

UNIVERSITAT ROVIRA I VIRGILI

AUDIO FINGERPRINTING PRACTICAL WORK

MULTIMEDIA SECURITY

Audio Fingerprinting Practical Work

JAVIER NOVELLA RUIZ

May 26, 2025



UNIVERSITAT
ROVIRA I VIRGILI

Contents

1	Introduction	2
1.1	Technologies used	2
2	Implementation	3
2.1	Module <code>bulddb</code>	4
2.2	Module <code>identify</code>	6
2.3	Fingerprint Database Structure	8
3	Testing	10

1 Introduction

The main goal is to implement an audio fingerprinting system that can identify the original track from different audio samples. The audio samples are divided in four types: clean samples, filtered samples, noisy samples, noisy filtered samples. These samples are short extracts from the original tracks, adding filters or noise to test the system in harsh conditions.

To the implementation a library of forty audio files have been used. Each audio file counts with four samples of the previously commented types (clean, filtered, noisy and noisy filtered). This means a total of sixteen samples per original audio file.

The work includes the development of two tools: `builddb` and `identify`.

- `builddb`: takes the original audios and creates a database with the fingerprint information.
- `identify`: takes a sample and matches the result against the database to return the corresponding original track.

1.1 Technologies used

After reading the Shazam paper [1] I could imagine an application with a lot of recursivity in the main tasks (reading files, processing files etc.). In programming recursivity usually means computationally demanding, so I tried to avoid slow interpreted languages like Python. The language chosen has been C# [2] with the SDK .NET 9.0. The main reason to use this language is that is an object-oriented high-level compiled language. It checked the application demandings, fast and counts with high-level tools to perform the desired tasks.

The main packages used for the implementation are:

- `NAudio`: library for working with audio files, playback, recording, and processing.
- `NWaves`: digital signal processing (DSP) library for audio analysis.
- `MessagePack`: compact binary serialization format for data exchange.

2 Implementation

The implementation is based on the Shazam algorithm [1] and it is divided in two different tools: buildddb and identify. Each tool has been developed in a separated C# module (see Figure 1).

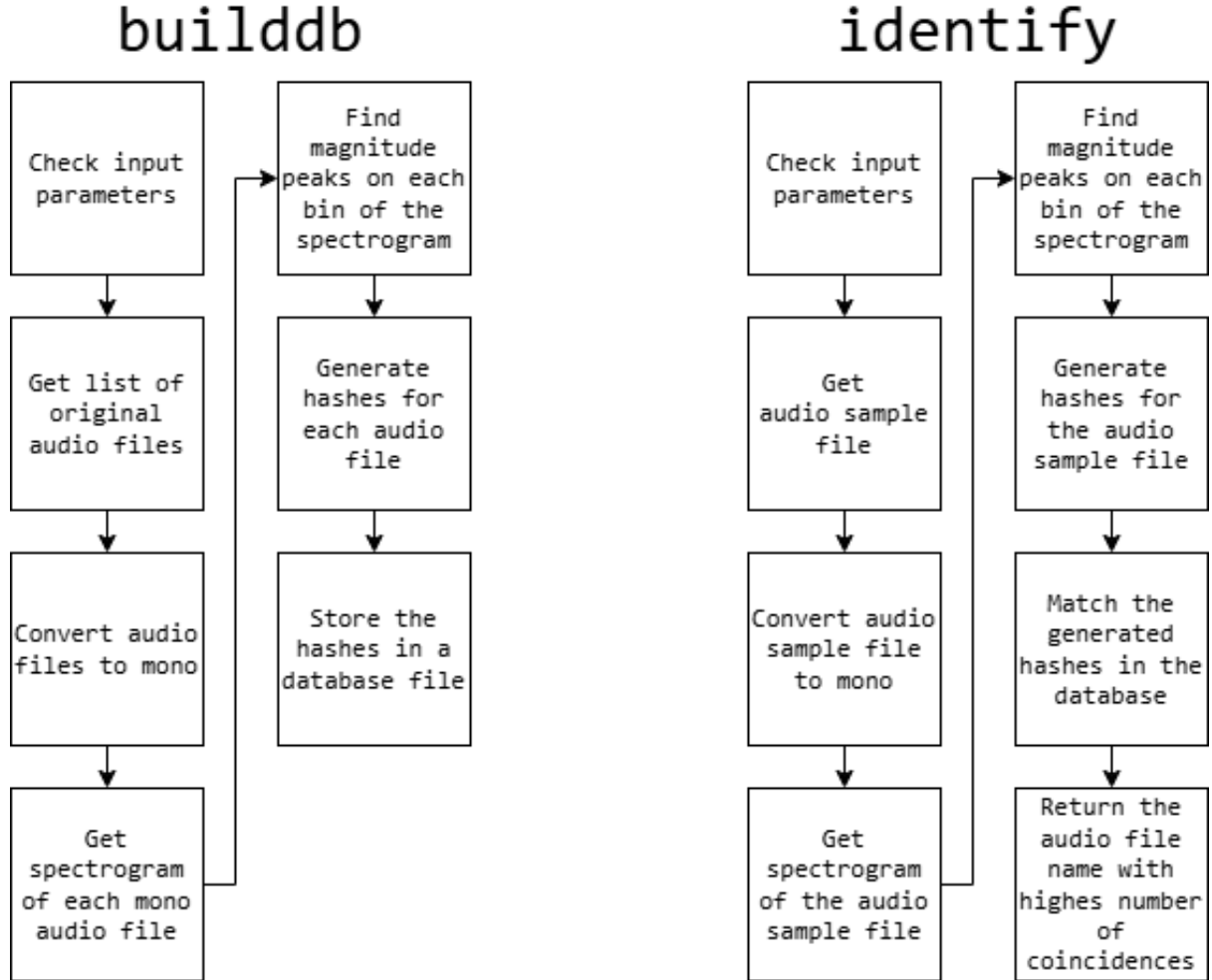


Figure 1: Modules buildddb and identify work flow

The main idea of this process is to split each audio file in small samples and generate a hash for each one. This hash is based on the magnitude peaks as a representative characteristic of the sample. This is done for each audio file and all the hashes are stored in a file that works as a database. During the identification, the same process is carried out with the provided audio sample and then the resultant hashes are collated with the database.

2.1 Module builddb

The module `builddb` is responsible for generating the audio fingerprint database. It takes as input a directory with original audio tracks and produces a serialized file containing the fingerprints of all the tracks. The process is based on the Shazam fingerprinting algorithm [1], and the core tasks are implemented in a state-driven manner, as shown in the flow diagram in Figure 1.

1. **CHECK_INPUT_PARAMETERS**: Validates the command-line arguments.
2. **READ_SONGS_LIST**: Retrieves the original audio files and stores their names and paths.
3. **CREATE_FOLDERS**: Generates folders for storing mono audio and spectrograms.
4. **CONVERT_SONGS_TO_MONO**: Converts stereo audio files to mono using `NAudio`, by averaging both channels. (see Figure 2).

```
case BuildDB.Program.State.CONVERT_SONGS_TO_MONO:
    for (int idx = 0; idx < Files.OriginalsPath.Length; idx++)
    {
        if (!File.Exists(BuildDB.Program.RESULTS_FOLDER_NAME + "\\\" + BuildDB.Audio.MONO_FILES_FOLDER_NAME + "\\\" + Files.OriginalsName[idx]))
        {
            using var audio = new AudioFileReader(Files.OriginalsPath[idx]);
            if (audio.WaveFormat.Channels == (int)BuildDB.Audio.Channels.STEREO)
            {
                var monoProvider = new StereoToMonoSampleProvider(audio) { LeftVolume = 0.5f, RightVolume = 0.5f };
                WaveFileWriter.CreateWaveFile6(BuildDB.Program.RESULTS_FOLDER_NAME + "\\\" + BuildDB.Audio.MONO_FILES_FOLDER_NAME + "\\\" + Files.OriginalsName[idx], monoProvider);
            }
            else if (audio.WaveFormat.Channels == (int)BuildDB.Audio.Channels.MONO)
            {
                File.Copy(Files.OriginalsPath[idx], BuildDB.Program.RESULTS_FOLDER_NAME + "\\\" + BuildDB.Audio.MONO_FILES_FOLDER_NAME + "\\\" + Files.OriginalsName[idx]);
            }
        }
        Files.MonoName[idx] = Files.OriginalsName[idx];
        Files.MonoPath[idx] = BuildDB.Program.RESULTS_FOLDER_NAME + "\\\" + BuildDB.Audio.MONO_FILES_FOLDER_NAME + "\\\" + Files.OriginalsName[idx];
    }
    State = BuildDB.Program.State.GET_SPECTROGRAMS;
    break;
```

Figure 2: Stereo to mono conversion averaging both channels using `NAudio`

5. **GET_SPECTROGRAMS**: Computes the magnitude spectrogram for each mono audio track using the Short-Time Fourier Transform (STFT) from `NWaves` [3]. For this step, each signal is segmented into overlapping frames of size 8192 samples (defined as `SPECTROGRAM_WINDOWS_SIZE`) with a hop size of 4096 samples between frames (`SPECTROGRAM_HOP_SIZE`). A Hann window function is applied to each frame to minimize spectral leakage. Only the magnitude component of the STFT is kept, discarding the phase information. The magnitude component is sufficient for detecting frequency peaks. The resulting time-frequency representation allows identification of local amplitude maxima (peaks) in both the time and frequency domains, which are used in the following step. The spectrograms are serialized using the `MessagePack`.

```
case BuildDB.Program.State.GET_SPECTROGRAMS:
    Array.Resize(ref MagnitudeSpectrograms, Files.MonoPath.Length);
    for (int idx = 0; idx < Files.MonoPath.Length; idx++)
    {
        if (!File.Exists(BuildDB.Program.RESULTS_FOLDER_NAME + "\\\" + BuildDB.Audio.SPECTROGRAM_FILES_FOLDER_NAME + "\\\" + Files.MonoName[idx] + ".msgpack"))
        {
            var fileStream = File.OpenRead(Files.MonoPath[idx]);
            var audio = new WaveFileReader(fileStream);
            var signal = audio.Channel[0];
            int windowSize = BuildDB.Audio.SPECTROGRAM_WINDOWS_SIZE;
            int hopSize = BuildDB.Audio.SPECTROGRAM_HOP_SIZE;
            var stft = new Stft(windowSize, hopSize, WindowType.Hann);
            MagnitudePhaseList magPhaseSpectrogram = stft.MagnitudePhaseSpectrogram(signal);
            MagnitudeSpectrogram[idx] = magPhaseSpectrogram.Magnitudes;
            File.WriteAllBytes(BuildDB.Program.RESULTS_FOLDER_NAME + "\\\" + BuildDB.Audio.SPECTROGRAM_FILES_FOLDER_NAME + "\\\" + Files.MonoName[idx] + ".msgpack", MessagePackSerializer.Serialize(MagnitudeSpectrogram[idx]));
        }
        else
        {
            MagnitudeSpectrogram[idx] = MessagePackSerializer.Deserialize<List<float>>((File.ReadAllBytes(BuildDB.Program.RESULTS_FOLDER_NAME + "\\\" + BuildDB.Audio.SPECTROGRAM_FILES_FOLDER_NAME + "\\\" + Files.MonoName[idx] + ".msgpack")));
        }
    }
    State = BuildDB.Program.State.FIND_PEAKS;
    break;
```

Figure 3: Spectrograms obtained using `NWaves`

6. **FIND_PEAKS**: In this step, local peaks are detected across each time frame of the magnitude spectrogram. The objective is to isolate significant frequency components that are likely to be robust under noise, filtering, or distortion. A local peak is defined as a frequency bin whose amplitude is higher than its surrounding neighbours within a radius of two bins (`IsLocalPeak` method) (see Figure 4), and which exceeds a minimum amplitude threshold (`MIN_PEAK_AMPLITUDE = 10.0`). For each frame:

```
1 reference
public static bool IsLocalPeak(float[] frame, int frameIdx)
{
    float current = frame[frameIdx];
    for (int i = -2; i <= 2; i++)
    {
        if (i == 0) continue;
        int idx = frameIdx + i;
        if (idx < 0 || idx >= frame.Length) continue;
        if (frame[idx] >= current)
            return false;
    }
    return true;
}
```

Figure 4: Function to detect the peaks within neighbours in a radius of two bins

- All candidate peaks are collected if they meet both criteria (local maximum and amplitude threshold).
 - These candidates are sorted by magnitude.
 - Only the top 10 peaks (`MAX_PEAKS_PER_FRAME`) are retained per frame.
7. **GENERATE_HASHES**: Fingerprints are generated by pairing each selected peak (referred to as an anchor) with other peaks within a target zone defined by frequency and time distance thresholds. The pairing process follows the next rules:
- For each anchor point, other peaks are considered within a window of frames and bins.
 - A hash is then created that encodes the frequency of the anchor, the frequency of the target peak, and the time delta.
 - A track ID and anchor time offset are also stored alongside the hash.
 - For efficiency, a maximum of 10,000 hashes are generated per track (`MAX_HASHES_PER_TRACK`).

The final hash format uses bit-shifting and masking to combine the frequency and time delta into a 32-bit integer:

$$\text{hash} = (f_1 \& 0x3FFF) \ll 20 \mid (f_2 \& 0x3FFF) \ll 10 \mid (\Delta t \& 0x3FFF) \quad (1)$$

Where f_1 and f_2 are frequency bin indices and Δt is the time difference. This format is inspired by Wang's original design [1].

```

case BuildDB.Program.State.GENERATE_HASHES:
{
    var fingerprints = new List<int hash, int offset, int trackId>();
    for (int trackId = 0; trackId < PeakMap.Count; trackId++)
    {
        for (int anchorTime = 0; anchorTime < PeakMap[trackId].Count; anchorTime++)
        {
            var anchorFrame = PeakMap[trackId][anchorTime];
            foreach (var (f1, mag1) in anchorFrame)
            {
                for (int t2 = anchorTime + 1; t2 <= anchorTime + BuildDB.Audio.HASH_TARGET_ZONE_TIME; t2++)
                {
                    foreach (var (f2, mag2) in PeakMap[trackId][t2])
                    {
                        if (Math.Abs(f2 - f1) > BuildDB.Audio.HASH_TARGET_ZONE_FREQ)
                        {
                            continue;
                        }
                        int deltaT = t2 - anchorTime;
                        int hash = (f1 & 0xFFFF) << 20 | (f2 & 0xFFFF) << 10 | (deltaT & 0xFFFF);
                        Fingerprints.Add((hash, anchorTime, trackId));
                        if (Fingerprints.Count > BuildDB.Audio.MAX_HASHES_PER_TRACK)
                        {
                            break;
                        }
                    }
                }
            }
        }
    }

    var fingerprintEntries = Fingerprints.Select(f => new FingerprintEntry(f.hash, f.offset, f.trackId)).ToList();
    var trackNames = new Dictionary<int, string>();
    for (int i = 0; i < Files.OriginalName.Length; i++)
    {
        trackNames[i] = Files.OriginalName[i];
    }
    var database = new FingerprintDatabase
    {
        Fingerprints = fingerprintEntries,
        TrackNames = trackNames
    };
    string dbPath = BuildDB.Program.RESULTS_FOLDER_NAME + "\\\" + args[1] + ".msgpack";
    var options = MessagePackSerializerOptions.Standard.AllowReadOnlyContractlessReaderResolver.Instance();
    File.WriteAllBytes(dbPath, MessagePackSerializer.Serialize(database, options));
    State = BuildDB.Program.State.CLEAN_TEMPORAL_FILES;
    break;
}
}

```

Figure 5: Generation of the hashes and serialization of the data

8. **CLEAN_TEMPORAL_FILES**: Temporary folders containing intermediate files (e.g., mono audio, spectrograms) are deleted to reduce storage footprint.
9. **SUCCESS**: Final state. A compact **MessagePack**-serialized fingerprint database is saved, ready for matching in the **identify** module.

The resultant database contains a list of fingerprints and their associated data (offset and track ID), as well as a mapping from track IDs to their original filenames. This is stored in an object of type **FingerprintDatabase**, which includes:

- A list of **FingerprintEntry** instances (hash, offset, trackId).
- A dictionary mapping track IDs to audio filenames.

2.2 Module identify

The module **identify** is responsible for determining the original audio track that best matches a given sample. It takes as input a fingerprint database (generated by **buildddb**) and an audio sample. The module extracts fingerprints from the sample and performs matching to identify the source track.

1. **CHECK_INPUT_PARAMETERS**: Verifies that the arguments include a valid fingerprint database path and an input audio file.
2. **READ_SONG**: Loads the input file and initializes metadata. As the identification is done for a single file, arrays for paths and names contain only one element.
3. **CREATE_FOLDERS**: Creates temporary folders for mono audio and spectrogram if they do not already exist.
4. **CONVERT_SONG_TO_MONO**: Converts the input audio to mono using the same logic as in **buildddb**.
5. **GET_SPECTROGRAMS**: Computes or loads the magnitude spectrogram of the sample using the same STFT parameters as in **buildddb**:
 - Window size: 8192 samples

- Hop size: 4096 samples
- Window function: Hann

The same spectral resolution is used for matching.

6. **FIND_PEAKS**: Local peaks in the spectrogram are extracted using the same criteria (amplitude threshold and neighborhood comparison) as the database fingerprints.
7. **GENERATE_HASHES**: Fingerprints are generated from the extracted peaks by pairing each anchor point with nearby peaks within the time-frequency target zone. The same 32-bit hash construction is used.
8. **LOAD_DB**: The fingerprint database is loaded from disk and deserialized using `MessagePack`. The resulting object is of type `FingerprintDatabase`, which includes:
 - A list of `FingerprintEntry` instances (hash, offset, trackId).
 - A dictionary mapping track IDs to audio filenames.

```
case Identify.Program.State.LOAD_DB:
    Files.GPath = args[1];
    var options = MessagePackSerializerOptions.Standard.WithResolver(ContractlessStandardResolver.Instance);
    FingerprintDatabase database = MessagePackSerializer.Deserialize<FingerprintDatabase>(File.ReadAllBytes(args[1]), options);
    DB = database.Fingerprints;
    TrackNames = database.TrackNames;
    State = Identify.Program.State.MATCH_HASHES;
    break;
```

Figure 6: Load of the MessagePack database

9. **MATCH_HASHES**: The list of sample hashes is compared against the loaded database. The algorithm constructs an index of the database by grouping fingerprints by hash value. Then, for each hash in the sample:
 - If the hash exists in the database, it retrieves a list of matching fingerprints.
 - For each match, it computes the time offset difference $\Delta t = t_{db} - t_{sample}$.
 - It increments a vote for the pair (`trackId`, Δt).

After all votes are counted, the best match is determined by finding the `trackId` with the highest number of aligned matches (votes). This voting strategy is based on the technique used in the Shazam algorithm [1]. Finally it returns to the console:

- The ID of the most probable matching track.
- The number of matching fingerprint alignments.
- The filename retrieved from `TrackNames`, if available.

If no strong match is found (i.e., no hash overlaps), the system reports that no match could be determined.


```

case Identify.Program.State.MATCH_RESULTS:
    var fingerprints = (fingerprints:fp => fp.hash).toList(fingerprints).Select(f => (f.hash, f.offset)).ToList();
    var hashes = DB.GroupBy(fp => fp.hash).ToDictionary(g => g.Key, g => g.Select(fp => (fp.trackId, fp.offset)).ToList());
    var votes = new Dictionary<int, TrackId> { };
    foreach (var hash, sampleOffset in sampleHashes)
    {
        if (hashes.ContainsKey(hash, out var matches))
        {
            foreach (var (trackId, offset) in matches)
            {
                int deltaOffset = offset - sampleOffset;
                var key = (trackId, deltaOffset);
                if (votes.ContainsKey(key))
                {
                    votes[key]++;
                }
                else
                {
                    votes[key] = 1;
                }
            }
        }
    }
    var best = votes.GroupBy(v => v.Key, trackId).Select(g => new { TrackId = g.Key, MaxVotes = g.Max(v => v.Value) }).OrderByDescending(x => x.MaxVotes).FirstOrDefault();
    if (best != null)
    {
        Console.WriteLine($"Best match: TrackId = {best.TrackId}, Matches = {best.MaxVotes}");
        if (TrackNames.TryGetValue(best.TrackId, out var name))
        {
            Console.WriteLine($"Filename: {name}");
        }
        else
        {
            Console.WriteLine("Track name not found in database.");
        }
    }
    else
    {
        Console.WriteLine("No match found.");
    }
    State = Identify.Program.State.IDLE_TEMPORAL_FILES;
    return;

```

Figure 7: Code for match and vote for the candidate TrackId

10. **CLEAN_TEMPORAL_FILES**: Deletes temporary mono and spectrogram files used during processing.
11. **SUCCESS**: The match result is printed, and the program terminates.

The **identify** module mirrors the operations of **builddb** to compare the samples and originals fingerprints.

2.3 Fingerprint Database Structure

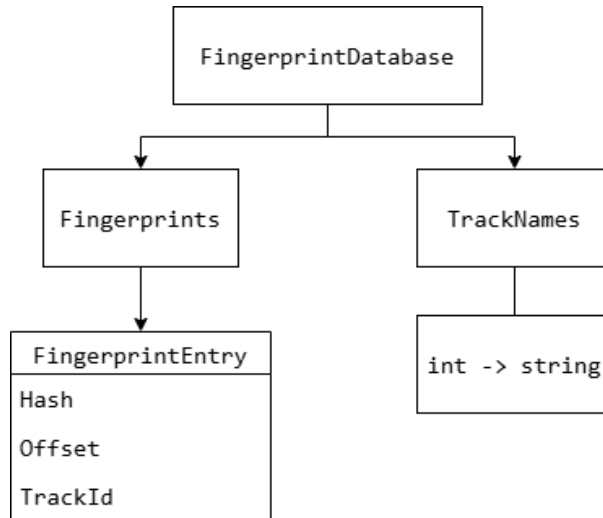


Figure 8: Structure of the **FingerprintDatabase** with **Fingerprints** and **TrackNames**

The fingerprint database is a data structure serialized in the **MessagePack** format. It is created by the **builddb** module and read by the **identify** module for comparison. It contains the following main components:

- **Fingerprints**: A list of records where each record contains:
 - **Hash**: A 32-bit integer fingerprint hash.
 - **Offset**: The time offset (in frames) of the anchor point in the original track.
 - **TrackId**: An integer identifying the audio track.

- **TrackNames:** A dictionary mapping each **TrackId** to its corresponding audio filename.

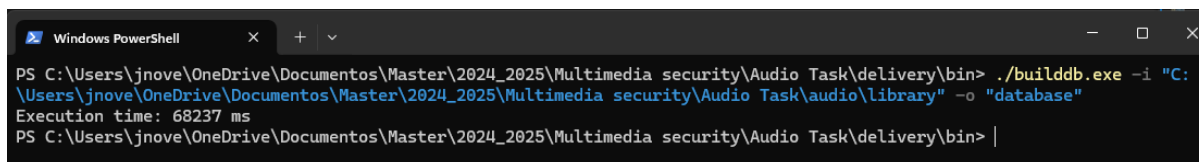
Each fingerprint is stored using the **FingerprintEntry** type, defined as a serializable record. The complete database is in the **FingerprintDatabase** class. The use of **MessagePack** allows the creations of minimal file size and fast binary serialization/deserialization compared to textual formats like JSON. With JSON memory expectations were thrown due to the high amount of data in the database. An example layout is shown in Figure 8.

3 Testing

To test the application first the database must be built using the program `builddb.exe`. For this the following command has to be executed:

```
builddb.exe -i <path_to_folder_with_audio_files> -o <name_of_the_database_generated>
```

The resultant database file with extension `.msgpack` is stored in the folder `results` that will be created in the same directory as the `builddb.exe` during the execution. In the testing computer (AMD Ryzen 5 2600 @ 3.8Ghz - 16 GB RAM DD3 @ 2166 Mhz) takes around 65 and 68 seconds to complete the process.



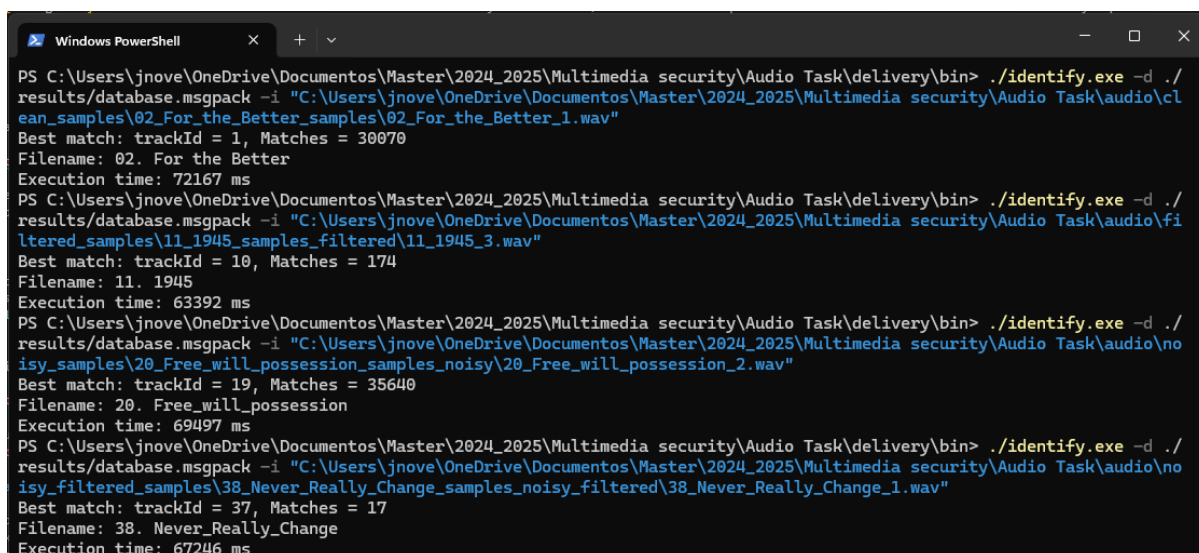
```
Windows PowerShell
PS C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\delivery\bin> ./builddb.exe -i "C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\audio\library" -o "database"
Execution time: 68237 ms
PS C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\delivery\bin> |
```

Figure 9: Successful execution of the `builddb` program.

Once the database is created the identification process can be carried out with the program `identify.exe`. To use it the next command must be executed:

```
identify.exe -d <path_to_the_database_file> -i <path_to_sample_audio_files>
```

The Successful rate with the tests done is 100%. The program has not been teste with all the samples available but all the test done have been successful. The time taken for the identification is always around 60-70 seconds. In the following image some random test are done with clean, filtered, noisy and noisy filtered samples.



```
Windows PowerShell
PS C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\delivery\bin> ./identify.exe -d ./results/database.msgpack -i "C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\audio\clean_samples\02_For_the_Better_samples\02_For_the_Better_1.wav"
Best match: trackId = 1, Matches = 30070
Filename: 02. For the Better
Execution time: 72167 ms
PS C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\delivery\bin> ./identify.exe -d ./results/database.msgpack -i "C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\audio\filtered_samples\11_1945_samples_filtered\11_1945_3.wav"
Best match: trackId = 10, Matches = 174
Filename: 11. 1945
Execution time: 63392 ms
PS C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\delivery\bin> ./identify.exe -d ./results/database.msgpack -i "C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\audio\noisy_samples\20_Free_will_possession_samples_noisy\20_Free_will_possession_2.wav"
Best match: trackId = 19, Matches = 35640
Filename: 20. Free_will_possession
Execution time: 69497 ms
PS C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\delivery\bin> ./identify.exe -d ./results/database.msgpack -i "C:\Users\jnove\OneDrive\Documentos\Master\2024_2025\Multimedia security\Audio Task\audio\noisy_filtered_samples\38_Never_Really_Change_samples_noisy_filtered\38_Never_Really_Change_1.wav"
Best match: trackId = 37, Matches = 17
Filename: 38. Never_Really_Change
Execution time: 67246 ms
```

Figure 10: Successful execution of the `identify` program.

References

- [1] A. L.-C. Wang, *An Industrial-Strength Audio Search Algorithm*. Shazam Entertainment, Ltd. Available at <https://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf>.
- [2] TCM Security, *Python vs C# Which One is Better in 2023?*. Available at <https://tcm-sec.com/python-vs-c-sharp/#:~:text=Performance%20and%20Speed,only%20%20seconds%20to%20run>.
- [3] NWaves DSP Library. Available at <https://github.com/ar1st0crat/NWaves>
- [4] P. Brossier, *Automatic annotation of musical audio for interactive systems*. PhD Thesis, Queen Mary University of London, 2006.
- [5] Y. Zhu and D. P. W. Ellis, *Estimating the similarity of short music excerpts using feature rank histograms*. In Proceedings of the 2008 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 1337–1340.