Browser Polygraph: Efficient Deployment of Coarse-Grained Browser Fingerprints for Web-Scale Detection of Fraud Browsers

Faezeh Kalantari
faezeh.kalantari@asu.edu
Arizona State University
Tempe, AZ, USA

Mehrnoosh Zaeifi
mzaeifi@asu.edu
Arizona State University
Tempe, AZ, USA

Yeganeh Safaei
ysafaeis@asu.edu
Arizona State University
Tempe, AZ, USA

Marzieh Bitaab
mbitaab@asu.edu
Arizona State University
Tempe, AZ, USA

Adam Oest
contact@adamoest.com
Amazon
Tempe, AZ, USA

Gianluca Stringhini
gian@bu.edu
Boston University
Boston, MA, USA

Yan Shoshitaishvili
yans@asu.edu
Arizona State University
Tempe, AZ, USA

Adam Doupé
doupe@asu.edu
Arizona State University
Tempe, AZ, USA

Abstract

In this paper, we address the prevalent issue of *account takeover* (ATO) fraud, which significantly impacts businesses through stolen user information. Websites have adopted *risk-based authentication*, incorporating browser fingerprinting techniques to counteract this threat. However, attackers have adapted by using anti-detect browsers, referred to as *fraud browsers*, to spoof user information effectively. While traditional fingerprinting methods are capable of identifying fraud browsers, they encounter scalability and performance challenges in risk-based systems. To address these issues, we developed Browser Polygraph, an ML-based tool that applies coarse-grained privacy-preserving fingerprints to assess browser authenticity and assigns *risk factors* to suspicious sessions. Coarse-grained fingerprints, by design, cannot be used for user tracking but only for fraud detection purposes. Deployed at a major financial company, Browser Polygraph has flagged suspicious sessions, enabling more targeted identification of potential fraud, thus enhancing the company's ability to tackle ATO attempts.

CCS Concepts

• **Security and privacy → Browser security**.

Keywords

Coarse-grained browser fingerprinting, Fraud browsers, Fraud detection

IMC '24, November 4–6, 2024, Madrid, Spain

ACM ISBN 979-8-4007-0592-2/24/11
https://doi.org/10.1145/3646547.3688455

ACM Reference Format:
Faezeh Kalantari, Mehrnoosh Zaeifi, Yeganeh Safaei, Marzieh Bitaab, Adam Oest, Gianluca Stringhini, Yan Shoshitaishvili, and Adam Doupé. 2024. Browser Polygraph: Efficient Deployment of Coarse-Grained Browser Fingerprints for Web-Scale Detection of Fraud Browsers. In *Proceedings of the 2024 ACM Internet Measurement Conference (IMC '24), November 4–6, 2024, Madrid, Spain.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3646547.3688455

1 Introduction

The war between fraudsters and website operators has been ever-present since the first login form was conceived. Fraudsters use any means necessary to steal cookies or login information from victims, and then use that stolen information to authenticate to the website as the victim [38, 62, 68]. Such attacks are called *account takeover fraud* or ATO [34]. ATOs cause significant financial damage—the FBI reports that victims reported 2.7 billion USD in losses due to Business Email Compromise (which is a fraud that is just a subset of ATO) in the year 2022 alone [16].

One way that website operators have attempted to protect users is through *risk-based authentication*—analyzing available information about the login attempt to assess the risk of it being an in-progress ATO attempt. One such technique is the matching of the user's current browser against prior browsers used by that account. Freeman et al. [17] used the `user-agent` to identify the browser, and follow-up work has leveraged the burgeoning field of browser fingerprinting (originally identified as a privacy threat to users, but, in this case, utilized "for good") [9, 13, 26, 45].

However, recent work shows that attackers can easily steal and replay browser fingerprints [29, 31]. In fact, criminals have turned this into a type of cybercrime-as-a-service [10]: When account information is stolen, the attacker also collects a browser profile. Underground marketplaces such as the Genesis Market (through which over 80 million stolen profiles for various services of over 2 million potentials were traded before the site was shut down by the FBI in 2023 [41]) facilitate the sale of these profiles on a massive scale. After acquiring stolen profiles, fraudsters load them

into specially-customizable web browser variants, so-called "anti-detect browsers" ostensibly created for privacy purposes, to bypass risk-based authentication and commit ATO fraud. In this paper, we refer to these browsers as *fraud browsers*. Notably, financial institutions are one of the most impacted sectors, facing numerous cases of fraud perpetrated through *fraud browsers* [3].

Fine-grained browser fingerprinting offers a solution to detecting fraud browsers through its user tracking techniques. These techniques generate detailed fingerprints using advanced browser fingerprinting techniques such as WebGL [11, 35], canvas [1, 15] and even CPU timing [43], which can then be compared with previously established user fingerprints to identify potential fraud. Nevertheless, through our consultation with a leading financial institution, herein referred to as *FinOrg*[1], we found that existing fine-grained fingerprinting methods face deployment challenges at *FinOrg* due to stringent constraints on fingerprint extraction time and data size. Current practices for thwarting fraud browsers predominantly rely on manual effort (studied by Azad et al. [7]) and may become obsolete with the release of new browser software.

Addressing these issues, our paper presents a prototype for the analysis of *efficient coarse-grained fingerprints* within large-scale Web traffic, aimed at fraud browsers detection. By coarse-grained browser fingerprints, we refer to a simplified and less detailed set of browser fingerprinting features designed specifically for fraud detection, optimized to meet resource and performance constraints (as defined by FinOrg). These fingerprints, in our paper, target discrepancies in the implementation of JavaScript APIs across different browsers. This novel, data-driven, and privacy-preserving strategy focuses on training a machine learning model via minimal coarse-grained features from *FinOrg* users. Our method aims to rigorously validate the authenticity of a browser session's reported `user-agent` string, which fraudsters typically manipulate to match their target's `user-agent`. We extend existing research in coarse-grained fingerprinting [6, 65], which are typically used to infringe user privacy, and instead apply our extended techniques to large-scale Web data, with a particular focus on privacy, coarse granularity, efficacy, and continuous detection adaptability.

In our prototype, we create a clustering model, called Browser Polygraph, that operates on privacy-preserving and coarse-grained fingerprints and determines a *risk factor* score that represents divergences between the actual identity of the observed browser and its identity as claimed by its `user-agent`. Browser Polygraph is tailored to meet the performance demands of large, real-world, financial companies, focusing on efficient and effective browser fingerprinting without delving into expensive, fine-grained analysis.

We deployed Browser Polygraph in production at FinOrg for 4.5 months, observing over 205,000 user sessions. Of these, Browser Polygraph identified 897 suspicious sessions, 2% of which were used in an ATO attack within 72 hours (compared an ATO prevalence of 0.43% across all sessions). Of the sessions that Browser Polygraph identified as *highly* risky, 5.83% were used in an ATO within 72 hours.

We designed Browser Polygraph with concept drift in mind, and built in mechanisms to detect when retraining is needed (e.g., due to new browser versions). We experimentally show that retraining was needed approximately after three months. This duration may vary depending on the specific features of new browser releases, but it can be automatically triggered by Browser Polygraph's drift detection module, increasing the resiliency of our approach in the cat-and-mouse game between attackers and defenders. Furthermore, in a separate evaluation, we assessed Browser Polygraph's efficacy in detecting a subset of fraudulent browser fingerprints generated by fraud browsers, demonstrating its potential in identifying these threats. In essence, Browser Polygraph is a preliminary prototype that represents a significant step forward in the automatic generation of coarse-grained browser fingerprints and their deployment in risk-based authentication systems. Our approach can be further enhanced by future advances in fine-grained browser fingerprinting, ensuring continued efficacy in the face of evasion by cyber criminals.

In summary, the contributions of this paper are as follows:

- We propose a new privacy-preserving idea of detecting lying browsers while maintaining the performance characteristics necessary for real-world deployment at FinOrg using efficient coarse-grained fingerprints.
- We leverage this idea into a clustering model, called Browser Polygraph, that outputs `risk factor` of suspicious user sessions, which can be used as part of risk-based authentication.
- We deployed Browser Polygraph at a major financial company FinOrg, where we demonstrated its effectiveness at detecting real-world fraud.
- We demonstrated Browser Polygraph's effectiveness (in an independent evaluation) in detecting a subset of fraudulent browser fingerprints created by fraud browsers.

2 Background

This paper builds on past research in browser fingerprinting, repurposing it to detect fraud browsers. In this section, we discuss related work in both areas and present the results of our investigation on the behaviors of fraud browsers.

2.1 Browser Fingerprinting

Browser fingerprinting research extensively probes the attributes and features of browsers that can divulge details about a user's underlying system—both software and hardware. These attributes and features can be used to uniquely identify the user's browser. JavaScript accessible attributes such as the `user-agent` [14], geolocation, timezone, screen resolution [8], and available fonts [2, 14, 15] have been classic sources of fingerprintable data. Recent advancements have added depth to this field by exploring more intricate parameters; for instance, APIs such as Canvas [1, 15], WebGL [11, 35], and WebRTC [15] have been explored. Beyond these, even nuanced elements such as AudioContext [15], CSS properties [47], specific JavaScript objects such as `navigator` and `screen` [36], Crypto API [43], the list of extensions and plugins [27, 28, 46, 63], and HTTP headers [27] serve as potential fingerprinting vectors.

[1]FinOrg is a major financial company that provides financial services through internet platforms.

2.2 Fraud Browsers

Privacy concerns have given rise to specialized browsers, commonly referred to as *anti-detect browsers*, designed to obscure user identity. These browsers allow a user to distort or hide a browser's identifying information, such as IP addresses, user-agents, JavaScript attributes, etc. However, these browsers are being actively misused by criminals—not for privacy reasons, but for conducting fraudulent activities. Therefore, we use the term *fraud browsers* in this paper to describe these browsers. Fraud browsers allow criminals to load profiles of the victim, typically stolen by phishing kits [62], and then later sold on online marketplaces [10]. Some marketplaces even have their own fraud browsers, such as the now-defunct Genesis Market.

Since addressing the challenges presented by fraud browsers is a vital element of our research, we examined several notable fraud browsers employed today. This examination revealed a commonality in their adeptness at customizing, randomizing, or concealing typical fingerprinting attributes, such as Timezone, Language, Screen Resolution, WebRTC, WebGL, Canvas, Audio Context, Geo Location, etc. Specifically, we analyzed browsers like, ClonBrowser [53], AntBrowser [50], Incogniton [56], GoLogin [55], Octo Browser [57] Linken Sphere [61], Sphere [58], VMLogin [59] CheB rowser [51], and AdsPower [48], noting usage of some of those in real-world fraud.

The uniqueness of each fraud browser lies in its functionality. For instance, Linken Sphere offers extensive customization, including browser or OS choice, fingerprint profiles, and cookie import capabilities. While offering fewer browser and OS customization options, Che Browser provides a unique service that allows users to purchase profiles for various Chrome versions. Incogniton and Octo Browser, both Chrome-based, support user profiles and cookie imports. GoLogin provides a wide range of operating system options and the capability to manage from 100 to thousands of user profiles. Our focus diverged from these software's extensive capabilities towards a more targeted approach. We concentrated on uncovering inconsistencies within the simplest and most direct JavaScript attributes rather than comparing their sophisticated spoofing or randomization techniques (details in Section 6).

2.3 Fraud Browsers' Behavior

We further delved into the behavior of fraud browsers to understand how they operate so that we could create strategies to detect them. We installed several fraud browsers (noted in Table 1) on a Windows machine and analyzed how their browser fingerprints change with altering the `user-agent`. This led to the categorization of these browsers into three primary strategies, with a fourth category identified for fraudulent use of legitimate browsers in spoofed environments:

Category 1: Browsers such as Linken Sphere and ClonBrowser which exhibit browser fingerprints that do not match any legitimate browser's fingerprint.
Category 2: Browsers in this category maintain a browser fingerprint that matches a legitimate browser fingerprint, but the fingerprint does not change when modifying the `user-agent` string.

Table 1: Fraud browsers categorized based on definitions in Section 2.3. Release dates with a preceding ~ are approximations based on related version release dates or the date of experimentation (Rel. = Release).

Browser	Rel. Date	Category	New Rel.?
Linken Sphere-8.93	April 2022	1	✗
ClonBrowser-4.6.6	~May 2023	1	✓
Incogniton-3.2.7.7	~May 2023	2	✓
Gologin-3.2.19	May 2023	2	✓
CheBrowser-0.3.38	~May 2023	2	✓
VMLogin-1.3.8.5	April 2023	2	✓
Octo Browser-1.10	September 2023	2	✓
Sphere-1.3	November 2023	2	✗
AntBrowser	May 2023	2	✗
AdsPower-4.12.27	December 2022	3	✓
AdsPower-5.4.20	April 2023	3	✓

Category 3: Browsers that adopt the browser's JavaScript engine and its browser fingerprint with each `user-agent` selection.
Category 4: Legitimate browsers (with same `user-agent` and fingerprint) used in spoofed environments to load stolen information.

Table 1 details the investigated fraud browsers and their categorization. Category **(1)** and **(2)** browsers are prevalent and exhibit detectable inconsistencies, making them susceptible to identification through coarse-grained browser fingerprints. Category **(3)** and **(4)** browsers (likely employed by more resourceful attackers), however, aim to approximate recreating a user's entire browser environment (and a genuine fingerprint), requiring sophisticated fraud detection techniques. As noted by Azad et al. [7], these precise spoofs often result in configurations that are challenging to distinguish from legitimate usage. Therefore, Browser Polygraph focuses on detecting Categories **(1)** and **(2)** browsers using coarse-grained fingerprinting, given its resource efficiency and practicality for our Web-scale browser authentication goals that we will explain in the next section.

3 Web-Scale Fingerprinting Requirements

We collaborated with FinOrg to understand their performance requirements based on the technical requirements of a real-world risk system deployed in a significant-traffic web application. This led to two core requirements: (1) fast response time, operating within 100 milliseconds, and (2) data extracted per-user should be minimal, under the threshold of one kilobyte.

Therefore, with these thresholds in mind, we evaluated the performance overhead of the state-of-the-art fingerprinting techniques mentioned in Section 2.1. Specifically, we evaluated the performance overhead of three prominent fine-grained fingerprining tools: FingerprintJS [54], AmIUnique [49], and ClientJS [52]. FingerprintJS and ClientJS were chosen due to their significant market share (as reported by Wappalyzer [60]) in browser fingerprinting. AmIUnique is an academic contribution from Laperdrix et al. [27].

Our analysis specifically focused on evaluating these tools' response times and data processing overhead. We calculated the average service time over five visits for each tool to assess response times accurately. In refining our analysis of storage needs for fraud detection, we shifted focus from the size of hashed data to the underlying data structure's size, which is crucial for hashing. This distinction is vital as our interest lies not in the hash values but in verifying whether a browser's specifics align with its claimed `user-agent`.

For FingerprintJS, we utilized their online tool for direct performance metrics. AmIUnique's data was gathered via its browser extension. ClientJS required a different approach; as an npm library, we embedded it into a basic Node.js application to produce an HTML page invoking the fingerprinting library. Before hashing, we logged the necessary data to the console to evaluate its storage requirements.

Table 2 demonstrates our evaluation results. While AmIUnique offers extensive fingerprint data collection through its browser extension, its service time and storage needs do not satisfy the requirements. FingerprintJS and ClientJS, although fast, fail to meet the data storage requirement. These results are in line with expectations, as fine-grained fingerprinting, which is required for uniquely identifying a specific user's browser, inherently requires extracting a significant amount of data.

Given the performance overhead of current fingerprinting techniques, particularly given the FinOrg requirements for high-traffic scenarios and real-time security requirements, we propose Browser Polygraph. Browser Polygraph is guided by two principal criteria:

(1) **Minimizing latency:** Targeting efficiency, Browser Polygraph is optimized to operate within real-time security constraints.
(2) **Minimizing data size:** It is designed to function effectively under stringent data size limitations.

The comparison in Table 2 underscores our approach. We aim for rapid response times akin to FingerprintJS and ClientJS while significantly reducing the fingerprint data size. This balance makes Browser Polygraph an optimal solution for Web-scale fraud detection, tailored for the stringent requirements of modern digital security landscapes, addressing both efficiency and scalability (we will explain the performance metrics of Browser Polygraph in Section 7.5). It is important to note that the comparison in Table 2 highlights timing and memory overhead as the key factors in the initial stage of fingerprint-based fraud detection, where browser data is collected. With respect to performance, fine-grained techniques are primarily designed for user tracking. However, not all of the detailed information collected by these techniques is necessarily efficient for identifying lying browsers, which is the main focus of the coarse-grained fingerprinting approach (Browser Polygraph) proposed in this paper.

Table 2: Comparison of time and storage requirements for fine-grained fingerprinting tools and our proposed Web-scale data-driven fraud detection solution.

Tool	Avg. service-time	Storage req.
AmIUnique	~1.5s	~60KB
FingerprintJS	51ms	~23KB
ClientJS	37ms	~10KB
Browser Polygraph	6ms	1KB

4 Threat Model

In this paper, we consider an adversary equipped with a victim's credentials (such as username, password, or cookies) and their browser fingerprint, which includes the `user-agent` and any other stolen browser information (which could be bought on the dark web [42]). The attacker's objective is to execute an Account Takeover (ATO) attack using a fraud browser. This adversary has the capability to customize and alter the settings of a chosen fraudulent browser to mimic the victim's fingerprint (the attributes of which have been discussed in Section 2.1). We assume that the attacker will, at a minimum, accurately set the most straightforward and critical attribute for browser fingerprint spoofing—the victim's `user-agent`.

On the defense front, the primary aim of this paper is to introduce a continuous, low-overhead, fingerprinting-based fraud detection mechanism that detects such ATO attacks. This defense mechanism is distinct from risk-based authentication schemes that validate a user's fingerprint upon logins [6, 29] or in response to access requests to sensitive actions. Our approach focuses on persistently monitoring and restricting access of fraud browsing sessions to the FinOrg's infrastructure, thereby minimizing the attack surface available to malicious actors.

5 Overview

In this research, we introduce a prototype that combines efficient coarse-grained browser fingerprinting with machine learning to identify suspicious browsing activity by detecting if a browser is lying about its `user-agent`. Notably, we are specifically *not* exploring new browser fingerprinting techniques. Instead, we focus on using established, yet effective, methods to detect user sessions showing inconsistent behaviors between the claimed browser and the actual browser. Our system is designed to seamlessly adjust to new browser updates, ensuring it remains effective against emerging suspicious browser fingerprints.

Our fingerprinting technique targets discrepancies in browsers' implementation of the JavaScript APIs, taking inspiration from the work of Akhavani et al. [4]. The evolving landscape of JavaScript APIs makes them a fertile ground for such fingerprinting efforts.

First, we generated JavaScript-based fingerprints by collecting fingerprints on many legitimate browser vendors and versions to identify those *candidate JavaScript attributes* that can be used for fingerprinting. During this selection process, our focus was on attributes that could be extracted within the performance constraints outlined in Section 3 and that uphold privacy standards—namely, these attributes should not enable user tracking but should be viable for fraud detection purposes.

However, we needed to identify the discriminatory fingerprintable features on real-world browsers, which we found are varied in number and different configurations than we could manually test. We deployed our candidate fingerprint collection script through our

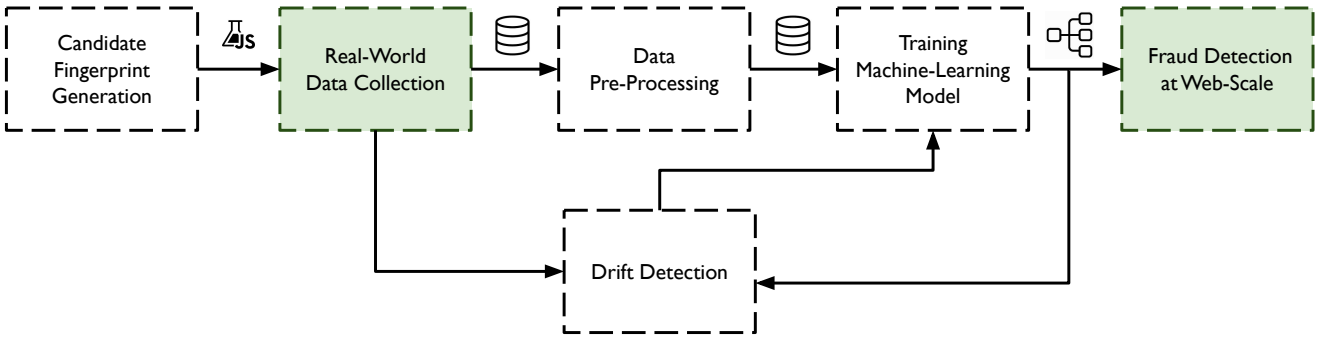

Figure 1: Overview of Browser Polygraph offering fraud detection at Web-scale. Rectangles in green and white colors suggest online vs. offline tasks.

collaboration with FinOrg. After gathering data for several months, we analyzed it, to identify the JavaScript-based fingerprints that were significant for detecting browser versions. The resulting set of coarse-grained browser fingerprints then served as the foundation for our machine learning model. The model's objective is straightforward: to determine if the `user-agent` of a browser matches the expected features of a similar but legitimate browser.

Figure 1 shows a high-level design of our prototype, which includes the following core components:

Candidate Fingerprint Generation. We ran multiple JavaScript fingerprint extractions on legitimate browser versions/vendors and identified *candidate JavaScript attributes* with considerable output variances between browsers.

Real-World Data Collection. We deployed a collection of the candidate fingerprintable JavaScript attributes on FinOrg, and collected this data.

Data Pre-Processing. After processing the collected candidate fingerprintable data, we analyzed it to identify key discriminatory fingerprintable features, to reduce the ultimate size of the collected data.

Training of Machine-Learning Model. After identifying the features, we trained a clustering algorithm. Its primary function is to group browsers together based on how they implement the JavaScript APIs.

Fraud Detection. With the clustering algorithm, Browser Polygraph identifies if the features of a browser (extracted using the coarse-grained fingerprints) map to the same cluster as the browser's `user-agent`. Non-matching is indicative of fraud, and we further calculate semantic similarity between the claimed browser and the fingerprint-matching browser to assign a `risk factor`.

Drift Detection. We can routinely assess the clustering algorithm's ability to correctly assign clusters to new browser versions, initiating retraining as needed to maintain accuracy.

Emphasizing *user experience*, our design minimizes operational disruptions, conducting most processes offline while handling real-time Data Collection and Fraud Detection.

6 Design

Now, we delve deeper into the structural and functional aspects of our fraud detection system Browser Polygraph. Each of the five components, previously highlighted in Section 5, will be examined in detail to offer a comprehensive understanding of their design and operation.

6.1 Candidate Fingerprint Generation

The central objective of this component is the identification of candidate JavaScript fingerprints that are suitable to be evaluated for their usefulness in the subsequent step of Real-World Data Collection. These candidate fingerprints aim to discern variations in the implementation of JavaScript APIs between legitimate browsers. Our emphasis, unlike prior work, is not on pinpointing specific browser versions/vendors but to achieve coarse-grained browser fingerprinting. As our evaluation in Section 7 will show, this granularity is sufficient to detect discrepancies that may emerge during fraud detection.

While the quantity of candidate fingerprints advanced to the subsequent phase is flexible, to collect data from FinOrg, they must align with the timing and data storage requirements outlined in Section 3. Additionally, it was essential that the candidate fingerprints be privacy-preserving and not enable user tracking. Therefore, rather than using all possible candidate fingerprints, we ran automated experiments on browser instances to identify distinguishing candidate fingerprints.

We tested browsers from Chrome-based and Firefox releases from mid-2017 to mid-2022. Notably, the testing environments varied: while we often used BrowserStack, sometimes local installations of browsers were used. Throughout this project, we only needed to perform the candidate fingerprint generation stage once, in mid-2022. Following this, we collected fingerprints for new browser releases as needed using BrowserStack. Overall, we gathered fingerprints from browser instances ranging from Chrome 59–119, Firefox 46–119 to Edge 17,18, and 80–119. In the following, we will explain the process of generating the list of candidate JavaScript fingerprints.

To select the most suitable JavaScript candidate fingerprints, we based the initial list on Akhavani et al.'s BrowserPrint [4], as it discussed browser fingerprinting via JavaScript attributes. BrowserPrint used the existence of a JavaScript attribute as a fingerprintable technique: the idea being that browsers introduce different features at different times, and these can uniquely identify the browser vendor and version. However, we anticipate these features to become less effective over time. Our insight is that, rather than storing the existence of every fingerprintable attribute, which requires not only storage but also continuous updating, we abstract the *shape* of a specific fingerprintable JavaScript prototype by capturing *only the number of properties of the JavaScript prototype*. A set of these are our *coarse-grained fingerprints*.

To determine the JavaScript objects to include in our candidate fingerprints, we prioritized the progression of JavaScript interfaces, assembling a comprehensive list from MDN [33] documentation, which comprised 1006 prototype names, and assessed them using `Object.getOwnPropertyNames` method to count the properties of each prototype. With the properties count serving as our coarse-grained fingerprints, we extracted all possible fingerprints from all candidate browsers, sorted these fingerprints based on their standard deviation across all candidate browsers, and identified the top 200 features from the sorted list (*deviation-based features*). The normalized standard deviation of the selected features ranges from 0.0012 to 1.3853.

As discussed earlier, BrowserPrint [4] utilizes the presence or absence of specific JavaScript properties as fingerprinting APIs, identifying 313 such JavaScript features, which they refer to as suspicious APIs. They analyzed these features for their unique fingerprinting behavior across various browser versions. Because these properties tend to appear or disappear over time, we categorize them as *time-based* features. We chose to integrate these features into our real-world data collection phase. However, since BrowserPrint's analysis covered browser versions up to 2020, and our study focuses on newer browsers released before mid-2022, we anticipated that this list might be outdated for our purposes. As we will demonstrate, only 6 remained relevant and were included in our data pre-processing stage (explained in Section 6.3).

Another lens through which browsers' evolution can be viewed is their varied implementations of certain functions, thus acting as potential fingerprinting indicators. Targeting no-input functions, we analyzed their behavior by navigating the top 100 websites on different browser instances (limited to Firefox and Chrome) using the VisibleJS [23] extension. However, the limited volatility observed in no-input functions led us to ignore those and narrow down our features to 513 JavaScript candidate fingerprints (*deviation-based* and *time-based*) for the subsequent phase (full list in Appendix-3).

6.2 Real-World Data Collection

After developing 513 JavaScript candidate fingerprints, we collaborated with FinOrg to collect coarse-grained candidate fingerprints of real-world browsers. We recorded coarse-grained candidate fingerprints alongside the `navigator.userAgent` attribute. FinOrg integrated our script into one flow of an online shopping platform, with continuous data collection over eight months. During this period, they provided us with periodic datasets, laying the groundwork for our analysis. The data comprised integer outputs from the 513 JavaScript candidate fingerprints, the `navigator.userAgent` string, and a fully anonymized `SessionID`, ensuring users' privacy (as described in Section A). We used four and a half months of this data (comprising 205k rows) for training of our machine learning model. In the following, we detail the process of refining the collected data and the methodology behind the training of our machine learning model.

6.3 Data Pre-Processing

As we delved into the collected data, unexpected patterns surfaced, challenging our initial hypotheses established during the Candidate Fingerprint Generation phase. A particular data sample from the first day of March revealed that 186 features exhibited a singular value across all samples. This was particularly notable among *time-based* features, where 40% displayed unique values, suggesting their diminishing relevance in newer browser versions. Consequently, after thoroughly reviewing the documentation and identifying vendor-specific distinctions, we refined our feature selection, focusing on 6 of *time-based* features that were more indicative of genuine browser behavior.

From analyzing the *deviation-based* features, we noted a significant portion exhibited unique values (30%), leading to their exclusion from further analysis. However, the remaining data revealed inconsistencies in some feature values among identical browser instances. Finding the root cause of these discrepancies is a difficult task: our only data is the coarse-grained fingerprint and a `user-agent`, where the `user-agent` does not match. This could be a fraud instance, or a benign instance, and therefore we conducted an in-depth manual analysis of the wider breadth of legitimate browsers, particularly of Firefox and Chrome browsers and their derivatives. This process uncovered specific facts that influenced those feature values (and we did not initially consider). Some of our notable revelations from real-world data are described in the following:

• **Firefox configuration influence**: Firefox has an array of security and performance configurations via the `about:config` page. Customizing these settings can inadvertently alter feature values. For instance, disabling Service Workers through `dom.serviceWorkers.enabled` can result in values related to `ServiceWorker` interfaces being zeroed. Additionally, manipulating settings such as `dom.element.transform-getters.enabled` can influence properties manifested through the `Element` interface.

• **Chrome extensions & flags**: Chrome's equivalent to Firefox's configuration page is `chrome:flags`. While it mostly pertains to experimental properties—and is arguably less frequently modified by the average user—it was deemed negligible for our study. However, Chrome extensions, which are widespread among users, were observed to impact fingerprinting values. For instance, the DuckDuckGo extension introduced two custom properties to the `Element` interface; this modification increments the integer value of one of our features by two. To assess the influence of common extensions, we revisited the set of default Chrome extensions previously identified by Picazo et al. [39] in 2020. At the time of our

review in April 2024, we found that, of the seven removable extensions reported, Google had integrated the functionality of six into Chrome as built-in apps, leaving only one as a default extension. Notably, the removal of these integrated apps and the remaining extension had no impact on the values of our feature set.

• **Brave**: a privacy-centric browser based on Chromium that could be misidentified as Chrome. Our investigation revealed while the examined version of Brave generated a `user-agent` identical to Chrome 111, there were discernible discrepancies in attribute values across certain interfaces, such as `Element`, compared to the genuine Chrome 111.

• **The Tor Browser**: Mirroring Brave's behavior, the examined version of Tor Browser presented a `user-agent` string aligning with Firefox version 102, yet the attribute values significantly deviated from those of the original Firefox 102. Given that Tor Browser's updates had lagged—being nearly a year behind the contemporary Firefox iteration—we deemed it prudent to exclude this browser instance from our analysis.

Based on this analysis, we adjusted our feature set, particularly for *deviation-based* attributes, some of which could be modified by user/browser privacy settings. We identified and excluded the most affected features to enhance the reliability of our coarse-grained fingerprints. Moreover, a detailed examination of real coarse-grained fingerprints highlighted minimal deviations in certain features, and we excluded those as well. From this process, we selected 22 *deviation-based* features, complemented by 6 *time-based* features. This refined selection forms our comprehensive set of 28 features, optimizing our fraud detection capabilities. The details of these features are cataloged in Table 8 of Appendix-1.

6.4 Training of Machine-Learning Model

In this section, we delve into our data-driven fraud detection system, Browser Polygraph. The core objective of Browser Polygraph is to verify the veracity of the browser's identity, specifically its `user-agent` string, and to identify any significant deviations that might indicate a falsified identity. Our dataset comprises 28 feature values for each user session, each linked to its respective `user-agent` string. Given that feature values tend to be consistent across similar browser releases, we opted for a semi-supervised learning approach for our model.

The initial phase of our methodology involves disregarding the `user-agent` strings and focusing solely on the feature values, which represent real-world browser fingerprints. This data is then employed to train our clustering model, designed to group browsers into distinct clusters. These clusters are formed based on the similarities in the implementation of JavaScript APIs by different browsers. For the clustering process, we selected the `kmeans` clustering algorithm due to its efficacy in handling such categorization tasks.

Post-training, the model is further refined by leveraging the ground truth of the `user-agent` strings. This step involves analyzing how these `user-agents` are distributed across the formed clusters, thereby enabling us to assess the model's accuracy in correlating the `user-agent` strings with the expected browser features. The following will provide an overview of the fitting and training processes.

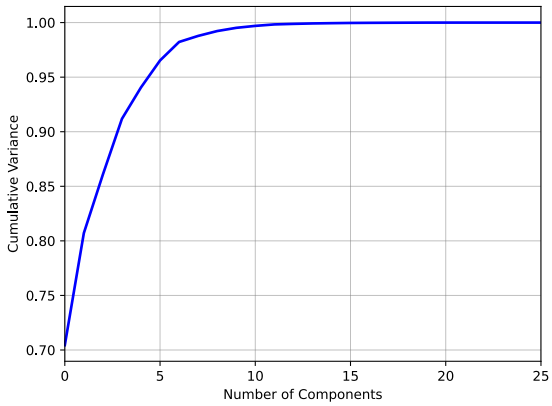

Figure 2: Cumulative variance based on the number of components in PCA.

6.4.1 Pre-Processing. Some of our features had large values which could skew the results of our model towards them. Therefore, we used Standard Scaler to scale some of our *deviation-based* attributes. The *time-based* attributes were already in the binary format which was suitable. Another crucial aspect of our preprocessing was outlier detection, for which we employed the Isolation Forest method [30]. This method was selected due to its effectiveness in identifying anomalies in high-dimensional datasets. By setting a threshold of 0.002%, we successfully filtered out data presumed to be outliers, ensuring a more refined and accurate dataset for subsequent analysis. Upon examining the data excluded by the 0.002% threshold, it is evident that none of the eliminated records (172 rows) corresponded to feature values indicative of a legitimate browser instance, as delineated in Section 6.1. Hence, this threshold selection is demonstrably appropriate.

6.4.2 Feature Selection. In the feature selection phase, we used Principal Component Analysis [21] (PCA) to streamline our dataset. PCA is a technique in machine learning that reduces dimensionality while retaining the most informative aspects of the data. It works by transforming the original features into a new set of variables, the principal components, which are orthogonal and maximize variance. The decision to use PCA was driven by its efficiency in simplifying complex data without significant loss of information. Analysis of the PCA results, particularly through the visualization shown in Figure 2, revealed that using seven components could capture over 98.5% of the cumulative variance in the dataset. This level of variance was deemed optimal for our analysis, striking a balance between simplicity and information retention.

6.4.3 Clustering. In this part of our analysis, having established seven as the optimal number of components through PCA, we employed the `kmeans` clustering algorithm on our dataset (kmeans was chosen due to its efficiency and straightforward implementation). The optimal number of clusters (k) was determined using the elbow method, as illustrated in Figure 3. This method plots the Within-Cluster Sum of Squares [20] (WCSS) against the number of clusters

to evaluate the variance within each cluster. A decrease in WCSS signifies a closer alignment of data points to their cluster centroids. In our analysis, the elbow graph demonstrated potential optimal cluster counts at $k = 3$, $k = 6$, and $k = 11$. At these points, the graph shows a noticeable "elbow", implying that additional clusters would not significantly improve the fit to the data. To further refine our decision, we analyzed the relationship between WCSS values and the number of clusters in Figure 4, where a pronounced increase at $k = 11$ suggested it as the ideal choice.

After selecting k=11 as the optimal number of clusters, we evaluated our model's accuracy. Since we are using a semi-supervised learning approach, our accuracy metric measures how well the clusters represent the `user-agent` instances. We define a cluster assignment as accurate if all instances of the same `user-agent` are assigned to the same cluster, specifically the cluster containing the majority of data points for that `user-agent` string. For instance, if most data points with the `user-agent` of Chrome 112 are assigned to cluster 0, we consider cluster 0 as the corresponding cluster for Chrome 112. Any data with `user-agent` of Chrome 112 that is assigned to a different cluster would be considered misclustered. Using this definition, our model achieved a clustering accuracy of 99.6%.

Beyond model accuracy, we further validated the `user-agents` and their corresponding clusters to ensure they matched our expectations. This process involved cross-referencing our results with legitimate browser instances from the Candidate Fingerprint Generation stage (Section 6.1). We confirmed that, in over 98% of the data, the corresponding clusters matched the expected behavior. However, we noted some discrepancies with older browser versions, such as Chrome 81 and Edge 17. On closer inspection, we found that these versions had few data points, in some cases less than 100 instances. This limited dataset occasionally led to misleading results, such as Chrome 81 fingerprints showing Firefox-like feature sets. For these specific `user-agents`, making up less than 2% of our traffic, we decided to align the cluster number with the feature values of legitimate browser instances, extracted in the Candidate Fingerprint Generation stage (Section 6.1). This adjustment was essential for maintaining the accuracy of our clustering.

The discrepancies observed with older browser versions can be attributed to the lack of sufficient descriptive data points, a challenge that is not uncommon in machine learning applications dealing with evolving technologies. To mitigate this issue in the future, a more extensive data collection process would be beneficial, akin to the detailed approach in Google Picasso's work [9]. Such a method provides a richer dataset, enabling more accurate clustering of even the lesser-known or outdated browser versions.

With prior adjustments, the distribution of `user-agents` across the selected clusters (with $k = 11$) is detailed in Table 3. To illustrate why our selected cluster size offers a more fitting analysis than other potential choices (3, 6, or 11), we examined the distribution of `user-agents` across a $k = 6$ cluster scenario. The outcomes highlighted in Table 9 (Appendix-2) versus our optimal selection in Table 3, affirm that our choice more effectively differentiates between browser releases, underscoring its suitability for our analysis. We have also explored the impact of changing model parameters on the accuracy of the model in Appendix-4, which further reinforces

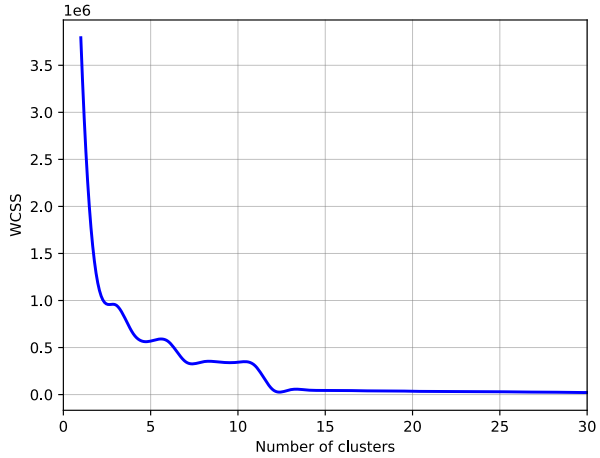

Figure 3: Elbow method to select the optimal number of clusters (WCSS vs. number of clusters).

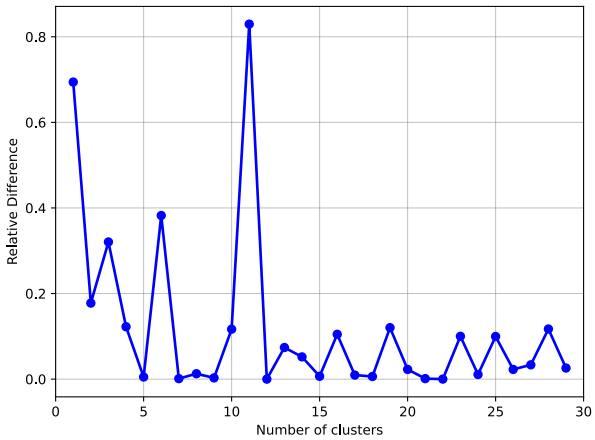

Figure 4: Relative WCSS vs. clusters: showing k=11 as an optimal number of clusters for our `kmeans` model.

that our choice of 28 features, 7 PCA components, and k=11 clusters lead to optimal accuracy for Browser Polygraph.

6.5 Fraud Detection

After successful training, the Browser Polygraph model can perform real-time monitoring and detection of potentially fraudulent activities in live Web traffic. The operational process of our Fraud Detection system is as follows:

When a user accesses FinOrg's website, JavaScript code collects the required feature values from their browser. These values are then analyzed by Browser Polygraph to identify the user's cluster affiliation. A key aspect of fraud detection involves comparing this predicted cluster with the cluster corresponding to the browser's declared `user-agent` (refer to Table 3). Any mismatch triggers our specialized risk analysis function.

Table 3: `user-agents` assigned to clusters with k=11.

Cluster	user-agents
0	Chrome 110-113, Edge 110-113
1	Firefox 101-114
2	Chrome 59-68, Firefox 51-91
3	Chrome 114, Edge 114
4	Chrome 69-89, Edge 79-89
5	Chrome 102-109, Edge 102-109
6	Edge 17-19, Firefox 46-50
9	Firefox 93-100
10	Chrome 90-101, Edge 90-101

Algorithm 1 Algorithm to calculate the `risk factor` based on a session's `user-agent` and its predicted cluster.

```
Input: sessUserAgent, predictedCluster, userAgentTable
Output: riskFactor
riskFactor ← ∞
for each uA in userAgentTable[predictedCluster] do
    if sessionUserAgent.vendor ≠ uA.vendor then
        distance ← 20
    else
        diff ← abs(sessUserAgent.version - uA.version)
        distance ← floor(diff / 4)
    end if
    if distance < riskFactor then
        riskFactor ← distance
    end if
end for
return riskFactor
```

The risk analysis function initially calculates the distance between a session's `user-agent` and every other `user-agent` in the predicted cluster (see Algorithm 1). The smallest of these distances is then employed as the session's `risk factor`. We define the distance between two `user-agent` strings based on the differences in their vendors and version numbers. For two `user-agents` with different vendors, the distance is at its maximum value. However, for `user-agents` from the same vendor, we first calculate the absolute numerical difference between their version numbers and then divide this difference by 4— which we empirically selected referring to Table 3. This method effectively reduces the likelihood of false negatives, especially when a browser instance is incorrectly assigned into a cluster of a similar vendor but a different release.

Browser Polygraph's strength lies in its efficiency and adaptability. The computationally intensive task of training the model is performed offline, ensuring that the real-time fraud detection process is possible. This not only minimizes delays during user interaction but also allows for ongoing system enhancements. In Section 7, we will explore the correlation between these `risk factors` and other risk metrics within FinOrg.

6.6 Drift Detection

An essential feature of Browser Polygraph is its ongoing evaluation mechanism to monitor the performance of the clustering algorithm. This includes regularly assessing the accuracy in assigning clusters to new browser releases. We implemented a robust drift detection strategy that actively identifies shifts in data patterns or browser behavior. When such shifts are detected, indicating a potential divergence from the established clustering norms, Browser Polygraph can automatically initiate a retraining process. This retraining ensures that Browser Polygraph remains up-to-date and accurately aligned with the latest browser versions' features.

More specifically, on designated dates, the automatic drift detection module evaluates the cluster number and accuracy of Browser Polygraph in clustering new browser releases. These dates are strategically chosen a few days after the latest releases of Firefox, Chrome, and Edge browsers. When the cluster number of a new browser release aligns with its closest release according to Table 3 with an accuracy rate above 98%, it indicates no significant shift in browser behavior, and no retraining is necessary. However, a change in the cluster number (compared to its closest release from Table 3) or a drop in the accuracy rate below 98% signals a shift in browser behavior, prompting Browser Polygraph to automatically initiate retraining. Section 7.3 discusses the process using Browser Polygraph's trained model (from March to mid-July 2023) to perform drift analysis on newly collected data (from late-July to October 2023).

7 Evaluation

7.1 Real-world Experiment

To demonstrate the practical application of Browser Polygraph in detecting fraud, we collaborated with FinOrg for data collection on one of their secondary purchase portals, as outlined in Section 6.2. We used 4.5 months of this data to train our `kmeans` model (Section 6.4). The resulting clusters formed the baseline for our fraud detection method (Table 3).

For training purposes, we analyzed 205k logged-in user sessions from March to mid-July 2023, which included `user-agents` from 113 different browser releases. FinOrg also provided session tags from their internal systems, specifically `Untrusted_IP`, `Untrusted_Cookie`, and `ATO` (Account Take-Over), used solely for evaluation purposes. These tags identified sessions from unfamiliar IPs (note that we did not receive the old or new IP, just that FinOrg was unfamiliar with the IPs), newly-established cookies, or those with suspicious activities linked to the browser fingerprint within a 72-hours period (meaning that FinOrg believed the account was involved in an ATO).

This dataset of only logged-in users minimized interference from web crawlers and bots, offering a solid foundation for our model. Post-training, Browser Polygraph flagged 897 sessions as suspicious during four-and-a-half-month period, considering various `risk factor` levels. This number of flagged sessions aligns with fraud data from similar studies, such as Varmedja et al. [64]'s work that used a popular credit card fraud database from Kaggle with 0.173% of reported credit card frauds in 2019 [22].

Browser Polygraph showed heightened accuracy in identifying potential fraud, with flagged sessions having a 27% and 26% higher

Table 4: Percentages of the traffic having Untrusted_IP, Untrusted_Cookie or ATO flags set in the entire traffic comparing to the batch flagged by Browser Polygraph and a randomly chosen batch.

Category	Untrusted_IP	Untrusted_Cookie	ATO
All users	51%	49%	0.43%
Flagged by Browser Polygraph (all)	78%	75%	2%
Flagged by Browser Polygraph (risk factor > 1)	93%	89%	3.89%
Flagged by Browser Polygraph (risk factor > 4)	94%	85%	5.83%
Randomly-chosen	48%	53%	0.22%

likelihood of being tagged as `Untrusted_IP` and `Untrusted_Cookie` flags, respectively. Sessions flagged by Browser Polygraph were also about five times more likely to be associated with `ATO` activities (Table 4). The correlation between Browser Polygraph's `risk factor` and the security tags was notable. Higher `riskfactors` correlated with increased cases of `Untrusted_IP`, `Untrusted_Cookie` and `ATO` flags, underscoring Browser Polygraph's effectiveness. Analysis indicated that lower `risk factors` were often assigned to sessions where the claimed `user-agent` shares the same vendor but only a slight variation in version number compared to other `user-agents` in the assigned cluster. Such cases might not always indicate fraud but could result from update inconsistencies or the use of certain extensions or browser configurations impacting attribute values (refer to Section 6.3).

In a supplementary experiment, we compare against a randomly selected 897 sessions from our dataset of 205k user sessions for further analysis. The results confirmed Browser Polygraph 's superior performance over the random selection method, demonstrating its utility in identifying potentially fraudulent sessions with increased precision (Table 4).

Despite Browser Polygraph's success in identifying sessions with `Untrusted_IP` and `Untrusted_Cookie` flags, only 2% of these 897 sessions have `ATO` flags (though this rate nearly doubled and tripled in batches with higher `risk factor`). This lower-than-expected detection rate may be due to various factors, such as the specific nature of the data (the tagging of ATO) or the context of system integration. To gain a better visibility, the next subsection will evaluate Browser Polygraph's effectiveness in detecting sessions originating from fraud browsers.

7.2 Fraud Browsers' Detection

We next evaluate the capability of Browser Polygraph to identify fraudulent attempts at spoofing legitimate browser fingerprints. Specifically, Browser Polygraph focuses on uncovering fraud within browsers classified under Categories **(1)** and **(2)** (in Table 1). It achieves this by analyzing discrepancies between the `user-agent` and the extracted fingerprints. To facilitate the evaluation, we established a private website equipped with the fingerprinting script developed during our *Real-World Data Collection* phase (Section 6.2). We were the only ones who accessed the website.

The primary objective is to assess Browser Polygraph's effectiveness in identifying fraudulent sessions. To this end, we selected several fraud browsers from Categories **(1)** and **(2)** (Table 1). These browsers were installed on a Windows machine, and multiple browser profiles were created for each, employing various `user-agents` representative of all clusters in Table 3. We then visited our private test website using each profile. In creating browser profiles, we tailored each one based on the browser's capacity for customization. This involved selecting a `user-agent` and their corresponding browser engines (if possible), aiming for diverse representation. Where feasible, for each cluster in Table 3, we generated two profiles using candidate `user-agents` from the same cluster. In cases where a fraud browser limited this capability, we opted for either randomized `user-agents` or those uniquely provided by the browser itself.

Our approach in this experimental setup was designed to assess the capability of Browser Polygraph to detect sessions initiated by fraud browsers. We passed the data collected in this experiment to Browser Polygraph's Fraud Detection module (trained in Section 6.4) to evaluate its effectiveness in identifying fraudulent browser fingerprints. The data presented in Table 5 highlights Browser Polygraph's effectiveness in detecting fraudulent sessions, evidenced by a high average `risk factor` for these sessions.

While Browser Polygraph demonstrated an impressive recall rate of 79% for the first three fraud browsers in Table 5, the recall rate for Sphere browser was observed at 67%, this outcome is primarily influenced by two factors. The first concerns the freely available version of the Sphere software, which is significantly outdated. This limits users' ability to customize their preferred user profiles. The second factor is the predominance of user profiles in this version that adhere to older versions of Chrome (e.g., Chrome 63, 64, 65), with Sphere 1.3 emulating a fingerprint similar to Chrome 61, all falling within Cluster 2. This latter reason also accounted for the non-flagged attempts by the first three browsers.

7.3 Drift Analysis

With the continual release of new browser versions and fraud browsers, a pertinent question arises: How long does our trained model in the Browser Polygraph remain effective? This subsection explains the behavior of Browser Polygraph's drift detection strategy (refer to Section 6.6) that focuses on determining the necessity and timing for retraining Browser Polygraph via an extensive drift analysis.

Referring to the trained model of Browser Polygraph in Table 3, for a new browser release, such as Chrome 115, if the feature set values do not significantly differ from the closest prior release (e.g., Chrome 114), we do not expect our drift detection algorithm from

Table 5: Fraud browsers' detection capability via Browser Polygraph.

Browser	Flagged Num	Not-Flagged Num	Avg. `risk factor`	Recall
GoLogin-3.3.23	12	4	11.66	75%
Incogniton-3.2.7.7	7	2	8.85	78%
Octo Browser-1.10	16	3	10.18	84%
Sphere-1.3	6	3	10.5	67%

Section 6.6 to trigger a retraining. However, if there is a drastic change in the values of the feature set, resulting in a different cluster number or a noticeable drop in accuracy, the algorithm will initiate a retraining process. To evaluate how the drift detection module accurately identifies retraining time, we used the clustering model from Section 6.4 that was trained from March to mid-July 2023. Subsequently, we obtained new data spanning from late-July to October 2023 from FinOrg, encompassing several releases of major browsers. With this updated dataset, we selected specific dates to evaluate the accuracy of Browser Polygraph in clustering newly released browsers. These dates were strategically chosen a few days following the latest Firefox release, with the newest Chrome and Edge versions released approximately one to two weeks earlier. Our focus was on two key metrics: determining the predominant cluster number for each new browser instance and calculating the percentage of browsers assigned to this cluster for each candidate browser, as detailed in Table 6.

Our findings indicate that new releases of Chrome, Firefox, and Edge were correctly clustered with an accuracy exceeding 99% for approximately three and half months. However, starting from late October, a shift in the assigned cluster number for Firefox 119 was observed, alongside a decline in the clustering accuracy for Chrome 119. This served as an indicator for retraining our clustering model. Further examination of the feature values for Firefox 119 confirmed substantial changes in the `Element` prototype's implementation compared to its predecessor, providing additional evidence supporting the hypothesis for the retraining time.

It is worth mentioning that our observations in this analysis align with the retraining signal from the drift detection module, which was triggered in October. Although in this analysis, our trained model in Browser Polygraph remained accurate for three and a half months, this duration may vary depending on the parameters of the model and the extent of changes in new browser releases.

Table 6: Drift Analysis of Browser Polygraph for data collected from late-July to October 2023.

Browser	Date	Cluster	Accuracy
Chrome 115		3	99.45
Firefox 115	07/25	1	99.3
Edge 115		3	100
Chrome 116		3	99.6
Firefox 116	08/25	1	99.99
Edge 116		3	99.88
Chrome 117		3	99.25
Firefox 117	09/25	1	99.81
Edge 117		3	99.94
Chrome 118		3	99.65
Firefox 118	10/23	1	99.46
Edge 118		3	99.91
Chrome 119		3	97.22
Firefox 119	10/31	10	98.57
Edge 119		3	99.84

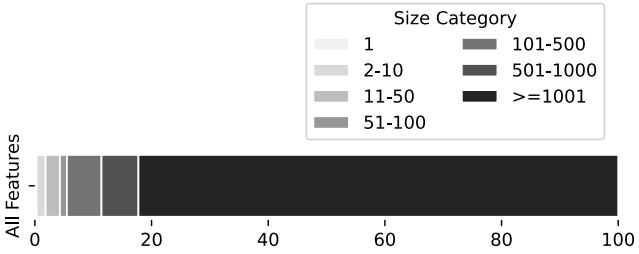

Figure 5: Percentage of fingerprints in anonymity sets.

7.4 Privacy Analysis

To evaluate the privacy-preserving nature of Browser Polygraph's feature set, we analyzed the anonymity sets and entropy of features within our dataset, which comprises 205k fingerprints. The former assesses how diverse the fingerprints are, while the latter provides insight into the diversity of individual features. Figure 5 displays the percentage of user fingerprints across anonymity sets of varying sizes. Notably, only 0.3% of our fingerprints are unique, a negligible rate compared to a browser fingerprinting study that reported 33.6% unique fingerprints [18]. Furthermore, while the same study found 8% of fingerprints in anonymity sets larger than 50, our study shows a significantly higher rate of 95.6%, making it nearly impossible to track individual users.

We also investigated Shannon entropy and normalized entropy for the collected attributes (`user-agent` and the integer outputs) as reported in Table 7 (sorted by normalized entropy). The `user-agent` was the most diverse feature in our dataset, yielding a normalized entropy value of 0.58. This value is the same as the normalized entropy of 0.58 for the `user-agent` string for AmIUnique study [18]. This similarity emphasizes that our data collection method does not increase user identifiability beyond what is already possible with the `user-agent` string alone, thus maintaining user anonymity.

7.5 Performance Analysis

To validate our adherence to the performance benchmarks set in Section 3, we measured Browser Polygraph's response time within

Table 7: Entropy information of selected features used in Browser Polygraph, sorted by normalized entropy.

Features	Entropy	Normalized Entropy
`user-agent`	5.97	0.58
`Object.getOwnPropertyNames(Element.prototype).length`	2.51	0.47
`Object.getOwnPropertyNames(SVGElement.prototype).length`	2.33	0.43
`Object.getOwnPropertyNames(Document.prototype).length`	2.17	0.42
`Object.getOwnPropertyNames(IntersectionObserver.prototype).length`	1.33	0.37
`HTMLVideoElement.prototype.hasOwnProperty('webkitDisplayingFullscreen')`	0.58	0.37
`Object.getOwnPropertyNames(CSSRule.prototype).length`	0.56	0.35
`Object.getOwnPropertyNames(StaticRange.prototype).length`	0.58	0.29

the experimental setup described in Section 7.2. Through a series of tests mirroring those applied to fine-grained fingerprinting solutions, Browser Polygraph demonstrated an average response time of 6ms. Additionally, the storage requirement for our fingerprints was only 1KB, which significantly surpasses the efficiency of the solutions in Table 2.

In Appendix-5, we compare the performance of Browser Polygraph's coarse-grained fingerprints with two fine-grained techniques for browser clustering (FingerprintJS and ClientJS) using synthetic data. The results demonstrate that Browser Polygraph performs better in clustering for this specific experiment. This is because Browser Polygraph's coarse-grained fingerprinting is fundamentally designed for detecting lying browsers, making it more efficient for browser clustering purposes.

8 Discussion

The most trivial step for configuring a fraud browser is setting the `user-agent` value—alignment of this value to the victim's actual `user-agent` however does not suffice for the adversary to fully spoof a victim's browser. A fraud browser could still show inconsistencies in its other configurations, and the primary goal of this paper is to identify such inconsistencies. Our solution, however, has some limitations:

Bypassing Browser Polygraph. Lin et al. [29] showed how stealing and spoofing actual user fingerprints could bypass the authentication process of critical websites. Their work can bypass Browser Polygraph in theory and it is essentially the problem of many JavaScript-based detection techniques; however, their current solution was not spoofing the output of `Object.getOwnPropertyNames` function for our feature set values, and it needs an additional update to cover it. While these cat-and-mouse games can be played indefinitely, the goal of this paper is *not* to identify new fingerprinting techniques but rather to introduce the concept of coarse-grained fingerprints that can be used in a production website with strict limitations on fingerprint extraction time and data size.

`user-agent` randomization. `user-agent` randomization is a common anti-fingerprinting strategy, potentially increasing false positives in Browser Polygraph. However, we do not recommend its application due to potential complications, such as triggering bot detection mechanisms [40].

Verification of new browsers. Browser Polygraph's current focus is on popular desktop browsers and those with analogous `user-agent` strings, due to the prevalence of fraud browsers offering more tools to spoof desktop rather than mobile browsers. Currently, it does not encompass mobile browsers or those with unique engines and `user-agent` strings. However, extending its verification capabilities to these areas is possible. This expansion would involve gathering baseline data for these new browsers, as detailed in Section 6.1. With baseline values and a thorough real-world dataset for any new browser type, adapting Browser Polygraph for detection becomes straightforward.

Deployment scope. The current deployment of Browser Polygraph leverages a coarse-grained browser fingerprinting technique, primarily designed for as a lightweight fraud detection mechanism in high-volume traffic scenarios. It is crucial to acknowledge that Browser Polygraph is not configured to detect fraudulent sessions conducted by sophisticated attackers, who may completely spoof the fingerprinting environment or use browsers classified under category 3 in Table 1. To counter these advanced spoofing efforts, more complex detection methods are necessary, perhaps binary analysis of each specific software, which can unearth unique software-specific fingerprintable attributes.

In our observations, for instance, AntBrowser [50] (similar to many other browsers) includes an `ANTBROWSER` object in its namespace and `antBrowser`-prefixed attributes on the `window` object, significantly increasing its fingerprintability. This is similar to the observation that Nikiforakis et al. [36] made where browser extensions that allow browser spoofing ironically make browsers *more* fingerprintable. While these attributes were initially identified through manual analysis, future work could automate this process, employing techniques such as fuzzing or binary analysis. Such advancements can enable identifying software-specific attributes and directly target the fingerprinting of fraud browsers.

Scale of the database. The current implementation of Browser Polygraph employs a dataset whose size does not yet pose significant storage challenges. However, should the dataset grow to an unmanageably large scale in the future, thereby impacting training efficiency, a viable solution would be the adoption of Stratified Sampling [32]. This approach is particularly effective for managing large datasets while ensuring the representativeness of diverse data segments. Using Stratified Sampling, we can efficiently maintain a representative sample of browser features, even from less popular browser instances, ensuring that our dataset remains manageable and comprehensive.

Manual efforts. While Browser Polygraph automates the majority of operations, such as candidate fingerprint generation, fraud detection, and drift detection, manual inspections were necessary

during the data pre-processing stage to address inconsistencies in *deviation-based* features. As discussed in Section 6.3, we manually assessed the impact of browser configurations on these features and determined whether to retain or discard specific features. Similarly, we manually installed popular extensions on Chrome to observe their influence on feature values. In case our selected features become ineffective over time—meaning that retraining Browser Polygraph does not achieve the required accuracy—or if there are drastic updates to browser implementations necessitating another round of pre-processing, we need to automate the manual inspections involved in this phase.

9 Related Work

This section discusses prior research relevant to our study on browser fingerprinting and fraud detection:

Risks of fingerprint spoofing. Liu et al. [31] explore how spoofing browser fingerprints can affect advertising content, while Lin et al. [29] demonstrate the potential for spoofed fingerprints to bypass websites authentication.

Nature of misuse. In the context of misuse, Kawase et al. [24] explore ATO within an online vehicle marketplace, presenting detection and prevention methods. Campobasso et al. [10] evaluate illegal sales of user fingerprints and tools for impersonating browser identities.

Advancements in browser fingerprinting. Nikiforakis et al. [36] focus on extracting browser family and version information using a set of techniques, such as ordering of methods and properties and detection of vendor-specific methods. Akhavani et al. [4] provide a complete list of JavaScript features for browser differentiation, but it might need revision via new browser releases. Schwarz et al. [44] propose automated techniques for inferring browser environments.

Browser fingerprinting in the wild. Durey et al. [13] and Senol et al. [45] explore the adoption of browser fingerprinting in real-world scenarios, with the latter extending the analysis to its effectiveness in fraud detection, particularly in safeguarding against unauthorized access and enhancing account security. Wu et al. [67] conduct a large-scale study of adversarial fingerprints—intentionally crafted by attackers to evade detection—and benign browser fingerprints, highlighting the role of browser fingerprinting in bot and fraud detection

Browser authentication techniques. Bursztein et al. [9] use graphical fingerprints for browser verification, and Laperdrix et al. [25] employ canvas challenges for authentication. Google offers Private State Tokens [19] as part of the Privacy Sandbox [19] project, providing a privacy-preserving method to validate users and potentially replacing third-party cookies. There are also proposals such as Web Environment Integrity API [66] which advocate for browser verification through attestation tokens. Additionally, Cloudflare's Privacy Pass [12] uses cryptographic tokens to authenticate browsers, allowing websites to verify legitimate traffic without tracking individual users.

Fraud browsers' detection. Traditionally, security researchers and industry professionals have relied on manual analysis to identify suspicious parameters in requests generated by fraud browsers, as highlighted by Azad et al. [7]. These mitigation strategies, often based on regular expressions, face obsolescence with each new release of fraud browsers. Compounding this issue, fraud browser vendors frequently update their software to align with new browser versions, thereby diminishing the effectiveness of existing fraud detection patterns. An automated prototype that adapts to ongoing browser updates is, therefore, a critical need in the industry.

10 Conclusion

In conclusion, this paper presents Browser Polygraph, a novel approach to addressing the challenges of detecting lying browsers in high-traffic environments. Browser Polygraph excels in merging the demand for detailed browser data with the scalability essential for real-time fraud detection. We show a new direction for browser fingerprints—coarse-grained fingerprints that can satisfy the performance constraints of a financial company FinOrg. This research marks a significant step forward in the fight against browser fingerprint spoofing and fraud browser detection, offering insight that could shape future developments in the field.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their valuable feedback, which helped us improve our paper. We also appreciate the support of the Department of Defense and the National Science Foundation (NSF) under grants No. CNS-2127232, CNS-2346845, and CNS-2419829. This work also relates to the Department of Navy award N00014-24-1-2193 issued by the Office of Naval Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research, the NSF or the US Government.

References

[1] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 674–689.

[2] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1129–1140.

[3] Gemini Advisory. 2023. Illicit Antidetect Platforms vs. Anti-Fraud Solutions. https://geminiadvisory.io/antidetect-platforms-vs-anti-fraud-solutions.

[4] Seyed Ali Akhavani, Jordan Jueckstock, Junhua Su, Alexandros Kapravelos, Engin Kirda, and Long Lu. 2021. Browserprint: An analysis of the impact of browser features on fingerprintability and web privacy. In *Information Security: 24th International Conference, ISC 2021, Virtual Event, November 10–12, 2021, Proceedings 24*. Springer, 161–176.

[5] Akhavani, Seyed Ali. 2024. bowserprint github. https://github.com/sa-akhavani/browserprint.

[6] Nampoina Andriamilanto, Tristan Allard, and Gaëtan Le Guelvouit. 2020. FPSelect: low-cost browser fingerprints for mitigating dictionary attacks against web authentication mechanisms. In *Proceedings of the 36th Annual Computer Security Applications Conference*. 627–642.

[7] Babak Amin Azad, Oleksii Starov, Pierre Laperdrix, and Nick Nikiforakis. 2020. Taming The Shape Shifter: Detecting Anti-fingerprinting Browsers. In *DIMVA 2020-17th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*.

[8] Károly Boda, Ádám Máté Földes, Gábor György Gulyás, and Sándor Imre. 2012. User tracking on the web via cross-browser fingerprinting. In *Information Security Technology for Applications: 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers 16*. Springer, 31–46.

[9] Elie Bursztein, Artem Malyshev, Tadek Pietraszek, and Kurt Thomas. 2016. Picasso: Lightweight device class fingerprinting for web clients. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. 93–102.

[10] Michele Campobasso and Luca Allodi. 2020. Impersonation-as-a-service: Characterizing the emerging criminal infrastructure for user impersonation at scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1665–1680.

[11] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-) browser fingerprinting via OS and hardware level features. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society.

[12] Cloudflare. 2024. Privacy Pass. https://developers.cloudflare.com/waf/tools/privacy-pass/ Accessed: August 19, 2024.

[13] Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2021. FP-Redemption: Studying browser fingerprinting adoption for the sake of web security. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 18th International Conference, DIMVA 2021, Virtual Event, July 14–16, 2021, Proceedings 18*. Springer, 237–257.

[14] Peter Eckersley. 2010. How unique is your web browser?. In *Privacy Enhancing Technologies: 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings 10*. Springer, 1–18.

[15] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 1388–1401.

[16] FBI. 2023. 2022 Internet Crime Report. https://www.ic3.gov/Media/PDF/AnnualReport/2022_IC3Report.pdf.

[17] David Freeman, Sakshi Jain, Markus Dürmuth, Battista Biggio, and Giorgio Giacinto. 2016. Who Are You? A Statistical Approach to Measuring User Authenticity.. In *NDSS*, Vol. 16. 21–24.

[18] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. 2018. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *Proceedings of the 2018 world wide web conference*. 309–318.

[19] Google. 2024. Privacy Sandbox. https://privacysandbox.com Accessed: August 19, 2024.

[20] Hestry Humaira and Rasyidah Rasyidah. 2020. Determining The Appropriate Cluster Number Using Elbow Method for K-Means Algorithm. In *Proceedings of the 2nd Workshop on Multidisciplinary and Applications (WMA) 2018*.

[21] Ian T Jolliffe. 2002. *Principal component analysis for special types of data*. Springer.

[22] Kaggle. 2024. Credit Card Fraud Detection. https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud.

[23] Soroush Karami, Faezeh Kalantari, Mehrnoosh Zaeifi, Xavier J Maso, Erik Trickel, Panagiotis Ilia, Yan Shoshitaishvili, Adam Doupé, and Jason Polakis. 2022. Unleash the simulacrum: shifting browser realities for robust {Extension-Fingerprinting} prevention. In *31st USENIX Security Symposium (USENIX Security 22)*. 735–752.

[24] Ricardo Kawase, Francesca Diana, Mateusz Czeladka, Markus Schüler, and Manuela Faust. 2019. Internet fraud: the case of account takeover in online marketplace. In *Proceedings of the 30th ACM Conference on Hypertext and Social Media*. 181–190.

[25] Pierre Laperdrix, Gildas Avoine, Benoit Baudry, and Nick Nikiforakis. 2019. Morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 43–66.

[26] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. 2020. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)* 14, 2 (2020), 1–33.

[27] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 878–894.

[28] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. 2021. Fingerprinting in style: Detecting browser extensions via injected style sheets. In *30th USENIX Security Symposium (USENIX Security 21)*. 2507–2524.

[29] Xu Lin, Panagiotis Ilia, Saumya Solanki, and Jason Polakis. 2022. Phish in sheep's clothing: Exploring the authentication pitfalls of browser fingerprinting. In *31st USENIX Security Symposium (USENIX Security 22)*. 1651–1668.

[30] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.

[31] Zengrui Liu, Prakash Shrestha, and Nitesh Saxena. 2022. Gummy browsers: targeted browser spoofing against state-of-the-art fingerprinting techniques. In *International Conference on Applied Cryptography and Network Security*. Springer, 147–169.

[32] Sharon L. Lohr. 2019. *Sampling: Design and Analysis*. Cengage Learning.

[33] MDN. 2022. MDN Web API. https://developer.mozilla.org/en-US/docs/Web/API.

[34] Grzergor Milka. 2018. Anatomy of account takeover. In *Enigma 2018 (Enigma 2018)*.

[35] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. *Proceedings of W2SP* 2012 (2012).

[36] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2013. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 541–555.

[37] U.S. Department of Health and Human Services. 2024. Human Subject Regulations Decision Charts: 2018 Requirements. https://www.hhs.gov/ohrp/regulations-and-policy/decision-charts-2018/index.html.

[38] Jeremiah Onaolapo, Enrico Mariconti, and Gianluca Stringhini. 2016. What happens after you are pwnd: Understanding the use of leaked webmail credentials in the wild. In *Proceedings of the 2016 Internet Measurement Conference*. 65–79.

[39] Pablo Picazo-Sanchez, Gerardo Schneider, and Andrei Sabelfeld. 2020. HMAC and "Secure Preferences": Revisiting Chromium-Based Browsers Security. In *Cryptology and Network Security: 19th International Conference, CANS 2020, Vienna, Austria, December 14–16, 2020, Proceedings 19*. Springer, 107–126.

[40] Radware. 2024. Detecting and Mitigating Highly Distributed Sophisticated Bot Attacks. https://www.radware.com/blog/application-protection/2023/11/detecting-and-mitigating-highly-distributed-sophisticated-bot-attacks.

[41] Rohan Goswami. 2023. Cybercrime marketplace Genesis Market shut by FBI, international law enforcement. https://www.cnbc.com/2023/04/04/genesis-market-shut-by-law-enforcement-in-cybercrime-operation.html.

[42] Iskander Sanchez-Rola, Leyla Bilge, Davide Balzarotti, Armin Buescher, and Petros Efstathopoulos. 2023. Rods with laser beams: understanding browser fingerprinting on phishing pages. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4157–4173.

[43] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. 2018. Clock around the clock: Time-based device fingerprinting. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1502–1514.

[44] Michael Schwarz, Florian Lackner, and Daniel Gruss. 2019. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits.. In *NDSS*.

[45] Asuman Senol, Alisha Ukani, Dylan Cutler, and Igor Bilogrevic. 2024. The Double Edged Sword: Identifying Authentication Pages and their Fingerprinting Behavior. In *Proceedings of the ACM on Web Conference 2024*. 1690–1701.

[46] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. 2017. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 329–336.

[47] Naoki Takei, Takamichi Saito, Ko Takasu, and Tomotaka Yamada. 2015. Web browser fingerprinting using only cascading style sheets. In *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*. IEEE.

[48] AdsPower Team. 2024. AdsPower. https://www.adspower.com.

[49] AmIUnique Team. 2024. AmIUnique. https://amiunique.org.

[50] AntBrowser Team. 2024. AntBrowser. https://antbrowser.pro.

[51] CheBrowser Team. 2024. CheBrowser. https://chebrowser.site.

[52] ClientJS Team. 2024. ClientJS. http://clientjs.org.

[53] ClonBrowser Team. 2024. ClonBrowser. https://www.clonbrowser.com.

[54] FingerprintJS Team. 2024. FingerprintJS. https://github.com/fingerprintjs/fingerprintjs.

[55] GoLogin Team. 2024. GoLogin. https://gologin.com.

[56] Incogniton Team. 2024. incogniton. https://incogniton.com.

[57] OctoBrowser Team. 2024. Octo Browser. https://octobrowser.net.

[58] Sphere Team. 2024. Sphere – A Browser for Anonymity. https://www.geeksforgeeks.org/sphere-a-browser-for-anonymity/y.

[59] VMLogin Team. 2024. VMLogin. https://www.vmlogin.us.

[60] Wappalyzer Team. 2024. Browser fingerprinting market share. https://www.wappalyzer.com/technologies/browser-fingerprinting.

[61] Tenebris. 2024. Linken Sphere. https://ls.tenebris.cc.

[62] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, et al. 2017. Data breaches, phishing, or malware? Understanding the risks of stolen credentials. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[63] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. 2019. Everyone is different: Client-side diversification for defending against extension fingerprinting. In *28th USENIX Security Symposium (USENIX Security 19)*. 1679–1696.

[64] Dejan Varmedja, Mirjana Karanovic, Srdjan Sladojevic, Marko Arsenovic, and Andras Anderla. 2019. Credit card fraud detection-machine learning methods. In *2019 18th International Symposium INFOTEH-JAHORINA (INFOTEH)*. IEEE, 1–5.

[65] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *27th USENIX Security Symposium (USENIX Security 18)*. 135–150.

[66] Ben Wiser. 2023. Web Environment Integrity. https://rupertbenwiser.github.io/Web-Environment-Integrity.

[67] Shujiang Wu, Pengfei Sun, Yao Zhao, and Yinzhi Cao. 2023. Him of Many Faces: Characterizing Billion-scale Adversarial and Benign Browser Fingerprints on Commercial Websites.. In *NDSS*.

[68] Mehrnoosh Zaeifi, Faezeh Kalantari, Adam Oest, Zhibo Sun, Gail-Joon Ahn, Yan Shoshitaishvili, Tiffany Bao, Ruoyu Wang, and Adam Doupé. 2024. Nothing Personal: Understanding the Spread and Use of Personally Identifiable Information in the Financial Ecosystem. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy*. 55–65.

A Ethics

This work does not raise any ethical issues since we ensure that all data collected is in accordance with FinOrg's data collection policy, which considers fingerprinting browsers for fraud detection purposes as a legitimate and legal use case. It's important to note, as discussed in Section 6.2, that the fingerprints we collected are only integer outputs, used exclusively for fraud detection purposes.

Furthermore, our coarse-grained fingerprints lack the granularity to identify specific user browsers, thereby preserving privacy. For instance, a sample from our dataset in March revealed that approximately 11% of sessions in a day shared a `user-agent` associated with a specific Chrome version on a Windows 10 machine, with 96% displaying identical coarse-grained fingerprints. This precludes individual user identification. To provide additional proof, we analyzed the anonymity sets and feature entropy in Section 7.4. Our analysis found that only 0.3% of the fingerprints in our dataset were unique, which is negligible compared to the 30% unique fingerprints reported in previous studies [18]. This reinforces that our approach does not enable tracking and is solely for fraud detection.

Additionally, examining the entropy of collected features (Table 7), the most diverse collected feature was the `user-agent`, indicating that our collected features do not provide additional information beyond what is already exposed by the `user-agent` string.

Moreover, the session identifiers shared with us by FinOrg were completely opaque and randomized, ensuring no possibility of tracing them back to individual users, thereby maintaining user privacy and data confidentiality. Importantly, no Personally Identifiable Information (PII) was captured; at no time did the research involve access to the users' identity or their IP addresses.

To determine the necessity of IRB approval, we referred to Chart 01 from the Human Subject Regulations Decision Charts [37], titled 'Is an Activity Human Subjects Research Covered by 45 CFR Part 46?' Based on this chart, we concluded that our project does not involve human subjects as defined under 45 CFR Part 46 and does not require IRB oversight.

Appendix-1

Here is the list of features used for training of Browser Polygraph. The *deviation-based* features focus on counting JavaScript properties from the list of all MDN prototypes. These are initially filtered by standard deviation across all tested browsers during the candidate fingerprint generation stage, and then further filtered during the data pre-processing stage due to inconsistencies in actual browser data. The *time-based* features were initially derived from BrowserPrint [5]'s work, which examines the presence or absence of properties in browser releases. However, many of these features were subsequently filtered out as they did not track browser changes after 2020.

Table 8: Features set used for training of Browser Polygraph.

Num	Feature	Type
1	`Object.getOwnPropertyNames(Element.prototype).length`	*deviation-based*
2	`Object.getOwnPropertyNames(Document.prototype).length`	*deviation-based*
3	`Object.getOwnPropertyNames(HTMLElement.prototype).length`	*deviation-based*
4	`Object.getOwnPropertyNames(SVGElement.prototype).length`	*deviation-based*
5	`Object.getOwnPropertyNames(SVGFEBlendElement.prototype).length`	*deviation-based*
6	`Object.getOwnPropertyNames(TextMetrics.prototype).length`	*deviation-based*
7	`Object.getOwnPropertyNames(Range.prototype).length`	*deviation-based*
8	`Object.getOwnPropertyNames(StaticRange.prototype).length`	*deviation-based*
9	`Object.getOwnPropertyNames(AuthenticatorAttestationResponse.prototype).length`	*deviation-based*
10	`Object.getOwnPropertyNames(HTMLVideoElement.prototype).length`	*deviation-based*
11	`Object.getOwnPropertyNames(ResizeObserverEntry.prototype).length`	*deviation-based*
12	`Object.getOwnPropertyNames(ShadowRoot.prototype).length`	*deviation-based*
13	`Object.getOwnPropertyNames(PointerEvent.prototype).length`	*deviation-based*
14	`Object.getOwnPropertyNames(IntersectionObserver.prototype).length`	*deviation-based*
15	`Object.getOwnPropertyNames(CanvasRenderingContext2D.prototype).length`	*deviation-based*
16	`Object.getOwnPropertyNames(CSSStyleSheet.prototype).length`	*deviation-based*
17	`Object.getOwnPropertyNames(AudioContext.prototype).length`	*deviation-based*
18	`Object.getOwnPropertyNames(HTMLLinkElement.prototype).length`	*deviation-based*
19	`Object.getOwnPropertyNames(HTMLMediaElement.prototype).length`	*deviation-based*
20	`Object.getOwnPropertyNames(WebGL2RenderingContext.prototype).length`	*deviation-based*
21	`Object.getOwnPropertyNames(WebGLRenderingContext.prototype).length`	*deviation-based*
22	`Object.getOwnPropertyNames(CSSRule.prototype).length`	*deviation-based*
23	`Navigator.prototype.hasOwnProperty('deviceMemory')`	*time-based*
24	`BaseAudioContext.prototype.hasOwnProperty('currentTime')`	*time-based*
25	`HTMLVideoElement.prototype.hasOwnProperty('webkitDisplayingFullscreen')`	*time-based*
26	`Screen.prototype.hasOwnProperty('orientation')`	*time-based*
27	`Window.prototype.hasOwnProperty('speechSynthesis')`	*time-based*
28	`CSSStyleDeclaration.prototype.hasOwnProperty('getPropertyValue')`	*time-based*

Appendix-2

Table 9: `user-agents` assigned to clusters by Browser Polygraph with a less optimal clusters' number choice, k=6.

Cluster	user-agents
0	Chrome 114, Edge 114
1	Chrome 90-96, Edge 90-96, Firefox 101-114
2	Chrome 97-113, Edge 97-113
3	Chrome 64-89, Edge 79-89
4	Chrome 59-63, Firefox 51-100
5	Edge 17,18, Firefox 46-50

Appendix-3

Here are the list of 200 features used for the *Real-World Data Collection* stage besides 313 features from browserprint's code [5]:

```
Object.getOwnPropertyNames(Element.prototype).length
Object.getOwnPropertyNames(Document.prototype).length
Object.getOwnPropertyNames(HTMLElement.prototype).length
Object.getOwnPropertyNames(SVGElement.prototype).length
Object.getOwnPropertyNames(Navigator.prototype).length
Object.getOwnPropertyNames(RTCIceCandidate.prototype).length
Object.getOwnPropertyNames(SVGFEBlendElement.prototype).length
Object.getOwnPropertyNames(TextMetrics.prototype).length
Object.getOwnPropertyNames(Range.prototype).length
Object.getOwnPropertyNames(StaticRange.prototype).length
Object.getOwnPropertyNames(RTCRtpReceiver.prototype).length
Object.getOwnPropertyNames(RTCPeerConnection.prototype).length
Object.getOwnPropertyNames(AuthenticatorAttestationResponse.prototype).length
Object.getOwnPropertyNames(FontFace.prototype).length
Object.getOwnPropertyNames(HTMLVideoElement.prototype).length
Object.getOwnPropertyNames(ResizeObserverEntry.prototype).length
Object.getOwnPropertyNames(ShadowRoot.prototype).length
Object.getOwnPropertyNames(RTCRtpSender.prototype).length
Object.getOwnPropertyNames(PointerEvent.prototype).length
Object.getOwnPropertyNames(Blob.prototype).length
Object.getOwnPropertyNames(ServiceWorkerRegistration.prototype).length
Object.getOwnPropertyNames(MediaSession.prototype).length
Object.getOwnPropertyNames(PaymentResponse.prototype).length
Object.getOwnPropertyNames(HTMLSourceElement.prototype).length
Object.getOwnPropertyNames(Clipboard.prototype).length
Object.getOwnPropertyNames(IDBTransaction.prototype).length
Object.getOwnPropertyNames(Performance.prototype).length
Object.getOwnPropertyNames(ServiceWorkerContainer.prototype).length
Object.getOwnPropertyNames(HTMLIFrameElement.prototype).length
Object.getOwnPropertyNames(PaymentRequest.prototype).length
Object.getOwnPropertyNames(RTCRtpTransceiver.prototype).length
Object.getOwnPropertyNames(IntersectionObserver.prototype).length
Object.getOwnPropertyNames(CanvasRenderingContext2D.prototype).length
Object.getOwnPropertyNames(CSSStyleSheet.prototype).length
Object.getOwnPropertyNames(BaseAudioContext.prototype).length
Object.getOwnPropertyNames(AudioContext.prototype).length
Object.getOwnPropertyNames(HTMLLinkElement.prototype).length
Object.getOwnPropertyNames(RTCDataChannel.prototype).length
Object.getOwnPropertyNames(WritableStream.prototype).length
Object.getOwnPropertyNames(DataTransferItem.prototype).length
Object.getOwnPropertyNames(DocumentFragment.prototype).length
Object.getOwnPropertyNames(HTMLMediaElement.prototype).length
```

```
Object.getOwnPropertyNames(StorageManager.prototype).length
Object.getOwnPropertyNames(HTMLSlotElement.prototype).length
Object.getOwnPropertyNames(Text.prototype).length
Object.getOwnPropertyNames(WebGL2RenderingContext.prototype).length
Object.getOwnPropertyNames(HTMLInputElement.prototype).length
Object.getOwnPropertyNames(WebGLRenderingContext.prototype).length
Object.getOwnPropertyNames(HTMLButtonElement.prototype).length
Object.getOwnPropertyNames(HTMLTextAreaElement.prototype).length
Object.getOwnPropertyNames(HTMLSelectElement.prototype).length
Object.getOwnPropertyNames(MediaRecorder.prototype).length
Object.getOwnPropertyNames(CountQueuingStrategy.prototype).length
Object.getOwnPropertyNames(ByteLengthQueuingStrategy.prototype).length
Object.getOwnPropertyNames(PerformanceMark.prototype).length
Object.getOwnPropertyNames(PerformanceMeasure.prototype).length
Object.getOwnPropertyNames(HTMLImageElement.prototype).length
Object.getOwnPropertyNames(SpeechSynthesisEvent.prototype).length
Object.getOwnPropertyNames(HTMLFormElement.prototype).length
Object.getOwnPropertyNames(IDBCursor.prototype).length
Object.getOwnPropertyNames(HTMLTemplateElement.prototype).length
Object.getOwnPropertyNames(CSSRule.prototype).length
Object.getOwnPropertyNames(Location.prototype).length
Object.getOwnPropertyNames(PaymentAddress.prototype).length
Object.getOwnPropertyNames(IntersectionObserverEntry.prototype).length
Object.getOwnPropertyNames(TextEncoder.prototype).length
Object.getOwnPropertyNames(ImageData.prototype).length
Object.getOwnPropertyNames(HTMLMetaElement.prototype).length
Object.getOwnPropertyNames(Crypto.prototype).length
Object.getOwnPropertyNames(GamepadButton.prototype).length
Object.getOwnPropertyNames(DOMMatrixReadOnly.prototype).length
Object.getOwnPropertyNames(MediaKeys.prototype).length
Object.getOwnPropertyNames(MessageEvent.prototype).length
Object.getOwnPropertyNames(IDBFactory.prototype).length
Object.getOwnPropertyNames(MediaDevices.prototype).length
Object.getOwnPropertyNames(OfflineAudioContext.prototype).length
Object.getOwnPropertyNames(URL.prototype).length
Object.getOwnPropertyNames(ScriptProcessorNode.prototype).length
Object.getOwnPropertyNames(SVGAnimatedNumberList.prototype).length
Object.getOwnPropertyNames(ServiceWorker.prototype).length
Object.getOwnPropertyNames(SensorErrorEvent.prototype).length
Object.getOwnPropertyNames(SVGAnimatedPreserveAspectRatio.prototype).length
Object.getOwnPropertyNames(Sensor.prototype).length
Object.getOwnPropertyNames(SVGAnimatedRect.prototype).length
Object.getOwnPropertyNames(SVGAnimatedString.prototype).length
Object.getOwnPropertyNames(Selection.prototype).length
Object.getOwnPropertyNames(SecurityPolicyViolationEvent.prototype).length
Object.getOwnPropertyNames(XPathExpression.prototype).length
Object.getOwnPropertyNames(SVGAnimatedNumber.prototype).length
Object.getOwnPropertyNames(SVGAnimatedTransformList.prototype).length
Object.getOwnPropertyNames(Screen.prototype).length
Object.getOwnPropertyNames(RTCTrackEvent.prototype).length
Object.getOwnPropertyNames(SVGAnimateElement.prototype).length
Object.getOwnPropertyNames(SVGAnimateMotionElement.prototype).length
Object.getOwnPropertyNames(RTCStatsReport.prototype).length
Object.getOwnPropertyNames(RTCSessionDescription.prototype).length
Object.getOwnPropertyNames(SVGAnimateTransformElement.prototype).length
Object.getOwnPropertyNames(ScreenOrientation.prototype).length
Object.getOwnPropertyNames(SVGAnimatedlengthList.prototype).length
Object.getOwnPropertyNames(XPathResult.prototype).length
Object.getOwnPropertyNames(SVGAngle.prototype).length
Object.getOwnPropertyNames(SVGAElement.prototype).length
Object.getOwnPropertyNames(SubtleCrypto.prototype).length
Object.getOwnPropertyNames(SVGAnimatedAngle.prototype).length
```

```
Object.getOwnPropertyNames(StyleSheetList.prototype).length
Object.getOwnPropertyNames(StyleSheet.prototype).length
Object.getOwnPropertyNames(StylePropertyMapReadOnly.prototype).length
Object.getOwnPropertyNames(StylePropertyMap.prototype).length
Object.getOwnPropertyNames(XPathEvaluator.prototype).length
Object.getOwnPropertyNames(SVGAnimatedBoolean.prototype).length
Object.getOwnPropertyNames(SharedWorker.prototype).length
Object.getOwnPropertyNames(StorageEvent.prototype).length
Object.getOwnPropertyNames(Storage.prototype).length
Object.getOwnPropertyNames(StereoPannerNode.prototype).length
Object.getOwnPropertyNames(SVGAnimatedEnumeration.prototype).length
Object.getOwnPropertyNames(SpeechSynthesisUtterance.prototype).length
Object.getOwnPropertyNames(SVGAnimatedInteger.prototype).length
Object.getOwnPropertyNames(SVGAnimatedlength.prototype).length
Object.getOwnPropertyNames(SpeechSynthesisErrorEvent.prototype).length
Object.getOwnPropertyNames(SourceBufferList.prototype).length
Object.getOwnPropertyNames(SourceBuffer.prototype).length
Object.getOwnPropertyNames(WebGLFramebuffer.prototype).length
Object.getOwnPropertyNames(PresentationConnection.prototype).length
Object.getOwnPropertyNames(Plugin.prototype).length
Object.getOwnPropertyNames(PluginArray.prototype).length
Object.getOwnPropertyNames(PopStateEvent.prototype).length
Object.getOwnPropertyNames(Presentation.prototype).length
Object.getOwnPropertyNames(PresentationAvailability.prototype).length
Object.getOwnPropertyNames(PresentationConnectionAvailableEvent.prototype).length
Object.getOwnPropertyNames(PresentationConnectionCloseEvent.prototype).length
Object.getOwnPropertyNames(PresentationConnectionList.prototype).length
Object.getOwnPropertyNames(PresentationReceiver.prototype).length
Object.getOwnPropertyNames(PresentationRequest.prototype).length
Object.getOwnPropertyNames(ProcessingInstruction.prototype).length
Object.getOwnPropertyNames(PictureInPictureWindow.prototype).length
Object.getOwnPropertyNames(PermissionStatus.prototype).length
Object.getOwnPropertyNames(PromiseRejectionEvent.prototype).length
Object.getOwnPropertyNames(PerformanceNavigationTiming.prototype).length
Object.getOwnPropertyNames(PerformanceObserver.prototype).length
Object.getOwnPropertyNames(PerformanceObserverEntryList.prototype).length
Object.getOwnPropertyNames(PerformancePaintTiming.prototype).length
Object.getOwnPropertyNames(Permissions.prototype).length
Object.getOwnPropertyNames(PerformanceResourceTiming.prototype).length
Object.getOwnPropertyNames(PerformanceServerTiming.prototype).length
Object.getOwnPropertyNames(PerformanceTiming.prototype).length
Object.getOwnPropertyNames(PeriodicWave.prototype).length
Object.getOwnPropertyNames(ProgressEvent.prototype).length
Object.getOwnPropertyNames(PublicKeyCredential.prototype).length
Object.getOwnPropertyNames(RTCDTMFToneChangeEvent.prototype).length
Object.getOwnPropertyNames(RTCCertificate.prototype).length
Object.getOwnPropertyNames(RTCDataChannelEvent.prototype).length
Object.getOwnPropertyNames(RTCDTMFSender.prototype).length
Object.getOwnPropertyNames(RTCPeerConnectionIceEvent.prototype).length
Object.getOwnPropertyNames(Response.prototype).length
Object.getOwnPropertyNames(PushManager.prototype).length
Object.getOwnPropertyNames(PushSubscription.prototype).length
Object.getOwnPropertyNames(PushSubscriptionOptions.prototype).length
Object.getOwnPropertyNames(RadioNodeList.prototype).length
Object.getOwnPropertyNames(ReadableStream.prototype).length
Object.getOwnPropertyNames(ResizeObserver.prototype).length
Object.getOwnPropertyNames(RelativeOrientationSensor.prototype).length
Object.getOwnPropertyNames(RemotePlayback.prototype).length
Object.getOwnPropertyNames(ReportingObserver.prototype).length
Object.getOwnPropertyNames(Request.prototype).length
Object.getOwnPropertyNames(SVGAnimationElement.prototype).length
Object.getOwnPropertyNames(XMLHttpRequestEventTarget.prototype).length
```

```
Object.getOwnPropertyNames(SVGCircleElement.prototype).length
Object.getOwnPropertyNames(TreeWalker.prototype).length
Object.getOwnPropertyNames(WebGLTexture.prototype).length
Object.getOwnPropertyNames(TextDecoderStream.prototype).length
Object.getOwnPropertyNames(TextEncoderStream.prototype).length
Object.getOwnPropertyNames(WebGLSync.prototype).length
Object.getOwnPropertyNames(TextTrack.prototype).length
Object.getOwnPropertyNames(TextTrackCue.prototype).length
Object.getOwnPropertyNames(TextTrackCueList.prototype).length
Object.getOwnPropertyNames(WebGLShaderPrecisionFormat.prototype).length
Object.getOwnPropertyNames(TextTrackList.prototype).length
Object.getOwnPropertyNames(TimeRanges.prototype).length
Object.getOwnPropertyNames(Touch.prototype).length
Object.getOwnPropertyNames(TouchEvent.prototype).length
Object.getOwnPropertyNames(TouchList.prototype).length
Object.getOwnPropertyNames(TrackEvent.prototype).length
Object.getOwnPropertyNames(TransformStream.prototype).length
Object.getOwnPropertyNames(WebGLTransformFeedback.prototype).length
Object.getOwnPropertyNames(TextDecoder.prototype).length
Object.getOwnPropertyNames(WebGLUniformLocation.prototype).length
Object.getOwnPropertyNames(SVGTitleElement.prototype).length
Object.getOwnPropertyNames(WebGLVertexArrayObject.prototype).length
Object.getOwnPropertyNames(SVGSymbolElement.prototype).length
Object.getOwnPropertyNames(SVGTextContentElement.prototype).length
Object.getOwnPropertyNames(SVGTextElement.prototype).length
Object.getOwnPropertyNames(SVGTextPathElement.prototype).length
Object.getOwnPropertyNames(SVGTextPositioningElement.prototype).length
Object.getOwnPropertyNames(SVGTransform.prototype).length
Object.getOwnPropertyNames(TaskAttributionTiming.prototype).length
Object.getOwnPropertyNames(SVGTransformList.prototype).length
Object.getOwnPropertyNames(SVGTSpanElement.prototype).length
Object.getOwnPropertyNames(SVGUnitTypes.prototype).length
Object.getOwnPropertyNames(SVGUseElement.prototype).length
Object.getOwnPropertyNames(SVGViewElement.prototype).length
```

Appendix-4

To conduct an internal performance analysis on our trained model, it is essential to first define our accuracy metric. As mentioned in this paper, we are using a semi-supervised learning method (i.e. clustering). Our defined accuracy metric is how accurately our clusters represent the `user-agent` instances. For this evaluation, we consider a cluster assignment to be accurate if all instances of the same `user-agent` are mapped to the same cluster number (i.e., the one with the majority of data points for that `user-agent` string).

$$\text{Model accuracy} = \frac{\text{Number of user-agents mapped to their corresponding clusters}}{\text{Total number of user-agents}} \tag{1}$$

For example, if the majority of the data points with the `user-agent` of Chrome110 are assigned to cluster 0, we consider cluster 0 as the corresponding cluster for Chrome110. Any data point with the `user-agent` of Chrome110 that is assigned to a different cluster number is considered misclustered. Thus, when discussing model accuracy in the context of the following sensitivity analysis, we refer to the percentage of data in the dataset that is correctly assigned to the appropriate cluster numbers (Formula 1).

For the Browser Polygraph model trained in this paper, as discussed in Section 6.4, we have determined that *28 features, 7 PCA components, and 11 clusters (k=11)* are optimal for our fraud detection objectives. With these parameters, our model achieves a 99.6% accuracy in correctly assigning `user-agent` instances to clusters. We will demonstrate how changes to the number of clusters, PCA components, and features impact the accuracy of the trained model in Browser Polygraph.

• **Number of clusters:** to evaluate the impact of varying the number of clusters on model accuracy, we trained our model using different numbers of clusters (5, 7, 9, 11, 13, 15, 17, and 19) while maintaining a consistent feature set of 28 attributes and 7 principal components. As shown in Table 10, model accuracy generally decreases as the number of clusters increases. With a higher number of clusters, browser instances with close versions are more likely to be misclustered. This occurs because increasing the number of clusters separates instances with closely aligned feature values, whereby assigning them to distinct clusters.

An important consideration arises: why not use fewer clusters (i.e., fewer than k=11)? In practice, selecting too few clusters provides attackers with greater flexibility to manipulate browser attributes, enabling them to evade detection by Browser Polygraph. From a fraud detection standpoint, while using a larger number of clusters might detect more potential fraud cases, it is essential not to compromise accuracy, as this can lead to an increase in false positives, i.e. incorrectly identifying a non-fraudulent browser instance as a fraud case. As illustrated in Table 10, there is a noticeable decrease in accuracy (0.2%) from k=11 to k=13, indicating that k=11 is an optimal choice for balancing cluster count and model accuracy.

• **Number of PCA components:** here, we evaluate the influence of the number of PCA components on our model's accuracy, while keeping the number of features at 28. Table 11 presents our analysis for PCA components equal to 6 and greater than 6. Each experiment is followed by a similar analysis as detailed in Section 6.4 to determine the optimal number of clusters. We observe an increase in accuracy from PCA=6 to PCA=7, indicating that the additional component provides valuable information for accurately clustering browser instances. However, adding more components beyond 7 leads to a decrease in accuracy due to the introduction of noise. This observation can be attributed to the "curse of dimensionality," a phenomenon that occurs when too many features—particularly irrelevant or noisy ones—are added to a model. In high-dimensional spaces, data points become more sparse, leading to reduced clustering accuracy. Therefore, 7 number of PCA components lead to the most accurate clustering.

• **Number of features:** here, we explore the influence of selecting more than 28 features on the training process. We analyzed all 513 features from our dataset of 205k rows. The values were sorted based on standard deviation (excluding those deemed improper in Section 6.3), and at each step, we selected 4 additional features from those with higher standard deviations. Out of the 4 features added in each step, two were present across the releases of Chrome, Edge, and Firefox browsers, highlighting differences between browser versions. Meanwhile, the other two features, which are absent in Firefox, were selected to emphasize the differences between Firefox and Chrome/Edge browsers. For each step, we extracted the optimal number of PCA components (similar to Figure 2) based on the analysis in Section 6.4 and also determined the optimal number of clusters according to the logic in Section 6.4, by observing the highest relative WSS. The list of added features and the results of our analysis for each step are shown in Table 12. Moving from 28 to 32 features does not affect the number of PCA components or clusters; however, it decreases model accuracy. We believe this accuracy reduction is due to the introduction of noise and the curse of dimensionality from the added features. However, increasing the number of features in the other steps (i.e., steps with 36 and 42 features) does not impact the number of PCA components, but it increases the number of optimal clusters. This is because the added features introduce additional dimensions to our data points, potentially leading to the formation of more clusters, which, in turn, reduces model accuracy. To summarize, the analysis in Table 12 shows a decrease in accuracy as the number of features and clusters increases. For instance, increasing the number of features from 28 to 32 causes

Table 10: Sensitivity analysis for increased number of clusters with 28 features and 7 PCA components.

Number of clusters	5	7	9	11	13	15	17	19
Model accuracy	99.88%	99.69%	99.58%	99.60%	99.40%	99.31%	99.29%	99.26%

Table 11: Sensitivity analysis for different number of PCA components with 28 features.

Number of PCA components	6	7	8	9	10
Optimal number of clusters	11	11	11	11	11
Model accuracy	99.54%	99.60%	99.46%	99.46%	99.46%

Table 12: Sensitivity analysis for different number of features.

Features	Names of added features	PCA	k	Model accuracy
28	Appendix-2	7	11	99.60%

32	`Object.getOwnPropertyNames(HTMLIFrameElement.prototype).length`
	`Object.getOwnPropertyNames(SVGAElement.prototype).length`
	`Object.getOwnPropertyNames(RemotePlayback.prototype).length`
	`Object.getOwnPropertyNames(StylePropertyMapReadOnly.prototype).length`
		7	11	99.52%

36	`Object.getOwnPropertyNames(Screen.prototype).length`
	`Object.getOwnPropertyNames(Request.prototype).length`
	`Object.getOwnPropertyNames(TouchEvent.prototype).length`
	`Object.getOwnPropertyNames(TaskAttributionTiming.prototype).length`
		7	12	99.41%

42	`Object.getOwnPropertyNames(PictureInPictureWindow.prototype).length`
	`Object.getOwnPropertyNames(ReportingObserver.prototype).length`
	`Object.getOwnPropertyNames(HTMLTemplateElement.prototype).length`
	`Object.getOwnPropertyNames(MediaSession.prototype).length`
		7	14	99.41%

the accuracy to drop by 0.08%, which, for a dataset of 200k rows, could result in up to 160 additional false positives in the fraud detection system (a 13% increase compared to a model with 28 features).

The three evaluations above clearly demonstrate that our chosen parameters in Browser Polygraph (28 features, 7 PCA components, and k=11 clusters) lead to optimal accuracy in clustering browser instances.

Appendix-5

In this section, a performance comparison between Browser Polygraph and fine-grained fingerprinting techniques is provided. Synthetic data is being used to compare Browser Polygraph coarse-grained fingerprints and two of the fine-grained fingerprinting techniques (FingerprintJS and ClientJS).

Before diving into the experiments conducted, we would like to reiterate that fine-grained fingerprinting techniques were originally designed for user tracking by collecting detailed information about a user's OS, browser, configuration, and underlying hardware. These techniques typically generate a JSON object containing these details, which is hashed to create a user identifier for tracking purposes. However, to provide a testbed for comparison between fine-grained techniques and our proposed method, we have to further process the JSON object generated by the fine-grained fingerprinting data to extract relevant information that can be used as the inputs of a clustering method.

In this synthetic setup, we used BrowserStack to launch Chrome, Edge, and Firefox browser instances on Windows 10 and 11. The synthetic data are collected from (1) our developed website for Browser Polygraph, (2) FingerprintJS website, and (3) a custom website for ClientJS. In this experiment, we extracted 430 fingerprints from Browser Polygraph, 382 fingerprints from FingerprintJS, and 391 fingerprints from ClientJS along with their corresponding `user-agent` strings. To prepare the FingerprintJS and ClientJS data for clustering, we had to interpret the fingerprints provided in the JSON format. To do so, for nested objects within the JSON, we flattened the data by creating separate columns for each key. Then, we converted all values into numerical formats: numeric values were left unchanged, boolean values were mapped to 0 and 1, and strings were encoded as numerical categories. Any missing values were assigned a default value of -1. Subsequently, columns with unique values across all data points were excluded. Additionally, for ClientJS, since some features were directly extracted from the `user-agent` string, we excluded those features as well. All other features were retained for clustering. Ultimately, 268 features were extracted for clustering FingerprintJS data while only 7 useful features from ClientJS were obtained, as ClientJS does not provide many descriptive browser features. We also used 28 features from Browser Polygraph as discussed in Section 6.4.

To perform clustering, we followed features scaling, PCA components selection, and cluster numbers selection as described in Section 6.4. Details and results of clustering for these three datasets are shown in Table 13. As can be seen, clustering model in Browser Polygraph achieved 100% accuracy as it did not miscluster any browser instance. However, FingerprintJS had 99.21% model accuracy and ClientJS had model accuracy of 93.63% which falls behind the other two techniques. The results show that the coarse-grained features of Browser Polygraph outperform fine-grained techniques in clustering. That is because the coarse-grained features of Browser Polygraph provide broad insights into the behavior and attributes of browsers. In fact, broad features offer a more comprehensive view of the data, aiding in effectively grouping similar browsers. This causes the clustering based on coarse-grained features to outperform clustering that relies on fine-grained fingerprinting features, which collect detailed and individualistic user tracking information.

Table 13: Comparison of clustering performance between Browser Polygraph and fine-grained fingerprinting techniques using synthetic data generated by BrowserStack across Windows 10 and Windows 11 (for Chrome, Edge, and Firefox).

Technique	Size of dataset	Features	PCA	k	Model accuracy
Browser Polygraph	430	28	13	14	100%
FingerprintJS	382	268	55	16	99.21%
ClientJS	391	7	2	5	93.60%

Similar to Windows 10 and 11 as mentioned above, we repeated the synthetic experiment to collect the data for Browser Polygraph, FingerprintJS, and ClientJS on macOS Sequoia and macOS Sonoma. The results are shown in Table 14. As can be seen, the performance of clustering for the three methods on macOS is similar to the performances observed for Windows.

Table 14: Comparison of clustering performance between Browser Polygraph and fine-grained fingerprinting techniques using synthetic data generated by BrowserStack across macOS Sequoia and macOS Sonoma (for Chrome, Edge, and Firefox).

Technique	Size of dataset	Features	PCA	k	Model accuracy
Browser Polygraph	320	28	11	14	100%
FingerprintJS	325	589	36	9	99.38%
ClientJS	327	4	2	15	85.93%

The analysis in this section highlights that the explored fine-grained fingerprinting techniques, in their current form and without additional preprocessing, may not be well-suited for clustering purposes.