform, with their magnitudes as the weights, where $x(n)$ represents the weighted frequency value, or magnitude, of bin number n, and $f(n)$ represents the center frequency of that bin.

$$\frac{\sum_{n=0}^{N-1} f(n)x(n)}{\sum_{n=0}^{N-1} x(n)}$$

Roll-off :

The smooth fall of response to zero at either end of the frequency range of a piece of audio equipment.

Zero-Crossing Rate (ZCR) :

The zero-crossing rate (ZCR) is the rate at which a signal changes from positive to zero to negative or from negative to zero to positive.

Accuracy per Epoch:

The accuracy per epoch sample is shown here for the predictions on the dataset using the DSSS technique -
As can be seen from the result in figure 27, using the DSSS technique, the accuracy for predicting the overall stegowork is 1 after just a few epochs of learning.

The accuracy per epoch sample is shown here for the predictions on the dataset using the Phase-Encoding technique -

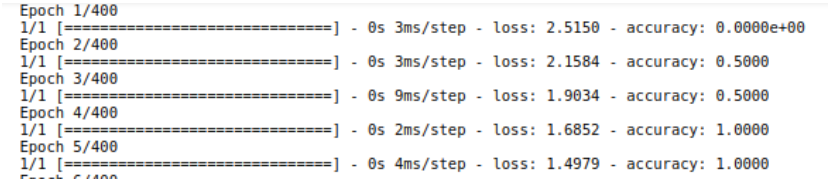


Figure 27: Accuracy per epoch

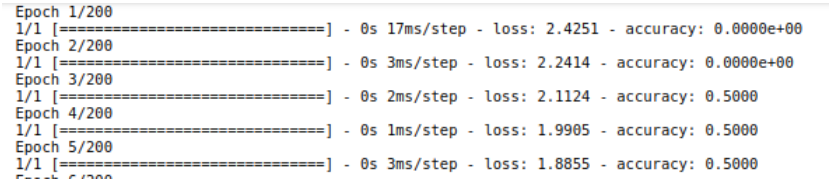


Figure 28: Accuracy per epoch

As can be seen from the result in figure 28, using the Phase-Encoding technique, the accuracy for predicting the overall stegowork is 0.5 after a few epochs of training the model.

Findings for readings:

Below is the image for the findings - (Label 0 stands for the original work, Label 1 for stego work 1 and Label 2 for stego work 2) using DSSS -

	chroma_stft	rmse	spectral_centroid	spectral_bandwidth	rolloff	zero_crossing_rate
0	0.209805	0.053749	784.025099	785.480247	1298.761907	0.049807
1	0.203358	0.057514	779.086546	798.692074	1313.338145	0.049969
2	0.203358	0.057514	779.088291	798.697218	1313.338145	0.049969

Figure 29: Findings from readings of all three works

Below is the image for the findings - (Label 0 stands for the original work, Label 1 for stego work 1 and Label 2 for stego work 2) using Phase Encoding technique -

	chroma_stft	rmse	spectral_centroid	spectral_bandwidth	rolloff	zero_crossing_rate
0	0.209805	0.053749	784.025099	785.480247	1298.761907	0.049807
1	0.209628	0.107401	777.489339	767.140220	1287.345812	0.049646
2	0.209628	0.107401	777.489339	767.140220	1287.345812	0.049646

Figure 30: Findings from readings of all three works

9 Conclusion

In this project, we have implemented several techniques of audio steganography, starting from Discrete Wavelet Transform to the Echo Hiding technique, followed by the Discrete Spread Spectrum. We have also managed to combine the Echo-hiding technique with the Discrete Wavelet Transform to provide some interesting results. In the subsequent section, we have moved on to deriving comparisons using plotting techniques such as general comparisons as well as the averaging method.

Section 3 analyzes the suitability of the discrete wavelet transform for steganographic embedding. For this, we constructed and implemented a steganographic algorithm that embeds and detects messages in the detail coefficients of audio files. Furthermore, we looked at a variety of parameters and made progress in finding parameter configurations that perform better than others. In section 5, we looked beyond the scope of the discrete wavelet transfer as a single entity, but as a building block in conjunction with other steganalysis algorithms. In this context, we explored the possibility of enhancing the echo hiding algorithm by passing specific bands extracted by the discrete wavelet transform as its input. The results, however, suggested, that this method may only be suitable for certain niche cases, as the embedding impact could only be reduced to an acceptable level by lowering the embedding capacity significantly.

As discussed, the Echo Hiding algorithm is basically a common watermarking technique. One goal of the project was to find an approach that is satisfiable in the understanding of steganography. The implementation in combination with a key stream is a functional solution to disguise the embedded work in the carrier signal. After some improvements, like the crossfader for the key stream or a suitable decay rate, the message can be embedded into the carrier signal via echoes. Also the test sets shows that it is possible to blur the detection of the hidden message more that the original algorithm. But to satisfied a steganographic approach it need more test further on and also to find the perfect parameters for the embedding, like delay, decay, key distribution and so on. For the future work it is possible to explore audio features to get an idea of the changes in the marked work. Also the an other calculation in combination with the cepstrum can clear the picture of an functional steganographic algorithm.

The Discrete Spread Spectrum Technique makes use of data that is transmitted after dividing into small pieces. Each piece is then allocated to a frequency channel across the spectrum. The Embedding data is modulated with a pseudo-random number and in turn multiplied with a mixer signal to generate the Stegowork. The Decoding step makes use of correlation, to correlate the extracted signal (stegowork) with the pseudo-random number. Thus can be done in two modes , the 'full' and 'valid' , with the former returning the convolution at each point of overlap between the stegowork and pseudo-random number. The latter returns the convolution product only for the points where the signals overlap completely. In our project , the bit error rate for both the above modes was found to be at least 0.25. Other unconventional techniques of hidden message detection were then employed such as using the sequential model in Machine Learning, that could distinguish the stegowork from the original work with an

accuracy of 1.

To aid us in testing and analyzing results, we developed several ways to compare two signals and show their relative difference (i.e. original signal and signal with an embedded message). Section 7 shows a selection of algorithms which give different perspectives of signal differences, combined with a visual representation for a quick and intuitive analysis. Some usage of these algorithms can be seen in figures 1 or 2 in section 3, where it proved very useful to visualize signal differences in certain scenarios involving distinct parameters.

In the final section, various methods of steganalysis have been covered, including the interpolation technique, comparing peaks and prominences between the stegowork and original work as well as Mel-Frequency cepstal coefficients and machine learning to compare findings based on several key features such as roll-off, spectral centroid and zero-crossing rate.

References

[1] Mohammed Salem Atoum. Steganography and watermarking: Review. `https://www.ijsr.net/archive/v7i6/ART20183644.pdf`.

[2] Ming Li, Yun Lei, Jian Liu, and Yonghong Yan. A novel audio watermarking in wavelet domain. In *International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, 2006.

[3] Qiuling Wu and Meng Wu. A novel robust audio watermarking algorithm by modifying the average amplitude in transform domain. In *Applied Sciences,*, 2018.

[4] Jannis Leuther and Sebastian Reichmann. Discrete wavelet transform watermarking framework & algorithm. `https://github.com/YxJannis/audio_stego_wt`.

[5] Gregory R. Lee, Ralf Gommers, Filip Wasilewski, Kai Wohlfahrt, and Aaron O'Leary. Pywavelets: A python package for wavelet analysis. In *Journal of Open Source Software, 4(36), 1237*, 2019.

[6] Bastian Bechthold. Soundfile python library documentation. `https://pysoundfile.readthedocs.io/en/latest/`, 2013.

[7] Jannis Leuther. Visualization of wavelet types in dwt embedding scenarios. `https://yxjannis.github.io/wavelets`.

[8] Shui-Hua Wang, Yu-Dong Zhang, Zhengchao Dong, and Preetha Phillips. *Wavelet Families and Variants*. Springer, 2018. `https://link.springer.com/chapter/10.1007/978-981-10-4026-9_6`.

[9] Frederik Sukop. Analyse und verbesserung von detektierungsmethoden echo-basierter wasserzeichentechnik anhand von verzoegerungens und daempfungsraten, 2020.

[10] Philharmonia Symphony Orchestra. Instrument sound samples. `https://philharmonia.co.uk/resources/sound-samples/`.

[11] Neha Baranwal and Kamalika Datta. Peak detection based spread spectrum audio watermarking using discrete wavelet transform. In *International Journal of Computer Applications Vol 24*, 2011.

[12] Sneha Mohanty. Discrete spread spectrum technique. https://github.com/SnehaMohanty/AudioSteganography/tree/main/discrete_spread_spectrum.

[13] Haniyeh Rafiee and Mohammad Fakhredanesh. Presenting a method for improving echo hiding. In *Journal of Computer and Knowledge Engineering, Vol. 2, No. 1*, 2019.

[14] Documentation microsoft wave file format. http://www.ievs.ch/projects/var/upload/Documentation%20Microsoft%20Wave%20File%20Format.pdf.

[15] Mel-frequency cepstrum. https://en.wikipedia.org/wiki/Mel-frequency_cepstrum.

[16] Tom Baeckstroem. Cepstrum and mfcc. https://wiki.aalto.fi/display/ITSP/Cepstrum+and+MFCC.

[17] Nagesh Singh Chauhan. Audio data analysis using deep learning with python (part 1). https://www.kdnuggets.com/2020/02/audio-data-analysis-deep-learning-python-part-1.html.