Sounds of Silence: A Study of Stability and Diversity of Web Audio Fingerprints

Shekhar Chalise

University of New Orleans, New Orleans, schalise@uno.edu

Follow this and additional works at: https://scholarworks.uno.edu/td

Recommended Citation
https://scholarworks.uno.edu/td/2900

This Thesis-Restricted is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Thesis-Restricted in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis-Restricted has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. For more information, please contact scholarworks@uno.edu.
Sounds of Silence: A Study of Stability and Diversity of Web Audio Fingerprints

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by
Shekhar Chalise
B.E. Tribhuvan University- Kathmandu, 2015
May, 2021
Acknowledgements

First of all, I would like to express my sincere gratitude to my advisor Dr. Krishna Phani Kumar Vadrevu for providing his invaluable guidance, support, and suggestions required for this research.

I would also like to thank Dr. Vassil Roussev and Dr. Hyunguk Yoo, for their kind consent to be a committee member for my thesis defense.

I would also like to thank my friends Mr. Rachindra Poudel, Mr. Pujan Pokhrel, and Mr. Nishan Rayamajhee for their invaluable suggestions. A very special thanks to my family and all my friends for their encouragement and support, which helped me in completion of this project.

I also want to express my sincere gratitude towards the University of New Orleans for providing me a creative space for research.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>Abstract</td>
<td>viii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Literature Review</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Browser Fingerprinting</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 History</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2 Fingerprints</td>
<td>5</td>
</tr>
<tr>
<td>2.1.3 Canvas Fingerprints</td>
<td>6</td>
</tr>
<tr>
<td>2.1.4 Font Fingerprints</td>
<td>7</td>
</tr>
<tr>
<td>2.1.5 Audio Fingerprints</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Background and Related Work</td>
<td>8</td>
</tr>
<tr>
<td>3. System Description</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Web Audio API</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Main Audio Node Interfaces</td>
<td>11</td>
</tr>
<tr>
<td>3.2.1 AudioNode</td>
<td>11</td>
</tr>
<tr>
<td>3.2.2 BaseAudioContext</td>
<td>11</td>
</tr>
<tr>
<td>3.2.3 AudioContext</td>
<td>11</td>
</tr>
<tr>
<td>3.2.4 OfflineAudioContext</td>
<td>12</td>
</tr>
<tr>
<td>3.2.5 AudioBuffer</td>
<td>12</td>
</tr>
<tr>
<td>3.2.6 OscillatorNode</td>
<td>12</td>
</tr>
<tr>
<td>3.2.7 DynamicCompressor</td>
<td>12</td>
</tr>
<tr>
<td>3.2.8 AnalyserNode</td>
<td>13</td>
</tr>
<tr>
<td>3.2.9 GainNode</td>
<td>13</td>
</tr>
<tr>
<td>3.2.10 ScriptProcessor</td>
<td>13</td>
</tr>
<tr>
<td>3.2.11 ChannelMergerNode</td>
<td>13</td>
</tr>
<tr>
<td>3.2.12 AudioDestinationNode</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Digital Signal Processing</td>
<td>14</td>
</tr>
<tr>
<td>3.3.1 Frequency Modulation</td>
<td>14</td>
</tr>
<tr>
<td>3.3.2 Amplitude Modulation</td>
<td>15</td>
</tr>
<tr>
<td>3.4 Fingerprinting Methods</td>
<td>16</td>
</tr>
<tr>
<td>3.4.1 DynamicCompressor Method</td>
<td>17</td>
</tr>
</tbody>
</table>
List of Figures

FIGURE Page
2.1 Panopticlick Experiment Table ........................................................... 5
2.2 Canvas Fingerprint Example.............................................................. 7
3.1 Frequency Modulation Example ......................................................... 15
3.2 Amplitude Modulation Example ......................................................... 16
3.3 DynamicCompressor Full Buffer Method ............................................. 17
3.4 Generated Triangle Wave ................................................................. 18
3.5 OscillatorNode Method ................................................................... 19
3.6 Hybrid (DynamicCompressor + OscillatorNode) Method ......................... 19
3.7 Custom Signal Hybrid Method .......................................................... 20
3.8 Generated Custom Wave .................................................................. 21
3.9 Audio Source Hybrid Method ........................................................... 22
3.10 Generated Audio Source Wave .......................................................... 23
3.11 Channel Merge Hybrid Method ......................................................... 24
3.12 Generated Channel Merge Wave ....................................................... 24
3.13 Amplitude Modulation Hybrid Method .............................................. 25
3.14 Generated Amplitude Modulation Wave ............................................. 26
3.15 Frequency Modulation Hybrid Method .......................................... 27
3.16 Generated Frequency Modulation Wave .......................................... 27
3.17 System Diagram........................................................................... 29
3.18 Fingerprinting Website................................................................. 31
4.1 Bar graphs showing grouping of the number of fingerprints observed for a fingerprinting method cont. ................................................................. 34
4.1 Bar graphs showing grouping of the number of fingerprints observed for a fingerprinting method ................................................................. 35
4.2 The frequency of the highest occurring fingerprint value in 30 iterations .......... 36
4.3 An example of distinct fingerprint groups generated with network graph based approach ........................................................................................................... 39
4.4 Browser compatibility of Web Audio API across different browsers................. 41
4.5 Browser/OS combination of distinct fingerprint group for DynamicCompressor..... 42
4.6 Browser/OS combination of distinct fingerprint group for OscillatorNode ......... 43
4.7 Browser/OS combination of distinct fingerprint group for Hybrid ...................... 43
4.8 Browser/OS combination of distinct fingerprint group for CustomSignalHybrid ... 44
4.9 Browser/OS combination of distinct fingerprint group for AudioSourceHybrid....... 44
4.10 Browser/OS combination of distinct fingerprint group for ChannelMergeHybrid .... 45
4.11 Browser/OS combination of distinct fingerprint group for AmplitudeModulation-Hybrid ..................................................................................... 45
4.12 Browser/OS combination of distinct fingerprint group for FrequencyModulation-Hybrid ..................................................................................... 46

B.1 The frequency of the highest occurring fingerprint value in 30 iterations for individual audio fingerprinting methods contd. ................................. 66
B.1 The frequency of the highest occurring fingerprint value in 30 iterations for individual audio fingerprinting methods ......................................................... 67
### List of Tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Minimum number of appearances of the most popular fingerprint for a user in 30 iterations</td>
</tr>
<tr>
<td>4.2</td>
<td>Average time in seconds to compute each audio fingerprinting method</td>
</tr>
<tr>
<td>4.3</td>
<td>Number of distinct and unique fingerprints and their entropy and normalized entropy</td>
</tr>
<tr>
<td>4.4</td>
<td>Browser mapping</td>
</tr>
<tr>
<td>4.5</td>
<td>Number of User-Agents spanning on more than 1 diverse fingerprint group</td>
</tr>
</tbody>
</table>
Abstract

Browser fingerprinting presents a grave threat to privacy as it allows user tracking even in private browsing modes. Prior measurement studies on HTML5-based fingerprinting have been limited to Canvas and WebGL but not Web Audio APIs. We aim to fill this gap by conducting the first large-scale systematic study of web audio fingerprints and studying their stability as well as diversity properties. Using MTurk and social media platforms, we collected 8 different audio fingerprints from 694 users.

Firstly, we show that the audio fingerprints are unstable unlike other fingerprinting methods with some users having as many as 20 different fingerprints. Despite this, we show that audio fingerprinting can still be used as an effective fingerprinting vector as most fingerprints tend to repeat quite often. We devised a graph-based fingerprint matching mechanism to measure the diversity of audio fingerprints. Our results show that audio fingerprints are much less diverse with only 45 distinct fingerprints among 694 users.

Keywords: Web Audio Fingerprints, Browser Fingerprints, Web Audio API, Canvas Fingerprints, Font Fingerprints, User-Agents Fingerprints, Tracking, Web Security
1. Introduction

“Browser fingerprinting” is a method of tracking web browsers by their browser configuration and settings information that is visible to the websites, rather than traditional tracking methods such as IP addresses and unique cookies. Browser fingerprinting is both difficult to detect and extremely difficult to thwart. A great example of Browser fingerprinting is Canvas fingerprinting. Canvas fingerprinting is a browser fingerprinting technique that uses HTML5 Canvas feature to draw text or 3D figures on a canvas when a user visits a website. The subtle difference in rendering that varies between users is tokenized to serve as an unique identifier that allows the website to remember the visitor. Another popular browser fingerprinting technique is the WebGL fingerprint. The WebGL hash is generated by rendering a pattern or low high entropy image (Image that does not compress well) and then creating a hash from the resulting rendered pixels. That image is rendered to a frame buffer with a fixed size so changing screen resolution will not affect the resulting hash.

There have been several large scale studies on Canvas Fingerprints and WebGL Fingerprints and the defenses have been proposed as well [1, 2, 3, 4, 5]. However, one area of fingerprinting that has been neglected is audio fingerprinting. Discovered by Englehardt et al. [6] in 2016 while crawling through 1 million websites, audio fingerprinting is fairly new to the browser fingerprinting world. There have been no large scale studies on audio fingerprinting and we aim to fill this gap in our research.

Stability of a browser fingerprint technique plays a pivotal role in its efficiency. The main reason behind browser fingerprinting popularity is its ability to compute a distinct fingerprint for a user when repeated multiple times even in private/incognito browsing mode. A browser fingerprint must be stable, fast and diverse to be used for tracking. First research question we try to answer in our research is that, **RQ1 - are audio fingerprinting stable enough to be used as a fingerprinting vector?** Computation time of a fingerprinting technique must be attainable to not add any computational overhead to the browser. If a fingerprinting technique takes too much time then it is not
effective in a sense that a user might leave the website before the fingerprint computation completes. The second research question we try to find an answer is that, **RQ2 - do audio fingerprints compute in a reasonable time frame?**

Recent security research has improved the security in the browser and made them less fingerprintable. At the same time, there has been an increasing amount of research on fingerprintability of web browsers [4, 2, 7, 8]. To address this gap for audio fingerprinting we study the effectiveness and diversity of our collected audio fingerprints at large. The third research question we try to answer in this research is **RQ3 - how diverse are audio fingerprints?** We also compare our audio fingerprinting technique to known fingerprinting methods such as Canvas, Font, User Agent, etc to answer our fourth research question, **RQ4 - in comparison to known fingerprinting techniques and how serious and effective are our audio fingerprinting techniques?**

Englehardt et al. found two audio fingerprinting techniques being used by the advertising company [6]. They found that the most popular script is from Liverail\(^1\), present on 512 sites and other scripts were present on less than 6 other websites. An additional technique was used on the OpenWPM project\(^2\) which is essentially an additional work by Englehardt et al. However, there can be several approaches to computing the audio fingerprints because of the interfaces provided by the Web Audio API [9]. The fifth question we try to answer in our research is that, **RQ5 - are there any additional techniques that can be used for audio fingerprinting?** In our research, we use three techniques discovered before and five new audio fingerprinting configurations that we developed to audio fingerprint a user and answer this question.

In the absence of a large scale study, industry has been doing contradictory things. For example, Brave, a privacy concerned browser has been implementing audio fingerprinting protection [10]. This is problematic in a sense that it adds computational overhead and it is also affecting the quality of the audio without actually realising its necessity. On the other hand, the Web Audio API developers from World Wide Web Consortium (W3C) believe - "This merely allows deduction of information already readily available by easier means (User Agent string), such as "this is browser

\(^{1}\)https://www.liverail.com/
\(^{2}\)https://audiofingerprint.openwpm.com/
However, both of these issues need a proper research study to pass a judgement on the correctness. Therefore, we raise a final research question, **RQ6 - are audio fingerprinting defenses necessary?**.

At the heart of our project lies an audio fingerprinting website that collects audio fingerprints as well as uses FingerprintJS [11] to capture canvas and font fingerprints and stores them in a database. The detailed system description is explained in Section 3. Our analysis and results are explained in detail in Section 4. In summary, our contributions are as follows:

1. We conducted the first-large systematic study of Web Audio API-based fingerprinting vectors by employing 633 web users. We measured both the stability and diversity of 8 audio fingerprints by repeating them 30 times for each user.

2. We designed, implemented and analyzed the effectiveness of 5 new audio fingerprinting vectors that made use of FFTs of audio source files and modulated and modified custom waveforms.

3. We designed a graph-based fingerprint matching mechanism to analyze the diversity of partially stable audio fingerprints.

4. We compared the diversity of audio fingerprints with other well-known fingerprinting vectors such as Canvas, font and User-Agent headers to allow the browser developers to take informed decisions when developing privacy protection features.

---

[3]https://www.w3.org/TR/webaudio/#priv-sec
2. Literature Review

2.1 Browser Fingerprinting

In this section, we answer the basic information encompassing the browser fingerprinting history, usages, types, current methods on the web and give an exhaustive overview of the researches led in the area of browser fingerprinting (BF) with a synopsis of current procedures.

2.1.1 History

Browser fingerprints were first discovered by Mayer in 2009 [12]. He concluded that the browser features and the plugins are unique to each user and can be used for tracking. He looked at the differences in browsing environments that can be exploited by the remote serve to identify the users. Mayer saw the "quirkiness" that comes with the hardware, operating system and the browser configuration can be used to track users on the internet. In his study 96.23% (1278 of 1328) participants could be uniquely identified.

Following Mayers in 2010, Eckersley conducted the Panopticlick Experiment from Electronic Frontier Foundation (EFF) [13]. The script calculates a uniqueness score based on the data that the web browser reveals during connections. Tests with Google Chrome 5, Opera 10.5 pre-alpha, Internet Explorer 8 and Mozilla Firefox 3.6 revealed that all four web browsers contained unique bits that could be used to identify them. This can be attributed to the web browser fingerprint database of the service as it contains more than 470,000 fingerprints. He also noted that the fingerprints are likely to reduce the uniqueness with the increase in users. The test results are displayed as a table on the screen as shown in Figure 2.1. Each bit of information identified by the test is displayed in its own row in the table revealing the bits of identifying information and how unique it is when compared to the whole database. The higher the number the worse. The experiment was able to identify 84% of the participants.
2.1.2 Fingerprints

The popularity of the web has seen an increase in tracking for statistical, marketing or commercial purposes. User tracking or web tracking allows you to track an Internet user anonymously and by various means while browsing your website. Browser fingerprinting (BFPs) is an incredibly accurate method of identifying unique browsers and tracking online activity. Browser fingerprinting has arrived at a condition of development where it is currently utilized by numerous organizations for various purposes. Several studies have demonstrated the development of this procedure along the years with both the disclosure of new properties and its spread on the web [6, 8, 4, 2]. Unlike cookies and local storage, fingerprint stays the same in incognito/private mode and even when browser data is purged.

**Fingerprint** is formally defined as a set of information elements defining a device or an instance of an application and **Fingerprinting** as the process by which an observer or attacker uniquely identifies a device or instance of an application based on a set of multiple information by RFC6973 [14]. Browser fingerprints (BFPs) have been defined in many ways on previous research [3, 15, 7]. In general, we can define the Browser Fingerprinting as a powerful method that the website utilizes to extract the device features to generate a single unique identifier which is then used to identify the user with a high probability. Based on the current trend and researches we discuss the primary fingerprinting mechanisms that are wild in the web in the section below.
2.1.3 Canvas Fingerprints

The HTML Canvas allows web applications to draw graphics in real time, with functions to support drawing shapes, arcs, and text to a custom canvas element[6]. This was demonstrated on the "Pixel Perfect" project by Mower et. al in 2012 [1]. The minute differences in rendering, smoothing, anti-aliasing and other features of the user devices can contribute to drawing an image differently on each machine allowing the exploitation of these differences to generate a unique fingerprint for a user. Canvas fingerprinting works by exploiting the HTML5 canvas element: when a user visits a website their browser is instructed to “draw” a hidden line of text or 3D graphic that is then rendered into a single digital token, a potentially unique identifier to track users without any actual identifier persistence on the machine. The technique is based on the fact that the same canvas image may be rendered differently on different computers. This happens for several reasons. At the image format level – web browsers use different image processing engines, image export options, compression level, the final images may get different checksum even if they are pixel-identical. At the system level – operating systems have different fonts, they use different algorithms and settings for anti-aliasing and sub-pixel rendering.

In plain English, what this means is that the HTML5 canvas element generates certain data, such as the font size and active background color settings of the visitor’s browser, on a website. This information serves as the unique fingerprint of every visitor. An example of the hidden canvas rendered on a user’s machine is shown in Figure 2.2
2.1.4 Font Fingerprints

Font list of a browser can also be utilized to fingerprint the users. Font fingerprinting techniques are based on measuring the onscreen dimensions of HTML elements filled with text pieces or single Unicode glyph. Font rendering in web browsers is affected by many factors, and these measurements may vary slightly. Font Enumeration attack is a brute-force method that tries different fonts from a sizable font-family dictionary. If the rendered element’s size differs from the default values, it means that the substituted font is present in the system. Shown below in code block 2.1 is an example of the font list of a user that is used to generate a unique identifier for that particular user.

```plaintext
```
2.1.5 Audio Fingerprints

Audio fingerprinting (AFPs) is a fingerprinting technique that is moderately new with regards to browser fingerprinting techniques. By using the Web Audio API [9] in modern browsers, one can misuse the subtle differences in the rendering of a fixed audio waveform, for instance a sine or triangle wave, to construct a unique fingerprint of a device. The Web Audio API gives an interface to create/process an audio by linking multiple modules like oscillators, compressor, and filters together to generate a very specific output. A continuous signal is passed through this pipeline to convert the continuous signal to discrete so that the computer can process the audio very easily in the form of a block called frames [8]. Each frame is composed of samples that represent the value of the audio stream at a specific point in time. While utilizing the Web Audio API to generate fingerprints, the code doesn’t gather sound played or recorded by your machine rather an identifier is generated by the code based on the frame values that varies based on a machine’s sound stack.

2.2 Background and Related Work

Audio Fingerprinting (AFPs) technique was first discovered by Englehardt et al. while crawling the web looking for trackers. AudioContext fingerprinting is one of the latest additions in a fingerprinter’s toolbox. They found scripts that process an audio signal generated with an OscillatorNode to fingerprint devices. The authors add that the fingerprinting process is similar to what is done with canvas fingerprinting as processed signals will present differences due to the software and hardware stack of the device. The relative novelty of this technique explains that scripts using this API were only found on a very small number of websites [6]. Researchers found that the first technique using AnalyzerNode to extract FFT to build fingerprints was not consistent and the results were flaky. However, the second technique with DynamicCompressor with full buffer was more promising for repeated visitors but they also saw collision between fingerprints and they were not particularly identifying. Their 1M web census saw the prevalence
of audio fingerprinting script in 5.6% of the websites. The detailed working mechanism of these methods are explained in Chapter 3

Laperdrix et al. collected 19,467 audio fingerprints from AmIUnique.org to mitigate the risk of audio fingerprinting by ‘breaking’ the stability of fingerprint over time [8]. They introduced very small noise directly into the audio processing routines of the browser so that tests using any number of AudioContext modules are all impacted. They operate AudioBuffers of the AudioNodeEngine as they contain the frames of the processed audio and decrease the volume of processed buffers by a factor ranging between 0.000 and 0.001. The changes made it impossible to detect modified sections from unmodified ones and produce a different hash as the audio routine for each browser session.

Queiroz et al. showed that the audio fingerprinting is capable of identifying the device’s class, based on features like device’s type, web browser’s version and rendering engine [16]. They also studied the stability of the two interfaces i.e AudioContext and OfflineAudioContext and chose to use OfflineAudioContext fingerprint users using several audio configurations. The researchers concluded that Web Audio API is capable of providing information about the devices only when employed with other Browser Fingerprinting techniques.

These works have shown how new web technologies such as Canvas API, Web GL API, Font, Audio and browser extensions are increasing the diversity of web browsers and making the users more prone to cookie-less tracking. While these research works have studied the privacy implications of browser fingerprinting, the security implications of these advanced fingerprinting techniques have not been studied well. In particular, no systematic research has been done to see how stable and diverse the audio fingerprinting can be. Our research primarily focuses on addressing this gap.
3. System Description

3.1 Web Audio API

The Web Audio API provides a powerful and versatile system for controlling audio on the Web, allowing developers to choose audio sources, add effects to audio, create audio visualizations, apply spatial effects (such as panning) and much more [9].

The Web Audio API is based on the concept of modular routing, which has its roots in analog synthesizers. There are audio sources, such as microphones, oscillators, and audio files, which provide some kind of audio signal. This API manages operations inside an Audio Context. Audio operations are performed with audio nodes, which are linked together to form an Audio Routing Graph. Multiple sources are supported within a single Audio Context. This modular design is highly flexible, allowing the creation of complex audio designs. They typically start with one or more sources. Node outputs can be linked to the inputs of others creating chains or webs of audio streams.[17] The APIs have been designed with a wide variety of use cases in mind. Ideally, it should be able to support any use case which could reasonably be implemented with an optimized C++ engine controlled via script and run in a browser. That said, modern desktop audio software can have very advanced capabilities, some of which would be difficult or impossible to build with this system. [18].

Those audio signals can be connected to other audio nodes which perform operations with that signal. These nodes include the Gain Node, which can raise or lower the volume of a signal; Filter Nodes, which changes how the signal sounds; and an Analyser Node, which provides real-time information about a signal which can be used to render a visualization.

Once the sound has been effected and is ready for output, it can be linked to the input of a AudioContext.destination (computer’s speakers), which sends the sound to the speakers. Note that this last connection is only required if you need the audio to be heard. [19]

A typical flow for Web Audio could look something like this:
1. Create audio context (AudioContext or OfflineAudioContext)

2. Create sources inside the context (e.g. `<audio>`, oscillator, streams)

3. Create effects nodes (e.g. reverb, flanger, panner, compression)

4. Choose a destination for the audio (e.g. speakers)

5. Connect the sources to the effects, and the effects to the destination

### 3.2 Main Audio Node Interfaces

In this section we discuss the different audio interfaces provided by the Web Audio API. These audio nodes are very important and are the base of our audio fingerprinting configurations. These nodes are used in combination to generate an audio fingerprinting and each of the nodes have their significance and a role to play in our fingerprinting methods.

#### 3.2.1 AudioNode

The AudioNode interface is a generic interface for representing an audio processing module. Each AudioNode has inputs and outputs, and multiple audio nodes are connected to build a processing graph. AudioNode.connect() method present in the AudioNode interface allows to connect the output of this node to be input into another node, either as audio data or as the value of an AudioParam [9].

#### 3.2.2 BaseAudioContext

The BaseAudioContext interface of the Web Audio API acts as a base definition for online and offline audio-processing graphs, as represented by AudioContext and OfflineAudioContext respectively. BaseAudioContext is not used directly, rather its used via one of these two inheriting interfaces.

#### 3.2.3 AudioContext

The AudioContext interface represents an audio-processing graph built from audio modules linked together, each represented by an AudioNode. An audio context controls both the creation
of the nodes it contains and the execution of the audio processing, or decoding [9].

3.2.4 OfflineAudioContext

The OfflineAudioContext interface is an AudioContext interface representing an audio-processing graph built from linked together AudioNodes. In contrast with a standard AudioContext, an OfflineAudioContext doesn’t render the audio to the device hardware; instead, it generates it, as fast as it can, and outputs the result to an AudioBuffer [9].

3.2.5 AudioBuffer

The AudioBuffer interface represents a short audio asset residing in memory, created from an audio file using the AudioContext.decodeAudioData() method, or from raw data using AudioContext.createBuffer(). Once put into an AudioBuffer, the audio can then be played by being passed into an AudioBufferSourceNode [9].

3.2.6 OscillatorNode

The OscillatorNode interface represents a periodic waveform, such as a sine wave. It is an AudioScheduledSourceNode audio-processing module that causes a specified frequency of a given wave to be created—in effect, a constant tone. An OscillatorNode is created using the BaseAudioContext.createOscillator() method. It always has exactly one output and no inputs [9].

3.2.7 DynamicCompressor

The DynamicsCompressorNode interface provides a compression effect, which lowers the volume of the loudest parts of the signal in order to help prevent clipping and distortion that can occur when multiple sounds are played and multiplexed together at once. This is often used in musical production and game audio. DynamicsCompressorNode is an AudioNode that has exactly one input and one output; it is created using the BaseAudioContext.createDynamicsCompressor() method [9].
3.2.8 AnalyserNode

The AnalyserNode interface represents a node able to provide real-time frequency and time-domain analysis information. It is an AudioNode that passes the audio stream unchanged from the input to the output, but allows you to take the generated data, process it, and create audio visualizations. An AnalyserNode has exactly one input and one output. The node works even if the output is not connected [9].

3.2.9 GainNode

The GainNode interface represents a change in volume. It is an AudioNode audio-processing module that causes a given gain to be applied to the input data before its propagation to the output. A GainNode always has exactly one input and one output, both with the same number of channels. The gain is a unit-less value, changing with time, that is multiplied to each corresponding sample of all input channels. If modified, the new gain is instantly applied, causing unaesthetic ‘clicks’ in the resulting audio [9].

3.2.10 ScriptProcessor

The createScriptProcessor() method of the BaseAudioContext interface creates a ScriptProcessorNode used for direct audio processing using JavaScript. This module is interconnected to two AudioBuffers: 1) Input data 2) Output data. This interface consists of an event handler function; onaudioprocess that takes the associated audioProcessingEvent and uses it to loop through each channel of the input buffer, and each sample in each channel and is triggered when the incoming AudioBuffer is ready to be processed.

3.2.11 ChannelMergerNode

The ChannelMergerNode interface, reunites different mono inputs into a single output. Each input is used to fill a channel of the output. This is useful for accessing each channel separately, e.g. for performing channel mixing where gain must be separately controlled on each channel. If ChannelMergerNode has one single output, but as many inputs as there are chan-
nels to merge; the number of inputs is defined as a parameter of its constructor and the call to
`AudioContext.createChannelMerger()`. In the case that no value is given, it will de-
fault to 6 [9].

### 3.2.12 AudioDestinationNode

The destination property of the `BaseAudioContext` interface returns an `AudioDestinationNode` representing the final destination of all audio in the context. It often represents an actual audio-rendering device such as your device’s speakers [9].

### 3.3 Digital Signal Processing

In this section we discuss two very popular methods of signal processing in Digital Signal Pro-
cessing. The knowledge of working mechanisms is vital to understand both of these processes. These methods are used in our research to generate Amplitude Modulation Hybrid and Frequency Modulation Hybrid Fingerprints. The exact details about the modulating and carrier signals are discussed in the Section 3.4.7 and 3.4.8 respectively.

#### 3.3.1 Frequency Modulation

Frequency modulation (FM) is the encoding of information in a carrier wave by varying the
instantaneous frequency of the wave. The technology is used in telecommunications, radio broad-
casting, signal processing, and computing. Frequency modulation is widely used for FM radio
broadcasting. It is also used in telemetry, radar, seismic prospecting, and monitoring newborns
for seizure, two-way radio systems, sound synthesis, magnetic tape-recording systems and some
video-transmission systems.

To generate a frequency modulated signal, the frequency of the radio carrier is changed in line
with the amplitude of the incoming radio signal. When the radio signal is modulated onto the radio
frequency carrier, the new radio frequency signal moves up and down in frequency. The amount
by which the signal moves up and down is important. It is known as the deviation and is normally
quoted as the number of kilohertz deviations. As an example the signal may have a deviation of
plus and minus 3 kHz, i.e. ±3 kHz. In this case the carrier is made to move up and down by 3 kHz
[20]. From Figure 3.1, it can be seen that the envelope of the signal follows the frequency of the modulating signal.

![Modulating signal vs Radio frequency signal]

Figure 3.1: Frequency Modulation Example

### 3.3.2 Amplitude Modulation

Amplitude modulation (AM) is a modulation technique used in electronic communication, most commonly for transmitting messages with a radio carrier wave. In amplitude modulation, the amplitude (signal strength) of the carrier wave is varied in proportion to that of the modulating signal, such as a radio signal. Although one of the earliest used forms of modulation it is still used today, mainly for long, medium and short wave broadcasting and for some aeronautical point to point communications.

In order that a radio signal can carry audio or other information for broadcasting or for two way
radio communication, it must be modulated or changed in some way. Although there are a number of ways in which a radio signal may be modulated, one of the easiest is to change its amplitude in line with variations of the sound. In this way the amplitude of the radio frequency signal varies in line with the instantaneous value of the intensity of the modulation. This means that the radio frequency signal has a representation of the sound wave superimposed in it [21]. From Figure 3.2, it can be seen that the envelope of the signal follows the contours of the modulating signal.

3.4 Fingerprinting Methods

In this section we discuss 8 fingerprinting methods used in this research to get the unique identifier of a user. First, we discuss the two methods namely DynamicCompressor and OscialltorNode that were found by Englehardt et al. during their 1-Million-Site Measurement and Analysis [6]. An additional technique used on the OpenWPM project 1. Then, we discuss a more advanced methods i.e. Custom Signal Hybrid, Audio Source Hybrid, Channel Merge Hybrid, Amplitude Modulation Hybrid and Frequency Modulation Hybrid that we introduced to audio fingerprint the

---

1https://audiofingerprint.openwpm.com/
users and try to answer the research question RQ5. The code snippet of all the audio fingerprinting methods are shown in the Appendix A

3.4.1 DynamicCompressor Method

![DynamicCompressor with full buffer method](image)

Figure 3.3: DynamicCompressor Full Buffer Method

Figure 3.3 shows the utilization of the OfflineAudioContext interface to audio fingerprint the sound stack of a device. Initially, a triangle wave is produced as shown in Figure 3.4 utilizing an OscillatorNode. The signal is then connected to DynamicsCompressorNode, to increment differences in processed audio between devices. The yield of this compressor is passed to the buffer of an OfflineAudioContext. Hashes generated from the values from the full buffer is used to audio fingerprint a user. This method was found wild in the web by trackers during their 1-Million-Site Measurement and Analysis [6].
3.4.2 OscillatorNode Method

Figure 3.5. shows the utilization of the AudioContext interface to audio fingerprint the sound stack of a device. Initially, a triangle wave is created as shown in Figure 3.4 using an OscillatorNode. This signal is then passed to an AnalyserNode and a ScriptProcessorNode. At last, the single goes through a GainNode whose gain is set zero to mute any output to AudioContext destination (For example, computer speakers). The AnalyserNode gives admittance to a Fast Fourier Transform (FFT) of the audio signal, which is caught utilizing the onaudioprocess event handler added by the ScriptProcessorNode. The subsequent FFT is taken care of into a hash and utilized as a unique audio fingerprint.

3.4.3 Hybrid Method

Figure 3.6. shows the combination of OscillatorNode and DynamicCompressor methods to generate a unique audio fingerprint of a user using AudioContext interface. A triangle wave is generated as shown in Figure 3.4 using OscillatorNode which is passed through
Figure 3.5: OscillatorNode Method

Figure 3.6: Hybrid (DynamicCompressor + OscillatorNode) Method
the DynamicsCompressorNode. This signal is then passed to an AnalyserNode and a ScriptProcessorNode. Similar to the OscillatorNode method the signal then goes through GainNode that is set to zero. We use AnalyserNode to FFT the audio signal, that is captured by onaudioprocess event handler added by the ScriptProcessorNode. The FFT is then hashed using MD5 which acts as an audio fingerprint.

3.4.4 Custom Signal Hybrid Method

![Custom Signal Hybrid Method](image)

Figure 3.7: Custom Signal Hybrid Method

Custom Signal Hybrid Method only differs from Hybrid method in terms of the signal. Instead of using a standard signal (Sine, Triangle, Sawtooth, Square) offered by the Web Audio API, we generate a periodic signal by defining the real and imaginary values as shown in Appendix A.4. Figure 3.7 shows the setup used to generate a unique audio fingerprint of a user using AudioContext interface for custom signal. A custom periodic wave is generated as shown in Figure 3.8 using `AudioContext.createPeriodicWave()` method which is passed through...
Figure 3.8: Generated Custom Wave

the DynamicsCompressorNode. This signal is then passed to an AnalyserNode and a ScriptProcessorNode as before. Similar as before we use AnalyserNode to FFT audio signal, that is captured by onaudioprocess event handler added by the ScriptProcessorNode. The FFT is then hashed using MD5 which acts as an audio fingerprint.
3.4.5 Audio Source Hybrid Method

Audio Source Hybrid is yet another new method that we introduced for audio fingerprinting. This method varies from all of the methods in terms of how the signal is generated. In this method, we used a 40 seconds audio clip [22] that was trimmed down to 5 seconds to make it shorter and suitable for fingerprinting. The audio clip is first decoded using `AudioContext.decodeAudioData()` method to get the audio buffer which is then utilized by `AudioContext.createBufferSource()` to generate the signal as shown in Figure 3.10. This signal is then passed on to the rest of the audio nodes as before. Similarly, the `AnalyserNode` is used to FFT audio signal, that is captured by `onaudioprocess` event handler added by the `ScriptProcessorNode` and then hashed using MD5 to generate an audio finger-
Figure 3.10: Generated Audio Source Wave

print. Figure 3.9 shows the method with different audio nodes used to generate an identifier.

3.4.6 Channel Merge Hybrid Method

Channel Merge Hybrid is another addition to audio fingerprinting methods which was created during this research. The idea is to generate multiple signals and merge them using a ChannelMergeNode. In this method we create four OscillatorNode with different frequencies and different signal types and merge them together to generate a single waveform. Specifically we create Sine OscillatorNode with frequency 440 HZ, Triangle OscillatorNode with frequency 10000 HZ, Square OscillatorNode with frequency 1880 HZ and Sawtooth OscillatorNode with frequency 22000 HZ. All of these OscillatorNode are then connected to the ChannelMergeNode. The output of the ChannelMergeNode is connected to the DynamicCompressorNode as before. The rest of the Audio Nodes remain the same as previous methods with FFT being captured and hashed through the onaudioprocess event handler added by the ScriptProcessorNode. Figure 3.11 shows the exact implementation of the Channel Merge Hybrid method and Figure 3.12 shows the output of the ChannelMergeNode.
Figure 3.11: Channel Merge Hybrid Method

Figure 3.12: Generated Channel Merge Wave
3.4.7 Amplitude Modulation Hybrid Method

Amplitude Modulation Hybrid is yet another contribution to the fingerprinting method introduced during this research. The major focus here is also on the signal generation. The rest of the audio nodes remains the same as previous methods where the FFT is being captured and hashed through the onaudioprocess event handler added by the ScriptProcessorNode to generate an audio fingerprint. We introduce two modulating waves: 1) Triangle, 440HZ 2) Square, 18HZ that are connected to their respective GainNode’s. The carrier wave is a Sine wave oscillating at 10000HZ that is connected to its own GainNode. The output of the both Modulating GainNodes are connected to carrier GainNode. This output is then connected to DynamicCompressor and the rest of the stack remains the same as well as the process. Figure 3.13 shows the exact implementation of the Amplitude Modulation Hybrid method and Figure 3.14 shows the output of the carrier GainNode.
3.4.8 Frequency Modulation Hybrid Method

Frequency Modulation Hybrid is another add to fingerprinting method introduced during this research. This method is analogous to Amplitude Modulation Hybrid Method discussed to 3.4.7. Similar to Amplitude Modulation Hybrid Method we introduce two modulating waves: 1) Triangle, 440HZ 2) Square, 18HZ that are connected to their respective GainNode’s whose gain is set to 60 and 30 respectively. The carrier wave is a Sine wave oscillating at 10000HZ. The output of the both Modulating GainNode’s are connected to carrier frequency. This output is then connected to DynamicCompressor. The rest of the audio nodes remains the same as previous methods where the FFT is being captured and hashed through the onaudioprocess event handler added by the ScriptProcessorNode to generate an audio fingerprint. Figure 3.15 shows the exact implementation of the Frequency Modulation Hybrid method and Figure 3.16 shows the output of the carrier wave.
Figure 3.15: Frequency Modulation Hybrid Method

Figure 3.16: Generated Frequency Modulation Wave
3.5 FingerprintJS2

FingerprintJS2 is one of the most popular browser fingerprinting library that collects 27 different attributes (at the time of our collection) of browser and hashes them into a single fingerprint called "vistorId" [11]. We primarily used FingerprintJS2 to obtain Canvas and Font Fingerprints. We also modified FingerprintJS’s font fingerprinting code to expand the list of font from 51 to 1094 [23]. The expanded list of fonts is listed in Appendix A.9.

3.6 IP Information

In addition to the FingerprintJS2 fingerprints we also collected IP information of each user that visited our fingerprinting site. We tried several third-party APIs to get user IP information but most of the third-parties were blocked by all of the mainstream browsers and some of the APIs required API-keys and had limits on the number of requests made. However, we were able to successfully get the IP information using a CloudFlare API [24]. The user-agent collected from this information was used for the further analysis of data in determining the type of device, type of browser and the browser versions used.

3.7 User Agent Information

We also used the Bowser package to collect the User Agent Information. The library is made to help to detect what browser the user has and gives a convenient API to filter the users depending on their browsers [25].
3.8 System Overview

Figure 3.17 shows our system overview of our audio fingerprinting project. The center of the system consists of the audio fingerprinting script written in JavaScript (Angular 11.0.4) that utilizes the Web Audio API to generate the fingerprints. The fingerprinting script also uses the popular fingerprinting FingerprintJS2 to get several components (28) used in browser fingerprinting. The IP information is collected using a cloudflare API service [24] and the User Agent is collected using Bowser [25]. The collected data is stored in the form of a JSON object in Google Firestore\(^2\). A cookie is set to each user once the user lands on our fingerprinting page. This cookie acts as an unique identifier in the database.

\(^2\)https://firebase.google.com/docs/firestore
Our script generates 8 different types of audio fingerprints discussed above. For a particular user, all the fingerprints (8) are generated 30 times and saved to the database in each iteration. The code-base resides in GitHub \(^3\) and the project has around 10000 lines of code.

### 3.9 Hosting Stack

Our entire project is hosted using Firebase Hosting \(^4\). The web page is hosted as a single page application. The hosting and the firestore is free till 1GiB Stored data, 10 GiB/month Network egress, 20K/day Document writes, 50K/day Document reads and Document deletes 20K/day.

### 3.10 User Study

We conducted our user study using Amazon’s Mechanical Turk \([26]\). Our aim was to get 5000 users for this study. We instructed each user to visit our website \(^5\) and click on the start button and stay on the page until they see a UserID/SurveyID. As soon as the user lands on our website, a unique cookie is set for that user that acts as an identifier in the database. We also instructed each user to submit their UserID/SurveyID once they are done to the MTurk so we can keep a proper track on the progress of our study. For any user the 8 audio fingerprints are generated 30 times and duration of each fingerprinting method is also saved in the database to analyse the average timings.

We had issues with the Apple iOS devices (iPhone, iPad, iPod) not functioning properly with the Web Audio API during testing. These devices were filtered as soon as the user lands on the website and were shown a message saying "Your iOS device is incompatible. Please try on any other Desktops/Laptops/Android Devices". However, the macOS devices(Macbooks/iMACs) had no issues with our setup. Furthermore, the AudioContext requires user interaction with the website to instantiate. This restriction made the fingerprinting page unusable without a user clicking on a button to start the fingerprinting. A solution to this problem was to add a "START" button that starts the fingerprinting for that particular user. The website was designed in an interactive way to inform the user of the progress of the fingerprinting with a progress bar and a message.

---

\(^3\)https://github.com/shekharchalise/audio-audiofingerprint  
\(^4\)https://firebase.google.com/docs/hosting  
\(^5\)https://audiocontextfingerprint.web.app/
Figure 3.18: Fingerprinting Website

saying "Please click on start and wait till the progress bar indicates 100%" and "Almost done!!!
Please don’t close the window" to avoid any mid termination of the website by the user that stops
our script from running and collecting the data. Figure 3.18 shows the setup of our website that
was used for the user study.
4. Evaluation

In this section we analyse the data collected during the user study. We also discuss the stability and diversity of the fingerprints. We collected data from 868 users however, after cleaning the data we only used data from 694 users. We discarded the data that were from the same IP address, ones that did not consist of the fingerprints, and the data consisting more than 30 iterations of data which might be caused due to various reasons like browser incompatibility, mid-termination of the fingerprinting website and multiple entry by the same user.

4.1 Stability Analysis

Stability is an important aspect of any fingerprinting method. Once a user fingerprint is collected, it must be reproducible for that user in a subsequent visit. Which means that for any fingerprinting method to increase its effect, it must be stable enough to be generated in a repeated fashion.

4.1.1 Statistical Analysis

We took a statistical approach to study the stability of the fingerprints. We wanted to run our 8 audio fingerprinting methods under 1 minute for each user. During the testing phase we observed that all of our audio fingerprints could be generated 30 times for each user under 1 minute. This made it possible to study the stability aspect as the fingerprints need to be reproduced in each iteration. To study the stability we grouped fingerprints based on the fingerprinting method used and the total number of fingerprints observed in 30 iterations for a particular user. For instance, a user with 2 different fingerprints in 30 iterations for OscillatorNode method falls into group 2, another user with 3 fingerprints in 30 iterations falls into group 3, and so on.

Ideally, a fingerprinting method should only yield one fingerprint even if it is run multiple times. But, the high instability of the computer hardware and software (Web Audio API), makes it not achievable in case of audio fingerprinting. Figure 4.1(a) - Figure 4.1(h) shows the bar graph
of the users with respect to the number of fingerprints observed in 30 iterations. The red curve indicates the percentage of users in each group of fingerprints observed. Figure 4.1(a) shows that 100% of the users have a total of one fingerprint in 30 iterations for the DynamicCompressor method making it the most stable method. Whereas, Figure 4.1(b) tells a completely different story. Although 71.46% users (496) had a single fingerprint which makes it less stable than DynamicCompressor, the remaining users did not generate random fingerprints and the fingerprints were repeating themselves in 30 iterations. For example, there were 9 users with 7 fingerprints in 30 iterations as shown in Figure 4.1(b) which shows the OscillatorNode method was stable to generate audio fingerprints where more than 80% of the users have less than or equal to 7 fingerprints in 30 iterations.

The stability was lowest for Figure 4.1(h), which observed less fingerprints in each group compared to the other methods. However, even for the least stable method, more than 80% of the users have less than or equal to 7 fingerprints in 30 iterations. The extreme number here is 24 for 1 user but not 30 so there is repetition of fingerprints for every user. There exists no column representing a group of size 30 for any audio fingerprinting method; this means that for every user and every fingerprinting vector, there were some fingerprint repetitions thus indicating stability. Results are similar for the reminder of our audio fingerprinting methods. Figure 4.1(f) shows the grouping for 610 users because some of the users were not able to generate the AudioSourceHybrid fingerprint due to its complex nature. This clearly answers our first research question on stability RQ1.
Figure 4.1: Bar graphs showing grouping of the number of fingerprints observed for a fingerprinting method cont.
4.1.2 Cumulative Distribution Function based approach

We used Cumulative Distribution Function (CDF) [27] to analyse the stability of our fingerprints as shown in Figure 4.2 to solidify our claim on stability. The popularity is the percentage of highest occurring fingerprint value in 30 iterations for each of the fingerprinting methods. Table 4.1 clearly shows that the fingerprints are stable considering the fact that the appearance of the most popular fingerprint occurs at least 2 times in 30 iterations for the AudioSource method. This means that even in the worst case the audio fingerprint repeated. Results are more promising for other fingerprinting methods where DynamicCompressor shows the best result with the same value occurring 100% of the times in 30 iterations. The graph shows that the OscillatorNode method shows that more than 80% of the users observed at least 25 repetitions (83.33%) in 30 iterations indicating stability. Appendix B.1 shows individual CDF plots for all the audio fingerprinting methods.
Figure 4.2: The frequency of the highest occurring fingerprint value in 30 iterations

<table>
<thead>
<tr>
<th>Fingerprinting Methods</th>
<th>Minimum number of appearances in 30 iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicCompressor</td>
<td>30</td>
</tr>
<tr>
<td>OscillatorNode</td>
<td>5</td>
</tr>
<tr>
<td>Hybrid</td>
<td>4</td>
</tr>
<tr>
<td>CustomSignalHybrid</td>
<td>5</td>
</tr>
<tr>
<td>AudioSourceHybrid</td>
<td>2</td>
</tr>
<tr>
<td>ChannelMergeHybrid</td>
<td>4</td>
</tr>
<tr>
<td>AmplitudeModulationHybrid</td>
<td>3</td>
</tr>
<tr>
<td>FrequencyModulationHybrid</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.1: Minimum number of appearances of the most popular fingerprint for a user in 30 iterations
4.2 Time Analysis

Time and computation is another aspect that is very important for any fingerprinting method. The computational overhead increases the time taken for a fingerprinting method to complete and raise suspicion. Therefore, the fingerprinting method must be computationally efficient to be generated in a certain time frame. In our research we wanted to study the timing stability of our audio fingerprinting methods. Our website collected time taken by each fingerprinting method for each iteration. A particular user running our script will have 30 different timings for each audio fingerprinting method. We first calculated the average time to generate each fingerprint by averaging the time taken to complete 30 iterations for each user for different audio fingerprinting methods. Then that average time is divided by the total users involved in the user study to generate average time to compute each fingerprint once. Table 4.2. shows the average time to generate 8 fingerprints is 1.275 seconds which is analysed with the data from 694 users. The highest time taken is 0.648 seconds by the DynamicCompressor method which is probably due its nature of completing the entire buffer rendering that happens in real time. This proves that our entire audio fingerprinting stack is computationally efficient and hence, answers our research question on timing stability RQ2 and further reinforces our claim on stability.

4.3 Diversity Analysis

The first defense to mitigate browser fingerprinting is to decrease the diversity of devices so that real fingerprints are hidden in noise. Diversity that is part of the device fingerprinting, is central to any browser fingerprinting technique. More diverse the fingerprints more easily identifiable the users and more efficient the technique. In this section we compare our audio fingerprinting methods with existing browser fingerprinting methods i.e. Canvas, Font and User-Agent Fingerprinting and try to answer our third research question.
### Table 4.2: Average time in seconds to compute each audio fingerprinting method

<table>
<thead>
<tr>
<th>Fingerprinting Methods</th>
<th>Avg Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicCompressor</td>
<td>0.648</td>
</tr>
<tr>
<td>OscillatorNode</td>
<td>0.106</td>
</tr>
<tr>
<td>Hybrid</td>
<td>0.105</td>
</tr>
<tr>
<td>CustomSignalHybrid</td>
<td>0.106</td>
</tr>
<tr>
<td>AudioSourceHybrid</td>
<td>0.074</td>
</tr>
<tr>
<td>ChannelMergeHybrid</td>
<td>0.103</td>
</tr>
<tr>
<td>AmplitudeModulationHybrid</td>
<td>0.068</td>
</tr>
<tr>
<td>FrequencyModulationHybrid</td>
<td>0.065</td>
</tr>
<tr>
<td><strong>CombinedFingerprint</strong></td>
<td><strong>1.275</strong></td>
</tr>
</tbody>
</table>

4.3.1 Network Graph based approach

As mentioned in Section 4.1.1 for a particular user there may be more than one fingerprint for a fingerprinting method in 30 iterations. For example for a user running our script can have different OscillatorNode fingerprints in different iterations. Unlike Canvas fingerprint where you run a code you get the same result in every iteration, we noticed that although stable, each user would get multiple fingerprints. To account for this instability we choose a network graph based approach using connected nodes to analyse the diversity.

Due to the unstable nature of the audio fingerprints it was not straightforward to do a statistical data analysis to determine the diversity. Our network graph based approach is based on creating a connected node graph between a fingerprint and its associated user for a particular fingerprinting method. To achieve this, we took all the fingerprints for a fingerprinting method and created a map between a fingerprint and a userID. A network graph was created with this data which had two nodes i.e fingerprints and userIDs. Figure 4.3 shows an example of distinct fingerprints group (connected components (CC)) for the collected data. We then calculated the number of distinct, unique, entropy, and normalized entropy of fingerprints for each fingerprinting method using the distinct fingerprint groups of this graph with 694 users as shown in Table 4.3. The number of
unique combined fingerprints is calculated by getting the union of all the unique fingerprints for each audio fingerprinting method. In comparison with canvas and font fingerprinting methods that shows 100 unique out of 170 and 232 unique out of 282 distinct users respectively, the best audio fingerprinting method being Hybrid shows only 21 unique out of 45 distinct users. If we compare our audio fingerprints with a previous work on Canvas fingerprinting by Gómez-Boix et al., which had 2,067,942 users with 78,037 distinct canvas fingerprints out of which 65,787 were unique [5], it does not yield a very good result in terms of unique and distinct audio fingerprints. Even with 694 users the best fingerprinting method being Hybrid could only produce 21 unique fingerprints out of 45 distinct users showing that although stable, audio fingerprints are not diverse enough. This answers our research question number four RQ4, only half way through with a comparison with canvas and font fingerprinting. To fully answer our question we compare the audio fingerprints with User-Agent in the Section 4.3.2.
<table>
<thead>
<tr>
<th>Fingerprinting Methods</th>
<th># of Users</th>
<th>Distinct</th>
<th>Unique</th>
<th>Entropy</th>
<th>Normalized Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canvas</td>
<td>694</td>
<td>170</td>
<td>100</td>
<td>5.718</td>
<td>0.605</td>
</tr>
<tr>
<td>Font</td>
<td>694</td>
<td>282</td>
<td>232</td>
<td>6.553</td>
<td>0.694</td>
</tr>
<tr>
<td>DynamicCompressor</td>
<td>694</td>
<td>36</td>
<td>19</td>
<td>2.460</td>
<td>0.261</td>
</tr>
<tr>
<td>OscillatorNode</td>
<td>694</td>
<td>41</td>
<td>20</td>
<td>3.712</td>
<td>0.391</td>
</tr>
<tr>
<td>Hybrid</td>
<td>694</td>
<td>45</td>
<td>21</td>
<td>3.875</td>
<td>0.410</td>
</tr>
<tr>
<td>CustomSignalHybrid</td>
<td>694</td>
<td>40</td>
<td>19</td>
<td>3.558</td>
<td>0.377</td>
</tr>
<tr>
<td>AudioSourceHybrid</td>
<td>610</td>
<td>36</td>
<td>19</td>
<td>5.173</td>
<td>0.587</td>
</tr>
<tr>
<td>ChannelMergeHybrid</td>
<td>694</td>
<td>46</td>
<td>21</td>
<td>4.327</td>
<td>0.458</td>
</tr>
<tr>
<td>AmplitudeModulationHybrid</td>
<td>694</td>
<td>46</td>
<td>21</td>
<td>4.679</td>
<td>0.496</td>
</tr>
<tr>
<td>FrequencyModulationHybrid</td>
<td>694</td>
<td>46</td>
<td>21</td>
<td>4.746</td>
<td>0.503</td>
</tr>
</tbody>
</table>

Table 4.3: Number of distinct and unique fingerprints and their entropy and normalized entropy

4.3.2 User-Agent Comparison

The User-Agent (UA) request header is a characteristic string that lets servers and network peers identify the application, operating system, vendor, and/or version of the requesting user agent [28]. A User-Agent can be known from the user agent header which is fairly simple, beside that even if a user uses a User-Agent switcher/defense, it is easy to figure out the User-Agent based on what Web API the browser is implementing. Every browser has their own implementation of the Web API and they can decide on the features they want to implement. Browser compatibility can be easily verified and the User-Agent can be easily extracted without any sophisticated technique by getting the differential implementation of Web APIs. Figure 4.4 shows the Web Audio API compatibility across different browsers. With regards to User-Agent we want to know if audio fingerprint diversity was simply an artefact of these implementation differences. In fact, the W3C (World Wide Web Consortium) documentation interestingly makes this exact same point "This merely allows deduction of information already readily available by easier means (User Agent

---

1https://caniuse.com/
string), such as "this is browser X running on platform Y". To get an answer to this question as well as to answer the research question RQ4 and ultimately answer the research question on diversity RQ3, we want to compare the audio fingerprints with User-Agent.

Figure 4.4: Browser compatibility of Web Audio API across different browsers

<table>
<thead>
<tr>
<th>Browser/OS</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung Internet for Android</td>
<td>Chrome</td>
</tr>
<tr>
<td>Yandex Browser</td>
<td>Chrome</td>
</tr>
<tr>
<td>Microsoft Edge</td>
<td>Chrome</td>
</tr>
<tr>
<td>Opera</td>
<td>Chrome</td>
</tr>
</tbody>
</table>

Table 4.4: Browser mapping

---

²https://www.w3.org/TR/webaudio/#priv-sec
We used the diverse fingerprint group (connected components(CC)) discussed in the Section 4.3.1 to further modify our code to find different User-Agent and Browser/Operating System (OS) combinations in one connected component. In order to plot the browser/OS combination in a more manageable way, we considered the browser and OS using the same base as the same entity. We considered Chrome and Firefox as the primary browsers and Windows, Linux, macOS and Linux as the primary OSs. Table 4.4 shows the mapping of different browsers to primary Browser. Figure 4.5 - 4.12 shows a homogeneity in browser and OS combination for all the audio fingerprinting vectors. The top text of each bar indicates the number of users / number of User-Agent associated with that distinct fingerprint group. We see that the majority of the groups either have the same browser or OS configuration. We see that the results are very uniform as we see that every distinct group is associated with only one browser either Firefox or Chrome but not both and at most 2 OSs. There is a lot of uniformity in terms of fingerprint groups and browser OS combination they are manifested in. Although every fingerprint group is associated with one browser, no browser is associated with only one fingerprint. For example Figure 4.5 shows 6 different fingerprint groups of Chrome/Android users with different numbers of users in each group. This means that each browser is associated with more than one fingerprint.

![Browser/OS Combination of connected components for DynamicCompressor](image)

Figure 4.5: Browser/OS combination of distinct fingerprint group for DynamicCompressor
Figure 4.6: Browser/OS combination of distinct fingerprint group for OscillatorNode

Figure 4.7: Browser/OS combination of distinct fingerprint group for Hybrid
Figure 4.8: Browser/OS combination of distinct fingerprint group for CustomSignalHybrid

Figure 4.9: Browser/OS combination of distinct fingerprint group for AudioSourceHybrid
Figure 4.10: Browser/OS combination of distinct fingerprint group for ChannelMergeHybrid

Figure 4.11: Browser/OS combination of distinct fingerprint group for AmplitudeModulationHybrid
We also wanted to see if audio fingerprints are a subset of User-Agents. In order to answer that we wanted to see if a single User-Agent is connected to multiple groups. We again utilized the distinct fingerprint group (connected component(CC) from 4.3.1 to find the total distinct and unique User-Agents in the user-study for each fingerprinting method. Then, used the difference between distinct and unique User-Agents to find the non-unique User-Agents for each distinct group to get the association between User-Agent and audio fingerprints. Table 4.5 shows that one User-Agent is spread across multiple groups for each fingerprinting technique. This means that two users having the same User-Agent can have different audio fingerprints. Unlike what the World Wide Web Consortium Document states we found that audio fingerprints can add incremental value to user-agent fingerprinting as there are a number of cases with same user-agent users having multiple audio fingerprints. Although audio fingerprints are better than User-Agents, we see that they do not hold notable value as Canvas and Font fingerprints. Thus, we arrive at a judgement about diversity of audio fingerprints saying that audio fingerprints are better than User-Agent fingerprint, however, they are not adequately diverse to be used as a standalone fingerprinting mechanism and answer our research question RQ3.
<table>
<thead>
<tr>
<th>Fingerprinting Methods</th>
<th># of UAs spanning &gt; 1 FP groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicCompressor</td>
<td>8</td>
</tr>
<tr>
<td>OscillatorNode</td>
<td>36</td>
</tr>
<tr>
<td>Hybrid</td>
<td>35</td>
</tr>
<tr>
<td>CustomSignalHybrid</td>
<td>36</td>
</tr>
<tr>
<td>AudioSourceHybrid</td>
<td>31</td>
</tr>
<tr>
<td>ChannelMergeHybrid</td>
<td>36</td>
</tr>
<tr>
<td>AmplitudeModulationHybrid</td>
<td>36</td>
</tr>
<tr>
<td>FrequencyModulationHybrid</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 4.5: Number of User-Agents spanning on more than 1 diverse fingerprint group
5. Discussion

5.1 Limitations

The analysis presented in this work has several methodological and measurement limitations. We tested our fingerprinting script using sine wave before settling for triangle wave, which showed similar results. Using other types of single such as square, saw-tooth and even changing the parameters of the AudioContext and OfflineAudioContext attributes may result in better unique fingerprints. However, we tested 8 different audio fingerprinting methods by obtaining FFTs of audio source files, modulated waves and custom waveforms, where we saw similar results across all the audio fingerprinting vectors. So we think that even if we try different waves and different audio parameters, we will end up with similar results. Secondly, in comparison to popular fingerprinting methods like Canvas and Font Fingerprinting based studies, our study is relatively small. However, even in the small size we see that audio fingerprints are less diverse in comparison to Canvas and Font, so we think that the results are going to hold up even in a bigger study. Thirdly, we had limited financial resources that hindered us to increase the number of users.

5.2 Future Work

We focus on the privacy concerned browser like Brave and understand their defenses against the audio fingerprinting. We want to understand the defenses proposed and study the robustness of these defenses. We also want to understand the value of audio fingerprints when used in conjunction with the existing browsers fingerprinting techniques like Canvas and Font fingerprinting. In addition, we would also like to conduct a larger user-study to solidify our findings in a more systematic and methodological way.
5.3 Conclusion

In this work, we tested the resilience of the mainstream browsers against audio fingerprinting on mobile and desktop. In order to do this, we devised an audio fingerprinting script based on earlier research [6, 8] and added 5 more techniques that collected 694 fingerprints across mobile and desktops. We analysed the stability of the audio fingerprinting methods which shows that the audio fingerprinting methods are stable when repeated multiple times in a timed setting. The diversity analysis, however shows that although the audio fingerprints are stable but they are not diverse enough to be used a single browser fingerprinting method by comparing it with the popular browser fingerprinting methods like Canvas, Font and User-Agent fingerprinting. Audio fingerprinting is important and has a tracking value which is better than User-Agent. Users with the same User-Agent can have different audio fingerprints but it does not have as significant value as Canvas and Font fingerprints. However, if used in conjunction with the more popular fingerprinting mechanism, the audio fingerprints can have an incremental value. Thus, with the statements presented here we finally answer the research question RQ6 saying that standalone audio fingerprints does not have as much diversity as existing fingerprinting mechanism, however, it does have incremental value, so further studies are needed.
References


getDynamicCompressorFingerprint(): Promise<any> {
    let sumBuffer = 0;
    let sumBufferHash = null;
    return new Promise((resolve, reject) => {
        try {
            let offlineAudioCtx = new ((<any>window).OfflineAudioContext || (<any>window).webkitOfflineAudioContext)(OFFLINEAUDICTX.
numberOfChannels, OFFLINEAUDICTX.length, OFFLINEAUDICTX.sampleRate);
            if (offlineAudioCtx) {
                let oscillator = offlineAudioCtx.createOscillator();
                oscillator.type = SIGNAL_TYPE;
                oscillator.frequency.value = FREQUENCY;
                // Create and configure compressor
                let compressor = offlineAudioCtx.createDynamicsCompressor();
                compressor.threshold && (compressor.threshold.value = COMPRESSOR.threshold);
                compressor.knee && (compressor.knee.value = COMPRESSOR.knee);
                compressor.ratio && (compressor.ratio.value = COMPRESSOR.ratio);
                // compressor.reduction && (compressor.reduction.value = -20);
                compressor.attack && (compressor.attack.value = COMPRESSOR.attack);
                compressor.release && (compressor.release.value = COMPRESSOR.release);
                // Connect nodes
                oscillator.connect(compressor);
                compressor.connect(offlineAudioCtx.destination);
                // Start audio processing
                oscillator.start(0);
                offlineAudioCtx.startRendering();
                offlineAudioCtx.oncomplete = ((evt: any) => {
                    sumBuffer = 0;
                    let MD5 = CryptoJS.algo.MD5.create();
                    for (let i = 0; i < evt.renderedBuffer.length; i++) {
                        MD5.update(evt.renderedBuffer.getChannelData(0)[i].toString());
                    }
                    const hash = MD5.finalize();
                    sumBufferHash = hash.toString(CryptoJS.enc.Hex);
                    for (let i = 4500; 5e3 > i; i++) {
                        sumBuffer += Math.abs(evt.renderedBuffer.getChannelData(0)[i]);
                    }
                    oscillator.disconnect();
                    //console.log("dynamicCompressor": sumBufferHash, "sum": sumBuffer);
                    //alert("dynamiccompressor " +sumBufferHash)
                    resolve({"hash": sumBufferHash, "sum": sumBuffer, "noFingerprint": false});
                });
            } else {
                // Code snippet for offlineAudioCtx
            }
        } catch (e) {
            reject(e);
        }
    });
}
Listing A.1: DynamicCompressor fingerprint generation code

```javascript
getOscillatorNodeFingerprint(): Promise<{}> {
    let oscillatorNode = [];
    let hash = null;
    return new Promise((resolve, reject) => {
        try {
            let audioCtx = ((<window>AudioContext) || (<window>webkitAudioContext))( );
            if (audioCtx) {
                let oscillator = audioCtx.createOscillator();
                let analyser = audioCtx.createAnalyser();
                let gain = audioCtx.createGain();
                let scriptProcessor = audioCtx.createScriptProcessor(SIGNATURE_PROCESSOR.bufferSize, SIGNATURE_PROCESSOR.numberOfInputChannels, SIGNATURE_PROCESSOR.numberOfOutputChannels);
                gain.gain.value = 1; // Disable volume
                analyser.fftSize = 2048;
                oscillator.type = SIGNAL_TYPE; // Set oscillator to output wave
                oscillator.connect(analyser); // Connect oscillator output to analyser input
                analyser.connect(scriptProcessor); // Connect analyser output to scriptProcessor input
                scriptProcessor.connect(gain); // Connect scriptProcessor output to gain input
                gain.connect(audioCtx.destination); // Connect gain output to audiocontext destination
                scriptProcessor.onaudioprocess = (async (event) => {
                    const bins = new Float32Array(analyser.frequencyBinCount);
                    analyser.getFloatFrequencyData(bins);
                    for (let i = 0; i < bins.length; i++) {
                        oscillatorNode.push(bins[i]);
                    }
                    analyser.disconnect();
                    scriptProcessor.disconnect();
                    gain.disconnect();
                    const audioFP = JSON.stringify(oscillatorNode);
                    hash = CryptoJS.MD5(audioFP).toString();
                    await audioCtx.close();
                    resolve({"hash": hash, "values": oscillatorNode, "noFingerprint": false });
                });
                oscillator.start(0);
            } else {
                reject({"hash": hash, "values": oscillatorNode, "noFingerprint": true });
            }
        } catch (u) {
            reject({"hash": hash, "values": oscillatorNode, "noFingerprint": true });
        }
    }):
}
```

Listing A.2: OscillatorNode fingerprint generation code

```javascript
getOscillatorNodeFingerprint(): Promise<{}> {
    let oscillatorNode = [];
    let hash = null;
    return new Promise((resolve, reject) => {
        try {
            let audioCtx = ((<window>AudioContext) || (<window>webkitAudioContext))( );
            if (audioCtx) {
                let oscillator = audioCtx.createOscillator();
                let analyser = audioCtx.createAnalyser();
                let gain = audioCtx.createGain();
                let scriptProcessor = audioCtx.createScriptProcessor(SIGNATURE_PROCESSOR.bufferSize, SIGNATURE_PROCESSOR.numberOfInputChannels, SIGNATURE_PROCESSOR.numberOfOutputChannels);
                gain.gain.value = 1; // Disable volume
                analyser.fftSize = 2048;
                oscillator.type = SIGNAL_TYPE; // Set oscillator to output wave
                oscillator.connect(analyser); // Connect oscillator output to analyser input
                analyser.connect(scriptProcessor); // Connect analyser output to scriptProcessor input
                scriptProcessor.connect(gain); // Connect scriptProcessor output to gain input
                gain.connect(audioCtx.destination); // Connect gain output to audiocontext destination
                scriptProcessor.onaudioprocess = (async (event) => {
                    const bins = new Float32Array(analyser.frequencyBinCount);
                    analyser.getFloatFrequencyData(bins);
                    for (let i = 0; i < bins.length; i++) {
                        oscillatorNode.push(bins[i]);
                    }
                    analyser.disconnect();
                    scriptProcessor.disconnect();
                    gain.disconnect();
                    const audioFP = JSON.stringify(oscillatorNode);
                    hash = CryptoJS.MD5(audioFP).toString();
                    await audioCtx.close();
                    resolve({"hash": hash, "values": oscillatorNode, "noFingerprint": false });
                });
                oscillator.start(0);
            } else {
                reject({"hash": hash, "values": oscillatorNode, "noFingerprint": true });
            }
        } catch (u) {
            reject({"hash": hash, "values": oscillatorNode, "noFingerprint": true });
        }
    }):
}
```
getHybridFingerprintWithAudioCtx(): Promise<any> {
    let hybridOscillatorNode = [];
    let hybridHash = null;

    return new Promise((resolve, reject) => {
        try {
            let audioCtx = new ((<any>window).AudioContext || (<any>window).webkitAudioContext)();
            if (audioCtx) {
                let oscillator = audioCtx.createOscillator();
                let analyser = audioCtx.createAnalyser();
                let gain = audioCtx.createGain();
                let scriptProcessor = audioCtx.createScriptProcessor(ScriptProcessor.bufferSize, ScriptProcessor.numberOfInputChannels, ScriptProcessor.numberOfOutputChannels);

                // Create and configure compressor
                let compressor = audioCtx.createDynamicsCompressor();
                compressor.threshold.setValueAtTime(Compressor.threshold, audioCtx.currentTime);
                compressor.knee.setValueAtTime(Compressor.knee, audioCtx.currentTime);
                compressor.ratio.setValueAtTime(Compressor.ratio, audioCtx.currentTime);
                compressor.attack.setValueAtTime(Compressor.attack, audioCtx.currentTime);
                compressor.release.setValueAtTime(Compressor.release, audioCtx.currentTime);

                gain.gain.value = 0; // Disable volume
                analyser.fftSize = 2048;
                oscillator.type = SIGNAL_TYPE; // Set oscillator to output triangle wave
                oscillator.connect(compressor); // Connect oscillator output to dynamic compressor
                compressor.connect(analyser); // Connect compressor to analyser
                analyser.connect(scriptProcessor); // Connect analyser output to scriptProcessor input
                scriptProcessor.connect(gain); // Connect scriptProcessor output to gain input
                gain.connect(audioCtx.destination); // Connect gain output to audiocontext destination

                scriptProcessor.onaudioprocess = (async (bins: any) => {
                    bins = new Float32Array(analyser.frequencyBinCount);
                    analyser.getFloatFrequencyData(bins);
                    for (let i = 0; i < bins.length; i++) {
                        hybridOscillatorNode.push(bins[i]);
                    }
                    analyser.disconnect();
                    scriptProcessor.disconnect();
                    gain.disconnect();
                    const audioFP = JSON.stringify(hybridOscillatorNode);
                    hybridHash = CryptoJS.MD5(audioFP).toString();
                    await audioCtx.close();
                    resolve({hash: hybridHash, values: hybridOscillatorNode, noFingerprint: false});
                });
                oscillator.start(0);
            } else {
                reject({hash: hybridHash, values: hybridOscillatorNode, noFingerprint: true});
            }
        } catch (u) {
            reject({hash: hybridHash, values: hybridOscillatorNode, noFingerprint: true});
        }
    })
})

Listing A.3: Hybrid fingerprint generation code
getCustomSignalHybridFingerprintAudioCtx(): Promise<any> {
  let hybridOscillatorNode = [];
  let hybridHash = null;
  return new Promise(async (resolve, reject) => {
    try {
      let audioCtx = new ((<any>window).AudioContext || (<any>window).webkitAudioContext)();
      if (audioCtx) {
        const OFFSET = 0.7;
        const pi = Math.PI;
        // https://medium.com/web-audio/phase-offsets-with-web-audio-wavetables-c7dc85ac3218
        // https://meettechniek.info/additional/additive-synthesis.html
        const real = new Float32Array(11);
        const imag = new Float32Array(11);
        real[0] = 0.360;
        real[1] = 0.760;
        real[2] = 0.120;
        real[3] = 0.745;
        real[4] = 0.235;
        real[5] = 0.145;
        real[6] = 0.545;
        real[7] = 0.675;
        real[8] = 0.585;
        real[9] = 0.685;
        real[10] = 0.115;
        real[11] = 0.660;
        imag[0] = pi/2;
        imag[1] = 0;
        imag[2] = pi/2;
        imag[3] = 0;
        imag[4] = pi/2;
        imag[5] = 0;
        imag[6] = pi/2;
        imag[7] = 0;
        imag[8] = pi/2;
        imag[9] = 0;
        imag[10] = pi/2;
        imag[11] = 0
        const wave = audioCtx.createPeriodicWave(real, imag, { disableNormalization: true });
        let oscillator = audioCtx.createOscillator();
        oscillator.frequency.value = 440;
        oscillator.setPeriodicWave(wave);
        const offset = audioCtx.createConstantSource();
        offset.offset.value = OFFSET;
        let analyser = audioCtx.createAnalyser();
        let gain = audioCtx.createGain();
        let scriptProcessor = audioCtx.createScriptProcessor(SCRIPT_PROCESSOR.bufferSize, SCRIPT_PROCESSOR.numberOfInputChannels, SCRIPT_PROCESSOR.numberOfOutputChannels);
        let compressor = audioCtx.createDynamicsCompressor();
        compressor.threshold.setValueAtTime(COMPRESSOR.threshold, audioCtx.currentTime);
        compressor.knee.setValueAtTime(COMPRESSOR.knee, audioCtx.currentTime);
        compressor.ratio.setValueAtTime(COMPRESSOR.ratio, audioCtx.currentTime);
        compressor.attack.setValueAtTime(COMPRESSOR.attack, audioCtx.currentTime);
        compressor.release.setValueAtTime(COMPRESSOR.release, audioCtx.currentTime);
Listing A.4: Custom Signal Hybrid fingerprint generation code

gain.gain.value = 0; // Disable volume
analyzer.fftSize = 2048;
oscillator.connect(compressor); // Connect oscillator output to dynamic compressor
compressor.connect(analyzer); // Connect oscillator output to dynamic compressor
analyzer.connect(scriptProcessor); // Connect analyser output to scriptProcessor input
scriptProcessor.connect(gain); // Connect scriptProcessor output to gain input
gain.connect(audioCtx.destination); // Connect gain output to audiocontext destination

scriptProcessor.onaudioprocess = (async (bins: any) => {
bins = new Float32Array(analyzer.frequencyBinCount);
analyzer.getFloatFrequencyData(bins);
for (let i = 0; i < bins.length; i++) {
    hybridOscillatorNode.push(bins[i]);
}
analyzer.disconnect();
scriptProcessor.disconnect();
gain.disconnect();
const audioFP = JSON.stringify(hybridOscillatorNode);
hybridHash = CryptoJS.MD5(audioFP).toString();
await audioCtx.close();
resolve({"hash": hybridHash, "values": hybridOscillatorNode, "noFingerprint": false});
});
oscillator.start(0);
offset.start();
} else {
    reject({"hash": hybridHash, "values": hybridOscillatorNode, "noFingerprint": true});
}
} catch (u) {
    reject({"hash": hybridHash, "values": hybridOscillatorNode, "noFingerprint": true});
}
}

getAudioSourceHybridFingerprintAudioCtx(): Promise<any> {
    let audioData = null;
    let hash = null;
    let analyserNodeData = [];
    return new Promise((resolve, reject) => {
        try {
            const audioContext = new ((<any>window).AudioContext || (<any>window).webkitAudioContext))();
            if (audioContext) {
                const sourceNode = audioContext.createBufferSource();
                const analyserNode = audioContext.createAnalyser();
                const gain = audioContext.createGain();
                const scriptProcessor = audioContext.createScriptProcessor(SCRIPT_PROCESSOR.bufferSize, SCRIPT_PROCESSOR.numberOfInputChannels, SCRIPT_PROCESSOR.numberOfOutputChannels);
                // Create and configure compressor
                const compressor = audioContext.createDynamicsCompressor();
                compressor.threshold.setValueAtTime(COMPRESSOR.threshold, audioContext.currentTime);
                compressor.knee.setValueAtTime(COMPRESSOR.knee, audioContext.currentTime);
                compressor.ratio.setValueAtTime(COMPRESSOR.ratio, audioContext.currentTime);
                compressor.attack.setValueAtTime(COMPRESSOR.attack, audioContext.currentTime);
            }
        } catch (e) {
            reject(e);
        }
    });
}
compressor.release.setValueAtTime(COMPRESSOR.release, audioContext.currentTime);
gain.gain.value = 0; // Disable volume
analyzerNode.fftSize = 2048;

// Now connect the nodes together
sourceNode.connect(compressor);
compressor.connect(analyzerNode);
analyzerNode.connect(scriptProcessor);
scriptProcessor.connect(gain);
gain.connect(audioContext.destination);

scriptProcessor.onaudioprocess = (async (event: any) => {
  const bins = new Float32Array(analyzerNode.frequencyBinCount);
analyzerNode.getFloatFrequencyData(bins);
  for (let i = 0; i < bins.length; i++) {
    analyzerNodeData.push(bins[i]);
  }
  const audioFP = JSON.stringify(analyzerNodeData);
  hash = CryptoJS.MD5(audioFP).toString();
  gain.disconnect();
  scriptProcessor.disconnect();
analyzerNode.disconnect();
  await audioContext.close();
  resolve({"hash": hash, "values": analyzerNodeData, "noFingerprint": false});
});

// Load the audio the first time through, otherwise play it from the buffer
if (audioData == null) {
  const request = new XMLHttpRequest();
  request.open('GET', '../../../assets/viper-05.ogg', true);
  request.responseType = 'arraybuffer';
  request.onload = (() => {
    audioContext.decodeAudioData(request.response, function (buffer) {
      audioData = buffer;
      sourceNode.buffer = buffer;
      sourceNode.start(0); // Play the sound now
      sourceNode.loop = false;
      function (e) {"Error with decoding audio data" + e};
    });
    request.send();
  })
  else {
    sourceNode.buffer = audioData;
    sourceNode.start(0); // Play the sound now
    sourceNode.loop = false;
  }
  else {
    reject({"hash": hash, "values": analyzerNodeData, "noFingerprint": true});
  }
  catch (u) {
    reject({"hash": hash, "values": analyzerNodeData, "noFingerprint": true});
  }
}
}

Listing A.5: Audio Source Hybrid fingerprint generation code
getChannelMergeHybridFingerprintAudioCtx(): Promise<any> {
  let hybridHash = null;
  let hybridOscillatorNode = [];
  return new Promise((resolve, reject) => {
    try {
      let audioCtx = new ((< any>window).AudioContext || (< any>window).webkitAudioContext)();
      if (audioCtx) {
        let oscillator1 = audioCtx.createOscillator();
        oscillator1.type = "sine";
        oscillator1.frequency.setValueAtTime(440, audioCtx.currentTime);
        let oscillator2 = audioCtx.createOscillator();
        oscillator2.type = "triangle";
        oscillator2.frequency.setValueAtTime(10000, audioCtx.currentTime);
        let oscillator3 = audioCtx.createOscillator();
        oscillator3.type = "square";
        oscillator3.frequency.setValueAtTime(1880, audioCtx.currentTime);
        let oscillator4 = audioCtx.createOscillator();
        oscillator4.type = "sawtooth";
        oscillator4.frequency.setValueAtTime(22000, audioCtx.currentTime);

        let channelMerger = audioCtx.createChannelMerger(4)
        oscillator1.connect(channelMerger, 0, 0);
        oscillator2.connect(channelMerger, 0, 1);
        oscillator3.connect(channelMerger, 0, 2);
        oscillator4.connect(channelMerger, 0, 3);

        let compressor = audioCtx.createDynamicsCompressor();
        let analyser = audioCtx.createAnalyser();
        let gain = audioCtx.createGain();
        let scriptProcessor = audioCtx.createScriptProcessor(SCRIPT_PROCESSOR.bufferSize, SCRIPT_PROCESSOR.numberOfInputChannels, SCRIPT_PROCESSOR.numberOfOutputChannels);

        compressor.threshold.setValueAtTime(COMPRESSOR.threshold, audioCtx.currentTime);
        compressor.knee.setValueAtTime(COMPRESSOR.knee, audioCtx.currentTime);
        compressor.ratio.setValueAtTime(COMPRESSOR.ratio, audioCtx.currentTime);
        compressor.attack.setValueAtTime(COMPRESSOR.attack, audioCtx.currentTime);
        compressor.release.setValueAtTime(COMPRESSOR.release, audioCtx.currentTime);

        gain.gain.value = 0; // Disable volume
        analyser.fftSize = 4096;

        channelMerger.connect(compressor); // Connect merger to compressor
        compressor.connect(analyser); // Connect compressor to analyser
        analyser.connect(scriptProcessor); // Connect analyser output to scriptProcessor input
        scriptProcessor.connect(gain); // Connect scriptProcessor output to gain input
        gain.connect(audioCtx.destination); // Connect gain output to audiocontext destination

        scriptProcessor.onaudioprocess = (async (bins: any) => {
          bins = new Float32Array(analyser.frequencyBinCount);
          analyser.getFloatFrequencyData(bins);
          for (let i = 0; i < bins.length; i++) {
            hybridOscillatorNode.push(bins[i]);
          }
          analyser.disconnect();
          scriptProcessor.disconnect();
          gain.disconnect();
        });
    }
  });
}

59
```javascript
const audioFP = JSON.stringify(hybridOscillatorNode);
hybridHash = CryptoJS.MD5(audioFP).toString();
await audioCtx.close();
resolve({"hash": hybridHash, "values": hybridOscillatorNode, "noFingerprint": false});
//start source
oscillator1.start(0);
oscillator2.start(0);
oscillator3.start(0);
oscillator4.start(0);
} else {
  reject({"hash": hybridHash, "values": hybridOscillatorNode, "noFingerprint": true});
}
} catch (u) {
  reject({"hash": hybridHash, "values": hybridOscillatorNode, "noFingerprint": true});
}
}

Listing A.6: Channel Merge Hybrid fingerprint generation code

getAmplitudeModulationHybridFingerprintAudioCtx(): Promise<any> {
  let hybridOscillatorNode = [];
  let hybridHash = null;
  return new Promise((resolve, reject) => {
    try {
      let audioCtx = new ((<any>window).AudioContext || (<any>window).webkitAudioContext))();
      if (audioCtx) {
        let mod = audioCtx.createOscillator();
        mod.frequency.setValueAtTime(18, audioCtx.currentTime);
        mod.type = "square"

        let modGain = audioCtx.createGain();
        modGain.gain.value = 30;

        let mod1 = audioCtx.createOscillator();
        mod1.frequency.setValueAtTime(440, audioCtx.currentTime);
        mod1.type = "triangle"

        let modGain1 = audioCtx.createGain();
        modGain1.gain.value = 60;

        let carrier = audioCtx.createOscillator();
        carrier.type = "sine"
        carrier.frequency.setValueAtTime(10000, audioCtx.currentTime);

        let carrierGain = audioCtx.createGain();
        carrierGain.gain.value = 1;

        let analyser = audioCtx.createAnalyser();
        let masterGain = audioCtx.createGain();
        masterGain.gain.value = 0; // Disable volume
      }
    }
  ```
mod . connect ( modGain ) ;
mod1 . connect ( modGain1 ) ;
mod . connect ( carrierGain . gain ) ;
mod1 . connect ( carrierGain . gain ) ;
carrier . connect ( carrierGain ) ;
// Create and configure compressor
let compressor = audioCtx . createDynamicsCompressor () ;
compressor . threshold . setValueAtTime ( COMPRESSOR . threshold , audioCtx . currentTime ) ;
compressor . knee . setValueAtTime ( COMPRESSOR . knee , audioCtx . currentTime ) ;
compressor . ratio . setValueAtTime ( COMPRESSOR . ratio , audioCtx . currentTime ) ;
compressor . attack . setValueAtTime ( COMPRESSOR . attack , audioCtx . currentTime ) ;
compressor . release . setValueAtTime ( COMPRESSOR . release , audioCtx . currentTime ) ;

let compressor = audioCtx . createDynamicsCompressor () ;
compressor . threshold . setValueAtTime ( COMPRESSOR . threshold , audioCtx . currentTime ) ;
compressor . knee . setValueAtTime ( COMPRESSOR . knee , audioCtx . currentTime ) ;
compressor . ratio . setValueAtTime ( COMPRESSOR . ratio , audioCtx . currentTime ) ;
compressor . attack . setValueAtTime ( COMPRESSOR . attack , audioCtx . currentTime ) ;
compressor . release . setValueAtTime ( COMPRESSOR . release , audioCtx . currentTime ) ;

let compressor = audioCtx . createDynamicsCompressor () ;
compressor . threshold . setValueAtTime ( COMPRESSOR . threshold , audioCtx . currentTime ) ;
compressor . knee . setValueAtTime ( COMPRESSOR . knee , audioCtx . currentTime ) ;
compressor . ratio . setValueAtTime ( COMPRESSOR . ratio , audioCtx . currentTime ) ;
compressor . attack . setValueAtTime ( COMPRESSOR . attack , audioCtx . currentTime ) ;
compressor . release . setValueAtTime ( COMPRESSOR . release , audioCtx . currentTime ) ;

let compressor = audioCtx . createDynamicsCompressor () ;
compressor . threshold . setValueAtTime ( COMPRESSOR . threshold , audioCtx . currentTime ) ;
compressor . knee . setValueAtTime ( COMPRESSOR . knee , audioCtx . currentTime ) ;
compressor . ratio . setValueAtTime ( COMPRESSOR . ratio , audioCtx . currentTime ) ;
compressor . attack . setValueAtTime ( COMPRESSOR . attack , audioCtx . currentTime ) ;
compressor . release . setValueAtTime ( COMPRESSOR . release , audioCtx . currentTime ) ;

let compressor = audioCtx . createDynamicsCompressor () ;
compressor . threshold . setValueAtTime ( COMPRESSOR . threshold , audioCtx . currentTime ) ;
compressor . knee . setValueAtTime ( COMPRESSOR . knee , audioCtx . currentTime ) ;
compressor . ratio . setValueAtTime ( COMPRESSOR . ratio , audioCtx . currentTime ) ;
compressor . attack . setValueAtTime ( COMPRESSOR . attack , audioCtx . currentTime ) ;
compressor . release . setValueAtTime ( COMPRESSOR . release , audioCtx . currentTime ) ;

let compressor = audioCtx . createDynamicsCompressor () ;
compressor . threshold . setValueAtTime ( COMPRESSOR . threshold , audioCtx . currentTime ) ;
compressor . knee . setValueAtTime ( COMPRESSOR . knee , audioCtx . currentTime ) ;
compressor . ratio . setValueAtTime ( COMPRESSOR . ratio , audioCtx . currentTime ) ;
compressor . attack . setValueAtTime ( COMPRESSOR . attack , audioCtx . currentTime ) ;
compressor . release . setValueAtTime ( COMPRESSOR . release , audioCtx . currentTime ) ;

// Connect carrier oscillator output to dynamic compressor
let carrierGain . connect ( compressor ) ;
// Connect analyser output to scriptProcessor input
let scriptProcessor . connect ( masterGain ) ;
// Connect gain output to audiocontext destination
scriptProcessor . onAudioProcess = ( async ( bins : any ) => {
  bins = new Float32Array ( analyser . frequencyBinCount ) ;
analyser . getFloatFrequencyData ( bins ) ;
for ( let i = 0 ; i < bins . length ; i ++ ) {
  hybridOscillatorNode . push ( bins [ i ] ) ;
}
analyser . disconnect () ;
scriptProcessor . disconnect () ;
masterGain . disconnect () ;
const audioFP = JSON . stringify ( hybridOscillatorNode ) ;
hybridHash = CryptoJS . MD5 ( audioFP ) . toString () ;
await audioCtx . close () ;
resolve ( { "hash": hybridHash , "values": hybridOscillatorNode , "noFingerprint": false } ) ;
});
carrier . start ( 0 ) ;
mod . start ( 0 ) ;
mod1 . start ( 0 ) ;
} else {
  reject ( { "hash": hybridHash , "values": hybridOscillatorNode , "noFingerprint": true } ) ;
}
} catch ( u ) {
  reject ( { "hash": hybridHash , "values": hybridOscillatorNode , "noFingerprint": true } ) ;
}
});
}

Listing A.7: Amplitude Modulation Hybrid fingerprint generation code

getFrequencyModulationHybridFingerprintAudioCtx () : Promise < any > {
  let hybridOscillatorNode = [] ;
  let hybridHash = null ;
  return new Promise ( ( resolve , reject ) => {
    try {
      let audioCtx = new (( < any > window ) . AudioContext || ( < any > window ) . webkitAudioContext ) () ;
      if ( audioCtx ) {
let mod = audioCtx.createOscillator();
mod.frequency.setValueAtTime(18, audioCtx.currentTime);
mod.type = "square"

let modGain = audioCtx.createGain();
modGain.gain.value = 30;

let mod1 = audioCtx.createOscillator();
mod1.frequency.setValueAtTime(440, audioCtx.currentTime);
mod1.type = "triangle"

let modGain1 = audioCtx.createGain();
modGain1.gain.value = 60;

let carrier = audioCtx.createOscillator();
carrier.type = "sine"
carrier.frequency.setValueAtTime(10000, audioCtx.currentTime);

let carrierGain = audioCtx.createGain();
carrierGain.gain.value = 1;

let analyser = audioCtx.createAnalyser();
let masterGain = audioCtx.createGain();
masterGain.gain.value = 0; // Disable volume

let scriptProcessor = audioCtx.createScriptProcessor(SCRIPT_PROCESSOR.bufferSize, SCRIPT_PROCESSOR.numberOfInputChannels,
SCRIPT_PROCESSOR.numberOfOutputChannels);

mod1.connect(modGain1);
modGain1.connect(carrier.frequency);
mod.connect(modGain);
modGain.connect(carrier.frequency);

// Create and configure compressor
let compressor = audioCtx.createDynamicsCompressor();
compressor.threshold.setValueAtTime(COMPRESSOR.threshold, audioCtx.currentTime);
compressor.knee.setValueAtTime(COMPRESSOR.knee, audioCtx.currentTime);
compressor.ratio.setValueAtTime(COMPRESSOR.ratio, audioCtx.currentTime);
compressor.attack.setValueAtTime(COMPRESSOR.attack, audioCtx.currentTime);
compressor.release.setValueAtTime(COMPRESSOR.release, audioCtx.currentTime);

carrier.connect(compressor); // Connect carrier output to analyser input
comparator.connect(analyser); // Connect compressor to analyser
analyser.connect(scriptProcessor); // Connect analyser output to scriptProcessor input
scriptProcessor.connect(masterGain); // Connect scriptProcessor output to gain input
masterGain.connect(audioCtx.destination); // Connect gain output to audiocontext destination

scriptProcessor.onaudioprocess = (async (bins: any) => {
  bins = new Float32Array(analyser.frequencyBinCount);
analyser.getFloatFrequencyData(bins);
for (let i = 0; i < bins.length; i++) {
  hybridOscillatorNode.push(bins[i]);
}
analyser.disconnect();
scriptProcessor.disconnect();
masterGain.disconnect();
const audioFP = JSON.stringify(hybridOscillatorNode);
Listing A.8: Frequency Modulation Hybrid fingerprint generation code

```javascript
hybridHash = CryptoJS.MD5(audioFP).toString();
await audioCtx.close();
resolve({"hash": hybridHash, "values": hybridOscillatorNode, "noFingerprint": false});
});
carrier.start(0);
mod.start(0);
mod.start(0);
} else {
  reject({"hash": hybridHash, "values": hybridOscillatorNode, "noFingerprint": true});
}
}
}
```

63-77
Figure B.1: The frequency of the highest occurring fingerprint value in 30 iterations for individual audio fingerprinting methods contd.
Figure B.1: The frequency of the highest occurring fingerprint value in 30 iterations for individual audio fingerprinting methods
Vita

The author, Shekhar Chalise, was born in Kathmandu, Nepal. He obtained his bachelor’s degree in Computer Engineering in 2015 from Tribhuvan University, Kathmandu, Nepal. He joined the University of New Orleans Computer Science Master’s program in Fall-2019. Shekhar has been working under Dr. Krishna Phani Kumar Vadrevu, in his Cybersecurity lab within the Department of Computer Science, at the University of New Orleans. He is also working as a Software Engineering Intern at LUCID LLC, New Orleans, LA and aspires to be a Software Engineer focusing on privacy and security. He worked on the study of stability and diversity of WebAudio fingerprints as part of his Master thesis in computer science.