MASARYK UNIVERSITY

FACULTY OF INFORMATICS

AudioContext Browser Fingerprinting

Bachelor's Thesis

JOSEF FROLA

Brno, Spring 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

AudioContext Browser Fingerprinting

Bachelor's Thesis

JOSEF FROLA

Advisor: Ing. Mgr. et Mgr. Zdeněk Říha, Ph.D.

Brno, Spring 2022


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

I declare that I have worked on this thesis independently, using only the primary and secondary sources listed in the bibliography.

Josef Frola

Advisor: Ing. Mgr. et Mgr. Zdeněk Říha, Ph.D.

Acknowledgements

I would like to express my gratitude to my advisor, Ing. Mgr. et Mgr. Zdeněk Říha, Ph.D., for his exceptional guidance throughout the writing of this thesis.

A massive thanks go to Ing. Štěpán Mráček, Ph.D., for supervising my work at ThreatMark and introducing me to this topic.

Lastly, I want to thank my friends, family, and other entities for helping me in various ways.

Abstract

Web browser fingerprinting is an integral part of device identification solutions. This thesis aims to describe and examine existing fingerprinting methods and find the best method of AudioContext fingerprinting. We describe the implementation details of the selected methods and explain the training and dataset of used machine learning models (simple regression model). We analyze the collected dataset by computing Shannon's entropy and examining the anonymity sets of the fingerprints. Finally, we evaluate and validate the newly trained models by using metrics such as Equal Error Rate (EER), Area Under the Receiver Operating Characteristic Curve (AUC), or Detection Error Tradeoff (DET) curve. We also discuss possible future problems with the selected fingerprinting methods.

Keywords

AudioContext, AudioContext fingerprinting, browser fingerprinting, device identification, device verification

Contents

List of Tables

List of Figures

1 Introduction

Third parties such as e-commerce and advertisement companies have always tried to track users on their Internet journeys. Without tracking, websites would not be able to present tailored ad content to their visitors. Among the tools allowing third parties to do so is web browser fingerprinting, identifying the combination of device and browser with which the user accesses the fingerprinted page. This practice is called Device Identification (DI), and it is very similar to facial recognition or verification of speakers [1] as it is a binary problem; the face either belongs to the person or does not. This similarity allows us to use the same evaluation tools for this problem, such as Equal Error Rate (EER), Area Under the Receiver Operating Characteristic Curve (AUC), or Detection Error Tradeoff (DET) curve.

ThreatMark, a cybersecurity company protecting users on their Internet endeavors, uses browser fingerprinting to fight fraud in their product called Anti-Fraud Suite (AFS). However, the current fingerprinting solution of AFS could benefit from the addition of AudioContext fingerprinting. Both ThreatMark's and other solutions benefit from higher fingerprint uniqueness. Adding new features to the existing browser fingerprint will most likely increase the fingerprint uniqueness, therefore increasing the precision of the solution.

This thesis describes web browser fingerprinting, focusing primarily on *AudioContext* fingerprinting methods from which we try to choose the most beneficial ones. We implement selected methods and deploy the fingerprinting solution to a production environment, from which we collect a dataset of browser fingerprints. The AFS utilizes a machine learning model, trained using the obtained dataset. This model predicts if two fingerprints originate from the same device. Finally, we evaluate and validate the performance of the models.

Chapter 2 explains what web browser fingerprinting is, describing currently used fingerprinting methods and fingerprinting history. Chapter 3 gives an overview of the Web Audio API, primarily the AudioContext interface and available AudioContext fingerprinting methods. Chapter 4 focuses on the implementation of selected methods. In chapter 5, we explain how the dataset was obtained and analyze the

dataset. Chapter 6 presents the machine learning model's architecture, training, and evaluation.

We implement the selected methods using JavaScript[1] and Python[2]. The training of the model and dataset analysis uses Python.

1. https://developer.mozilla.org/en-US/docs/Web/JavaScript
2. https://www.python.org/

2 Web Browser Fingerprinting

For most people, the gateway to the Internet is a web browser installed on their device (PC, laptop, mobile phone). We can obtain various information about the hardware and software of the device from these web browsers, for example, Chrome[1] or Safari[2]. The hardware-specific information can be a screen or window resolution, number of processing units, graphics card vendor, or the operating system of the device. Users can also use virtual machines (VirtualBox, VMware), where the hardware information we obtain depends on the implementation of the virtual machine. The browser itself provides information about its vendor and version, installed fonts or plugins, and other features. These features form the web browser fingerprint, which serves multiple purposes [2].

This section explains what is and how web browser fingerprinting works. Furthermore, we describe how third parties obtain the browser fingerprint and its' purpose. We also shortly describe web browser fingerprinting history.

2.1 History of Browser Fingerprinting

Third parties have been trying to identify users almost from the beginning of the Internet. Historically, the only methods for identifying users taken into account were cookies and IP addresses. The use of these methods has not vanished; they now serve as an addition to browser fingerprinting or vice versa. The problem is that users can block cookies and hide their IP addresses behind VPN/NAT. In reaction to this, third parties were forced to find new ways to track the users on their Internet journey reliably. This event is called the discovery of browser fingerprinting.

In a publication by Mayer [3] from 2009, he analyzed that the nuances between various browsing environments can be used by third parties to track the users. He used the term *"quirkiness"* by which he meant the combination of information like the operating sys-

1. `https://www.google.com/chrome`
2. `https://www.apple.com/safari`

tem, the hardware specification, and the web browser configuration. He approached his experiment by collecting the content of various browser API objects, such as *navigator*, *screen*, *navigator.plugins* and *navigator.mimeTypes*. He collected data of 1328 users who accessed his experiment web page and stated that 1278 (96.23%) users had a unique set of properties and therefore were uniquely identifiable. Due to the small scale of his experiment, he added that a more general conclusion could not be drawn [4, p. 4].

Electronic Frontier Foundation (EFF) published an experiment conducted by Peter Eckersley a year later [5]. In this study, 470 161 fingerprints were collected in the range of two weeks. Data were collected by communicating on socials and popular websites. Using data from Hypertext Transfer Protocol (HTTP) [6] headers, JavaScript, plugins, and more, 94.2% of fingerprints collected by his solution were unique. The outcome of this study proved that browser fingerprinting can be done on a large scale and that most users are identifiable while browsing the Internet.

2.2 Browser Fingerprinting Methods

As we mentioned before, the web browser fingerprint is a set of properties. We obtain these properties through the use of multiple fingerprinting methods. Many of these properties are directly available from the browser API, and obtaining them does not require any specialized methods. These properties generally have fewer information bits used for fingerprinting. In contrast, other methods such as canvas or WebGL fingerprinting require non-trivial effort (both technically and computationally); therefore, their entropy is usually higher. The output of these *"non-trivial"* methods is usually a hash string [4, 7].

2.2.1 Cookies

An HTTP cookie (web cookie, browser cookie) is a name-value pair and metadata. Cookies can be set with Javascript running in the user's browser or via the HTTP *Set-Cookie* header. Both methods store the cookie in the user's browser. Values of cookies used for browser fingerprinting are usually not volatile. The value in fingerprinting cookie

usually supplements the fingerprinting solution and serves as a label of the browser fingerprint [8, p. 260].

The long-lived cookies are practical for browser identification, especially in fraud prevention solutions. For example, if we set the cookie to expire in one year, we have a label linking all users' visits. We use this label to compare only fingerprints created with this label linked to the users' browser. Also, we can use the label reversely. Let us assume these session scenarios.

1. Fingerprint is unchanged, and the label has changed.
2. Fingerprint is different, and the label has changed.
3. Fingerprint is different, and the label stayed the same.

In the first scenario, the cookie might have expired; therefore new value was present on an identical fingerprint. Also, this scenario could indicate the use of anonymous browser sessions, and it does not indicate fraudulent behavior. If the user logins with a different label and fingerprint, the user has used a new device or new browser on an already recognized device, or someone else is accessing the account. In this case, other aspects of the session need to be checked, such as geolocation or timezone. The third scenario might be caused by a major browser update or the fraudster stealing the cookie.

2.2.2 User-Agent

The User-Agent[3] request-header value is a string containing a distinct value for specific browsers. This string (example below) contains essential information for any web browser fingerprinting solution. Example:

```
Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:47.0)
Gecko/20100101 Firefox/47.0
```

There are two main approaches used to utilize this information. First, we can take the value as-is and compare it or hash the string to

3. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent

obtain an ID. The second approach parses the string and separates the data into individual properties. These properties can be later used, for example, to construct a feature vector as an input to a decision tree algorithm [9].

2.2.3 Fonts

This method obtains a list of fonts installed in the user's browser and system. The method used to utilize the Flash[4] plugin to obtain all fonts installed on the device in the past. Because Flash was deprecated, various new methods were introduced by fingerprinting solution vendors.

Javascript code executed on the client-side is currently used to obtain this information. The code creates a *div* element a tries to render a string with selected font. The browser first checks the device to see if the font is present and if not, it uses its own. The script then measures the dimension of the div depending on the generated font. Also, it should be noted that the solutions are using hidden *iframes*, so the user does not notice the insertion of elements into the web page.

When utilizing a fonts list in any fingerprinting solution, it is crucial to remember that this method is time-consuming because of the insertions in the background. To shorten the time needed to obtain the list, we must select a subset of fonts that best fit our use case [10].

2.3 Third-parties

Third parties want to gather fingerprints for many reasons. Most of these reasons are not beneficial for the users as such gathering of user data primarily aims to increase the profits of the third parties or their customers. The most common usage of such tracking is targeted marketing. The subsequent usage is for security in fraud prevention/protection solutions or law enforcement agencies to track criminals on the Internet.

Most third-party solutions are trying to provide an ID based on the collected data. The fewer ID collisions, the better the solution. There

4. `https://www.adobe.com/products/flashplayer/end-of-life.html`

are not many ways the users can protect themself, and sometimes if they try to do so, they can even be more trackable.

Another use case is a solution that returns the probability/distance that the fingerprint belongs to the same device by comparing two or more fingerprints. Security solutions often use this method to determine if the user enters the website from the same device as usual. The higher the distance between two fingerprints, the higher the probability of fingerprints originating from different devices.

2.4 ThreatMark

ThreatMark is a cybersecurity company specializing in fraud prevention [11]. Founded in 2015, the firm has been growing rapidly while creating solutions to help fight fraud. ThreatMarks main selling point is a product called AFS [12]. The company specializes in protecting customers of financial organizations such as banks. Notably, the company has won many awards and is also a representative vendor in the Gartner's Market Guide for Online Fraud Detection [13].

2.4.1 Anti-Fraud Suite

The Anti-Fraud Suite (AFS) is a full-stack fraud prevention platform utilizing behavioral intelligence with transaction risk analysis, user identification, and many more. Among the many detection abilities of AFS are the abilities to detect malware, phishing, bot sessions, or Remote Access Tool (RAT) attacks. The malware detection works for mobile devices using Android and iOS operating systems. Phishing detection utilizes a JavaScrip probe observing changes made to the DOM[5] tree. To detect the RAT attacks, AFS examines the user's mouse movement patterns and scans open ports to detect the usage of some remote access tools.

The AFS utilizes web browser fingerprinting to detect login attempts from fraudsters' devices to detect fraudulent web sessions. AFS uses a selected combination of available fingerprinting methods for its fingerprinting component. Besides collecting the fingerprint,

5. `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`

AFS uses HTTP cookies to label the fingerprints. Every browser has a cookie(device tag) marking the browser on the user's device. It is important to note that cookies are not safe, and fraudsters can easily steal the device tag. Therefore, the need for fingerprinting solution apart from the cookie to recognize the user's device.

3 AudioContext Fingerprinting

The AudioContext fingerprinting method utilizes *Web Audio API*[1] operations executed inside an audio context. The API provides a way to operate the combination of a device's audio hardware and software. By fingerprinting the AudioContext, we can obtain valuable information to distinguish different combinations of hardware, software, and the browser itself.

AudioContext fingerprinting is being regarded as infrequently used. The inferior utilization of this method might be caused by its relative novelty [4, p. 6] or, as shown in Section 5.4.2, by the lower fingerprint uniqueness. Still, this method provides solid over-the-time stability, meaning that the information obtained from the method does not tend to change over some period of time.

3.1 Web Audio API

As mentioned above, the Web Audio API allows the utilization of audio operations in the web browser. By using Javascript, we can create an AudioContext object providing an interface to control various audio modules linked together. These modules are AudioNode[2] objects linked together, forming an audio routing graph (Figure 3.1). AudioNode object is a generic interface serving as an audio processing module [14].

3.1.1 AudioContex

AudioContext[3] interface gives us control over the creation and execution of the audio processing. It is considered good practice to create a single instance of AudioContext, which we can use for different operations.

---

1. `https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API`
2. `https://developer.mozilla.org/en-US/docs/Web/API/AudioNode`
3. `https://developer.mozilla.org/en-US/docs/Web/API/AudioContext`

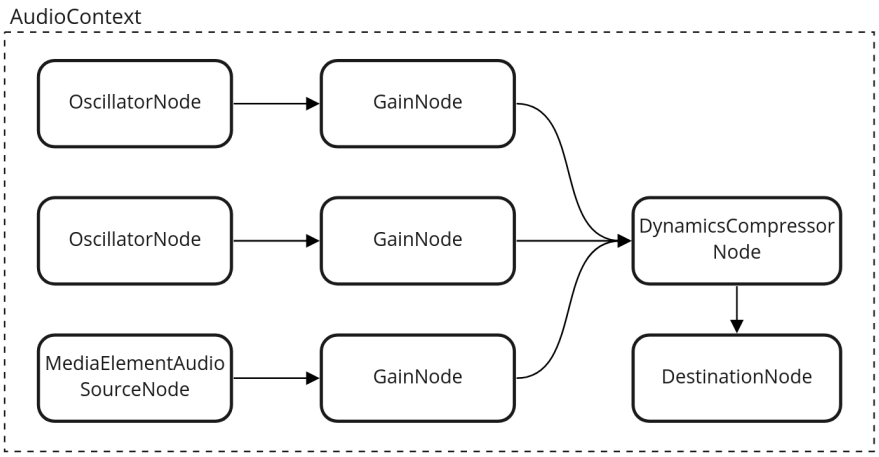


Figure 3.1: Example of AudioNodes creating an audio routing graph.

OfflineAudioContext[4] is an interface of AudioContext. This interface allows us to render audio without accessing the device's hardware. The OfflineAudioContext generates the audio data as fast as possible and saves the generated data to an AudioBuffer stored in the browser memory. AudioBuffer data is internally depicted in Linear Pulse Code Modulation (LPCM) [15] encoding, and the values are 32-bit floats in the range from `-1.0` to `1.0` [14].

3.1.2 OscillatorNode

The OscillatorNode[5] mathematically generates audio samples and serves as a source node. We can generate periodic waveforms using this interface instead of loading and playing an audio file in the fingerprinting solution. When creating the oscillator node, we have multiple options on how to customize it. We can set the oscillator to generate different waves such as sine (default), square, sawtooth, or triangle and set a custom frequency [14].

4. `https://developer.mozilla.org/en-US/docs/Web/API/OfflineAudioContext`
5. `https://developer.mozilla.org/en-US/docs/Web/API/OscillatorNode`

3.1.3 DynamicsCompressorNode

The DynamicsCompressorNode [6] helps prevent distortion or clipping when multiple sounds are played simultaneously and allows to lower the volume of the loudest parts of the signal. It is also worth noting that the compressor node has precisely one input and one output. The DynamicsCompressorNode inherits properties from AudioNode and allows high customizability [14]. These properties are represented like AudioParam[7] values and the value ranges are shown in Table B.1.

- **Threshold** represents the value in decibels (dB) above which the compression starts taking effect.
- **Knee** is a decibel value specifying the range above the threshold where the curve smoothly transitions to the compressed portion.
- **Ratio** is a decibel value representing the amount of change for 1 dB in output needed in the input.
- **Reduction** represents the float value setting the strength of gain reduction applied to the signal.
- **Attack** represents the number of seconds needed to reduce the gain by 10 dB.
- **Release** is a opposite of attack property. It is the number of seconds needed to increase the gain by 10 dB.

3.2 Available Fingerprintig Methods

The previous section established the building blocks required for AudioContext fingerprinting; it is now possible to examine and describe various fingerprinting methods. The web page `https://audiofingerprint.openwpm.com/` gives an excellent overview of all the available information about how to utilize AudioContext fingerprinting. This

6. `https://developer.mozilla.org/enUS/docs/Web/API/DynamicsCompressorNode`
7. `https://developer.mozilla.org/en-US/docs/Web/API/AudioParam`

web page is a part of a study concluded by Steven Englehardt and Arvind Narayana to monitor and describe available fingerprinting methods used by third parties [16].

In the beginning, we can check if the AudioContext interface is available and usable in the browser, giving us a single bit of information. It is worth mentioning that the AudioContext interface is available in most of the current browsers but cannot be started without user input. This input can be a button click, screen swipe, and others.

3.2.1 AudioContext Properties

Another easily obtainable set of information is the AudioContext object's properties. The object has two read-only properties: *baseLatency* and *outputLatency*. Other properties are the *destination* property and properties obtained by creating a new AnalyserNode[8] and examining its properties. The AudioContext properties are:

- **baseLatency**[9] represents the number of seconds of processing latency when the AudioContext passes the audio from the AudioDestinationNode to the audio subsystem. We can set this property during the initialization of the AudioContext by using *latencyHint* option, but some browsers may ignore it.

- **outputLatency**[10] read-only property returns an estimation of the output latency in seconds. This property is only available in Firefox-based browsers.

- **sampleRate**[11] is a float number representing the sample rate of audio context in samples per second. The default value varies depending on the output device.

8. https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode
9. https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/baseLatency
10. https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/outputLatency
11. https://developer.mozilla.org/en-US/docs/Web/API/BaseAudioContext/sampleRate

- **state**[12] is a read-only property of BaseAudioContext interface. It gives us information describing the state of current AudioContext. The possible values are `suspended`, `running` or `closed`.

- **destination** is an AudioDestination[13] interface, representing the output of the audio graph. The node has no output as it is the last node in the whole chain of audio nodes. If we use this node with an OfflineAudioContext, this node will act as the recorder of the audio data. The AudioDestinationNode inherits properties from *AudioNode*, its parent, and has one extra property *maxChannelCount*. Rest of the properties are listed in Table 3.1.

- **AnalyserNode** provides real-time frequency and time-domain analysis information. The node does not manipulate the audio data it receives, but it allows to analyzes the received data and creates audio visualizations. Besides the inherited properties from AudioNode, the AnalyserNode has a few different properties:

  - *fftSize* represents the size of Fast Fourier Transform (FFT).
  - *frequencybinCount* is half of the value of fftSize.
  - *minDecibels* represents the minimum decibel value for FFT scaling
  - *maxDecibels* represents the maximum decibel value for FFT scaling
  - *smoothingTimeConstant* is an average between the last and current buffer the AnalyserNode processed.

12. `https://developer.mozilla.org/en-US/docs/Web/API/BaseAudioContext/state`
13. `https://developer.mozilla.org/en-US/docs/Web/API/AudioDestinationNode`

Table 3.1: Example of fingerprintable AudioContext properties.

Property	Source	Value
baseLatency	AC	0.011609977324263039
outputLatency	AC	0
sampleRate	AC	44100
state	AC	"running"
maxChannelCount	ADN	2
numberOfInputs	ADN	1
numberOfOutputs	ADN	0
channelCount	ADN	2
channelCountMode	ADN	"explicit"
channelInterpretation	ADN	"speakers"
fftSize	AN	2048
frequencybinCount	AN	1024
minDecibels	AN	-100
maxDecibels	AN	-30
smoothingTimeConstant	AN	0.8
numberOfInputs	AN	1
numberOfOutputs	AN	1
channelCount	AN	2
channelCountMode	AN	"max"
channelInterpretation	AN	"speakers"

AC - AudioContext, ADN - AudioDestinationNode, AN - AnalyserNode

3.2.2 Fingerprint Using DynamicsCompressor

This method utilizes `OfflineAudioContext`, OscillatorNode, and DynamicsCompressorNode. The method is executed in the following sequence:

1. Create a new OfflineAudioContext.
2. Initialize OscillatorNode with triangle wave and frequency of 10 000 Hz on the created audio context.
3. Create DynamicsCompressorNode and set its properties to desired values.

- threshold to `-50`
- knee to `40`
- ratio to `12`
- reduction to `-20`
- attack to `0`
- release to `0.25`

4. Attach the oscillator node to the compressor node and the compressor node to the output node, the AudioDestinationNode.
5. Start the oscillator and instruct the audio context to start rendering.
6. When the rendering is complete (oncomplete[14] event), there are a few options on how to handle the obtained data.
   (a) Iterate over the data buffer and feed the data to a hashing function resulting in a `hash string`.
   (b) Iterate over a selected number of values and sum up the absolute of these values. This option produces a `single float number`.

3.2.3 Fingerprint Using OscillatorNode

The main difference between this and the previous method is that this method uses AudioContext instead of OfflineAudioContext. This method also does not utilize DynamicsCompressor in any way. This method instead uses the AnalyserNode and ScriptPrecessor to analyze the outputs of the oscillator. The steps to obtain the fingerprint are the following:

1. Create a new AudioContext.
2. Create an OscillatorNode and customize it as needed.

14. `https://developer.mozilla.org/en-US/docs/Web/API/OfflineAudioContext/complete_event`

3. Create GainNode[15], AnalyserNode, and ScriptProcessorNode[16].
    - Set the `gain` property of the GainNode to `0`, as we do not want to produce sound on the user's audio hardware.
4. Connect the oscillator node to the analyser node, the script processor to the gain node, and the gain to the output node.
5. Start the oscillator.
6. After the `onaudioprocess` event occurs, create a new `Float32Array`[17] to which the data from the AnalyserNode are stored.
7. Disconnect all the created nodes.
8. The output is an `Float32Array`of frequency data.

3.2.4 Hybrid of OscillatorNode and DynamicsCompressor

This method is almost identical to the previously described method. The only difference is that this method connects a DynamicsCompressorNode to the AnalyserNode; the rest of the steps stays the same. The output of this method is also a `Float32Array` of frequency data.

15. `https://developer.mozilla.org/en-US/docs/Web/API/GainNode`
16. `https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode`
17. `https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Float32Array`

4 Implementation

The implementation description will not mention all details regarding the architecture and functionality of AFS because of NDA policies. We only highlight parts related to AudioContext fingerprinting, which are a part of a fully working fingerprinting solution. These components are a Javascript probe running in the user's browser and a server-side component written in Python.

The probe collects various information from the browser, constructing a fingerprint message. Part of this message is the information about audio context, which we examine in this work. This fingerprint message is then sent for processing to the backend component via an HTTP request. The message processing consists of storing the data in a database from which the data are being requested to perform machine learning model training and fingerprint comparison.

4.1 JavaScript Probe

In our implementation, the AudioContext fingerprint consists of three main parts. These parts are the one-bit information telling us if the browser supports AudioContext, the selection of AudioContext properties, and a sum of oscillator buffer data.

We do not use all fingerprinting methods mentioned in the previous chapter because the information received by these methods would be redundant. We are allowed to do this because the combination of AudioContext properties and one hash-like method gives us almost identical information as if we used properties and several hash-like methods. The yielded information is identical because the hash-like methods tend to have similar entropy and over-the-time stability.

4.1.1 AudioContext Support

This method gives us a single bit of information which we store to the `boolean` *AudioCtx.support* attribute. This attribute tells us if the AudioContext object is available for use in the fingerprinted browser. To obtain this information we use the `typeof` on *window.AudioContext* or *window.webkitAudioContext* and if the returned type is a `"function"`

```
1   function audioCtxSupport () {
2     var AudioContext =
3       window.AudioContext
4       || window.webkitAudioContext;
5     var audioCtxSupport = typeof AudioContext
        === "function" ? true : false;
6
7     return audioCtxSupport;
8   }
```

Listing 4.1: JavaScript function of AudioContext support fingerprinting method.

we set the boolean value to `true`, otherwise to `false`. Below, we show the Javascript code used to obtain the data regarding the support of AudioContext in the browser.

4.1.2 AudioContext Properties

The Princeton example [16] of audio context property fingerprinting utilizes properties with low entropy. Therefore, we decided not to include these in our solution. Among these properties are properties obtained by creating AnalyserNode, listed in Table 3.1. These properties are initialized with browser-dependent defaults, so we do not bother creating the said node. Another property we left out was the *state* property of the AudioContext object. This value provides information about the state of the AudioContext when the probe starts. For example, in Mozilla Firefox, the value is always set to `"suspended"`, indicating the use of the said browser. This information is redundant as it is easily obtainable from the user-agent string. Also, this property has only three possible values, indicating low information gain. The last property we left out from our selection is *outputLatency* because it is supported only on firefox-based browsers.

Below, we show the approach on how to obtain the propertiers. First, we check if *AudioContext* object is available; if not, we fall back to *webkitAudioContext*. The function then returns an object containing copied values.

```
1   function audioCtxProperties() {
2   var p = {};
3
4   var AudioContext = window.AudioContext ||
       window.webkitAudioContext;
5   var ctx = new AudioContext();
6   var dest = ctx.destination;
7
8   p["BaseLatency"] = ctx.baseLatency;
9   p["SampleRate"] = ctx.sampleRate;
10  p["MaxChannelCount"] = dest.maxChannelCount;
11  p["NumberOfInputs"] = dest.numberOfInputs;
12  p["NumberOfOutputs"] = dest.numberOfOutputs;
13  p["ChannelCount"] = dest.channelCount;
14  p["ChannelCountMode"] = dest.channelCountMode;
15  p["ChannelInterpretation"] = dest.
       channelInterpretation;
16
17  return p;
18  }
```

Listing 4.2: JavaScript function of AudioContext properties fingerprinting method.

4.1.3 AudioContext Buffer Sum

For our hash-like method, we choose OfflineAudioContext in combination with DynamicsCompressor. This method is described in Section 3.2.2 and Figure 4.1 shows the audio data are processed. The output of our implementation is a sum of absolute values from the *renderedBuffer*. It is a sum of the last five hundred values the buffer contains. The values oscillate between 1 and -1, depending on the browser. Figure 4.2 presents that both Chrome and Firefox runs result in different values, thus resulting in different sums. Besides the browser architecture, other factors also affect the results, such as sample rate or operating system.

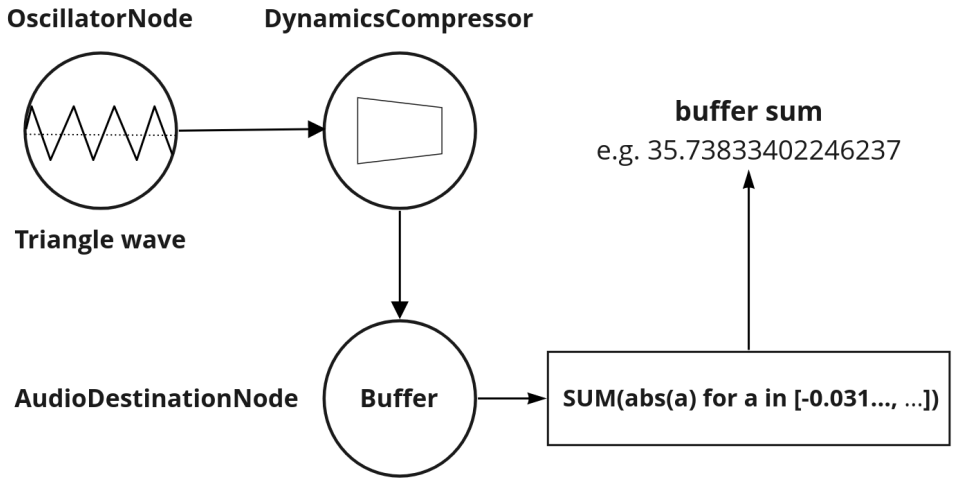


Figure 4.1: Flow of the buffer sum fingerprinting method.

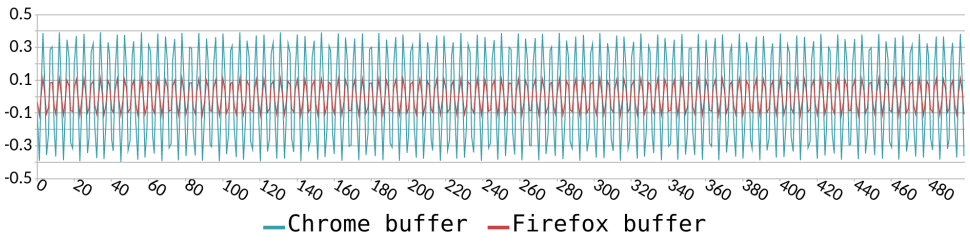


Figure 4.2: Example of buffer values of the buffer sum fingerprinting method on Chrome and Firefox.

Table 4.1 shows examples of values returned by this method from various environments. All Chrome-based browsers return the same value when fingerprinted on Linux systems. All browsers listed in the table, besides Firefox and Safari, use the Blink browser engine, which results in the same values on different versions or different browsers. Moreover, Firefox, a Gecko-based browser, has the same values on Linux systems, but the value changes when switching to a different operating system and CPU model. When the implementation of Web Audio API does not change between versions, the fingerprinted values do not change too. This behavior gives us great over-the-time stability.

Table 4.1: Example values of AudioContext buffer sum fingerprirnt on different combinations of devices and browsers.

OS	CPU	Browser	Value
Ubuntu 20.04	Intel i7-8550U	Chrome 101	`124.04347527516074`
		Edge 101	`124.04347527516074`
		Opera 86	`124.04347527516074`
		Firefox 99	`35.73833402246237`
Fedora 35	Intel i7-8665U	Chrome 101	`124.04347527516074`
		Firefox 99	`35.73833402246237`
RHEL 8.5	Intel i7-8650U	Chrome 100	`124.04347527516074`
		Firefox 94	`35.73833402246237`
Windows 11	Ryzen 5 3600	Chrome 101	`124.04347527516074`
		Firefox 99	`35.7383295930922`
macOS 12.3.1	Intel Core i5	Chrome 101	`124.04347657808103`
		Safari 15.4	`124.04345808873768`

However, the information gain from this method is rather small; we explain this in Section 5.4.

The following code block presents a minimal working solution to obtain the sum of the *renderedBuffer*. However, in the full implementation, we created a *tryResume* function which, depending on the *state* property of the current AudioContext object, tries to start rendering of the context. The function tries to start the context three times every half a second and, if unsuccessful, abandons the effort and returns `null`.

```
1   var bufferSum = 0;
2   function audioCtxBufferSum() {
3     var AudioContext =
4       window.OfflineAudioContext
5       || window.webkitOfflineAudioContext;
6     var ctx = new AudioContext(1, 5000, 44100);
7     var oscillator = ctx.createOscillator();
8     oscillator.type = "triangle";
9     oscillator.frequency.value = 1e4;
10    var compressor =
11      ctx.createDynamicsCompressor();
12    compressor.threshold.value = -50;
13    compressor.knee.value = 40;
14    compressor.ratio.value = 12;
15    compressor.reduction.value = -20;
16    compressor.attack.value = 0;
17    compressor.release.value = 0.25;
18
19    oscillator.connect(compressor);
20    compressor.connect(ctx.destination);
21    oscillator.start(0);
22    ctx.startRendering();
23
24    ctx.oncomplete = function (e) {
25      for (
26          var i = 4500;
27          i < e.renderedBuffer.length; i++
28        ) {
29        bufferSum += Math.abs(
30          e.renderedBuffer.getChannelData(0)[i]
31        );
32      }
33      compressor.disconnect();
34    };
35  }
```

Listing 4.3: JavaScript function of AudioContext buffer sum fingerprinting method.

4.1.4 Possible Problems and Side Effects

It is worth mentioning that the AudioContext interface is available in almost all browsers, but in some, the audio context is not allowed to start without the user interacting with the web page. Among these are iOS 11 or older, Google Chrome since version 71 [17], and other Chrome-based web browsers. In these browsers, the rendering of audio data starts if triggered by a user action or gesture.

The AudioContext not starting right away causes a side effect in these browsers. When the web page waits for the user's gesture, it prints a warning message to the web console, informing the user that the AudioContext was not allowed to start. Figure 4.3 the content of the warning message. This behavior makes the probe visible in one place besides the network and sources manager in the developer web tools and leads to customers asking why the bank needs to use their audio. This somewhat blows the cover of the fingerprinting component being almost invisible and might be a reason why we drop AudioContext fingerprinting in the future.

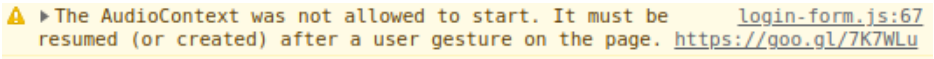


Figure 4.3: A warning appears in Chrome browser console during the creation of AudioContext.

Another problem occurs when the browser rejects starting the audio processing because the page is in the background. This behavior is observable in iOS 12 and newer [14].

We work around both these problems by stalling/delaying the audio context processing using timeouts and eventually giving up if no user action/gesture occurs in a certain timeframe. The issue with the warning message has no other solution than removing the AudioContext fingerprinting part from the JavaScript probe.

4.2 Python Backend

The Python code block listed below represents the AudioContext fingerprint data structure used in the backend application. The data

classes were created using non-standard library `pydantic` [18], giving us data validation using python type annotations. Only *support* attribute is mandatory because not all browsers support the AudioContext interface. Section 4.1.2 explains the selection of AudioContext properties in a great detail.

```
1   from typing import Optional
2   from pydantic import BaseModel
3
4   class AudioCtxProperties(BaseModel):
5     base_latency: Optional[float]
6     sample_rate: Optional[int]
7     max_channel_count: Optional[int]
8     number_of_inputs: Optional[int]
9     number_of_outputs: Optional[int]
10    channel_count: Optional[int]
11    channel_count_mode: Optional[str]
12    channel_interpretation: Optional[str]
13
14  class AudioCtx(BaseModel):
15    support: bool
16    properties: Optional[AudioCtxProperties]
17    buffer_sum: Optional[float]
```

Listing 4.4: AudioContext fingerprint data classes used in the backend Python application.

5 Dataset

To create the dataset, we needed to deploy a new version of AFS containing the new AudioContext fingerprinting solution to a production environment. The updated fingerprinting probe would then collect new fingerprints with added AudioContext fingerprints needed to retrain the current machine learning model of AFS. For the data collection, we choose the production environment of Slovenská Sporiteľňa (SLSP), one of ThreatMark's customers.

5.1 Data Collection

Figure 5.1 shows a histogram of the dataset provided by ThreatMark, containing 226 846 fingerprints created between August 2021 to December 2021. The spikes represent a new browser version releases, which tend to create new fingerprints as some web APIs do sometimes change. The dataset consists of a list of users who have a set of *json* fingerprint objects linked to their device tags. A *device tag* is a random string value obtained from a cookie, which is used to link the fingerprints to the users.

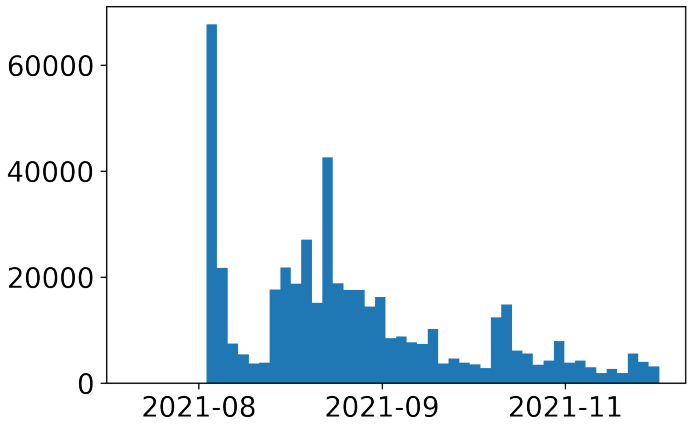


Figure 5.1: Histogram showing the creation time of fingerprints collected from SLSP production.

When loading the fingerprints, we create a Python dataclass *Fingerprint* with the following properties. These are *creation_time*, the creation time of the fingerprint, *fingerprint_id*, a string ID of the fingerprint, *data*, a dictionary representing the fingerprint object, and *user_agent*, a *UserAgent* object. The *data* property contains a user-agent string, the AudioContext Fingerprint, and other fingerprinted information. Because users can have more device tags linked to their alias, we create a dictionary where the keys are a combination of the *device tag* and *alias* of the user. The value stored behind this key is a set of *Fingerprint* objects fingerprinted on this *device tag* cookie under this user alias.

5.2 Data Cleaning

5.2.1 User-agents

One *device tag* should have linked fingerprints only from one combination of operating system and browser. This means that *device tag* created on Chrome contains only fingerprints from Chrome. If this condition is not met, we delete such fingerprints found on the examined key.

5.2.2 Missing AudioContext Fingerprints

When preparing the dataset for the *AudioContext-only* model, the dataset could not contain fingerprints with missing AudioContext data. This step was critical, as these are the only data, besides user-agent, needed to train the AudioContext-only model.

5.3 Data Processing

5.3.1 Platforms

We divide the dataset into multiple categories, referred to as *platforms*. A platform is a dictionary key constructed from the UserAgent[1] object *os.family* and *browser.family*. We distinguish only the major

1. `https://pypi.org/project/user-agents/`

Table 5.1: Fingerprint platform names used to devide the dataset.

Operating system		Browser
Windows		Firefox
iOS		Chrome
Mac OS X		Opera
Android	X	IE
Linux		Safari
Other		Edge
		Other

operating systems; the rest is marked as `"other"`. The same logic applies to web browser families. The produced key is, for example, `"Windows:Chrome"`. Table 5.1 shows the Cartesian product of operating system families and browser families, giving us all platform names used to divide the dataset.

5.3.2 Diff Feature Vector

The input of the model is a vector describing changes between two fingerprints. It is vital to point out that the order of the fingerprints matters. The first argument of the diffing function is the old fingerprint, and the second argument is the newer fingerprint. The resulting diff feature vector of two identical fingerprints is a zero vector.

5.3.3 Positive Samples

For each platform, we select positive and negative samples. Positive samples are diff feature vectors computed from the fingerprints belonging to the same device tag, while the negative feature vectors originate from two different device tags but have a matching platform.

For each *device tag* we create N positive samples, where N is the number of unique fingerprints found on this *device tag*. The first sample is the newest fingerprint on the device tag compared with itself, so that the model can learn how the feature vector looks when the fingerprint remains the same. The additional *N-1* samples are the remaining distinct fingerprints compared in a way that the first input is the older fingerprint compared with the closes newer one. For example, if we

have four fingerprints named FP1, FP2, FP3, and FP4, where FP1 is the newest and FP4 is the oldest, we obtain the following positive samples:

1. Diff of FP1 and FP1.
2. Diff of FP2 and FP1.
3. Diff of FP3 and FP2.
4. Diff of FP4 and FP3.

We limit the number of samples in each platform so that the platforms with the most samples do not outclass the less frequent platforms, for example `"Windows:Chrome"` outclassing `"Linux:Firefox"`. By not doing so, the model would become biased towards more prominent platforms, not taking the smaller ones into account.

5.3.4 Negative Samples

We select two random fingerprints from the same platform for each positive sample and compare them to create a negative sample. We skip fingerprints belonging to the same user alias. Even though the devices have different device tags, they are likely originate from the same device.

5.4 Dataset Analysis

5.4.1 Entropy

The Shannon's entropy [19] is defined by the following equation:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log p(x_i) \tag{5.1}$$

Dividing the Shannon's entropy by the maximum entropy $\log_b(n)$ results in normalized entropy (efficiency) [20]. Efficiency is a value between 0 and 1; the higher the value, the higher the diversity index.

$$H_n(X) = -\sum_{i=1}^{n} \frac{p(x_i) \log_b p(x_i)}{\log_b n} \tag{5.2}$$

Table 5.2: Entropy of the AudioContext fingerprint features.

Feature	Shannon Entropy	Efficiency
support	0.01	0.00
buffer sum	2.66	0.13
properties	1.02	0.05
All	**3.31**	**0.17**

By calculating the entropy of each fingerprint feature, we can quantify how beneficial the collected AudioContext fingerprints are for the model. Table 5.2 shows that the entropy of all features is `3.31`, giving us a efficiency of `0.17`. The buffer sum has an entropy of `2.66`, thus making it more effective than the method presented by Yinzhi Cao et al. [21], but also less effective than calculating the entropy of the whole wave; Englehardt et al. [16] claim it to be 5.4. The entropy of property features shown in the Table 5.3 equals to `1.02`, showing that most features from this method are not valuable.

Table 5.3: Entropy of the AudioContext property fingerprint features.

Feature	Shannon Entropy	Efficiency
base latency	0.00	0.00
sample rate	0.66	0.03
max channel count	0.40	0.02
number of inputs	0.01	0.00
number of outputs	0.01	0.00
channel count	0.01	0.00
channel count mode	0.01	0.00
channel interpretation	0.01	0.00
All	**1.02**	**0.05**

5.4.2 Anonymity Sets

By dividing each platform into anonymity sets, we could visualize how each platform's data look. Each anonymity set represents devices with the same fingerprint. If the fingerprint is unique, we do not want it to be in a separate set containing just one record. Instead, we group

all these unique fingerprints to a set called *unique_devices*, telling us how many AudioContext fingerprints are unique between all devices.

Figure 5.2 shows the anonymity sets of `"Mac OS X:Chrome"` platform. The *unique_devices* set represents only a 4.43% of all fingerprints, while the /textttdark blue anonymity set shows that more than a third of all fingerprints are the same. Other platforms have an even smaller set of unique AudioContext fingerprints, implying that the AudioContext fingerprinting method cannot work independently.

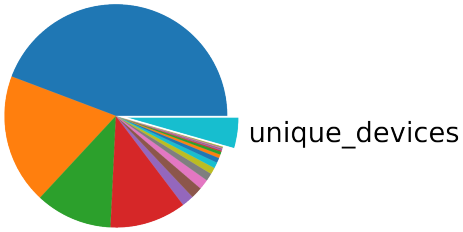


Figure 5.2: Distribution of anonymity sets recorded on "Mac OS X:Chrome" platform.

5.4.3 Most Common Values

The processing of the dataset also showed which data are valuable or not. We achieved this simply by counting the occurrences of all values of properties of the AudioContext fingerprints we collected. Among these are properties of AudioContext *numberOfInputs*, *numberOfOutputs*, *channelCountMode*, and *channelInterpretation*. All these properties had a uniform value among the 226 846 fingerprints.

5.4.4 Over-the-time Stability

We found out that only around `0.35%` of the AudioContext fingerprints tend to change over 90 days. Figure 5.3 shows which features of the fingerprint change, showing that the most prone to change is the *buffer sum* feature at `0.25%` of changed values over 90 days. These findings tell us that the AudioContext fingerprints have good over-the-time stability.

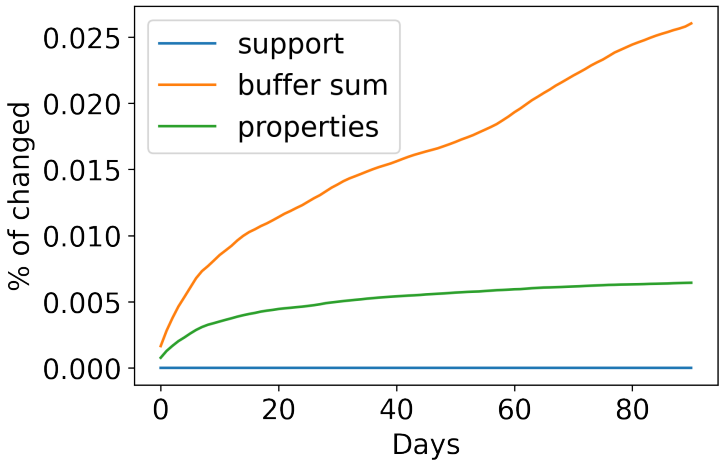


Figure 5.3: Change rate over of AudioContext fingerprint features in the span of 90 days.

6 Model

The machine learning model provided by ThreatMark is a neural network model built using the Keras on top of TensorFlow [22]. No changes were made to the supplied model.

The model is a relatively simple regression model that consists of three layers. The first layer is a *Dense* [1] layer with a `"relu"` [23] activation function. There we also specify the input shape with the *input_dim* argument, giving the argument a sample of the dataset to learn the format of the data. The layer mentioned before is followed by a *Dropout*[2] layer, applying dropout to the input. In our case, the layer is configured to have a dropout of rate of `0.1`. The last layer is a another Dense layer, set with activation parametr of `"sigmoid"`, resulting in onedimensional output space. Table B.2 shows the summary of the layers.

The model returns a value between `0` and `1`. The closer the returned value is to `0`, the more confident the model is that the fingerprints are different, and conversely, the closer the value is to `1`, the fingerprints should originate from the same device.

6.1 Training

The training configuration of the model goes as follows. The optimizer is set to `"adam"`[3] to utilize the *Adam* algorithm. The loss function attribute is set to `"binary_crossentropy"`[4], computing the cross-entropy loss between true labels and predicted labels. The metric is set to `"accuracy"` as we aim to calculate how often the predictions equals the labels.

We retrained the model using a dataset with AudioContext data only and a dataset containing all fingerprint data enriched by AudioContext fingerprint. By doing so, we obtained two models, which

1. `https://keras.io/api/layers/core_layers/dense/`
2. `https://keras.io/api/layers/regularization_layers/dropout/`
3. `https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam`
4. `https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryCrossentropy`

weights we exported. The weights provide us with a plug-in functionality because we can switch the model weights directly in the product when a better model is trained. If we change the diff function by adding new features, we only need to retrain the model and export the weights. To plug the model in, we export the weights from the Keras model ($\mathbf{W}_1$, $\mathbf{b}_1$, $\mathbf{W}_2$, $\mathbf{b}_2$) and compute the prediction p from input $\mathbf{i}$ as:

$$p = \frac{1}{1 + \exp(-\mathbf{o}_2)} \tag{6.1}$$

where $\mathbf{o}_2 = \mathbf{o}_1 \cdot \mathbf{W}_2 + \mathbf{b}_2$ and $\mathbf{o}_1 = relu(\mathbf{i} \cdot \mathbf{W}_1 + \mathbf{b}_1)$.

6.2 Evaluation

When evaluating, we use the original model provided by ThreatMark as a baseline against which we compare the newly retrained model. One of the main goals of this work is to measure if adding the diff vector of AudioContext fingerprint results in an increase or decrease in the model's accuracy.

We also evaluated the *AudioContext-only* model, which is a model trained only using a diff vector of AudioContext fingerprint. The only reason we trained a model using only AudioContext fingerprints is that we cannot share complete details of the ThreatMarks fingerprinting solution. By training this model, we obtained demonstration material showing the process of how the real model was trained without giving away the corporate know-how.

6.2.1 Receiver Operating Characteristic

First, we used the Receiver Operating Characteristic Curve (ROC Curve) and Area Under the ROC Curve (UAC) [24]. The curve plots True Positive Rate (TPR),

$$TPR = \frac{TP}{TP + FN} \tag{6.2}$$

and False Positive Rate (FPR).

$$FPR = \frac{FP}{FP + TN} \tag{6.3}$$

Table 6.1: Confusion matrix used to for the model evaluation.

		Actual Values	
		Positive (1)	Negative (0)
Predicted	Positive (1)	TP	FP
	Negative (0)	FN	TN

True-positive means that the prediction is correct, and true-negative means that the returned value does not reflect the truth. The same logic applies to false-positive and false-negative records, as shown in Table 6.1.

Table 6.2 shows that the AUC of the baseline model is `0.99675`. By retraining the model with added AudioContext diff vector, we achieved an AUC of `0.99722`, thus making this a measly **0.05%** increase of the prediction accuracy of the model. Also, the table shows the AUC of the AudioContext-only being only `0.71136`, which is slightly more than what we expected.

As the ROC curve is not the best tool to visually present this slight evolution, we had to use other tools described in the following section.

Table 6.2: Area Under the ROC Curve of the models.

Model	AUC
Original + AudioContext	`0.9972200412140385`
Baseline	`0.9967575204400178`
AudioContext only	`0.7113697186496813`

6.2.2 Detection Error Tradeoff and Equal Error Rate

Detection Error Tradeoff (DET) [25] curves can give us more direct feedback on the detection error tradeoff. To obtain the DET Curve, we need to plot the False Acceptance Rate (FAR) and False Rejection Rate (FRR).

Figure 6.1 shows us the two DET curves of the baseline model and the newly trained model. The curve of the newly trained model does tend to go just below the curve of the baseline model while coping the shape with almost perfect precision. In conclusion, these DET curves show that the increase in accuracy is negligible and does not show any indicative information on how we could better integrate the added AudioContext diff vector into the model.

Equal Error Rate (EER) [26] describes a point where the FRR and FAR are equal, in other words, where they intersect. By multiplying this value by 100, we obtain the model's error rate percentage. The EER tells us how likely the model is to predict two same fingerprints as different and two different fingerprints as the same.

The EER of the baseline model and the newly trained model is also shown in Figure 6.1. By adding the AudioContext fingerprint diff vector to the model, the EER of the model decreased from 1.94678% to 1.79557% (Table 6.3). This slight decrease shows that the added vector is not useless, but the added value is insignificant.

Table 6.3 and Figure C.1 also show that the AudioContext-only model has a terrible EER of 45.7652%, yet again proving that the model can not function on its own.

Table 6.3: Equal Error Rate (EER) of the models.

Model	EER
AudioContext only	45,7652%
Baseline	1,94678%
Original + AudioContext	1,79557%

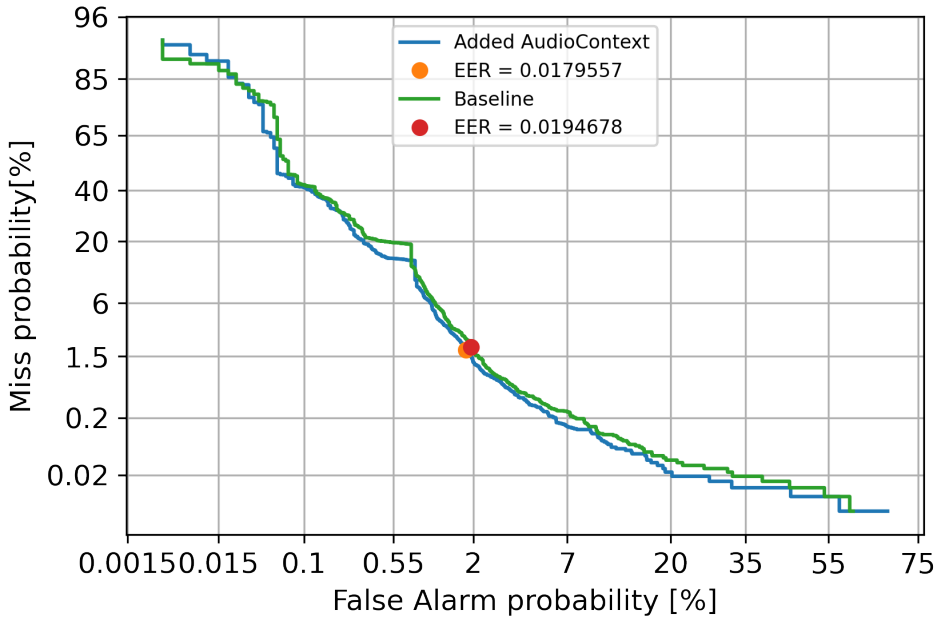


Figure 6.1: DET curves and EER of the baseline model and the model with added AudioContext fingerprints diff vector.

6.3 Validation

We validated only the model trained on the full dataset because the evaluation of the model utilizing only the AudioContext fingerprints showed, as expected, to be unusable.

6.3.1 Manual Validation

To validate that the model works, we selected fingerprints from two platforms, those being `"iOS:Safari"` and `"Windows:Firefox"`. We then filtered keys with at least six (could not find more in the dataset) distinct fingerprints to have a decent dataset. To achieve this, we had to select fingerprints originating only from Windows 10 and iOS version 12.4.9 because not too many new fingerprints were created during the three months creation period of our dataset. Then the data were redirected to two groups, one for the same device comparison and

the other for a different device comparison. We found four suitable combinations of device tags and user aliases and let the model make the predictions. After manual validation of the result (predictions), the model predicted values of the model trained on full dataset corresponded to the truth.

7 Conclusion

In this thesis, we expanded ThreatMark's device identification solution by adding an AudioContext browser fingerprint to their product AFS.

We analyzed the methods currently used to obtain browser fingerprints on the web. We discussed why browser fingerprinting happens and the methods used to obtain these browser fingerprints. The main aim was to analyze the methods used to obtain the AudioContext fingerprint. We created an overview of these methods from which we choose the most fitting ones for our use case. We also pointed out possible issues that might occur when using these fingerprinting methods.

We created a JavaScript snippet to obtain the AudioContext fingerprint from the web browsers. Also, we designed the data format of how the fingerprint will be handled in the backend application. The codes were then successfully connected to the AFS.

After releasing the new version of AFS containing the AudioContext features, the solution was deployed to the production environment of SLSP, from where we obtained the dataset used to train the prediction models. We used the dataset to train two models, one of them for the AFS and the second for presentation purposes, as we could not share the details regarding the whole dataset. We analyzed the dataset in great detail, calculating the entropy and anonymity sets. The analysis showed that the AudioContext fingerprint does not provide enough unique information but is relatively stable in time.

We evaluated the models using metrics such as AUC and EER, showing a slight increase of just 0.05% in the model's accuracy and a slight decrease in EER after adding the AudioContext diff vector.

7.1 Future Work

7.1.1 Better properties selection

The entropy measurements and finding the most common values showed that specific properties we collect have almost no value for the fingerprinting model. By removing these from the fingerprint collection, we could reduce the size of the fingerprint message HTTP request,

thus reducing the traffic size for users. Also, reducing the fingerprint size would save some database space for AFS's other features.

7.1.2 Other Fingerprinting Methods

By implementing the fingerprinting method presented by Englehardt et al. [16] we could have a method with higher entropy. The downside of this method is that we would need to send all the wave values to the backend with an adequately trained model or generate a hash of these values. The first option dramatically increases the size of the fingerprint message, and the second would not be much more useful than our current solution.

7.1.3 Privacy Changes

As browser manufacturers are trying to protect their users' privacy, we must consider that these methods might become obsolete. Privacy-based browsers such as Brave[1] are known to have already implemented some protection against AudioContext fingerprinting [14]. Brave uses a method called *farbling* to randomize the audio samples returned by the audio context stack, thus making methods such as *buffer sum* unusable. Notably, attempts such as FPGuard [27] were made to protect the users from being fingerprinted, although they showed mixed results.

1. `https://brave.com/`

Listings

Bibliography

1. BIMBOT, Frédéric et al. A Tutorial on Text-Independent Speaker Verification. *EURASIP Journal on Advances in Signal Processing*. 2004, vol. 2004, pp. 1–22.
2. GABRYEL, Marcin; KOCIĆ, Milan. Fingerprint Device Parameter Stability Analysis. In: RUTKOWSKI, Leszek; SCHERER, Rafał; KORYTKOWSKI, Marcin; PEDRYCZ, Witold; TADEUSIEWICZ, Ryszard; ZURADA, Jacek M. (eds.). *Artificial Intelligence and Soft Computing*. Cham: Springer International Publishing, 2021, pp. 464–472. ISBN 978-3-030-87897-9.
3. MAYER, Jonathan R. "Any person... a pamphleteer": Internet Anonymity in the Age of Web 2.0. *Undergraduate Senior Thesis, Princeton University*. 2009, vol. 85.
4. LAPERDRIX, Pierre; BIELOVA, Nataliia; BAUDRY, Benoit; AVOINE, Gildas. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)*. 2020, vol. 14, no. 2, pp. 1–33.
5. ECKERSLEY, Peter. How Unique Is Your Web Browser? In: ATALLAH, Mikhail J.; HOPPER, Nicholas J. (eds.). *Privacy Enhancing Technologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–18. ISBN 978-3-642-14527-8.
6. FIELDING, Roy; GETTYS, Jim; MOGUL, Jeffrey; FRYSTYK, Henrik; MASINTER, Larry; LEACH, Paul; BERNERS-LEE, Tim. *Hypertext transfer protocol–HTTP/1.1*. RFC 2616, 1999.
7. UPATHILAKE, Randika; LI, Yingkun; MATRAWY, Ashraf. A classification of web browser fingerprinting techniques. In: *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*. 2015, pp. 1–5. Available from DOI: 10.1109/NTMS.2015.7266460.
8. DABROWSKI, Adrian; MERZDOVNIK, Georg; ULLRICH, Johanna; SENDERA, Gerald; WEIPPL, Edgar. Measuring Cookies and Web Privacy in a Post-GDPR World. In: CHOFFNES, David; BARCELLOS, Marinho (eds.). *Passive and Active Measurement*. Cham: Springer International Publishing, 2019, pp. 258–270. ISBN 978-3-030-15986-3.

9. BRIJAIN, Mr; PATEL, R; KUSHIK, MR; RANA, K. A survey on decision tree algorithm for classification. 2014.

10. GÓMEZ-BOIX, Alejandro; LAPERDRIX, Pierre; BAUDRY, Benoit. Hiding in the Crowd: An Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In: *Proceedings of the 2018 World Wide Web Conference*. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 309–318. WWW '18. ISBN 9781450356398. Available from DOI: 10.1145/3178876.3186097.

11. *Company: ThreatMark: Anti-fraud and Threat Detection Experts*. 2021. Available also from: https://www.threatmark.com/company/#.

12. *ThreatMark Anti-Fraud Suite (AFS)*. 2021. Available also from: https://www.threatmark.com/products/anti-fraud-suite-afs/.

13. JAKUBICEK, Lukas. *Threatmark listed as a representative vendor in Gartner's Market Guide for Online Fraud Detection*. 2020. Available also from: https://www.threatmark.com/representative-vendor-gartners-market-guide-online-fraud-detection/.

14. COPLAND, Savannah. *How the web audio API is used for audio fingerprinting*. FingerprintJS, 2021. Available also from: https://fingerprintjs.com/blog/audio-fingerprinting.

15. *Sustainability of digital formats: Planning for Library of Congress Collections*. 2008. Available also from: https://www.loc.gov/preservation/digital/formats/fdd/fdd000011.shtml.

16. ENGLEHARDT, Steven; NARAYANAN, Arvind. Online Tracking: A 1-Million-Site Measurement and Analysis. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1388–1401. CCS '16. ISBN 9781450341394. Available from DOI: 10.1145/2976749.2978313.

17. CHROMIUM SOURCE CODE, François BeaufortDives into. *Autoplay policy in Chrome*. Google. Available also from: https://developer.chrome.com/blog/autoplay/#webaudio.

18. *Models*. Available also from: https://pydantic-docs.helpmanual.io/usage/models/#private-model-attributes.

19. RÉNYI, Alfréd. On measures of entropy and information. In: *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*. 1961, vol. 4, pp. 547–562.

20. KUMAR, Uma; KUMAR, Vinod; KAPUR, J N. Normalized measures of entropy. *International Journal Of General System*. 1986, vol. 12, no. 1, pp. 55–69.

21. CAO, Yinzhi; LI, Song; WIJMANS, Erik. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In: *NDSS*. 2017. Available from DOI: 10.14722/ndss.2017.23152.

22. *Tensorflow*. Available also from: https://www.tensorflow.org/.

23. HARA, Kazuyuki; SAITO, Daisuke; SHOUNO, Hayaru. Analysis of function of rectified linear unit used in deep learning. In: *2015 International Joint Conference on Neural Networks (IJCNN)*. 2015, pp. 1–8. Available from DOI: 10.1109/IJCNN.2015.7280578.

24. *Classification: Roc curve and AUC | machine learning crash course | google developers*. Google. Available also from: https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc.

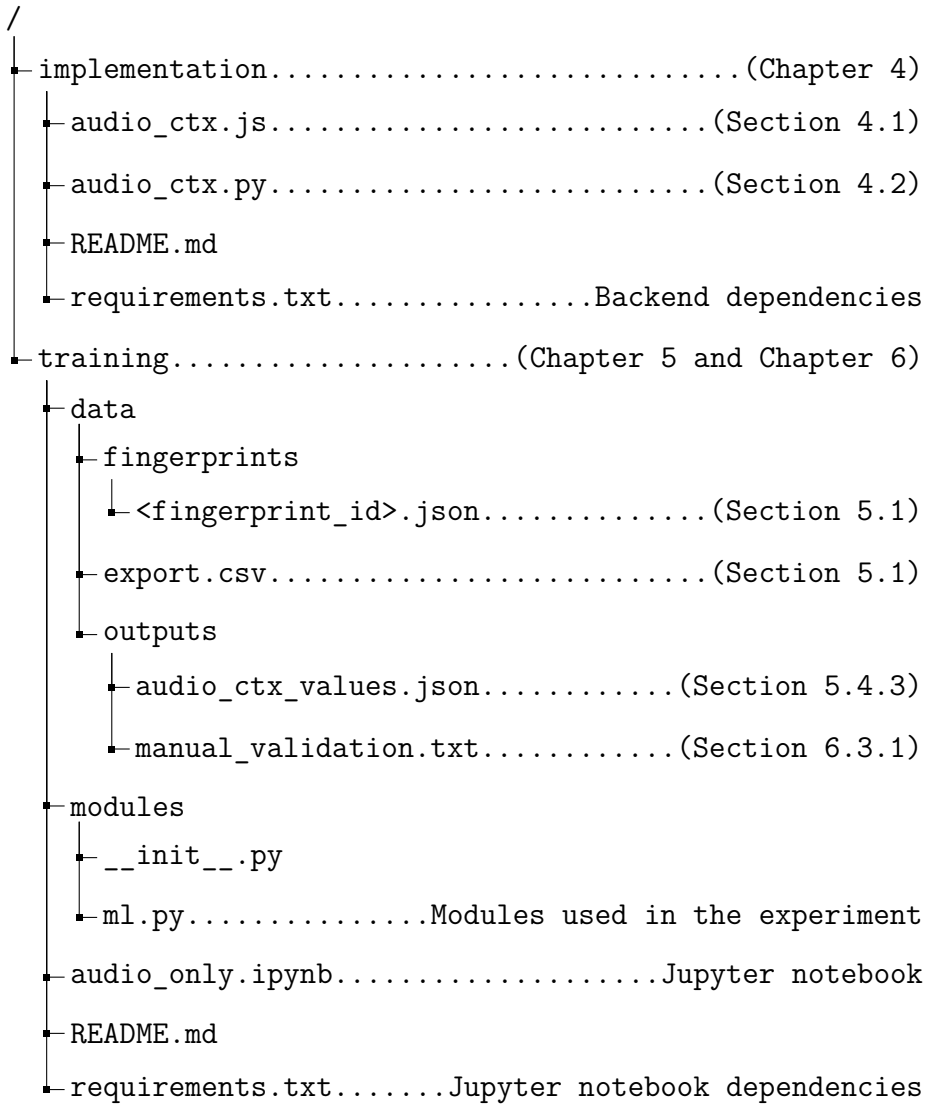
25. MARTIN, Alvin; DODDINGTON, George; KAMM, Terri; ORDOWSKI, Mark; PRZYBOCKI, Mark. *The DET curve in assessment of detection task performance*. 1997. Technical report. National Inst of Standards and Technology Gaithersburg MD.

26. BRUMMER, Niko. *Measuring, refining and calibrating speaker and language information extracted from speech*. 2010. PhD thesis. Stellenbosch: University of Stellenbosch.

27. FAIZKHADEMI, Amin; ZULKERNINE, Mohammad; WELDEMARIAM, Komminist. FPGuard: Detection and Prevention of Browser Fingerprinting. In: SAMARATI, Pierangela (ed.). *Data and Applications Security and Privacy XXIX*. Cham: Springer International Publishing, 2015, pp. 293–308. ISBN 978-3-319-20810-7.

A Source code

The attached archive is split into two parts. The `implementation` directory contains code that was used for data collection and is now utilized in AFS. The `training` directory consists of an example dataset used for the model training and an example Jupyter notebook executed only on AudioContext fingerprints. Both `README` files describe each section in greater detail.



B Tables

Table B.1: Example of default, minimal, and maximal values of DynamicsCompressorNode.

Property	Default value	Minimal value	Maximal value
threshold	-24	-100	0
knee	30	0	40
ratio	12	1	20
reduction	0	-20	0
attack	0.003	0	1
release	0.25	0	1

Table B.2: Summary of the machine learning model layers and its parameters.

Layer name (type)	Output Shape	Param #
dense (Dense)	(None, 75)	750
dropout (Dropout)	(None, 75)	0
dense_1 (Dense)	(None, 1)	76

Total params: 826; Trainable params: 826; Non-trainable params: 0

C Figures

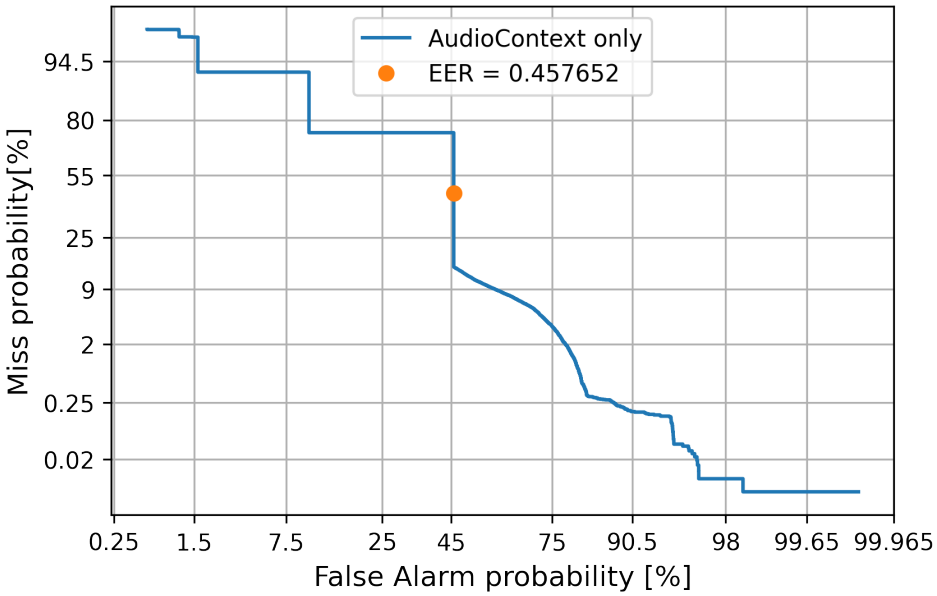


Figure C.1: DET curve and EER of the AudioContext-only model.