Ultrasonic DTMF Steganography: Technical Overview of Encoding and Decoding

Yassine TAMANI
github.com/rabbyt3s

December 21, 2024

Abstract

We present an ultrasonic DTMF-based method for embedding text characters into a WAV file, using frequencies in the 16–20 kHz range. The goal is to hide data in an audio track with minimal audible artifacts. We provide technical details on how the encoding script disperses characters in the time-domain and how the decoding script leverages band-pass filtering and FFT-based peak detection to recover the message.

1 Introduction

Dual-Tone Multi-Frequency (DTMF) has traditionally been used for transmitting control signals (digits, letters) via pairs of low and high frequencies. In this project, we *shift* those frequencies to the ultrasonic range (approximately 16–20 kHz) so that humans generally cannot hear them, yet they remain detectable by straightforward DSP techniques.

Our approach centers on two main scripts:

1. `encode.py`: Reads a WAV file, generates ultrasonic DTMF tones for each character in the message, and injects them into the audio.
2. `decode.py`: Reads the resulting WAV, applies a band-pass filter near 15–21 kHz, and detects the specific frequency pairs to reconstruct the hidden text.

2 Encoding Process (encode.py)

2.1 Frequency Matrix

We define two arrays:

- `FREQS_LOW = [16000, 16400, 16800, 17200, 17600, 18000, 18400]`
- `FREQS_HIGH = [18800, 19200, 19600, 20000, 20400]`

and an `EXTENDED_MATRIX` mapping characters (A–Z, 0–9, etc.) to indices in these arrays. Each character is thus represented by one low and one high ultrasonic frequency.

2.2 Tone Generation

For each character, we:

1. Identify the low and high ultrasonic frequencies,
2. Generate two sine waves (e.g., $A\sin(2\pi f_{\text{low}}\,t)$ and $B\sin(2\pi f_{\text{high}}\,t)$),
3. Sum them (with optional fade-in/out to avoid clicks) and normalize the amplitude.

2.3 Dispersion in the Audio Track

Given an original audio of length L samples and N characters to encode, the script divides the audio into segments so that each character's tone is inserted in a distinct region. Specifically:

- `segment_length = L // (N + 1)`,
- For character index i ($i = 1, \ldots, N$), we place the character's ultrasonic signal near $i \times$ `segment_length`.

We add the tone (scaled by `ultrasonic_amp`) to the original waveform, ensuring the amplitude is kept below clipping level.

2.4 Final Normalization

If the resulting "mixed" signal surpasses amplitude 1.0 in floating-point WAV, we scale everything down slightly to prevent clipping. The final WAV is then written out, typically at 44.1 kHz sample rate so that 20 kHz signals are within the Nyquist limit.

3 Decoding Process (decode.py)

3.1 Band-Pass Filtering

Since the hidden signals lie around 16–20 kHz, we first apply a band-pass filter, e.g., from 15–21 kHz. This significantly attenuates the normal audio content below 15 kHz.

3.2 Block-Wise FFT and Peak Detection

We split the filtered signal into blocks of duration `chunk_duration`, e.g., 0.2 s. In each block:

1. Multiply by a Hanning window to reduce FFT edge artifacts.
2. Compute the magnitude of the FFT.
3. Find peaks exceeding a threshold (e.g., $0.15 \times$ the block's max amplitude).
4. Attempt to match exactly one "low" ultrasonic freq and one "high" ultrasonic freq from our known arrays.

If we detect exactly one match for each sub-band, we identify the corresponding character from `EXTENDED_MATRIX`.

3.3 Character Confidence and Gaps

We impose a minimum gap (`min_gap`) between detected characters to avoid duplication when blocks overlap. If a block yields a strong detection but arrives too soon after the previous one, it's discarded.

Finally, we keep only characters above a certain confidence threshold relative to the block's maximum amplitude. This helps remove spurious detections in noisy segments.

4 Visual Inspection

We generate four main plots to confirm the injection is (or is not) visible:

1. **`wave_original.png`**: Full waveform of the *original* audio.
2. **`wave_encoded.png`**: Full waveform after ultrasonic embedding.
3. **`wave_diff.png`**: The difference (encoded minus original), showing only injected energy.
4. **`spectrogram_comparison.png`**: Spectrogram top (original), bottom (encoded), illustrating newly added high-frequency bands.

Example Figures

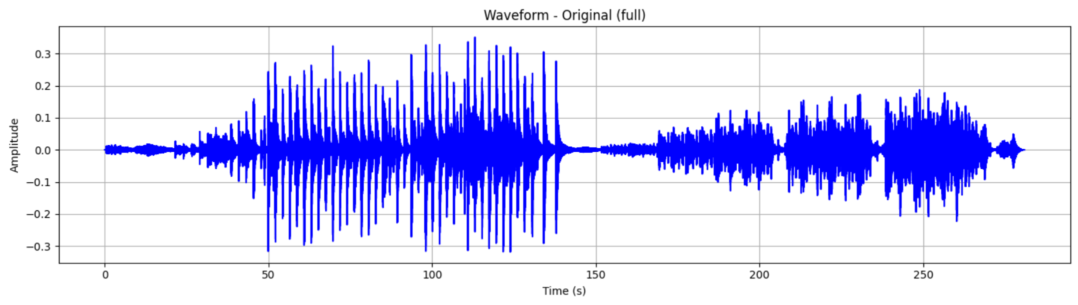

Figure 1: `wave_original.png`: Baseline audio signal.

Original Waveform

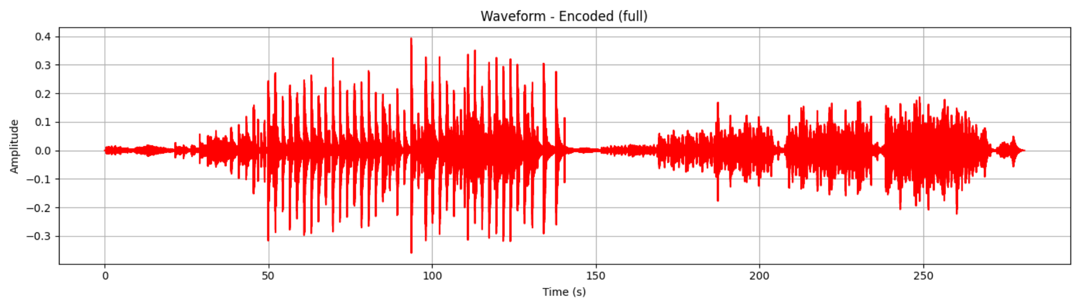

Figure 2: `wave_encoded.png`: Audio now containing ultrasonic tones.

Encoded Waveform

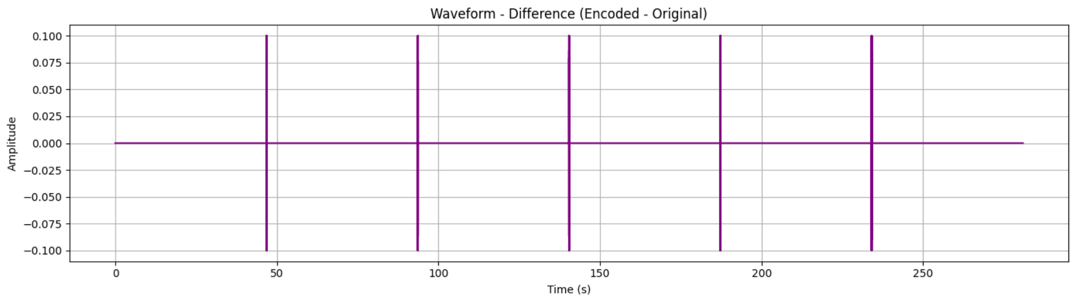

Figure 3: `wave_diff.png`: The injection's time-domain footprint (encoded − original).

Difference Waveform

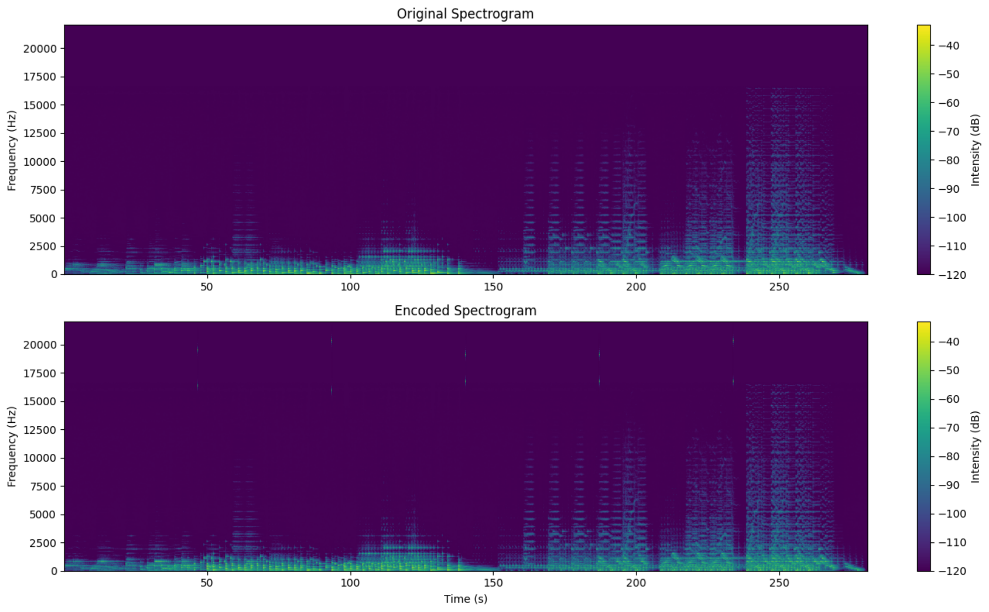

Figure 4: `spectrogram_comparison.png`: High-frequency tones near 16–20 kHz in the encoded track.

Spectrogram Comparison

5 Discussion

5.1 Applications

This ultrasonic DTMF approach can be used for:

- Low-rate steganography, embedding short messages in standard audio,
- Broadcasting hidden instructions or triggers between devices capable of ultrasonic detection,
- Experimental watermarking or beacon-like signals in a public environment.

5.2 Limitations

- **Hardware Response**: Many consumer-grade speakers and mics strongly attenuate or distort signals above 18–20 kHz.
- **Lossy Compression**: MP3 or AAC often remove high frequencies, destroying the hidden data.
- **Data Capacity**: Each character uses two distinct frequencies; large messages become cumbersome or risk increased audibility if amplitude is raised.

6 Conclusion

By mapping DTMF logic into an ultrasonic frequency range, we can hide simple text messages without notably affecting the audible portion of the waveform. The `encode.py` script disperses these tones in time, while `decode.py` recovers them with a band-pass filter and block-wise FFT detection. Although practical deployment depends on hardware capabilities and uncompressed formats, this method serves as a straightforward demonstration of ultrasonic steganography.

Code Availability

All scripts, including `encode.py`, `decode.py`, and optional utilities for visualization, are found at:

`https://github.com/rabbyt3s`

Author

Yassine TAMANI
GitHub: rabbyt3s